

Experiments with MATLAB

Cleve Moler
October 4, 2011

Copyright 2011

Cleve Moler

Electronic edition published by MathWorks, Inc.

<http://www.mathworks.com/moler>

Contents

Preface	iii
1 Iteration	1
2 Fibonacci Numbers	17
3 Calendars and Clocks	33
4 Matrices	45
5 Linear Equations	63
6 Fractal Fern	75
7 Google PageRank	83
8 Exponential Function	97
9 T Puzzle	113
10 Magic Squares	123
11 TicTacToe Magic	141
12 Game of Life	151
13 Mandelbrot Set	163
14 Sudoku	183
15 Ordinary Differential Equations	199
16 Predator-Prey Model	213
17 Orbits	221
18 Shallow Water Equations	241

19	Morse Code	247
20	Music	263

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. TM.

For more information about relevant MathWorks policies, see:

http://www.mathworks.com/company/aboutus/policies_statements

Preface

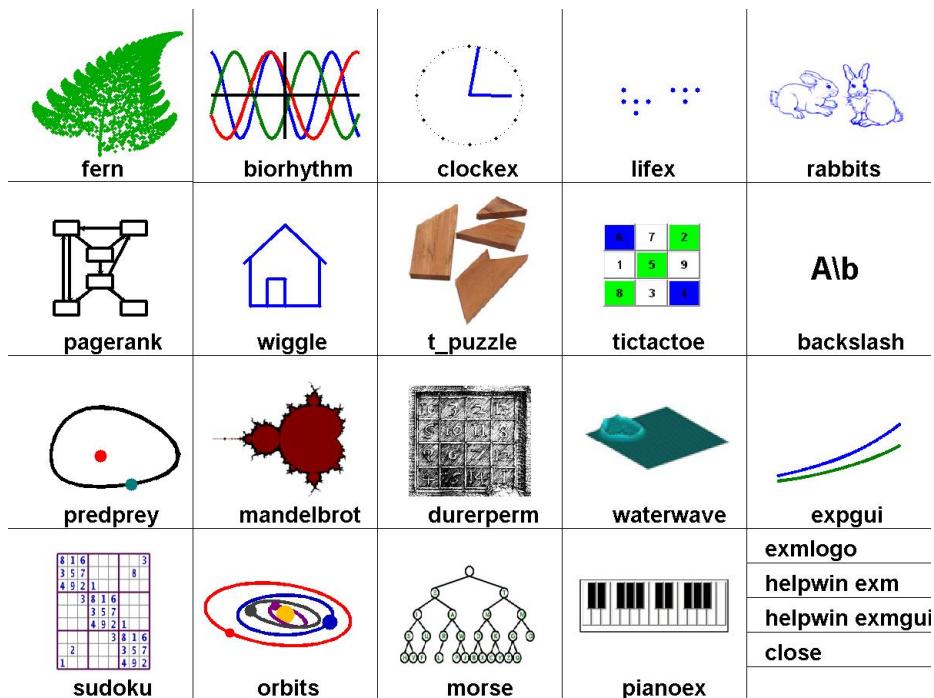


Figure 1. *exmgui* provides a starting point for some of the experiments.

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

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, ϕ , 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 ϕ 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
```

$$\begin{aligned}
 X &= \sqrt{1+x} \\
 X^2 &= 1+x \\
 X^2 - X - 1 &= 0 \\
 X &= \frac{1 \pm \sqrt{5}}{2} \\
 \phi &= \frac{1 + \sqrt{5}}{2}
 \end{aligned}$$

Figure 1.1. Compute the fixed point by hand.

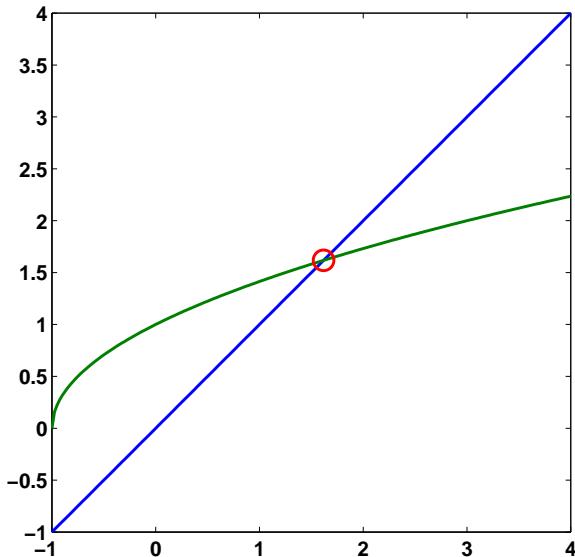


Figure 1.2. A fixed point at $\phi = 1.6180$.

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 ϕ . 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
```

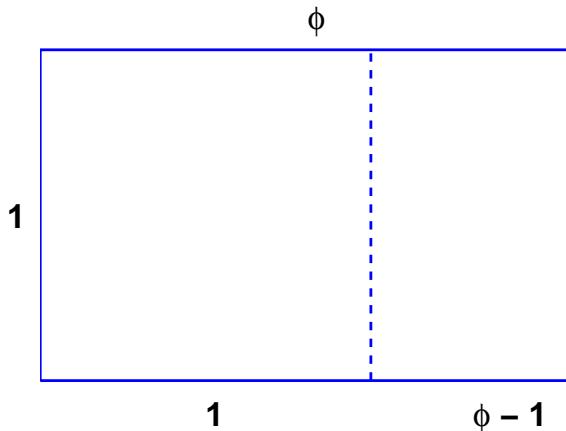


Figure 1.3. The golden rectangle.

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 ϕ :

$$\frac{1}{\phi} = \frac{\phi - 1}{1}.$$

Multiplying both sides of this equation by ϕ 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 ϵ , 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 π . 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.

$$\begin{array}{lll}
 \frac{43^2}{4^{(3^2)}} & \frac{-3^4}{(-3)^4} & \sin 1 \\
 \frac{(4^3)^2}{\sqrt[4]{32}} & \sqrt[4]{-3} & \sin \frac{\pi}{3} \\
 -2^{-4/3} & (\arcsin 1)/\pi
 \end{array}$$

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 .

$c = \text{something involving } f$

(b) Write a MATLAB statement that converts temperature in Celsius, c , to Fahrenheit, f .

$f = \text{something involving } c$

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, π^ϕ or ϕ^π ?

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 ϕ . 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 ϕ ?

```
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.

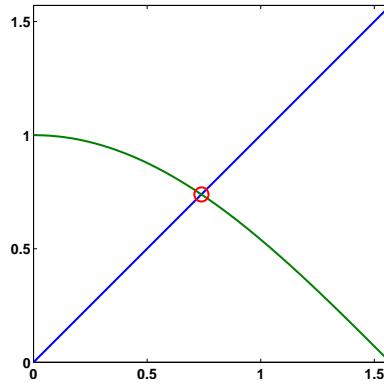


Figure 1.4. Fixed point of $x = \cos(x)$.

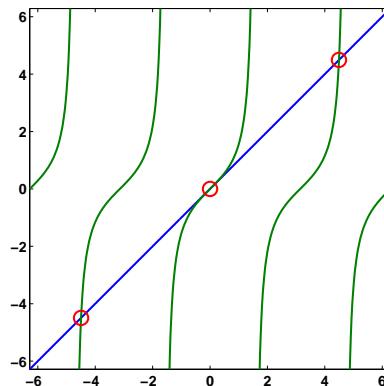


Figure 1.5. Three fixed points of $x = \tan(x)$

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])
```

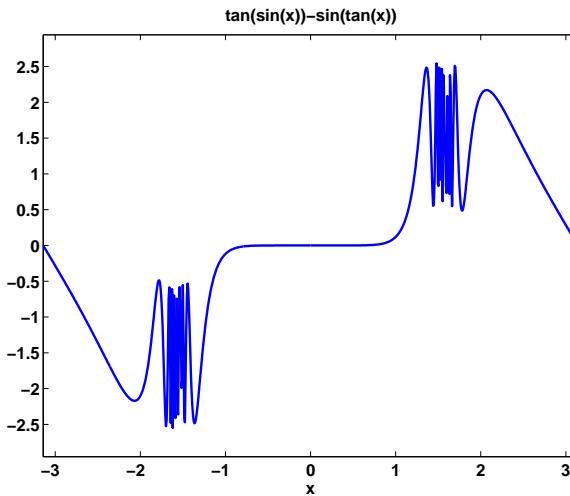


Figure 1.6. A wiggly function.

(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*(-3:1:3))
```

(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*

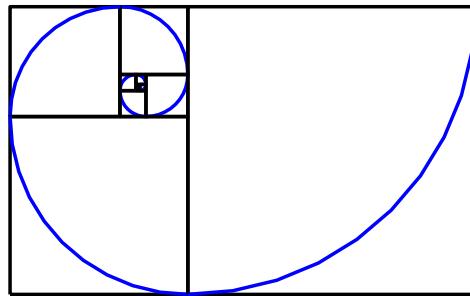


Figure 1.7. A spiral formed from golden rectangles and inscribed quarter circles.

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

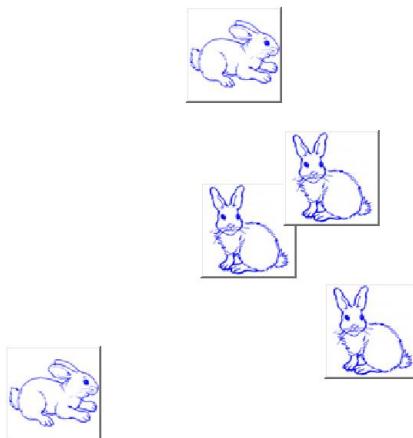


Figure 2.1. *Fibonacci's rabbits.*

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 ϕ can be expressed as an infinite continued fraction.

$$\phi = 1 + \frac{1}{1 + \frac{1}{1 + \frac{1}{\dots}}}$$

To verify this claim, suppose we did not know the value of this fraction. Let

$$x = 1 + \frac{1}{1 + \frac{1}{1 + \frac{1}{\dots}}}$$

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 ϕ ,

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 ϕ_n denote the continued fraction truncated after n terms. ϕ_n is a rational approximation to ϕ . Let's express ϕ_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 ϕ_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:

$$\lim_{n \rightarrow \infty} \frac{f_n}{f_{n-1}} = \phi.$$

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

```
2.000000000000000
1.500000000000000
1.66666666666667
1.600000000000000
1.625000000000000
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 ρ . 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 ρ , namely ϕ 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

$$f_0 = c_1 + c_2 = 1,$$

$$f_1 = c_1\phi + c_2(1 - \phi) = 1.$$

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:

$$c_1 = \frac{\phi}{2\phi - 1},$$

$$c_2 = -\frac{(1 - \phi)}{2\phi - 1}.$$

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 γ ?
- (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

$$\begin{aligned}d_n &= 0, \quad n \leq 0 \\d_1 &= 1, \\d_2 &= 1\end{aligned}$$

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 δ ?
- (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 .

$$\begin{aligned}F(x) &= \sum_{n=1}^{\infty} f_n x^n \\&= x + 2x^2 + 3x^3 + 5x^4 + 8x^5 + 13x^6 + \dots\end{aligned}$$

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 + \dots
\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

$$f_{k+1}x^{k+1} = f_kx^k + f_{k-1}x^{k-1}$$

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 \wedge 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.

$$\begin{aligned}
 F(x) &= \sum_{n=1}^{\infty} f_n x^n \\
 &= x + 2x^2 + 3x^3 + 5x^4 + 8x^5 + \dots \\
 &= x + (1+1)x^2 + (2+1)x^3 + (3+2)x^4 + (5+3)x^5 + \dots \\
 &= x + x^2 + x(x + 2x^2 + 3x^3 + 5x^4 + \dots) + x^2(x + 2x^2 + 3x^3 + \dots) \\
 &= x + x^2 + xF(x) + x^2F(x)
 \end{aligned}$$

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 3

Calendars and Clocks

Computations involving time, dates, biorhythms and Easter.

Calendars are interesting mathematical objects. The Gregorian calendar was first proposed in 1582. It has been gradually adopted by various countries and churches over the four centuries since then. The British Empire, including the colonies in North America, adopted it in 1752. Turkey did not adopt it until 1923. The Gregorian calendar is now the most widely used calendar in the world, but by no means the only one.

In the Gregorian calendar, a year y is a *leap year* if and only if y is divisible by 4 and not divisible by 100, or is divisible by 400. In MATLAB the following expression must be **true**. The double ampersands, '**&&**', mean "and" and the double vertical bars, '**||**', mean "or".

```
mod(y,4) == 0 && mod(y,100) ~= 0 || mod(y,400) == 0
```

For example, 2000 was a leap year, but 2100 will not be a leap year. This rule implies that the Gregorian calendar repeats itself every 400 years. In that 400-year period, there are 97 leap years, 4800 months, 20871 weeks, and 146097 days. The average number of days in a Gregorian calendar year is $365 + \frac{97}{400} = 365.2425$.

The MATLAB function **clock** returns a six-element vector **c** with elements

```
c(1) = year  
c(2) = month  
c(3) = day  
c(4) = hour  
c(5) = minute  
c(6) = seconds
```

Copyright © 2011 Cleve Moler
MATLAB® is a registered trademark of MathWorks, Inc.™
October 4, 2011

The first five elements are integers, while the sixth element has a fractional part that is accurate to milliseconds. The best way to print a `clock` vector is to use `fprintf` or `sprintf` with a specified *format string* that has both integer and floating point fields.

```
f = '%6d %6d %6d %6d %6d %.3f\n'
```

I am revising this chapter on August 2, 2011, at a few minutes after 2:00pm, so

```
c = clock;
fprintf(f,c);
```

produces

```
2011      8      2     14      2    19.470
```

In other words,

```
year = 2011
month = 8
day = 2
hour = 14
minute = 2
seconds = 19.470
```

The MATLAB functions `datenum`, `datevec`, `datestr`, and `weekday` use `clock` and facts about the Gregorian calendar to facilitate computations involving calendar dates. Dates are represented by their *serial date number*, which is the number of days since the theoretical time and day over 20 centuries ago when `clock` would have been six zeros. We can't pin that down to an actual date because different calendars would have been in use at that time.

The function `datenum` returns the date number for any `clock` vector. For example, using the vector `c` that I just found, the current date number is

```
datenum(c)
```

is

```
734717.585
```

This indicates that the current time is a little over halfway through day number 734717. I get the same result from

```
datenum(now)
```

or just

```
now
```

The `datenum` function also works with a given year, month and day, or a date specified as a string. For example

```
datenum(2011,8,2)
```

and

```
datenum('Aug. 2, 2011')
```

both return

```
734717
```

The same result is obtained from

```
fix(now)
```

In two and a half days the date number will be

```
datenum(fix(now+2.5))  
734720
```

Computing the difference between two date numbers gives an elapsed time measured in days. How many days are left between today and the first day of next year?

```
datenum('Jan 1, 2012') - datenum(fix(now))  
ans =  
152
```

The `weekday` function computes the day of the week, as both an integer between 1 and 7 and a string. For example both

```
[d,w] = weekday(datenum(2011,8,2))
```

and

```
[d,w] = weekday(now)
```

both return

```
d =  
3  
w =  
Tue
```

So today is the third day of the week, a Tuesday.

Friday the 13th

Friday the 13th is unlucky, but is it unlikely? What is the probability that the 13th day of any month falls on a Friday? The quick answer is 1/7, but that is not quite right. The following code counts the number of times that Friday occurs on the various weekdays in a 400 year calendar cycle and produces figure 3.1. (You can also run `friday13` yourself.)

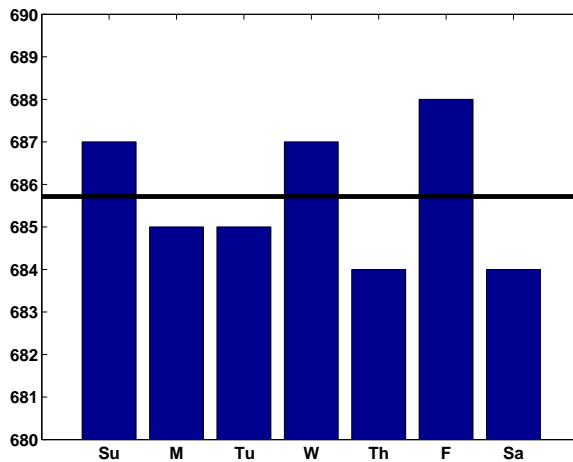


Figure 3.1. The 13th is more likely to be on Friday than any other day.

```
c = zeros(1,7);
for y = 1601:2000
    for m = 1:12
        d = datenum([y,m,13]);
        w = weekday(d);
        c(w) = c(w) + 1;
    end
end
c
bar(c)
axis([0 8 680 690])
avg = 4800/7;
line([0 8], [avg avg], 'linewidth',4,'color','black')
set(gca,'xticklabel',{'Su','M','Tu','W','Th','F','Sa'})
```

c =

687	685	685	687	684	688	684
-----	-----	-----	-----	-----	-----	-----

So the 13th day of a month is more likely to be on a Friday than any other day of the week. The probability is $688/4800 = .143333$. This probability is close to, but slightly larger than, $1/7 = .142857$.

Biorhythms

Biorhythms were invented over 100 years ago and entered our popular culture in the 1960s. You can still find many Web sites today that offer to prepare personalized biorhythms, or that sell software to compute them. Biorhythms are based on the notion that three sinusoidal cycles influence our lives. The physical cycle has a

period of 23 days, the emotional cycle has a period of 28 days, and the intellectual cycle has a period of 33 days. For any individual, the cycles are initialized at birth.

Figure 3.2 is my biorhythm, which begins on August 17, 1939, plotted for an eight-week period centered around the date this is being revised, July 27, 2011. It shows that I must be in pretty good shape. Today, I am near the peak of my intellectual cycle, and my physical and emotional cycles peaked on the same day less than a week ago.

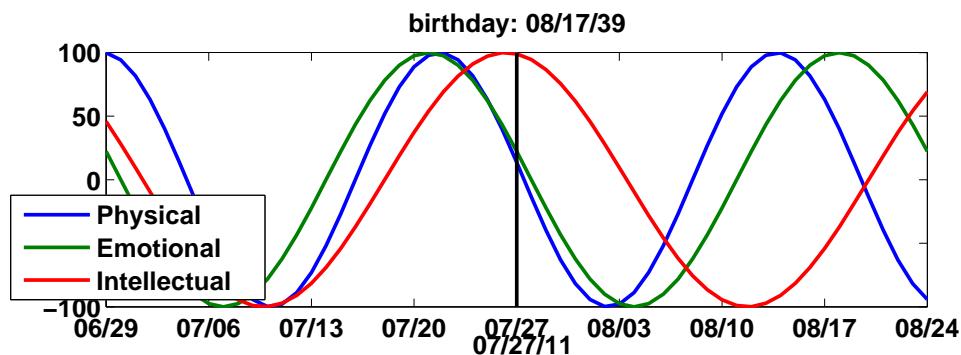


Figure 3.2. My biorhythm.

A search of the United States Government Patent and Trademark Office database of US patents issued between 1976 and 2010 finds 147 patents that are based upon or mention biorhythms. (There were just 113 in 2007.) The Web site is

<http://patft.uspto.gov>

The date and graphics functions in MATLAB make the computation and display of biorhythms particularly convenient. The following code segment is part of our program `biorhythm.m` that plots a biorhythm for an eight-week period centered on the current date.

```
t0 = datenum('Aug. 17, 1939')
t1 = fix(now);
t = (t1-28):1:(t1+28);
y = 100*[sin(2*pi*(t-t0)/23)
           sin(2*pi*(t-t0)/28)
           sin(2*pi*(t-t0)/33)];
plot(t,y)
```

You see that the time variable `t` is measured in days and that the trig functions take arguments measured in radians.

When is Easter?

Easter Day is one of the most important events in the Christian calendar. It is also one of the most mathematically elusive. In fact, regularization of the observance of Easter was one of the primary motivations for calendar reform. The informal rule is that Easter Day is the first Sunday after the first full moon after the vernal equinox. But the ecclesiastical full moon and equinox involved in this rule are not always the same as the corresponding astronomical events, which, after all, depend upon the location of the observer on the earth. Computing the date of Easter is featured in Don Knuth's classic *The Art of Computer Programming* and has consequently become a frequent exercise in programming courses. Our MATLAB version of Knuth's program, `easter.m`, is the subject of several exercises in this chapter.

Recap

```
%% Calendar Chapter Recap
% This is an executable program that illustrates the statements
% introduced in the Calendar Chapter of "Experiments in MATLAB".
% You can access it with
%
%     calendar_recap
%     edit calendar_recap
%     publish calendar_recap
%
% Related EXM programs
%
%     biorhythm.m
%     easter.m
%     clockex.m
%     friday13.m

%% Clock and fprintf
format bank
c = clock
f = '%6d %6d %6d %6d %6d %9.3f\n'
fprintf(f,c);

%% Modular arithmetic
y = c(1)
is_leapyear = (mod(y,4) == 0 && mod(y,100) ~= 0 || mod(y,400) == 0)

%% Date functions
c = clock;
dnum = datenum(c)
dnow = fix(now)
```

```

xmas = datenum(c(1),12,25)
days_till_xmas = xmas - dnow
[~,wday] = weekday(now)

%% Count Friday the 13th's
c = zeros(1,7);
for y = 1:400
    for m = 1:12
        d = datenum([y,m,13]);
        w = weekday(d);
        c(w) = c(w) + 1;
    end
end
format short
c

%% Biorhythms
bday = datenum('8/17/1939')
t = (fix(now)-bday) + (-28:28);
y = 100*[sin(2*pi*t/23)
          sin(2*pi*t/28)
          sin(2*pi*t/33)];
plot(t,y)
axis tight

```

Exercises

3.1 *Microcentury.* The optimum length of a classroom lecture is one *microcentury*. How long is that?

3.2 $\pi \cdot 10^7$. A good estimate of the number of seconds in a year is $\pi \cdot 10^7$. How accurate is this estimate?

3.3 `datestr`. What does the following program do?

```

for k = 1:31
    disp(datestr(now,k))
end

```

3.4 `calendar` (a) What does the `calendar` function do? (If you have the Financial Toolbox, `help calendar` will also tell you about it.) Try this:

```

y = your birth year    % An integer, 1900 <= y <= 2100.
m = your birth month   % An integer, 1 <= m <= 12.

```

```
calendar(y,m)
```

- (b) What does the following code do?

```
c = sum(1:30);
s = zeros(12,1);
for m = 1:12
    s(m) = sum(sum(calendar(2011,m)));
end
bar(s/c)
axis([0 13 0.80 1.15])
```

- 3.5 *Another clock.* What does the following program do?

```
clf
set(gcf,'color','white')
axis off
t = text(0.0,0.5,' ','fontsize',16,'fontweight','bold');
while 1
    s = dec2bin(fix(86400*now));
    set(t,'string',s)
    pause(1)
end
```

Try

```
help dec2bin
```

if you need help.

- 3.6 *How long?* You should not try to run the following program. But if you were to run it, how long would it take? (If you insist on running it, change both 3's to 5's.)

```
c = clock
d = c(3)
while d == c(3)
    c = clock;
end
```

- 3.7 *First datenum.* The first countries to adopt the Gregorian calendar were Spain, Portugal and much of Italy. They did so on October 15, 1582, of the new calendar. The previous day was October 4, 1582, using the old, Julian, calendar. So October 5 through October 14, 1582, did not exist in these countries. What is the MATLAB serial date number for October 15, 1582?

- 3.8 *Future datenum's.* Use `datestr` to determine when `datenum` will reach 750,000. When will it reach 1,000,000?

3.9 *Your birthday.* On which day of the week were you born? In a 400-year Gregorian calendar cycle, what is the probability that your birthday occurs on a Saturday? Which weekday is the most likely for your birthday?

3.10 *Ops per century.* Which does more operations, a human computer doing one operation per second for a century, or an electronic computer doing one operation per microsecond for a minute?

3.11 *Julian day.* The Julian Day Number (JDN) is commonly used to date astronomical observations. Find the definition of Julian Day on the Web and explain why

$$\text{JDN} = \text{datenum} + 1721058.5$$

In particular, why does the conversion include an 0.5 fractional part?

3.12 *Unix time.* The Unix operating system and POSIX operating system standard measure time in seconds since 00:00:00 Universal time on January 1, 1970. There are 86,400 seconds in one day. Consequently, Unix time, `time_t`, can be computed in MATLAB with

$$\text{time_t} = 86400 * (\text{datenum}(y, m, d) - \text{datenum}(1970, 1, 1))$$

Some Unix systems store the time in an 32-bit signed integer register. When will `time_t` exceed 2^{31} and overflow on such systems?

3.13 *Easter.*

- (a) The comments in `easter.m` use the terms “golden number”, “epact”, and “metonic cycle”. Find the definitions of these terms on the Web.
- (b) Plot a `bar` graph of the dates of Easter during the 21-st century.
- (c) How many times during the 21-st century does Easter occur in March and how many in April?
- (d) On how many different dates can Easter occur? What is the earliest? What is the latest?
- (e) Is the date of Easter a periodic function of the year number?

3.14 *Biorhythms.*

- (a) Use `biorhythm` to plot your own biorhythm, based on your birthday and centered around the current date.
- (b) All three biorhythm cycles start at zero when you were born. When do they return to this initial condition? Compute the m , the least common multiple of 23, 28, and 33.

$$m = \text{lcm}(\text{lcm}(23, 28), 33)$$

Now try

$$\text{biorhythm}(\text{fix}(\text{now}) - m)$$

What is special about this biorhythm? How old were you, or will you be, m days after you were born?

(c) Is it possible for all three biorhythm cycles to reach their maximum at exactly the same time? Why or why not? Try

```
t = 17003
biorhythm(fix(now)-t)
```

What is special about this biorhythm? How many years is t days? At first glance, it appears that all three cycles are reaching maxima at the same time, t . But if you look more closely at

```
sin(2*pi*t./[23 28 33])
```

you will see that the values are not all exactly 1.0. At what times near $t = 17003$ do the three cycles actually reach their maxima? The three times are how many hours apart? Are there any other values of t between 0 and the least common multiple, m , from the previous exercise where the cycles come so close to obtaining a simultaneous maximum? This code will help you answer these questions.

```
m = lcm(lcm(23,28),33);
t = (1:m)';
find(mod(t,23)==6 & mod(t,28)==7 & mod(t,33)==8)
```

(d) Is it possible for all three biorhythm cycles to reach their minimum at exactly the same time? Why or why not? When do they nearly reach a simultaneous minimum?

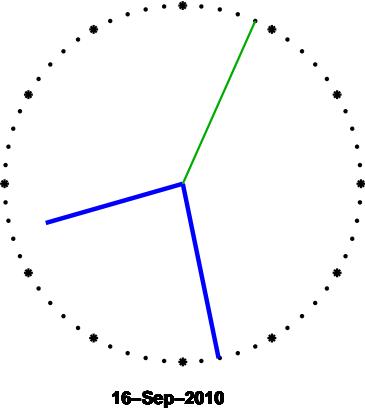


Figure 3.3. *clockex*

3.15 *clockex*. This exercise is about the `clockex` program in our `exm` toolbox and shown in figure 3.3.

(a) Why does `clockex` use trig functions?

- (b) Make `clockex` run counter-clockwise.
- (c) Why do the hour and minute hands in `clockex` move nearly continuously while the second hand moves in discrete steps.
- (d) The second hand sometime skips two marks. Why? How often?
- (e) Modify `clockex` to have a digital display of your own design.

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

$$sx = \begin{pmatrix} sx_1 \\ sx_2 \end{pmatrix}$$

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

$$A = \begin{pmatrix} 4 & -3 \\ -2 & 1 \end{pmatrix}$$

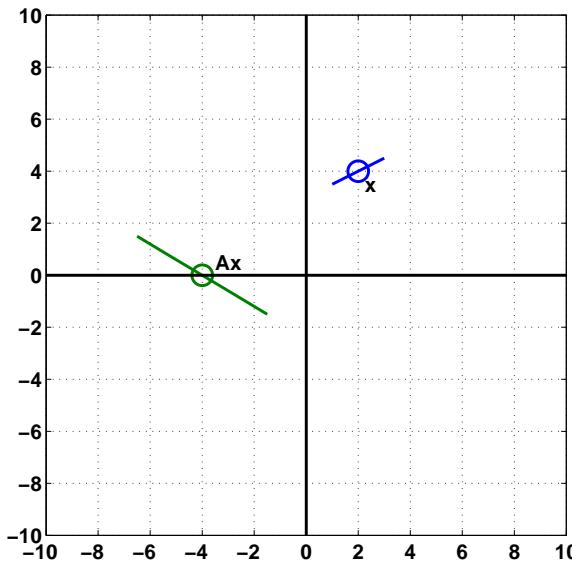


Figure 4.1. Matrix multiplication transforms lines through x to lines through Ax .

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,

$$x^T = (2 \ 4)$$

$$A^T = \begin{pmatrix} 4 & -2 \\ -3 & 1 \end{pmatrix}$$

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 \mathbf{x}' for the transpose of \mathbf{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

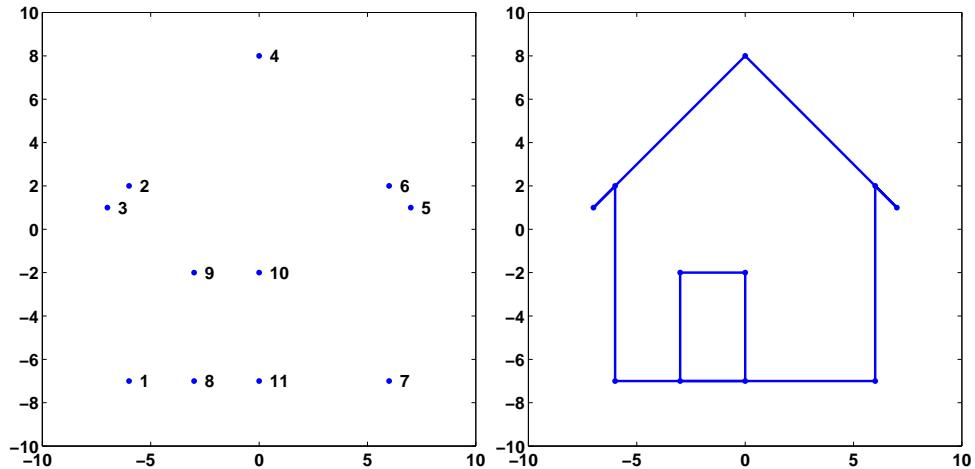


Figure 4.2. Connect the dots.

```
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

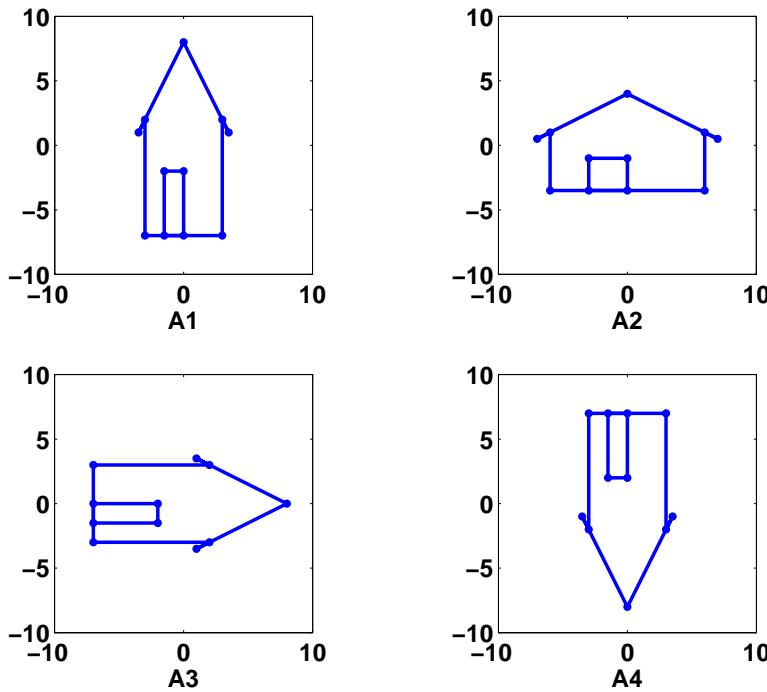


Figure 4.3. The effect of multiplication by scaling matrices.

right angles to each other and parallel to the axes. The matrix A_1 shrinks the first coordinate to reduce the width of the house while the height remains unchanged. The matrix A_2 shrinks the second coordinate to reduce the height, but not the width. The matrix A_3 interchanges the two coordinates while shrinking one of them. The matrix A_4 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
```

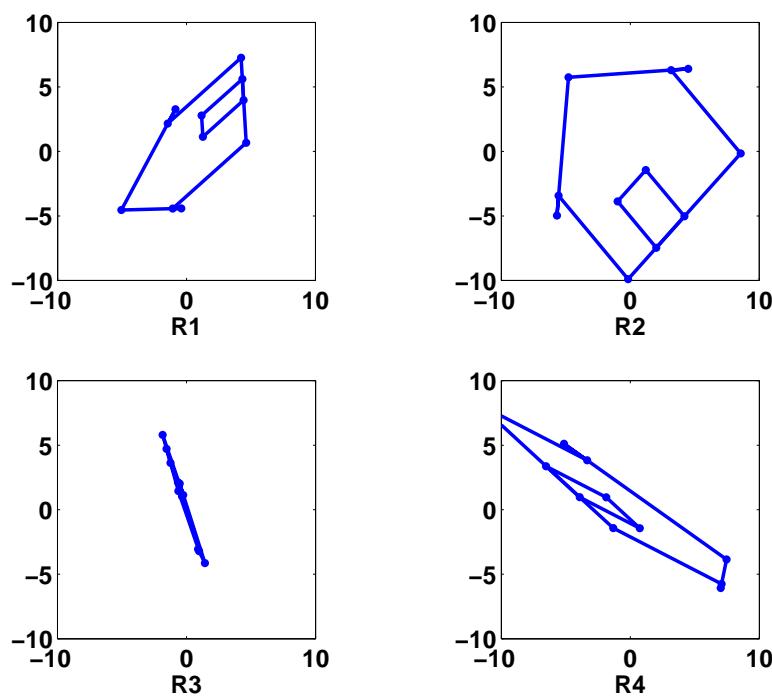


Figure 4.4. The effect of multiplication by random matrices.

Figure 4.4 shows the effect of multiplication by these four matrices on the house. Matrices R_1 and R_4 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 R_2 is close to a rotation, which we will discuss shortly. Matrix R_3 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 θ . Figure 4.5 shows the effect of multiplication by the plane rotations with $\theta = 15^\circ, 45^\circ, 90^\circ$, and 215° .

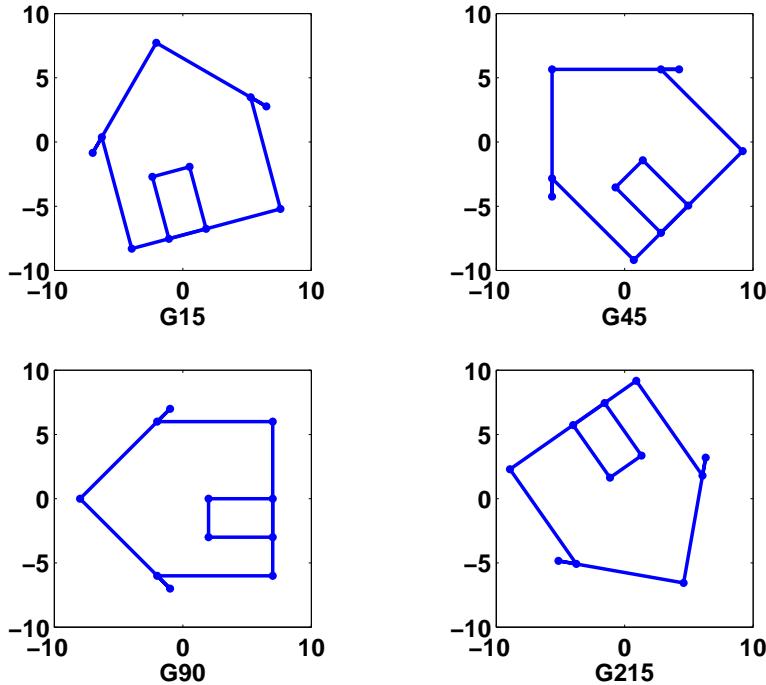


Figure 4.5. The affect of multiplication by plane rotations though $15^\circ, 45^\circ, 90^\circ$, and 215° .

```
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 G_{45} is fairly close to the random matrix R_2 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 θ computed by

```
theta = (4*abs(t-round(t))-1) * thetamax;
```

varies in a sawtooth fashion between `-thetamax` and `thetamax`. The graph of θ as a function of t is shown in figure 4.6. Each value of θ produces a corresponding plane rotation $G(\theta)$. Then

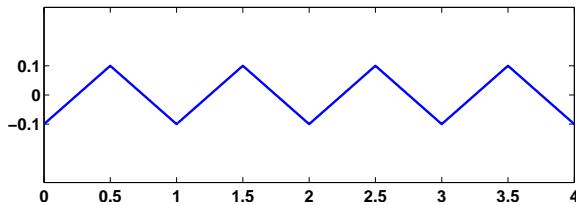


Figure 4.6. Wiggle angle θ

```
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].'
```

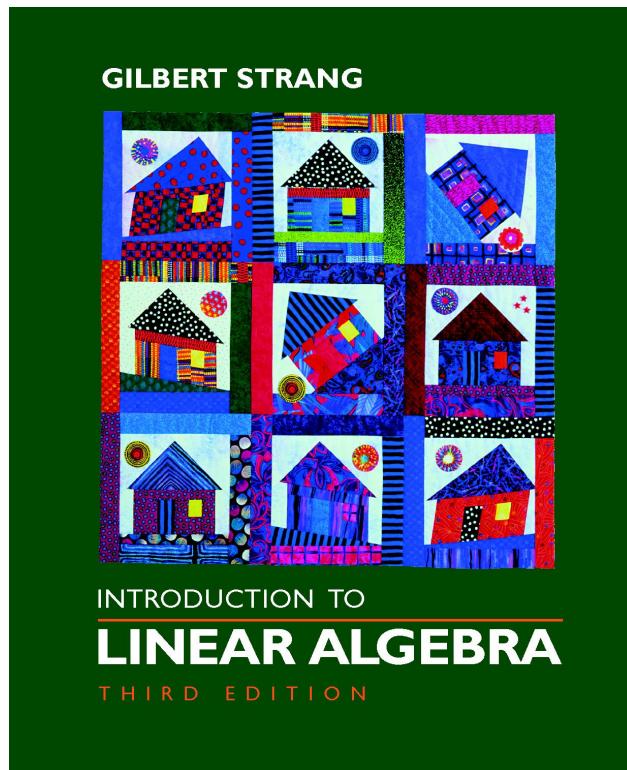


Figure 4.7. The cover of Gilbert Strang's textbook shows a quilt by Chris Curtis.

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 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.
% 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)`

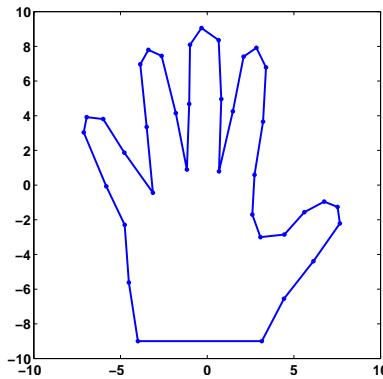


Figure 4.8. A hand.

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

$$0x = b$$

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.

$$\begin{aligned} 3x_1 + 12x_2 + x_3 &= 2.36 \\ 12x_1 + 2x_3 &= 5.26 \\ 2x_2 + 3x_3 &= 2.77 \end{aligned}$$

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.

$$\mathbf{x} = \mathbf{b}/\mathbf{A}$$

The two operations $\mathbf{A}\backslash\mathbf{b}$ and \mathbf{b}/\mathbf{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 $7\backslash 21$ 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

$$\begin{aligned}\mathbf{A} = \\ \begin{array}{ccc} 3 & 12 & 1 \\ 12 & 0 & 2 \\ 6 & 0 & 1 \end{array}\end{aligned}$$

and

$$\begin{aligned}\mathbf{b} = \\ \begin{array}{c} 2.36 \\ 5.26 \\ 2.77 \end{array}\end{aligned}$$

At first glance, this does not look like much of a change. However,

$$\mathbf{x} = \mathbf{A}\backslash\mathbf{b}$$

produces

`Warning: Matrix is singular to working precision.`

$$\begin{aligned}\mathbf{x} = \\ \begin{array}{c} \text{NaN} \\ -\text{Inf} \\ \text{Inf} \end{array}\end{aligned}$$

`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

$$\begin{aligned} b = \\ 2.36 \\ 5.26 \\ 2.63 \end{aligned}$$

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.

$$\begin{aligned} x = \\ 0.29 \\ 0.05 \\ 0.89 \\ A*x = \\ 2.36 \\ 5.26 \\ 2.63 \end{aligned}$$

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.

$$\begin{aligned} y = A(1:2,1:2)\backslash b(1:2); \\ y(3) = 0 \end{aligned}$$

$$\begin{aligned} y = \\ 0.44 \\ 0.09 \\ 0.00 \\ A*y = \\ 2.36 \\ 5.26 \\ 2.63 \end{aligned}$$

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 columns. 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

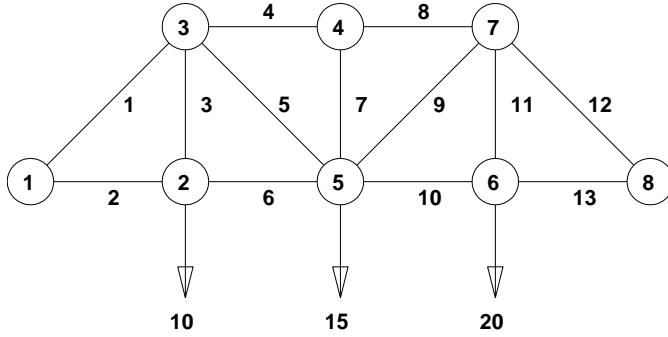


Figure 5.1. A plane truss.

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 :

$$\begin{aligned} \text{Joint 2:} \quad & f_2 = f_6, \\ & f_3 = 10; \end{aligned}$$

$$\begin{aligned} \text{Joint 3:} \quad & \alpha f_1 = f_4 + \alpha f_5, \\ & \alpha f_1 + f_3 + \alpha f_5 = 0; \end{aligned}$$

$$\begin{aligned} \text{Joint 4:} \quad & f_4 = f_8, \\ & f_7 = 0; \end{aligned}$$

$$\begin{aligned} \text{Joint 5:} \quad & \alpha f_5 + f_6 = \alpha f_9 + f_{10}, \\ & \alpha f_5 + f_7 + \alpha f_9 = 15; \end{aligned}$$

$$\begin{aligned} \text{Joint 6:} \quad & f_{10} = f_{13}, \\ & f_{11} = 20; \end{aligned}$$

$$\begin{aligned} \text{Joint 7:} \quad & f_8 + \alpha f_9 = \alpha f_{12}, \\ & \alpha f_9 + f_{11} + \alpha f_{12} = 0; \end{aligned}$$

$$\text{Joint 8:} \quad f_{13} + \alpha f_{12} = 0.$$

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.

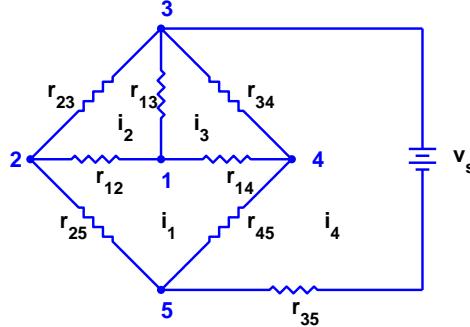


Figure 5.2. A resistor network.

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 $v_k, k = 1, \dots, 4$, denote the voltage difference between each of the first four nodes and node number 5 and let $i_k, k = 1, \dots, 4$, 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:

$$R\mathbf{i} = \mathbf{b}.$$

Here \mathbf{i} is the current vector,

$$\mathbf{i} = \begin{pmatrix} i_1 \\ i_2 \\ i_3 \\ i_4 \end{pmatrix},$$

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

$$x \rightarrow Ax + b,$$

with different matrices A and vectors b . These are known as *affine transformations*. The most frequently used transformation has

$$A = \begin{pmatrix} 0.85 & 0.04 \\ -0.04 & 0.85 \end{pmatrix}, \quad b = \begin{pmatrix} 0 \\ 1.6 \end{pmatrix}.$$

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

```

```

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

$$-3 \leq x_1 \leq 3, \quad 0 \leq x_2 \leq 10,$$

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

$$b = \begin{pmatrix} 0 \\ 0 \end{pmatrix}, \quad b = \begin{pmatrix} 1/2 \\ 0 \end{pmatrix}, \quad \text{and} \quad b = \begin{pmatrix} 1/4 \\ \sqrt{3}/4 \end{pmatrix}.$$

Chapter 7

Google PageRank

The world's largest matrix computation. (This chapter is out of date and needs a major overhaul.)

One of the reasons why Google™ is such an effective search engine is the PageRank™ algorithm developed by Google's founders, Larry Page and Sergey Brin, when they were graduate students at Stanford University. PageRank is determined entirely by the link structure of the World Wide Web. It is recomputed about once a month and does not involve the actual content of any Web pages or individual queries. Then, for any particular query, Google finds the pages on the Web that match that query and lists those pages in the order of their PageRank.

Imagine surfing the Web, going from page to page by randomly choosing an outgoing link from one page to get to the next. This can lead to dead ends at pages with no outgoing links, or cycles around cliques of interconnected pages. So, a certain fraction of the time, simply choose a random page from the Web. This theoretical random walk is known as a *Markov chain* or *Markov process*. The limiting probability that an infinitely dedicated random surfer visits any particular page is its PageRank. A page has high rank if other pages with high rank link to it.

Let W be the set of Web pages that can be reached by following a chain of hyperlinks starting at some root page, and let n be the number of pages in W . For Google, the set W actually varies with time, but by June 2004, n was over 4 billion. Let G be the n -by- n *connectivity matrix* of a portion of the Web, that is, $g_{ij} = 1$ if there is a hyperlink to page i from page j and $g_{ij} = 0$ otherwise. The matrix G can be huge, but it is very sparse. Its j th column shows the links on the j th page. The number of nonzeros in G is the total number of hyperlinks in W .

Copyright © 2011 Cleve Moler
MATLAB® is a registered trademark of MathWorks, Inc.™
October 4, 2011

Let r_i and c_j be the row and column sums of G :

$$r_i = \sum_j g_{ij}, \quad c_j = \sum_i g_{ij}.$$

The quantities r_j and c_j are the *in-degree* and *out-degree* of the j th page. Let p be the probability that the random walk follows a link. A typical value is $p = 0.85$. Then $1 - p$ is the probability that some arbitrary page is chosen and $\delta = (1 - p)/n$ is the probability that a particular random page is chosen. Let A be the n -by- n matrix whose elements are

$$a_{ij} = \begin{cases} pg_{ij}/c_j + \delta & : c_j \neq 0 \\ 1/n & : c_j = 0. \end{cases}$$

Notice that A comes from scaling the connectivity matrix by its column sums. The j th column is the probability of jumping from the j th page to the other pages on the Web. If the j th page is a dead end, that is has no out-links, then we assign a uniform probability of $1/n$ to all the elements in its column. Most of the elements of A are equal to δ , the probability of jumping from one page to another without following a link. If $n = 4 \cdot 10^9$ and $p = 0.85$, then $\delta = 3.75 \cdot 10^{-11}$.

The matrix A is the transition probability matrix of the Markov chain. Its elements are all strictly between zero and one and its column sums are all equal to one. An important result in matrix theory known as the *Perron–Frobenius theorem* applies to such matrices. It concludes that a nonzero solution of the equation

$$x = Ax$$

exists and is unique to within a scaling factor. If this scaling factor is chosen so that

$$\sum_i x_i = 1,$$

then x is the *state vector* of the Markov chain and is Google's PageRank. The elements of x are all positive and less than one.

The vector x is the solution to the singular, homogeneous linear system

$$(I - A)x = 0.$$

For modest n , an easy way to compute x in MATLAB is to start with some approximate solution, such as the PageRanks from the previous month, or

```
x = ones(n,1)/n
```

Then simply repeat the assignment statement

```
x = A*x
```

until successive vectors agree to within a specified tolerance. This is known as the *power method* and is about the only possible approach for very large n .

In practice, the matrices G and A are never actually formed. One step of the power method would be done by one pass over a database of Web pages, updating weighted reference counts generated by the hyperlinks between pages.

The best way to compute PageRank in MATLAB is to take advantage of the particular structure of the Markov matrix. Here is an approach that preserves the sparsity of G . The transition matrix can be written

$$A = pGD + ez^T$$

where D is the diagonal matrix formed from the reciprocals of the outdegrees,

$$d_{jj} = \begin{cases} 1/c_j & : c_j \neq 0 \\ 0 & : c_j = 0, \end{cases}$$

e is the n -vector of all ones, and z is the vector with components

$$z_j = \begin{cases} \delta & : c_j \neq 0 \\ 1/n & : c_j = 0. \end{cases}$$

The rank-one matrix ez^T accounts for the random choices of Web pages that do not follow links. The equation

$$x = Ax$$

can be written

$$(I - pGD)x = \gamma e$$

where

$$\gamma = z^T x.$$

We do not know the value of γ because it depends upon the unknown vector x , but we can temporarily take $\gamma = 1$. As long as p is strictly less than one, the coefficient matrix $I - pGD$ is nonsingular and the equation

$$(I - pGD)x = e$$

can be solved for x . Then the resulting x can be rescaled so that

$$\sum_i x_i = 1.$$

Notice that the vector z is not actually involved in this calculation.

The following MATLAB statements implement this approach

```
c = sum(G,1);
k = find(c~=0);
D = sparse(k,k,1./c(k),n,n);
e = ones(n,1);
I = speye(n,n);
x = (I - p*G*D)\e;
x = x/sum(x);
```

The power method can also be implemented in a way that does not actually form the Markov matrix and so preserves sparsity. Compute

```
G = p*G*D;
z = ((1-p)*(c~=0) + (c==0))/n;
```

Start with

```
x = e/n
```

Then repeat the statement

```
x = G*x + e*(z*x)
```

until x settles down to several decimal places.

It is also possible to use an algorithm known as *inverse iteration*.

```
A = p*G*D + delta
x = (I - A)\e
x = x/sum(x)
```

At first glance, this appears to be a very dangerous idea. Because $I - A$ is theoretically singular, with exact computation some diagonal element of the upper triangular factor of $I - A$ should be zero and this computation should fail. But with roundoff error, the computed matrix $I - A$ is probably not exactly singular. Even if it is singular, roundoff during Gaussian elimination will most likely prevent any exact zero diagonal elements. We know that Gaussian elimination with partial pivoting always produces a solution with a small residual, relative to the computed solution, even if the matrix is badly conditioned. The vector obtained with the backslash operation, $(I - A)\backslash e$, usually has very large components. If it is rescaled by its sum, the residual is scaled by the same factor and becomes very small. Consequently, the two vectors x and $A*x$ equal each other to within roundoff error. In this setting, solving the singular system with Gaussian elimination blows up, but it blows up in exactly the right direction.

Figure 7.1 is the graph for a tiny example, with $n = 6$ instead of $n = 4 \cdot 10^9$. Pages on the Web are identified by strings known as *uniform resource locators*, or *URLs*. Most URLs begin with `http` because they use the *hypertext transfer protocol*. In MATLAB, we can store the URLs as an array of strings in a *cell array*. This example involves a 6-by-1 cell array.

```
U = {'http://www.alpha.com'
      'http://www.beta.com'
      'http://www.gamma.com'
      'http://www.delta.com'
      'http://www.rho.com'
      'http://www.sigma.com'}
```

Two different kinds of indexing into cell arrays are possible. Parentheses denote subarrays, including individual cells, and curly braces denote the contents of the cells. If k is a scalar, then $U(k)$ is a 1-by-1 cell array consisting of the k th cell in U , while $U\{k\}$ is the string in that cell. Thus $U(1)$ is a single cell and $U\{1\}$ is the string '`http://www.alpha.com`'. Think of mail boxes with addresses on a city street. $B(502)$ is the box at number 502, while $B\{502\}$ is the mail in that box.

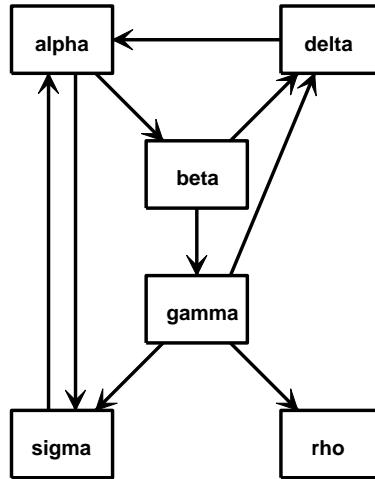


Figure 7.1. A tiny Web.

We can generate the connectivity matrix by specifying the pairs of indices (i, j) of the nonzero elements. Because there is a link to `beta.com` from `alpha.com`, the $(2, 1)$ element of G is nonzero. The nine connections are described by

```
i = [ 2 6 3 4 4 5 6 1 1]
j = [ 1 1 2 2 3 3 3 4 6]
```

A sparse matrix is stored in a data structure that requires memory only for the nonzero elements and their indices. This is hardly necessary for a 6-by-6 matrix with only 27 zero entries, but it becomes crucially important for larger problems. The statements

```
n = 6
G = sparse(i,j,1,n,n);
full(G)
```

generate the sparse representation of an n -by- n matrix with ones in the positions specified by the vectors i and j and display its full representation.

0	0	0	1	0	1
1	0	0	0	0	0
0	1	0	0	0	0
0	1	1	0	0	0
0	0	1	0	0	0
1	0	1	0	0	0

The statement

```
c = full(sum(G))
```

computes the column sums

```
c =
2      2      3      1      0      1
```

Notice that $c(5) = 0$ because the 5th page, labeled `rho`, has no out-links.

The statements

```
x = (I - p*G*D)\e
x = x/sum(x)
```

solve the sparse linear system to produce

```
x =
0.3210
0.1705
0.1066
0.1368
0.0643
0.2007
```

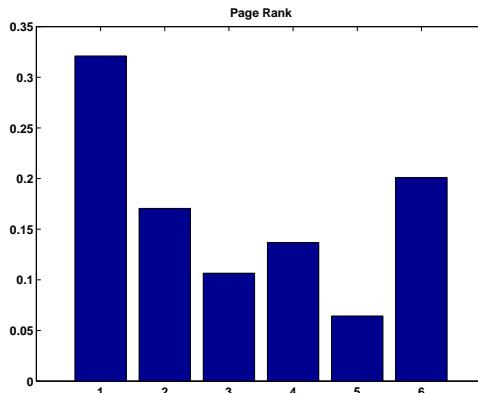


Figure 7.2. Page Rank for the tiny Web

The `bar` graph of \mathbf{x} is shown in figure 7.2. If the URLs are sorted in PageRank order and listed along with their in- and out-degrees, the result is

	page-rank	in	out	url
1	0.3210	2	2	http://www.alpha.com
6	0.2007	2	1	http://www.sigma.com
2	0.1705	1	2	http://www.beta.com
4	0.1368	2	1	http://www.delta.com
3	0.1066	1	3	http://www.gamma.com
5	0.0643	1	0	http://www.rho.com

We see that `alpha` has a higher PageRank than `delta` or `sigma`, even though they all have the same number of in-links. A random surfer will visit `alpha` over 32% of the time and `rho` only about 6% of the time.

For this tiny example with $p = .85$, the smallest element of the Markov transition matrix is $\delta = .15/6 = .0250$.

```
A =
0.0250    0.0250    0.0250    0.8750    0.1667    0.8750
0.4500    0.0250    0.0250    0.0250    0.1667    0.0250
0.0250    0.4500    0.0250    0.0250    0.1667    0.0250
0.0250    0.4500    0.3083    0.0250    0.1667    0.0250
0.0250    0.0250    0.3083    0.0250    0.1667    0.0250
0.4500    0.0250    0.3083    0.0250    0.1667    0.0250
```

Notice that the column sums of `A` are all equal to one.

The `exm` toolbox includes the program `surfer`. A statement like

```
[U,G] = surfer('http://www.www.ddd.aaa',n)
```

starts at a specified URL and tries to surf the Web until it has visited `n` pages. If successful, it returns an `n`-by-1 cell array of URLs and an `n`-by-`n` sparse connectivity matrix. The function uses `urlread`, which was introduced in MATLAB 6.5, along with underlying Java utilities to access the Web. Surfing the Web automatically is a dangerous undertaking and this function must be used with care. Some URLs contain typographical errors and illegal characters. There is a list of URLs to avoid that includes `.gif` files and Web sites known to cause difficulties. Most importantly, `surfer` can get completely bogged down trying to read a page from a site that appears to be responding, but that never delivers the complete page. When this happens, it may be necessary to have the computer's operating system ruthlessly terminate MATLAB. With these precautions in mind, you can use `surfer` to generate your own PageRank examples.

The statement

```
[U,G] = surfer('http://www.harvard.edu',500)
```

accesses the home page of Harvard University and generates a 500-by-500 test case. The graph generated in August 2003 is available in the `exm` toolbox. The statements

```
load harvard500
spy(G)
```

produce a `spy` plot (Figure 7.3) that shows the nonzero structure of the connectivity matrix. The statement

```
pagerank(U,G)
```

computes page ranks, produces a bar graph (Figure 7.4) of the ranks, and prints the most highly ranked URLs in PageRank order.

For the `harvard500` data, the dozen most highly ranked pages are

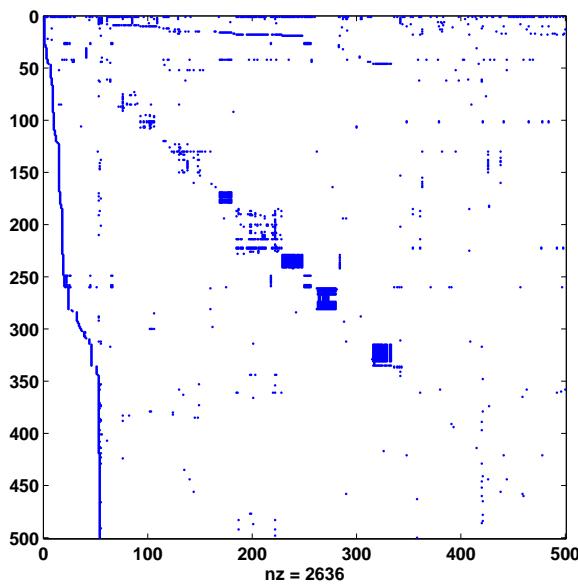


Figure 7.3. Spy plot of the harvard500 graph.

	page-rank	in	out	url
1	0.0843	195	26	http://www.harvard.edu
10	0.0167	21	18	http://www.hbs.edu
42	0.0166	42	0	http://search.harvard.edu:8765/ custom/query.html
130	0.0163	24	12	http://www.med.harvard.edu
18	0.0139	45	46	http://www.gse.harvard.edu
15	0.0131	16	49	http://www.hms.harvard.edu
9	0.0114	21	27	http://www.ksg.harvard.edu
17	0.0111	13	6	http://www.hsph.harvard.edu
46	0.0100	18	21	http://www.gocrimson.com
13	0.0086	9	1	http://www.hsdm.med.harvard.edu
260	0.0086	26	1	http://search.harvard.edu:8765/ query.html
19	0.0084	23	21	http://www.radcliffe.edu

The URL where the search began, `www.harvard.edu`, dominates. Like most universities, Harvard is organized into various colleges Harvard Medical School, the Harvard Business School, and the Radcliffe Institute. You can see that the home pages of these schools have high PageRank. With a different sample, such as the one generated by Google itself, the ranks would be different.

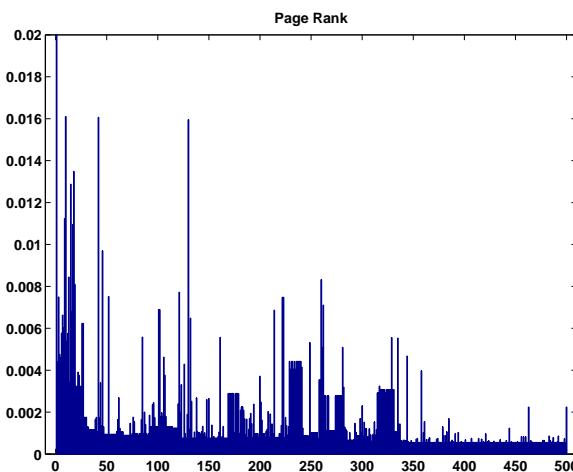


Figure 7.4. *PageRank of the harvard500 graph.*

Further Reading

Further reading on matrix computation includes books by Demmel [?], Golub and Van Loan [?], Stewart [?, ?], and Trefethen and Bau [?]. The definitive references on Fortran matrix computation software are the LAPACK Users' Guide and Web site [?]. The MATLAB sparse matrix data structure and operations are described in [?]. Information available on Web sites about PageRank includes a brief explanation at Google [?], a technical report by Page, Brin, and colleagues [?], and a comprehensive survey by Langville and Meyer [?].

Recap

```

%% Page Rank Chapter Recap
% This is an executable program that illustrates the statements
% introduced in the Page Rank Chapter of "Experiments in MATLAB".
% You can access it with
%
%     pagerank_recap
%     edit pagerank_recap
%     publish pagerank_recap
%
% Related EXM programs
%
%     surfer
%     pagerank

%% Sparse matrices

```

```
n = 6
i = [2 6 3 4 4 5 6 1 1]
j = [1 1 2 2 3 3 3 4 6]
G = sparse(i,j,1,n,n)
spy(G)

%% Page Rank
p = 0.85;
delta = (1-p)/n;
c = sum(G,1);
k = find(c~=0);
D = sparse(k,k,1./c(k),n,n);
e = ones(n,1);j
I = speye(n,n);
x = (I - p*G*D)\e;
x = x/sum(x)

%% Conventional power method
z = ((1-p)*(c~=0) + (c==0))/n;
A = p*G*D + e*z;
x = e/n;
oldx = zeros(n,1);
while norm(x - oldx) > .01
    oldx = x;
    x = A*x;
end
x = x/sum(x)

%% Sparse power method
G = p*G*D;
x = e/n;
oldx = zeros(n,1);
while norm(x - oldx) > .01
    oldx = x;
    x = G*x + e*(z*x);
end
x = x/sum(x)

%% Inverse iteration
x = (I - A)\e;
x = x/sum(x)

%% Bar graph
bar(x)
title('Page Rank')
```

Exercises

7.1 Use `surfer` and `pagerank` to compute PageRanks for some subset of the Web that you choose. Do you see any interesting structure in the results?

7.2 Suppose that `U` and `G` are the URL cell array and the connectivity matrix produced by `surfer` and that `k` is an integer. Explain what

`U{k}, U(k), G(k,:), G(:,k), U(G(k,:)), U(G(:,k))`

are.

7.3 The connectivity matrix for the `harvard500` data set has four small, almost entirely nonzero, submatrices that produce dense patches near the diagonal of the `spy` plot. You can use the zoom button to find their indices. The first submatrix has indices around 170 and the other three have indices in the 200s and 300s. Mathematically, a graph with every node connected to every other node is known as a *clique*. Identify the organizations within the Harvard community that are responsible for these near cliques.

7.4 A Web connectivity matrix G has $g_{ij} = 1$ if it is possible to get to page i from page j with one click. If you multiply the matrix by itself, the entries of the matrix G^2 count the number of different paths of length two to page i from page j . The matrix power G^p shows the number of paths of length p .

- (a) For the `harvard500` data set, find the power p where the number of nonzeros stops increasing. In other words, for any q greater than p , `nnz(G^q)` is equal to `nnz(G^p)`.
- (b) What fraction of the entries in G^p are nonzero?
- (c) Use `subplot` and `spy` to show the nonzeros in the successive powers.
- (d) Is there a set of interconnected pages that do not link to the other pages?

7.5 The function `surfer` uses a subfunction, `hashfun`, to speed up the search for a possibly new URL in the list of URLs that have already been processed. Find two different URLs on The MathWorks home page <http://www.mathworks.com> that have the same `hashfun` value.

7.6 Figure 7.5 is the graph of another six-node subset of the Web. In this example, there are two disjoint subgraphs.

- (a) What is the connectivity matrix G ?
- (b) What are the PageRanks if the hyperlink transition probability p is the default value 0.85?
- (c) Describe what happens with this example to both the definition of PageRank and the computation done by `pagerank` in the limit $p \rightarrow 1$.

7.7 The function `pagerank(U,G)` computes PageRanks by solving a sparse linear

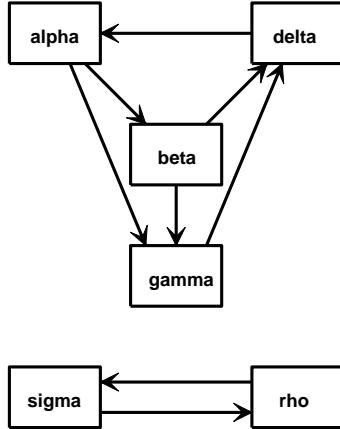


Figure 7.5. Another tiny Web.

system. It then plots a bar graph and prints the dominant URLs.

(a) Create `pagerank1(G)` by modifying `pagerank` so that it just computes the PageRanks, but does not do any plotting or printing.

(b) Create `pagerank2(G)` by modifying `pagerank1` to use inverse iteration instead of solving the sparse linear system. The key statements are

```

x = (I - A)\e
x = x/sum(x)

```

What should be done in the unlikely event that the backslash operation involves a division by zero?

(c) Create `pagerank3(G)` by modifying `pagerank1` to use the power method instead of solving the sparse linear system. The key statements are

```

G = p*G*D
z = ((1-p)*(c~=0) + (c==0))/n;
while termination_test
    x = G*x + e*(z*x)
end

```

What is an appropriate test for terminating the power iteration?

(d) Use your functions to compute the PageRanks of the six-node example discussed in the text. Make sure you get the correct result from each of your three functions.

7.8 Here is yet another function for computing PageRank. This version uses the power method, but does not do any matrix operations. Only the link structure of the connectivity matrix is involved.

```
function [x,cnt] = pagerankpow(G)
```

```
% PAGERANKPOW PageRank by power method.
% x = pagerankpow(G) is the PageRank of the graph G.
% [x,cnt] = pagerankpow(G)
%      counts the number of iterations.

% Link structure

[n,n] = size(G);
for j = 1:n
    L{j} = find(G(:,j));
    c(j) = length(L{j});
end

% Power method

p = .85;
delta = (1-p)/n;
x = ones(n,1)/n;
z = zeros(n,1);
cnt = 0;
while max(abs(x-z)) > .0001
    z = x;
    x = zeros(n,1);
    for j = 1:n
        if c(j) == 0
            x = x + z(j)/n;
        else
            x(L{j}) = x(L{j}) + z(j)/c(j);
        end
    end
    x = p*x + delta;
    cnt = cnt+1;
end
```

(a) How do the storage requirements and execution time of this function compare with the three `pagerank` functions from the previous exercise?

(b) Use this function as a template to write a function that computes PageRank in some other programming language.

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

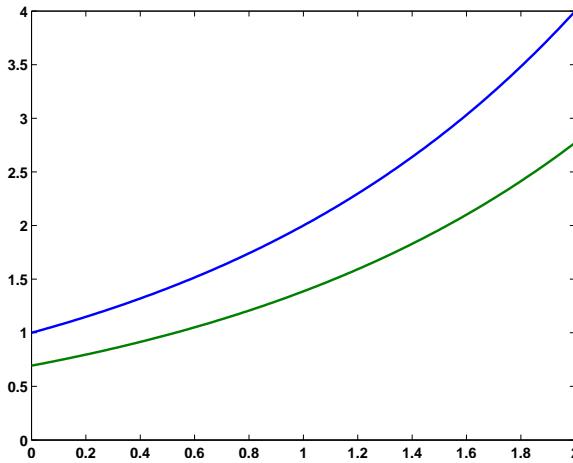


Figure 8.1. The blue curve is the graph of $y = 2^t$. The green curve is the graph of the rate of growth, $\dot{y} = dy/dt$.

We know what 2^t means if t is an integer, 2^t is the t -th power of 2.

$$2^{-1} = 1/2, \quad 2^0 = 1, \quad 2^1 = 2, \quad 2^2 = 4, \dots$$

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.

$$\begin{aligned} 2^{1/2} &= \sqrt{2} = 1.4142\dots, \\ 2^{5/3} &= \sqrt[3]{2^5} = 3.1748\dots, \\ 2^{355/113} &= \sqrt[113]{2^{355}} = 8.8250\dots \end{aligned}$$

In principle, 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, $\dot{y}(t)/y(t)$.

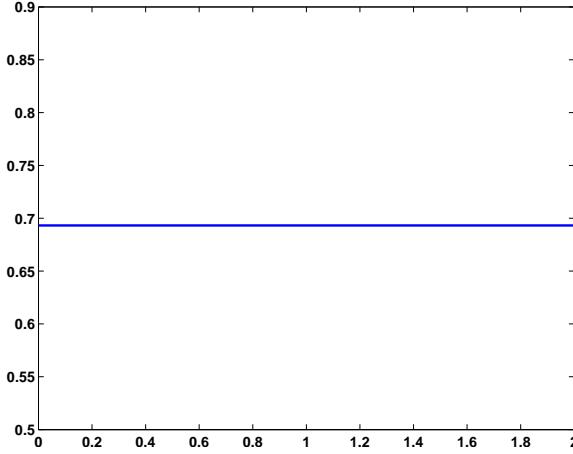


Figure 8.2. The ratio, \dot{y}/y .

```
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, $\dot{y}/y = 0.6931\dots$, 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 $\dot{y}/y = 1.0986\dots$. 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

$$e = \lim_{h \rightarrow 0} (1 + h)^{1/h}$$

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

0.5000000000000000 2.2500000000000000
0.2500000000000000 2.4414062500000000
0.1250000000000000 2.565784513950348
0.0625000000000000 2.637928497366600
0.0312500000000000 2.676990129378183
0.0156250000000000 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

$$\left(1 + \frac{1}{2251799813685248}\right)^{2251799813685248}$$

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.

$$e = 2.718281828\dots$$

Let's derive a more useful representation of the exponential function. Start by putting t back in the picture.

$$\begin{aligned} e^t &= \left(\lim_{h \rightarrow 0}(1+h)^{1/h}\right)^t \\ &= \lim_{h \rightarrow 0}(1+h)^{t/h} \end{aligned}$$

Here is the *Binomial Theorem*.

$$(a+b)^n = a^n + na^{n-1}b + \frac{n(n-1)}{2!}a^{n-2}b^2 + \frac{n(n-1)(n-2)}{3!}a^{n-3}b^3 + \dots$$

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 binomial theorem with $a = 1$, $b = h$ and $n = t/h$.

$$\begin{aligned} (1+h)^{t/h} &= 1 + (t/h)h + \frac{(t/h)(t/h-1)}{2!}h^2 + \frac{(t/h)(t/h-1)(t/h-2)}{3!}h^3 + \dots \\ &= 1 + t + \frac{t(t-h)}{2!} + \frac{t(t-h)(t-2h)}{3!} + \dots \end{aligned}$$

Now let h go to zero. We get the power series for the exponential function.

$$e^t = 1 + t + \frac{t^2}{2!} + \frac{t^3}{3!} + \dots + \frac{t^n}{n!} + \dots$$

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, π , 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

$$e = \text{expex}(1)$$

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.

$$\begin{aligned}y(t+2h) &= (1+rh)y(t+h) - p \\&= (1+rh)^2y(t) - ((1+rh)+1)p \\y(t+3h) &= (1+rh)^3y(t) - ((1+rh)^2+(1+rh)+1)p \\y(t+nh) &= (1+rh)^ny(0) - ((1+rh)^{n-1} + \dots + (1+rh)+1)p \\&= (1+rh)^ny(0) - ((1+rh)^n - 1)/(1+rh - 1)p\end{aligned}$$

Solve for p

$$p = (1+rh)^n / ((1+rh)^n - 1)rh y_0$$

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

$$\begin{aligned}y_0/n \\= 555.56\end{aligned}$$

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 θ 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 definition. 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.

$$\begin{aligned}\frac{d}{d\theta} e^{i\theta} &= ie^{i\theta} \\ \frac{d^2}{d\theta^2} e^{i\theta} &= i^2 e^{i\theta} = -e^{i\theta}\end{aligned}$$

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 θ .

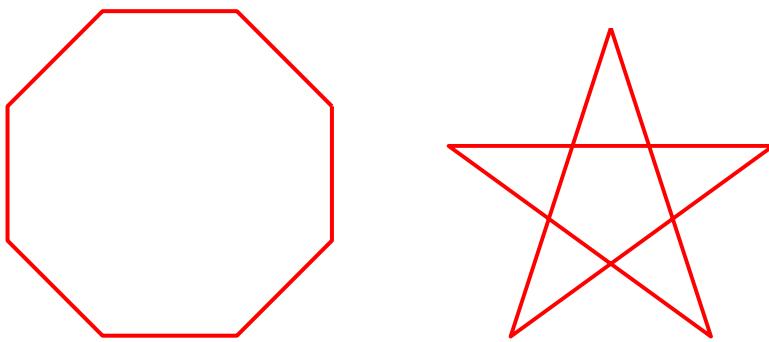


Figure 8.3. Two plots of $e^{i\theta}$.

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 θ , 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 π

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

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 octagon.

```
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 9

T Puzzle

A classic puzzle demonstrates complex arithmetic.



Figure 9.1. The wooden T puzzle. Photo courtesy of Shop New Zealand,
<http://www.shopnewzealand.co.nz>.

I first saw the wooden T puzzle shown in figure 9.1 at Puzzling World in

Wanaka, New Zealand. They told me that it was their most popular puzzle. I have since learned that it was a well-known toy in the 1800s and an advertising tool in the early 1900s. The underlying mathematics involves geometry, trigonometry, and arithmetic with complex numbers. The `t_puzzle` program in the `exm` toolbox demonstrates some useful programming techniques.

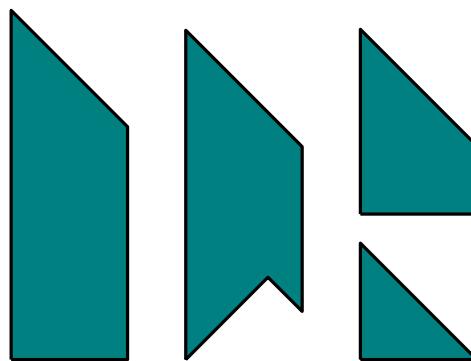


Figure 9.2. *The four pieces.*

Figure 9.2 shows the electronic version of the four pieces. They all have the same width, but different heights. One of them has an unshapely chunk cut out of it, resulting in an irregular pentagon.

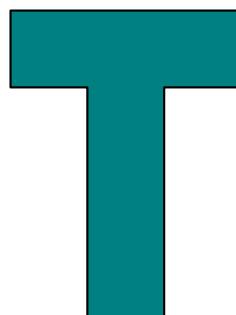


Figure 9.3. *The T.*

It turns out to be possible to arrange the four pieces to form the capital “T” shape shown in figure 9.3, as well as the arrow and the rhombus shapes shown in figure 4.4. What happened to those all of those 45° angles and what happened to that chunk?

If you do a Google search on “T-puzzle” you can quickly see how to solve the puzzle and form the T, but please try `t_puzzle` for a while before you go surfing for the solution. If you click near the center of one of the four pieces, you can move

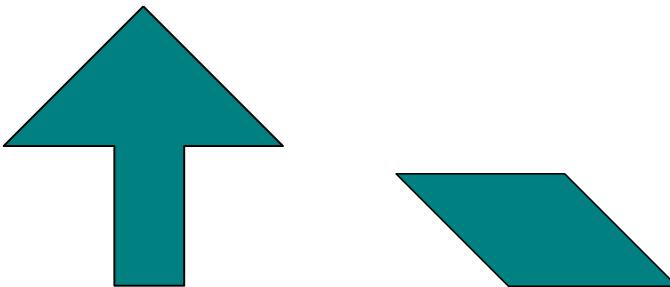


Figure 9.4. The arrow and the rhombus.

it horizontally and vertically. If you click near one of the vertices, you can rotate a piece about its center. If you click with the right mouse button, or, on a one-button mouse, hold down the control key while you click, you can flip a piece over horizontally, changing its right/left-handed orientation. If you click in the subplot on the lower left of the window, you can cycle through images of the three shapes.

The key to effective computation involving the T-puzzle pieces is the use of complex arithmetic. A complex number is formed from a pair of real numbers and the imaginary unit $i = \sqrt{-1}$. For example,

$$z = 3 + 4i$$

The *real* part is 3 and the *imaginary* part is 4. This is the *Cartesian* representation of a complex number.

Addition of complex numbers simply involves addition of the real parts and of the imaginary parts.

$$\begin{aligned} & (3 + 4i) + (5 - i) \\ &= (3 + 5) + (4i - i) \\ &= 8 + 3i \end{aligned}$$

Multiplication of complex numbers is more interesting because it makes use of the fact that $i^2 = -1$.

$$\begin{aligned} & (3 + 4i) \cdot (5 - i) \\ &= (3 \cdot 5 + (4i) \cdot (-i)) + (3 \cdot (-i) + (4i) \cdot 5) \\ &= (15 + 4) + (-3 + 20)i \\ &= 19 + 17i \end{aligned}$$

A fundamental fact involving complex numbers is *Euler's formula*.

$$e^{i\phi} = \cos \phi + i \sin \phi$$

If you are not familiar with e or Euler's formula, see our chapter on the "Exponential Function" and the Wikipedia entry on "Euler's Identity". Or, you can just accept the formula as convenient notation.

Setting $\phi = \pi$ leads to

$$\begin{aligned} e^{i\pi} &= \cos \pi + i \sin \pi \\ &= -1 \end{aligned}$$

Moving the -1 to the left hand side produces a beautiful equation involving five basic mathematical quantities, e , i , π , 1 , and 0 .

$$e^{i\pi} + 1 = 0$$

The *polar form* of a complex number is

$$z = x + iy = re^{i\phi}$$

where

$$\begin{aligned} r &= |z| = \sqrt{x^2 + y^2} \\ x &= r \cos \phi \\ y &= r \sin \phi \end{aligned}$$

Our T puzzle program uses the fact that multiplication by $e^{i\theta}$ rotates a complex number through an angle θ . To see this, let

$$\begin{aligned} w &= e^{i\theta} \\ z &= re^{i\phi} \end{aligned}$$

then

$$w \cdot z = e^{i\theta} \cdot re^{i\phi} = re^{i(\theta+\phi)}$$

In MATLAB the letter `i` can be used in any of three different roles. It can be an iteration counter,

```
for i = 1:10
```

or a subscript,

```
A(i,k)
```

or the imaginary unit.

```
z = 3 + 4i
```

The polar form of a complex number `z` is obtained with the MATLAB functions `abs(z)` and `angle(z)`. The quantity $e^{i\phi}$ is written `exp(i*phi)`. For example

```
z = 3 + 4i
r = abs(z)
phi = angle(z)
w = r*exp(i*phi)
```

produces

```

z =
3.0000 + 4.0000i
r =
5
phi =
0.9273
w =
3.0000 + 4.0000i

```

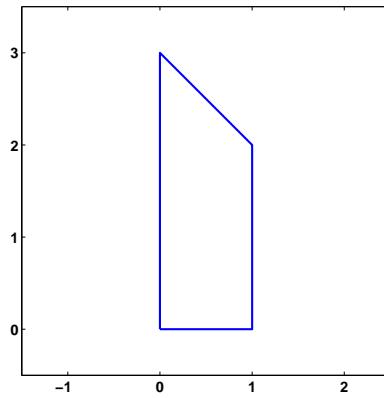


Figure 9.5. The complex coordinates of the vertices are $0 + 0i$, $1 + 0i$, $1 + 2i$, and $0 + 3i$.

The largest of the four pieces of the T puzzle can be represented in MATLAB by the statement

```
z = [0 1 1+2i 3i 0]
```

The vector z has five complex entries. The first two elements are 0 and 1; their imaginary parts are zero. The third element is $1 + 2i$; its real part is 1 and its imaginary part is 2. After that comes $3i$; its real part is zero and its imaginary part is 3. The last element of z is a repeat of the first so that the line drawn by the statement

```
line(real(z),imag(z))
```

returns to the origin. The result is figure 9.5.

With this representation, the translations and rotations required by the graphics in `t_puzzle` can be programmed easily and efficiently. Translation is very easy. The statement

```
z = z - (3-i)/2
```

repositions the piece in figure 9.5 at one of the corners. Do you see which corner?

Rotations are also easily done. The statements

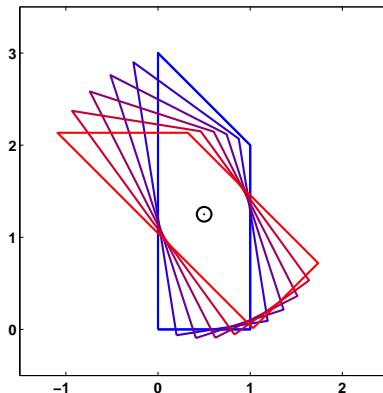


Figure 9.6. Rotating through multiples of nine degrees.

```
mu = mean(z(1:end-1))
theta = pi/20
omega = exp(i*theta)
z = omega*(z - mu) + mu
```

rotate the piece about its center through 9° in the counterclockwise direction. Figure 9.6 shows the result of repeating these statements several times. Let's look at each step more carefully. The statement

```
mu = mean(z(1:end-1))
```

drops the last element of z because it is a repeat of the first and then computes the complex average of the coordinates of the vertices. This gives the coordinates of what we can call the center of the polygon. The angle

$$\theta = \pi/20$$

is 9° in radians. The statements

```
omega = exp(i*theta)
z = omega*(z - mu) + mu
```

translate the piece so that its center is at the origin, do a complex scalar times vector multiplication to produce the rotation, and then translate the result back to its original center.

Recap

```
%% T Puzzle Chapter Recap
% This is an executable program that illustrates the statements
% introduced in the T Puzzle Chapter of "Experiments in MATLAB".
% You can access it with
```

```

%
%     puzzle_recap
%     edit puzzle_recap
%     publish puzzle_recap
%
% Related EXM programs
%
%     t_puzzle

%% The T Puzzle
close all
figure
t_puzzle

%% Polar Form of a Complex Number.
z = 3 + 4i
r = abs(z)
phi = angle(z)
z_again = r*exp(i*phi)

%% A Puzzle Piece.
figure
z = [0 1 1+2i 3i 0]
line(real(z),imag(z),'color','red')
axis([-2.5 1.5 -.5 4.0])

%% Translation.
z = z - (3-i)/2
line(real(z),imag(z),'color','blue')

%% Rotation.
mu = mean(z(1:end-1));
theta = pi/10;
omega = exp(i*theta);
z = omega*(z - mu) + mu
line(real(z),imag(z),'color','green')

```

Exercises

9.1 *Complex numbers.* Express each of these numbers in polar form.

i
 -4
 $3 - 4i$

$$8 + 15i$$

Express each of these numbers in Cartesian form.

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

$$e^{i\pi/3}$$

$$e^i$$

$$e^3$$

9.2 *Stop sign.* Try this for $n = 8$ and for other small integer values of n . Describe and explain the results.

```
n = 8
z = exp(2*pi*i*(0:n)'/n)
plot(z, '-o')
axis square
s = sum(z)
```

9.3 *Strip.* How many different ways are there to form the shape in figure 9.7 with the T-puzzle pieces?



Figure 9.7. A strip.

9.4 *Area.* If the width of each of the T-puzzle pieces is one unit, what are their areas?

9.5 *Symmetry.* Which one of our three shapes – T, arrow and rhombus – does not have an axis of symmetry?

9.6 *Jumping rotation.* Click near a vertex of one of the T-puzzle pieces and rotate the piece slowly. You should see that the rotation is not smooth, but proceeds in discrete jumps. Why? How large are the jumps? How does `t_puzzle` accomplish this?

9.7 *Snappy translation.* Drag one of the T-puzzle pieces until it is close to, but not exactly touching, another. When you release the mouse button you sometimes see the piece snap into place. Under what circumstances does this happen? How does `t_puzzle` accomplish it?

9.8 *Rotation.* Reproduce figure 9.6.

9.9 *Different puzzles.* Do a Google search on “T-puzzle”. Include the quotes and hyphen in the search string so that you get an exact match. Some of the Web pages have pieces with different sizes than the ones we have described here.

- (a) How many different versions of the T-puzzle are there on the Web?
- (b) Can you make all three of our shapes – the T, arrow, and rhombus – with the pieces shown on these Web sites.
- (c) Modify our `t_puzzle` to use the set of pieces from one of the Web sites.

Chapter 10

Magic Squares

With origins in centuries old recreational mathematics, magic squares demonstrate MATLAB array operations.

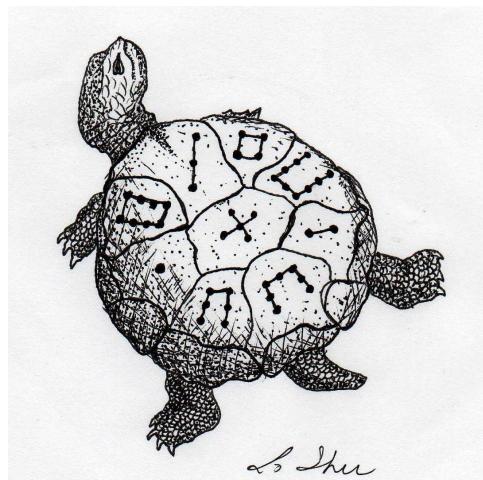


Figure 10.1. *Lo Shu. (Thanks to Byerly Wiser Cline.)*

Magic squares predate recorded history. An ancient Chinese legend tells of a turtle emerging from the Lo river during a flood. The turtle's shell showed a very unusual pattern – a three-by-three grid containing various numbers of spots. Of

course, we do not have any eye-witness accounts, so we can only imagine that the turtle looked like figure 10.1. Each of the three rows, the three columns, and the two diagonals contain a total of 15 spots. References to Lo Shu and the Lo Shu numerical pattern occur throughout Chinese history. Today, it is the mathematical basis for *Feng Shui*, the philosophy of balance and harmony in our surroundings and lives.

An n -by- n magic square is an array containing the integers from 1 to n^2 , arranged so that each of the rows, each of the columns, and the two principal diagonals have the same sum. For each $n > 2$, there are many different magic squares of order n , but the MATLAB function `magic(n)` generates a particular one.

MATLAB can generate Lo Shu with

```
A = magic(3)
```

which produces

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

The command

```
sum(A)
```

sums the elements in each column to produce

```
15     15     15
```

The command

```
sum(A')
```

transposes the matrix, sums the columns of the transpose, and then transposes the results to produce the row sums

```
15
15
15
15
```

The command

```
sum(diag(A))
```

sums the main diagonal of A, which runs from upper left to lower right, to produce

```
15
```

The opposite diagonal, which runs from upper right to lower left, is less important in linear algebra, so finding its sum is a little trickier. One way to do it makes use of the function that “flips” a matrix “upside-down.”

```
sum(diag(flipud(A)))
```

produces

```
15
```

This verifies that A has equal row, column, and diagonal sums.

Why is the magic sum equal to 15? The command

```
sum(1:9)
```

tells us that the sum of the integers from 1 to 9 is 45. If these integers are allocated to 3 columns with equal sums, that sum must be

```
sum(1:9)/3
```

which is 15.

There are eight possible ways to place a transparency on an overhead projector. Similarly, there are eight magic squares of order three that are rotations and reflections of A. The statements

```
for k = 0:3
    rot90(A,k)
    rot90(A',k)
end
```

display all eight of them.

8	1	6	8	3	4
3	5	7	1	5	9
4	9	2	6	7	2
6	7	2	4	9	2
1	5	9	3	5	7
8	3	4	8	1	6
2	9	4	2	7	6
7	5	3	9	5	1
6	1	8	4	3	8
4	3	8	6	1	8
9	5	1	7	5	3
2	7	6	2	9	4

These are all the magic squares of order three. The 5 is always in the center, the other odd numbers are always in the centers of the edges, and the even numbers are always in the corners.

Melancholia I is a famous Renaissance engraving by the German artist and amateur mathematician Albrecht Dürer. It shows many mathematical objects, including a sphere, a truncated rhombohedron, and, in the upper right hand corner, a magic square of order 4. You can see the engraving in our figure 10.2. Better yet, issue these MATLAB commands



Figure 10.2. Albrecht Dürer's *Melancolia*, 1514.

```
load durer
whos
```

You will see

X	648x509	2638656	double array
caption	2x28	112	char array
map	128x3	3072	double array

The elements of the array X are indices into the gray-scale color map named map.
The image is displayed with

```
image(X)
colormap(map)
axis image
```

Click the magnifying glass with a “+” in the toolbar and use the mouse to zoom in on the magic square in the upper right-hand corner. The scanning resolution becomes evident as you zoom in. The commands

```
load detail
```

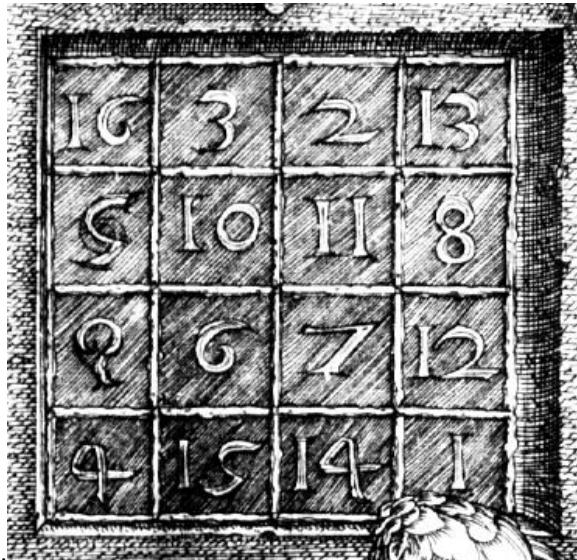


Figure 10.3. Detail from *Melancolia*.

```
image(X)
colormap(map)
axis image
```

display the higher resolution scan of the area around the magic square that we have in figure 10.3.

The command

```
A = magic(4)
```

produces a 4-by-4 magic square.

```
A =
16      2      3     13
 5     11     10      8
 9      7      6     12
 4     14     15      1
```

The commands

```
sum(A), sum(A'), sum(diag(A)), sum(diag(flipud(A)))
```

yield enough 34's to verify that A is indeed a magic square.

The 4-by-4 magic square generated by MATLAB is not the same as Dürer's magic square. We need to interchange the second and third columns.

```
A = A(:, [1 3 2 4])
```

changes A to

```
A =
 16    3    2   13
  5   10   11    8
  9    6    7   12
  4   15   14    1
```

Interchanging columns does not change the column sums or the row sums. It usually changes the diagonal sums, but in this case both diagonal sums are still 34. So now our magic square matches the one in Dürer's etching. Dürer probably chose this particular 4-by-4 square because the date he did the work, 1514, occurs in the middle of the bottom row.

The program `durerperm` interchanges rows and columns in the image produced from `detail` by interchanging groups of rows and columns in the array `X`. This is not especially important or useful, but it provides an interesting exercise.

We have seen two different 4-by-4 magic squares. It turns out that there are 880 different magic squares of order 4 and 275305224 different magic squares of order 5. Determining the number of different magic squares of order 6 or larger is an unsolved mathematical problem.

For a magic square of order n , the magic sum is

$$\mu(n) = \frac{1}{n} \sum_{k=1}^{n^2} k$$

which turns out to be

$$\mu(n) = \frac{n^3 + n}{2}.$$

Here is the beginning of a table of values of the magic sum.

n	$\mu(n)$
3	15
4	34
5	65
6	111
7	175
8	260

You can compute $\mu(n)$ in MATLAB with either

```
sum(diag(magic(n)))
```

or

```
(n^3 + n)/2
```

The algorithms for generating matrix square fall into three distinct cases:

odd, n is odd.

singly-even, n is divisible by 2, but not by 4.

doubly-even, n is divisible by 4.

The best known algorithm for generating magic squares of odd order is de La Loubere's method. Simon de La Loubere was the French ambassador to Siam in the late 17th century. I sometimes refer to his method as the "nor'easter algorithm", after the winter storms that move northeasterly up the coast of New England. You can see why if you follow the integers sequentially through `magic(9)`.

47	58	69	80	1	12	23	34	45
57	68	79	9	11	22	33	44	46
67	78	8	10	21	32	43	54	56
77	7	18	20	31	42	53	55	66
6	17	19	30	41	52	63	65	76
16	27	29	40	51	62	64	75	5
26	28	39	50	61	72	74	4	15
36	38	49	60	71	73	3	14	25
37	48	59	70	81	2	13	24	35

The integers from 1 to n^2 are inserted along diagonals, starting in the middle of first row and heading in a northeasterly direction. When you go off an edge of the array, which you do at the very first step, continue from the opposite edge. When you bump into a cell that is already occupied, drop down one row and continue.

We used this algorithm in MATLAB for many years. Here is the code.

```
A = zeros(n,n);
i = 1;
j = (n+1)/2;
for k = 1:n^2
    is = i;
    js = j;
    A(i,j) = k;
    i = n - rem(n+1-i,n);
    j = rem(j,n) + 1;
    if A(i,j) ~= 0
        i = rem(is,n) + 1;
        j = js;
    end
end
```

A big difficulty with this algorithm and resulting program is that it inserts the elements one at a time – it cannot be *vectorized*.

A few years ago we discovered an algorithm for generating the same magic squares of odd order as de La Loubere's method, but with just four MATLAB matrix operations.

```
[I,J] = ndgrid(1:n);
```

```
A = mod(I+J+(n-3)/2,n);
B = mod(I+2*J-2,n);
M = n*A + B + 1;
```

Let's see how this works with $n = 5$. The statement

```
[I,J] = ndgrid(1:n)
```

produces a pair of matrices whose elements are just the row and column indices, i and j .

```
I =
1   1   1   1   1
2   2   2   2   2
3   3   3   3   3
4   4   4   4   4
5   5   5   5   5
```

```
J =
1   2   3   4   5
1   2   3   4   5
1   2   3   4   5
1   2   3   4   5
1   2   3   4   5
```

Using these indices, we generate two more matrices. The statements

```
A = mod(I+J+1,n)
B = mod(I+2*J-2,n)
```

produce

```
A =
3   4   0   1   2
4   0   1   2   3
0   1   2   3   4
1   2   3   4   0
2   3   4   0   1
```

```
B =
1   3   0   2   4
2   4   1   3   0
3   0   2   4   1
4   1   3   0   2
0   2   4   1   3
```

Both A and B are fledgling magic squares. They have equal row, column and diagonal sums. But their elements are not the integers from 1 to n^2 . Each has duplicated elements between 0 and $n - 1$. The final statement

```
M = n*A+B+1
```

produces a matrix whose elements are integers between 1 and n^2 and which has equal row, column and diagonal sums. What is not obvious, but is true, is that there are no duplicates. So M must contain *all* of the integers between 1 and n^2 and consequently is a magic square.

```
M =
17    24      1      8     15
23      5      7     14     16
  4      6     13     20     22
 10     12     19     21      3
 11     18     25      2      9
```

The doubly-even algorithm is also short and sweet, and tricky.

```
M = reshape(1:n^2,n,n)';
[I,J] = ndgrid(1:n);
K = fix(mod(I,4)/2) == fix(mod(J,4)/2);
M(K) = n^2+1 - M(K);
```

Let's look at our friend `magic(4)`. The matrix M is initially just the integers from 1 to 16 stored sequentially in 4 rows.

```
M =
 1      2      3      4
 5      6      7      8
 9     10     11     12
13     14     15     16
```

The logical array K is true for half of the indices and false for the other half in a pattern like this.

```
K =
 1      0      0      1
 0      1      1      0
 0      1      1      0
 1      0      0      1
```

The elements where K is false, that is 0, are left alone.

```
.
 2      3      .
 5      .      .      8
 9      .      .      12
 .
14     15      .
```

The elements where K is true, that is 1, are reversed.

```
16      .      .      13
 .
11     10      .
 .
 7      6      .
 4      .      .      1
```

The final result merges these two matrices to produce the magic square.

The algorithm for singly even order is the most complicated and so we will give just a glimpse of how it works. If n is singly even, then $n/2$ is odd and `magic(n)` can be constructed from four copies of `magic(n/2)`. For example, `magic(10)` is obtained from `A = magic(5)` by forming a block matrix.

$$\begin{bmatrix} A & A+50 \\ A+75 & A+25 \end{bmatrix}$$

The column sums are all equal because `sum(A) + sum(A+75)` equals `sum(A+50) + sum(A+25)`. But the row sums are not quite right. The algorithm must finish by doing a few swaps of pieces of rows to clean up the row sums. For the details, issue the command.

```
type magic
```

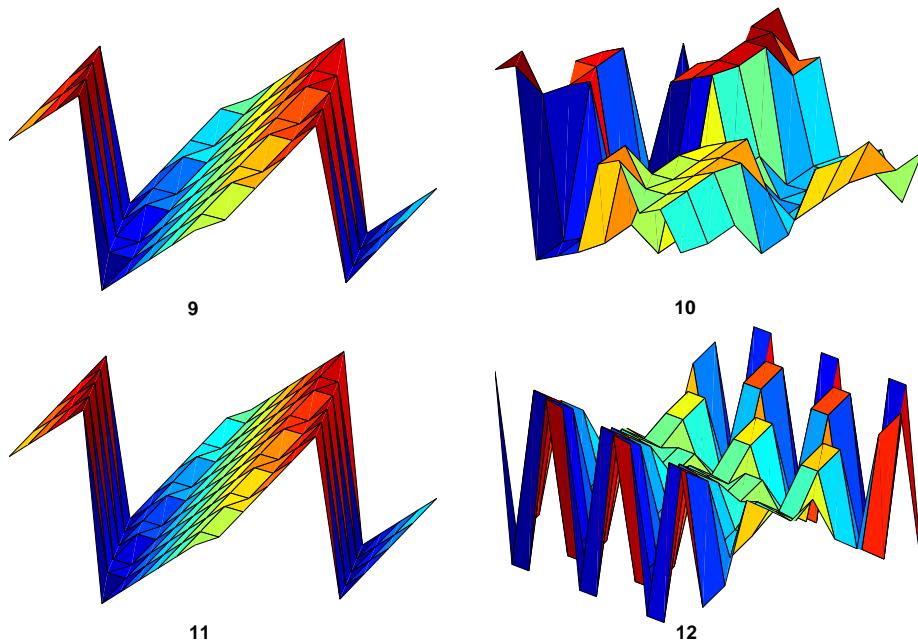


Figure 10.4. Surf plots of magic squares of order 9, 10, 11, 12.

Let's conclude this chapter with some graphics. Figure 10.4 shows

```
surf(magic(9))    surf(magic(10))
surf(magic(11))    surf(magic(12))
```

You can see the three different cases – on the left, the upper right, and the lower right. If you increase each of the orders by 4, you get more cells, but the global shapes remain the same. The odd n case on the left reminds me of Origami.

Further Reading

The reasons why MATLAB has magic squares can be traced back to my junior high school days when I discovered a classic book by W. W. Rouse Ball, *Mathematical Recreations and Essays*. Ball lived from 1850 until 1925. He was a fellow of Trinity College, Cambridge. The first edition of his book on mathematical recreations was published in 1892 and the tenth edition in 1937. Later editions were revised and updated by another famous mathematician, H. S. M Coxeter. The thirteenth edition, published by Dover in 1987, is available from many booksellers, including Powells and Amazon.

```
http://www.powells.com/cgi-bin/biblio?inkey=17-0486253570-0
http://www.amazon.com/Mathematical-Recreations-Essays-Dover-Books/dp/0486253570
```

There are dozens of interesting Web pages about magic squares. Here are a few authors and links to their pages.

Mutsumi Suzuki	http://mathforum.org/te/exchange/hosted/suzuki/MagicSquare.html
Eric Weisstein	http://mathworld.wolfram.com/MagicSquare.html
Kwon Young Shin	http://user.chollian.net/~brainstm/MagicSquare.htm
Walter Trump	http://www.trump.de/magic-squares

Recap

```
%% Magic Squares Chapter Recap
% This is an executable program that illustrates the statements
% introduced in the Magic Squares Chapter of "Experiments in MATLAB".
% You can access it with
%
%     magic_recap
%     edit magic_recap
%     publish magic_recap
%
% Related EXM programs
%
%     magic
%     ismagical

%% A Few Elementary Array Operations.
format short
A = magic(3)
sum(A)
```

```
sum(A')',
sum(diag(A))
sum(diag(flipud(A)))
sum(1:9)/3
for k = 0:3
    rot90(A,k)
    rot90(A',k)
end

%% Durer's Melancolia
clear all
close all
figure
load durer
whos
image(X)
colormap(map)
axis image

%% Durer's Magic Square
figure
load detail
image(X)
colormap(map)
axis image
A = magic(4)
A = A(:,[1 3 2 4])

%% Magic Sum
n = (3:10)';
(n.^3 + n)/2

%% Odd Order
n = 5
[I,J] = ndgrid(1:n);
A = mod(I+J+(n-3)/2,n);
B = mod(I+2*J-2,n);
M = n*A + B + 1

%% Doubly Even Order
n = 4
M = reshape(1:n^2,n,n)';
[I,J] = ndgrid(1:n);
K = fix(mod(I,4)/2) == fix(mod(J,4)/2);
M(K) = n^2+1 - M(K)
```

```

%% Rank
figure
for n = 3:20
    r(n) = rank(magic(n));
end
bar(r)
axis([2 21 0 20])

%% Ismagical
help ismagical
for n = 3:10
    ismagical(magic(n))
end

%% Surf Plots
figure
for n = 9:12
    subplot(2,2,n-8)
    surf(rot90(magic(n)))
    axis tight off
    text(0,0,20,num2str(n))
end
set(gcf,'color','white')

```

Exercises

10.1 *ismagic*. Write a MATLAB function `ismagic(A)` that checks if A is a magic square.

10.2 *Magic sum*. Show that

$$\frac{1}{n} \sum_{k=1}^{n^2} k = \frac{n^3 + n}{2}.$$

10.3 *durerperm*. Investigate the `durerperm` program. Click on two different elements to interchange rows and columns. Do the interchanges preserve row and column sums? Do the interchanges preserve the diagonal sums?

10.4 *Colormaps*. Try this.

```

clear
load detail
whos

```

You will see three matrices in your workspace. You can look at all of `map` and `caption`.

```
map
caption
```

The matrix `X` is 359-by-371. That's 133189 elements. Look at just a piece of it.

```
X(101:130,101:118)
```

The elements of `X` are integers in the range

```
min(min(X))
max(max(X))
```

The commands

```
image(X)
axis image
```

produce a pretty colorful display. That's because the elements of `X` are being used as indices into the default colormap, `jet(64)`. You can use the intended colormap instead.

```
colormap(map)
```

The array `map` is a 64-by-3 array. Each row, `map(k,:)`, specifies intensities of red, green and blue. The color used at point `(i,j)` is `map(X(i,j),:)`. In this case, the colormap that comes with `detail` has all three columns equal to each other and so is the same as

```
colormap(gray(64))
```

Now experiment with other colormaps

```
colormap(hot)
colormap(cool)
colormap(copper)
colormap(pink)
colormap(bone)
colormap(flag)
colormap(hsv)
```

You can even cycle through 101 colormaps.

```
for p = 0:.001:1
    colormap(p*hot+(1-p)*pink)
    drawnow
end
```

You can plot the three color components of a colormap like `hot` with

```
rgbplot(hot)
```

This is what TV movie channels do when they *colorize* old black and white films.

10.5 *Knight's tour*. Do you know how a knight is allowed to move on a chess board? The `exm` function `knightstour` generates this matrix, K.

```
K =
 50   11   24   63   14   37   26   35
 23   62   51   12   25   34   15   38
 10   49   64   21   40   13   36   27
 61   22    9   52   33   28   39   16
 48    7   60    1   20   41   54   29
 59    4   45    8   53   32   17   42
  6   47    2   57   44   19   30   55
   3   58    5   46   31   56   43   18
```

If you follow the elements in numerical order, you will be taken on a *knight's tour* of K. Even the step from 64 back to 1 is a knight's move.

Is K a magic square? Why or why not?

Try this.

```
image(K)
colormap(pink)
axis square
```

Select the *data cursor* icon on the figure tour bar. Now use the mouse to take the knight's tour from dark to light on the image.

10.6 *ismagical*. The `exm` function `ismagical` checks for four different magical properties of square arrays.

Semimagic: all of the columns and all of rows have the same sum.

Magic: all of the columns, all of rows and both principal diagonals have the same sum.

Panmagic: all of the columns, all of rows and all of the diagonals, including the broken diagonals in both directions, have the same sum.

Associative: all pairs of elements on opposite sides of the center have the same sum. For example, this matrix that has all four properties.

```
M =
 10   18    1   14   22
 11   24    7   20    3
 17    5   13   21    9
 23    6   19    2   15
   4   12   25    8   16
```

Here is one of the broken diagonals. Its sum is $\mu(5) = 65$.

$$\begin{array}{ccccc}
 & . & . & . & 14 \\
 & . & . & . & 3 \\
 17 & . & . & . & . \\
 & 6 & . & . & . \\
 & . & 25 & . & .
 \end{array}$$

All of the broken diagonals in both directions have the same sum, so M is panmagic. One pair of elements on opposite sides of the center is 24 and 2. Their sum is twice the center value. All pairs of elements on opposite sides of the center have this sum, so M is associative.

- (a) Use `ismagical` to verify that M has all four properties.
- (b) Use `ismagical` to investigate the magical properties of the matrices generated by the MATLAB `magic` function.
- (c) Use `ismagical` to investigate the magical properties of the matrices generated by this algorithm for various odd n and various values of a_0 and b_0 ,

```

a0 = ...
b0 = ...
[I,J] = ndgrid(1:n);
A = mod(I+J-a0,n);
B = mod(I+2*J-b0,n);
M = n*A + B + 1;

```

- (d) Use `ismagical` to investigate the magical properties of the matrices generated by this algorithm for various odd n and various values of a_0 and b_0 ,

```

a0 = ...
b0 = ...
[I,J] = ndgrid(1:n);
A = mod(I+2*J-a0,n);
B = mod(I+3*J-b0,n);
M = n*A + B + 1;

```

10.7 Inverse. If you have studied matrix theory, you have heard of matrix inverses. What is the matrix inverse of a magic square of order n ? It turns out to depend upon whether n is odd or even. For odd n , the matrices `magic(n)` are nonsingular. The matrices

```
X = inv(magic(n))
```

do not have positive, integer entries, but they do have equal row and column sums.

But, for even n , the determinant, `det(magic(n))`, is 0, and the inverse does not exist. If `A = magic(4)` trying to compute `inv(A)` produces an error message.

10.8 Rank. If you have studied matrix theory, you know that the rank of a matrix is the number of linearly independent rows and columns. An n -by- n matrix is singular if its rank, r , is not equal to its order. This code computes the rank of the magic squares up to order 20, generated with

```

for n = 3:20
    r(n) = rank(magic(n));
end

```

The results are

n =	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
r =	3	3	5	5	7	3	9	7	11	3	13	9	15	3	17	11	19	3

Do you see the pattern? Maybe the bar graph in figure 10.5 will help. You can see

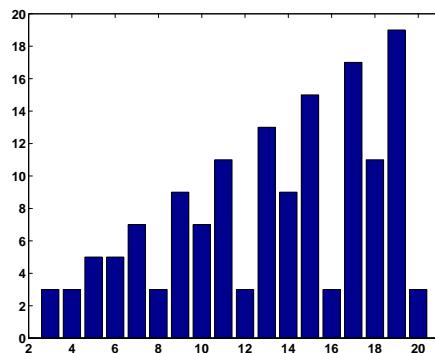


Figure 10.5. Rank of magic squares.

that the three different algorithms used to generate magic squares produce matrices with different rank.

n	rank
odd	n
even, not divisible by 4	n/2+2
divisible by 4	3

Chapter 11

Tic Tac Toe Magic

Three simple games are related in a surprising way. And, the programming of the game play is instructive.

The first of the three games is Pick15. Harold Stark, who was then at the University of Michigan, told me about the game in the late 1960s. I suspect that this is the first time you have heard of it.

The game involves two players. You start by listing the single digit numbers from 1 to 9. You then take turns selecting numbers from the list, attempting to acquire three numbers that add up to 15. Each number can be chosen only once. You may eventually acquire more than three numbers, but you must use exactly three of them to total 15. If neither player can achieve the desired total, the game is a draw.

For example, suppose that Green and Blue are playing. They start with the list.

List : 1 2 3 4 5 6 7 8 9

Green :

Blue :

Suppose Green has the first move and chooses 8. Then Blue chooses 4 and Green chooses 2. Now Blue should respond by choosing 5 to prevent Green from getting $2 + 5 + 8 = 15$. Here is the situation after the first two rounds.

List : 1 2 3 4 5 6 7 8 9

Green : 2 8

Blue : 4 5

Now Green chooses 6 to block $4 + 5 + 6 = 15$ from Blue. This is actually an advantageous move for Green because it gives her two different ways to win, $1+6+8$ and $2+6+7$. Blue cannot block both. If Blue chooses 7, then Green chooses 1 to win. If Blue chooses 1, then Green chooses 7 to win. The final position might be

List :	1	2	3	4	5	6	7	8	9
Green :						6	7	8	
Blue :				4	5				

Note that Green also has $7+8=15$, but this does not count because there are only two numbers in the sum.

Figure 11.1 shows the starting position for the Pick15 option in our MATLAB `tictactoe` program. When you play against the computer, your moves are shown in green and the responses from the program are shown in blue.



Figure 11.1. Starting position for Pick15.

Figure 11.2 shows the position for our example game after two moves from each player. Green now has to choose 6 to block Blue.



Figure 11.2. Position after two moves from each player.

Figure 11.3 shows the final position for our example game. Green has won with $2+6+7 = 15$.



Figure 11.3. Final position for Pick15.

Please take time out from reading this chapter to try this game a few times yourself, playing against a friend on paper or against our MATLAB program. I think you will find that Pick15 is more challenging than it sounds at first.

The second of our three games is familiar worldwide. It is called “TicTacToe” in the United States, “Noughts and Crosses” in Great Britain, and has many other names in many other countries and languages. Our program `tictactoe` uses green and blue instead of X’s and O’s. The objective, of course, is to get your color on all three squares in a row, column, or diagonal. Figure 11.6 shows typical output.

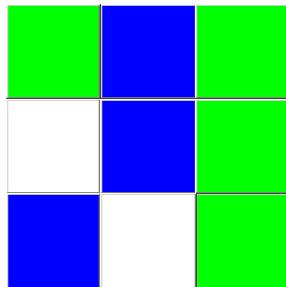


Figure 11.4. Typical output from a game of Tic Tac Toe. Green has won with the third column.

Our program `tictactoe` uses a naive three-step strategy.

- If possible, make a winning move.
- If necessary, block a possible winning move by the opponent.
- Otherwise, pick a random empty square.

This strategy will lead to a win only when the opponent makes a error. And, even though it is possible for the player with the second move to always force a draw, this strategy will not accomplish that.

Our third game, Magic15, introduces the Lo-Shu 3-by-3 magic square. Now we see that Pick15 is actually Tic Tac Toe played on a magic square. The rows, columns and main diagonals of the magic square provide all possible ways of having three distinct numbers that sum to 15. Winning moves in Pick15 correspond to winning moves in Tic Tac Toe. All three games are actually the same game with different displays.

8	1	6
3	5	7
4	9	2

Figure 11.5. Initial configuration for a game of Magic3.

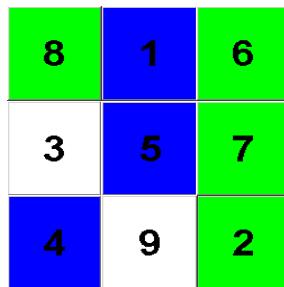


Figure 11.6. Green has won with $6+7+2 = 15$ in the third column.

Game Play

The `tictactoe` program operates three different views of the same game. You are the green player and the computer is the blue player. The state of the game is carried in a 3-by-3 matrix `X` whose entries are +1 for cells occupied by green, -1 for cells occupied by blue, and 0 for as yet unoccupied cells. Obviously, the game begins with `X = zeros(3,3)`

Here is the portion of main program that is executed when you, as the green player, click on one of the buttons or cells. It retrieves `i` and `j`, the matrix indices corresponding to the active button and then checks if the current game already has a winner. If not, it makes your move by setting `X(i,j) = 1` and recursively calls `tictactoe` to let blue take a turn.

```

switch
case 'green'
    [i,j] = find(gcbo == B);
    if winner(X)
        return
    end
    X(i,j) = 1;
    tictactoe('blue')

```

Here is the corresponding blue portion of the main program. If the current game does not already have a winner, it calls the `strategy` function to get the indices for a move and makes that move.

```

case 'blue'
if winner(X)
    return
end
[i,j] = strategy(X,-1);
X(i,j) = -1;
...
end

```

Briefly, then, the green moves are determined by user button clicks and the blue moves are determined by the `strategy` function.

Here is the function that checks to see if there is a winner. It looks for any column, row, or diagonal whose elements sum to $3*p$ where $p = 1$ for green and $p = -1$ for blue. The only way a sum can be $3*p$ is if all three elements are equal to p .

```
function p = winner(X)
% p = winner(X) returns
%   p = 0, no winner yet,
%   p = -1, blue has won,
%   p = 1, green has won,
%   p = 2, game is a draw.

for p = [-1 1]
    s = 3*p;
    win = any(sum(X) == s) || any(sum(X') == s) || ...
          sum(diag(X)) == s || sum(diag(fliplr(X))) == s;
    if win
        return
    end
end
p = 2*all(X(:) ~= 0);
```

Here is the naive, but fairly effective strategy that the computer uses against you. It first checks to see if it can make a winning move. If not, it then checks to see if it needs to block your pending winning move. It neither of these happens, it simply chooses a random empty cell. This crude strategy makes the play interesting. It is possible for you to beat the computer if its random choices are not optimal. The exercises suggest some improvements in this strategy.

```
function [i,j] = strategy(X,p);
% [i,j] = strategy(X,p) is a better, but not perfect, move for player p.

% Appear to think.
pause(0.5)

% If possible, make a winning move.
[i,j] = winningmove(X,p);

% Block any winning move by opponent.
if isempty(i)
    [i,j] = winningmove(X,-p);
end

% Otherwise, make a random move.
if isempty(i)
```

```
[i,j] = find(X == 0);
m = ceil(rand*length(i));
i = i(m);
j = j(m);
end
```

Finally, here is the function that seeks a winning move by either player. It looks for rows, columns, or diagonals with sums equal to $2*p$ for either value of p .

```
function [i,j] = winningmove(X,p);
% [i,j] = winningmove(X,p) finds any winning move for player p.

s = 2*p;
if any(sum(X) == s)
    j = find(sum(X) == s);
    i = find(X(:,j) == 0);
elseif any(sum(X') == s)
    i = find(sum(X') == s);
    j = find(X(i,:) == 0);
elseif sum(diag(X)) == s
    i = find(diag(X) == 0);
    j = i;
elseif sum(diag(fliplr(X))) == s
    i = find(diag(fliplr(X)) == 0);
    j = 4 - i;
else
    i = [];
    j = [];
end
```

The remainder of the `tictactoe` function is responsible for the gui.

Let's see how all this works on an example. We begin with the position shown in figure 11.2, after two moves by each player. The state of the game at this point is

```
X =
1     0     0
0    -1     0
-1     0     1
```

It is green's move. Green needs to click the 6 button to prevent blue from having $4+5+6 = 15$. Or, it is probably easier to look at $M = \text{magic}(3)$

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

You see that 4 and 5 are on the antidiagonal, so green needs the 6 to prevent a win in the TicTacToe version of this game. Either way, this corresponds to $(i, j) = (1, 3)$ and setting $X(i, j) = 1$.

```
X =
1     0     1
0    -1     0
-1    0     1
```

Now it is blue's turn. Both the first row and the third column of X have sums equal to 2, corresponding to the fact that green has two pending winning moves. With $p = -1$, the `strategy` function looks for a block by calling `winningmove(X, -p)`. The result is $(i, j) = (2, 3)$, although it just as well could have been $(i, j) = (1, 2)$ if `winningmove` made its checks in another order. This leads to

```
X =
1     0     1
0    -1    -1
-1    0     1
```

On the next turn, green finds that the sum along the first row is 2, and so sets $X(1, 2) = 1$ to make the sum 3. This gives

```
X =
1     1     1
0    -1    -1
-1    0     1
```

and green proclaims a win.

It would be possible to implement recursive backtracking strategies like we describe in the Sudoku chapter. But there are only a few hundred possible games, so the backtracking would be exhaustive. And boring – all the games would be draws.

Recap

```
%% TicTacToe Chapter Recap
% This is an executable program that illustrates the statements
% introduced in the TicTacToe Chapter of "Experiments in MATLAB".
% You can access it with
%
%     tictactoe_recap
%     edit tictactoe_recap
%     publish tictactoe_recap
%
% Related EXM programs
%
%     tictactoe
```

```
%% tictactoe/winner

% function p = winner(X)
% % p = winner(X) returns
% %   p = 0, no winner yet,
% %   p = -1, blue has won,
% %   p = 1, green has won,
% %   p = 2, game is a draw.
%
% for p = [-1 1]
%   s = 3*p;
%   win = any(sum(X) == s) || any(sum(X') == s) || ...
%         sum(diag(X)) == s || sum(diag(fliplr(X))) == s;
%   if win
%     return
%   end
% end
% p = 2*all(X(:) ~= 0);

%% tictactoe/strategy

% function [i,j] = strategy(X,p);
% [i,j] = strategy(X,p) is a move for player p.
%
% % Appear to think.
% pause(0.5)
%
% % If possible, make a winning move.
% [i,j] = winningmove(X,p);
%
% % Block any winning move by opponent.
% if isempty(i)
%   [i,j] = winningmove(X,-p);
% end
%
% % Otherwise, make a random move.
% if isempty(i)
%   [i,j] = find(X == 0);
%   m = ceil(rand*length(i));
%   i = i(m);
%   j = j(m);
% end

%% tictactoe/winningmove
```

```
% function [i,j] = winningmove(X,p);
% [i,j] = winningmove(X,p) finds any winning move for player p.
%
% s = 2*p;
% if any(sum(X) == s)
%     j = find(sum(X) == s);
%     i = find(X(:,j) == 0);
% elseif any(sum(X') == s)
%     i = find(sum(X') == s);
%     j = find(X(i,:) == 0);
% elseif sum(diag(X)) == s
%     i = find(diag(X) == 0);
%     j = i;
% elseif sum(diag(fliplr(X))) == s
%     i = find(diag(fliplr(X)) == 0);
%     j = 4 - i;
% else
%     i = [];
%     j = [];
% end
```

Exercises

11.1 *Traditional.* Modify `tictactoe.m` so that it uses traditional X's and O's.

11.2 *Win.* Is it possible to win against `tictactoe` with its naive strategy?

11.3 *First move.* Modify `tictactoe` so that the computer takes the first move.

11.4 *Center square.* Modify the strategy used by `tictactoe.m` so that, before taking a random move, it takes the center square if it is available. Does this improve the program's chances of winning or forcing a draw?

11.5 *xkcd.* Implement the complete tictactoc strategy available from the Web comic strip `xkcd` by Randal Munroe at

<http://xkcd.com/832>

11.6 *Computer versus computer.* Modify `tictactoe.m` so that the computer plays against itself. Count the number of draws and wins for both sides. Run a large number of games, with and without the addition of the center square strategy.

Chapter 12

Game of Life

Conway's Game of Life makes use of sparse matrices.

The “Game of Life” was invented by John Horton Conway, a British-born mathematician who is now a professor at Princeton. The game made its public debut in the October 1970 issue of *Scientific American*, in the “*Mathematical Games*” column written by Martin Gardner. At the time, Gardner wrote

This month we consider Conway's latest brainchild, a fantastic solitaire pastime he calls “life”. Because of its analogies with the rise, fall and alternations of a society of living organisms, it belongs to a growing class of what are called “simulation games” – games that resemble real-life processes. To play life you must have a fairly large checkerboard and a plentiful supply of flat counters of two colors.

Of course, today we can run the simulations on our computers.

The *universe* is an infinite, two-dimensional rectangular grid. The *population* is a collection of grid cells that are marked as *alive*. The population evolves at discrete time steps known as *generations*. At each step, the fate of each cell is determined by the vitality of its eight nearest neighbors and this rule:

- A live cell with two live neighbors, or any cell with three live neighbors, is alive at the next step.

The fascination of Conway's Game of Life is that this deceptively simple rule leads to an incredible variety of patterns, puzzles, and unsolved mathematical problems – just like real life.

If the initial population consists of only one or two live cells, it expires in one step. If the initial population consists of three live cells then, because of rotational

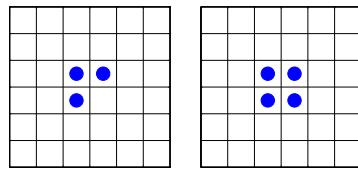


Figure 12.1. A pre-block and a block.

and reflexive symmetries, there are only two different possibilities – the population is either L-shaped or I-shaped. The left half of figure 12.1 shows three live cells in an L-shape. All three cells have two live neighbors, so they survive. The dead cell that they all touch has three live neighbors, so it springs to life. None of the other dead cells have enough live neighbors to come to life. So the result, after one step, is the population shown in the right half of figure 12.1. This four-cell population, known as the *block*, is stationary. Each of the live cells has three live neighbors and so lives on. None of the other cells can come to life.

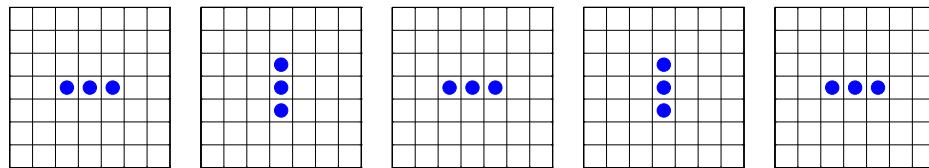


Figure 12.2. A blinker blinking.

The other three-cell initial population is I-shaped. The two possible orientations are shown in first two steps of figure 12.2. At each step, two end cells die, the middle cell stays alive, and two new cells are born to give the orientation shown in the next step. If nothing disturbs it, this *blinker* keeps blinking forever. It repeats itself in two steps; this is known as its *period*.

One possible four-cell initial population is the block. Discovering the fate of the other four-cell initial populations is left to an exercise.

The beginning of the evolution of the most important five-cell initial population, known as the *glider*, is shown in figure 12.3. At each step two cells die and two new ones are born. After four steps the original population reappears, but it has moved diagonally down and across the grid. It continues to move in this direction forever, eventually disappearing out of our field of view, but continuing to exist in the infinite universe.

The fascination of the Game of Life cannot be captured in these static figures. Computer graphics lets you watch the dynamic development. We will show just more one static snapshot of the evolution of an important larger population. Figure 12.4 is the *glider gun* developed by Bill Gosper at MIT in 1970. The portion of the population between the two static blocks oscillates back and forth. Every 30 steps, a glider emerges. The result is an infinite stream of gliders that fly out of the field of view.

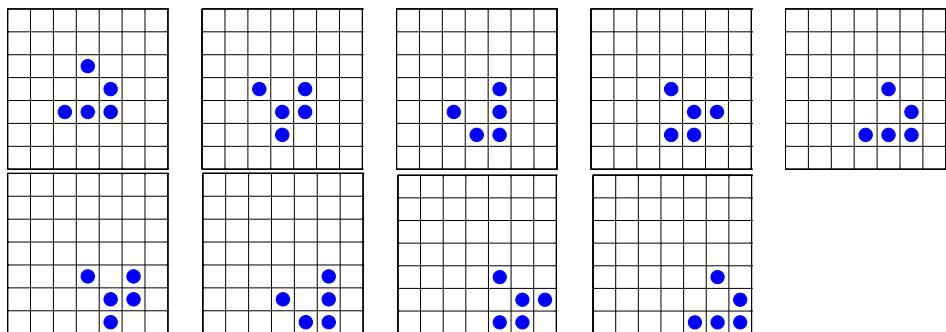


Figure 12.3. A glider gliding.

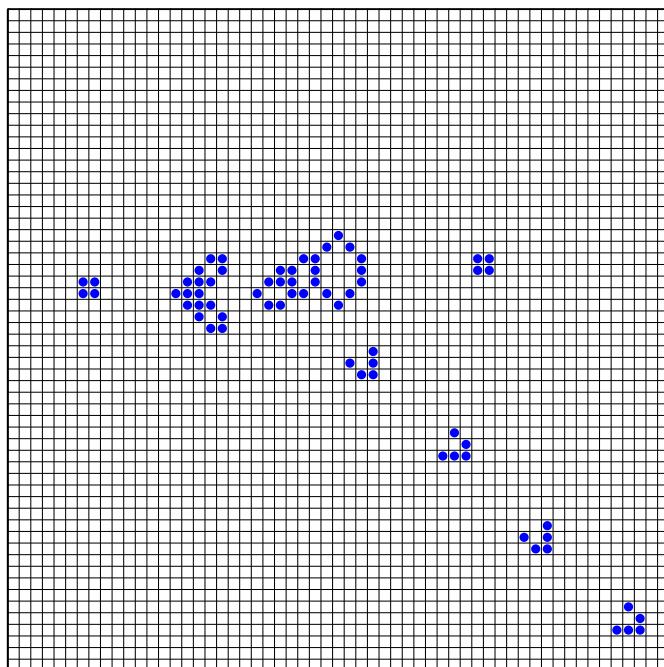


Figure 12.4. Gosper's glider gun.

MATLAB is a convenient environment for implementing the Game of Life. The universe is a matrix. The population is the set of nonzero elements in the matrix. The universe is infinite, but the population is finite and usually fairly small. So we can store the population in a finite matrix, most of whose elements are zero, and increase the size of the matrix if necessary when the population expands. This is the ideal setup for a *sparse* matrix. Conventional storage of an n -by- n matrix requires n^2 memory. But sparse storage of a matrix X requires just three vectors, one integer and one floating point vector of length $\text{nnz}(X)$ – the number of nonzero

elements in \mathbf{X} – and one integer vector of length n , not n^2 , to represent the start of each column. For example, the snapshot of the Gosper glider gun in figure 12.4 is represented by an 85-by-85 matrix with 68 nonzero entries. Conventional full matrix storage would require $85^2 = 7225$ elements. Sparse matrix storage requires only $2 \cdot 65 + 85 = 221$ elements. This advantage of sparse over full storage increases as more gliders are created, the population expands, and n increases.

The `exm` toolbox includes a program called `lifex`. (MATLAB itself has a simpler demo program called `life`.) The initial population is represented by a matrix of 0's and 1's. For example,

```
G = [1 1 1; 1 0 0; 0 1 0]
```

produces a single glider

```
G =
1     1     1
1     0     0
0     1     0
```

The universe is represented by a sparse n -by- n matrix \mathbf{X} that is initially all zero. We might start with $n = 23$ so there will be a 10 cell wide border around a 3-by-3 center. The statements

```
n = 23;
X = sparse(n,n)
```

produce

```
X =
All zero sparse: 23-by-23
```

The initial population is injected in the center of the universe. with the statement

```
X(11:13,11:13) = G
```

This produces a list of the nonzero elements

```
X =
(11,11)      1
(12,11)      1
(11,12)      1
(13,12)      1
(11,13)      1
```

We are now ready to take the first step in the simulation. Whether cells stay alive, die, or generate new cells depends upon how many of their eight neighbors are alive. The statements

```
n = size(X,1);
p = [1 1:n-1];
q = [2:n n];
```

generate index vectors that increase or decrease the centered index by one, thereby accessing neighbors to the left, right, up, down, and so on. The statement

$$\begin{aligned} Y = & X(:, p) + X(:, q) + X(p, :) + X(q, :) + \dots \\ & X(p, p) + X(q, q) + X(p, q) + X(q, p) \end{aligned}$$

produces a sparse matrix with integer elements between 0 and 8 that counts how many of the eight neighbors of each interior cell are alive. In our example with the first step of the glider, the cells with nonzero counts are

$Y =$			
(9,9)	1	(10,9)	2
(11,9)	2	(12,9)	1
(9,10)	2	(10,10)	2
(11,10)	3	(12,10)	2
(13,10)	1	(9,11)	3
(10,11)	3	(11,11)	5
(12,11)	1	(13,11)	1
(9,12)	2	(10,12)	1
(11,12)	3	(12,12)	1
(13,12)	1	(9,13)	1
(10,13)	1	(11,13)	1

The basic rule of Life is

A live cell with two live neighbors, or any cell with three live neighbors, is alive at the next step.

This is implemented with the single MATLAB statement

$$X = (X \& (Y == 2)) | (Y == 3)$$

The two characters '==' mean "is equal to". The '&' character means "and". The '|' means "or". These operations are done for all the cells in the interior of the universe. In this example, there are four cells where Y is equal to 3, so they survive or come alive. There is one cell where X is equal to 1 and Y is equal to 2, so it survives. The result is

$X =$	
(11,11)	1
(12,11)	1
(10,12)	1
(11,12)	1
(12,13)	1

Our glider has taken its first step.

One way to use `lifex` is to provide your own initial population, as either a full or a sparse matrix. For example, you create your own fleet of gliders fleet with

$$G = [1 \ 1 \ 1; \ 1 \ 0 \ 0; \ 0 \ 1 \ 0]$$

```
S = sparse(15,15);
for i = 0:6:12
    for j = 0:6:12
        S(i+(1:3),j+(1:3)) = G;
    end
end
lifex(S)
```

The Web page

http://www.argentum.freeserve.co.uk/lex_home.htm

is the home of the “Life Lexicon”, maintained by Stephen Silver. Among thousands of the facts of Life, this 160-page document lists nearly 450 different initial populations, together with their history and important properties. We have included a text copy of the Lexicon with the `exm` toolbox in the file

`exm/lexicon.txt`

`Lifex` can read initial populations from the Lexicon. Calling `lifex` with no arguments,

`lifex`

picks a random initial population from the Lexicon. Either

`lifex('xyz')`

or

`lifex xyz`

will look for a population whose name begins with `xyz`. For example, the statements

```
lifex pre-block
lifex block
lifex blinker
lifex glider
```

start with the simple populations that we have used in this introduction. The statement

`lifex Gosper`

provides Gosper’s glider gun.

By default, the initial population is surrounded by a strip of 20 dead border cells to provide a viewing window. You can change this to use `b` border cells with

`lifex('xyz',b)`

If the population expands during the simulation and cells travel beyond this viewing window, they continue to live and participate even though they cannot be seen.

Further Reading

The Wikipedia article is a good introduction.

http://en.wikipedia.org/wiki/Conway's_Game_of_Life

Another good introduction is available from Math.com, although there are annoying popups and ads.

<http://www.math.com/students/wonders/life/life.html>

If you find yourself at all interested in the Game of Life, take a good look at the Lexicon, either by reading our text version or by visiting the Web page.

http://www.argentum.freeserve.co.uk/lex_home.htm

Recap

```
%% Life Chapter Recap
% This is an executable program that illustrates the statements
% introduced in the Life Chapter of "Experiments in MATLAB".
% You can access it with
%
%     life_recap
% It does not work so well with edit and publish.
%
% Related EXM programs
%
%     lifex

% Generate a random initial population
X = sparse(50,50);
X(21:30,21:30) = (rand(10,10) > .75);
p0 = nnz(X);

% Loop over 100 generations.
for t = 1:100

    spy(X)
    title(num2str(t))
    drawnow

    % Whether cells stay alive, die, or generate new cells depends
    % upon how many of their eight possible neighbors are alive.
    % Index vectors increase or decrease the centered index by one.

    n = size(X,1);
    p = [1 1:n-1];
```

```

q = [2:n n];

% Count how many of the eight neighbors are alive.

Y = X(:,p) + X(:,q) + X(p,:) + X(q,:)
    + ...
    X(p,p) + X(q,q) + X(p,q) + X(q,p);

% A live cell with two live neighbors, or any cell with
% three live neighbors, is alive at the next step.

X = (X & (Y == 2)) | (Y == 3);

end

p100 = nnz(X);
fprintf('%5d %5d %8.3f\n', p0, p100, p100/p0)

```

Exercises

12.1 *life vs. lifex* The MATLAB demos toolbox has an old program called `life`, without an `x`. In what ways is it the same as, and in what ways does it differ from, our `exm gui lifex`?

12.2 *Four-cell initial populations*. What are all of the possible four-cell initial populations, and what are their fates? You can generate one of the four-cell populations with

```

L = [1 1 1; 1 0 0];
lifex(L,4)

```

12.3 *Lexicon*. Describe the behavior of each of these populations from the Lexicon. If any is periodic, what is its period?

```

ants
B-52
blinker puffer
diehard
Canada goose
gliders by the dozen
Kok's galaxy
rabbits
R2D2
spacefiller
wasp
washerwoman

```

12.4 *Glider collisions.* What happens when:

- A glider collides head-on with a block?
- A glider side-swipes a block?
- Two gliders collide head-on?
- Two gliders clip each others wings?
- Four gliders simultaneously leave the corners of a square and head towards its center?

See also: `lifex('4-8-12')`.

12.5 *Factory.* How many steps does it take the `factory` to make one glider?

12.6 *R-pentomino.* Of all the possible five-cell initial populations, the only one that requires a computer to determine its ultimate behavior is the one that Conway dubbed the *R-pentomino*. It is shown in figure 12.5 and can be generated by

```
R = [0 1 1; 1 1 0; 0 1 0]
```

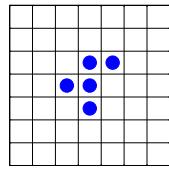


Figure 12.5. The *R-pentomino*.

As the simulation proceeds, the population throws off a few gliders, but otherwise remains bounded. If you make `b` large enough, the statement

```
lifex(R,b)
```

shows all of the bounded behavior. How large does this `b` have to be? What is the maximum population during the evolution? How many gliders are produced? How many steps does it take for the population to stabilize? How many blinkers are present in the stabilized population? What is size of the stabilized population?

12.7 *Execution time.* Display actual computer execution time by inserting `tic` and `toc` in `lifex.m`. Place the single statement

```
tic
```

before the start of the inner loop. Change the call of `caption` to

```
caption(t,nnz(X),toc)
```

Make the appropriate modifications in the `caption` subfunction at the end of `lifex.m`. Demonstrate your modified program on a few interesting examples.

12.8 *The Ark.* Run

```
lifex('ark',128)
```

for a few minutes. About how much time does it take on your computer to do one step? According to the Lexicon, the `ark` requires 736692 steps to stabilize. About how much time will it take on your computer for the `ark` to stabilize?

12.9 *Houses*. Check out

```
lifex(houses)
```

12.10 *Checkerboards*. This code generates an n -by- n checkerboard of 0's and 1's.

```
[I,J] = meshgrid(1:n);
C = (mod(I+J,2)==0);
```

What are the stabilization times and final populations for n -by- n checkerboard initial populations with $3 \leq n \leq 30$?

12.11 *Early exit*. Where should these code segments

```
Xsave = X;
```

and

```
if isequal(X,Xsave)
    break
end
```

be inserted in `life_recap`?

12.12 *Symmetry*. What is the effect of inserting the statement

```
X = (X + X' > 0)
```

after the generation of `X` at the beginning of `life_recap`.

12.13 *Live or die?* Insert these statements before the first executable statement of `life_recap`.

```
P = zeros(10000,2);
for s = 1:10000
```

Insert these statements after the last executable statement.

```
P(s,:) = [p0 p100];
end
```

Deactivate or remove the `spy`, `title`, `pause` and `fprintf` statements. Run the resulting program and then generate histograms of the results.

```
R = P(:,2)./P(:,1);
hist(P(:,1),30)
hist(P(:,2),30)
hist(R,30)
```

- (a) Is the typical population growing or declining after 100 generations?
- (b) Do you recognize the shape of the histogram of the initial population, $P(:,1)$? What is $\text{mean}(P(:,1))$? Why?
- (c) I don't know enough about probability to describe the distribution of the final population, $P(:,2)$. Can anybody help me out?

12.14 *Gosper's glider gun.* Replace the random initial population generator in `life_recap` by this code:

```
% Gosper glider gun

gun = ['
        ,           +           ,
        ,           + +          ,
        ,           ++          ++  ,
        ,           +  +          ++  ,
        ' ++         +  +          ++  ,
        ' ++         +  + ++     + +  ,
        ,           +  +          +   ,
        ,           +  +          ,
        ,           ++          ];
```



```
X = sparse(76,76);
X(34:42,20:57) = (gun=='+');
spy(X)
```

- (a) How many steps does it take for the gun to emit a glider?
- (b) What happens when a glider meets the boundary? How is this different from `lifex('gosper')`?

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,

$$z_{k+1} = z_k^2, \quad k = 0, 1, \dots$$

For which initial values z_0 does this sequence remain bounded as $k \rightarrow \infty$? It is easy to see that this set is simply the unit disc, $|z_0| \leq 1$, 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

$$z_{k+1} = z_k^2 + z_0, \quad k = 0, 1, \dots$$

remain bounded at $k \rightarrow \infty$. That's it. That's the entire definition. It's amazing that such a simple definition can produce such fascinating complexity.

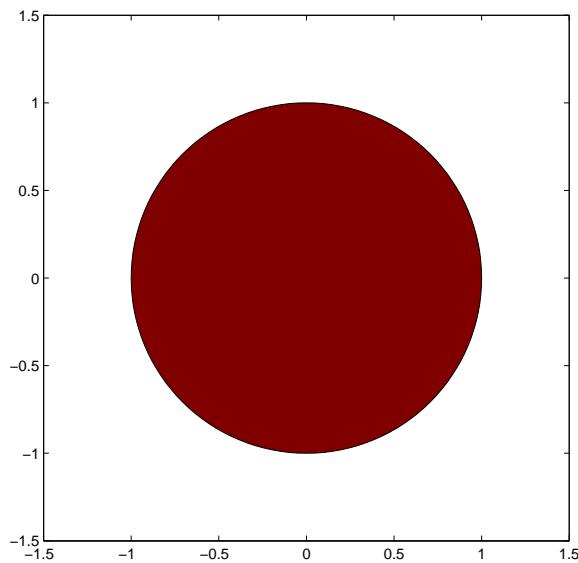


Figure 13.1. The unit disc is shown in red. The boundary is simply the unit circle. There is no intricate fringe.

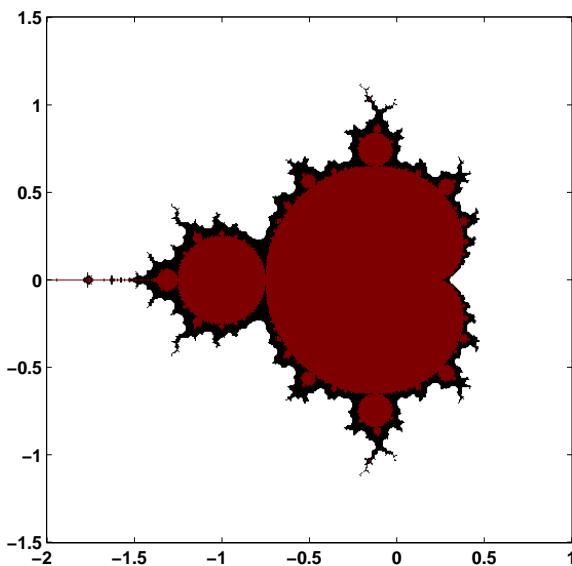


Figure 13.2. The Mandelbrot set is shown in red. The fringe just outside the set, shown in black, is a region of rich structure.

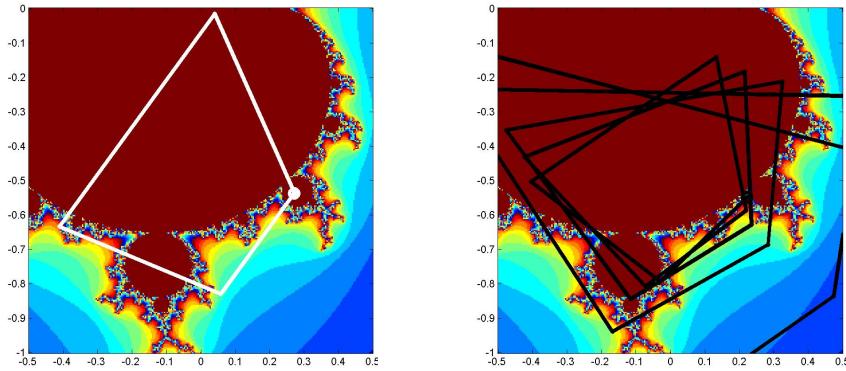


Figure 13.3. Two trajectories. $z_0 = .25 - .54i$ generates a cycle of length four, while nearby $z_0 = .22 - .54i$ generates an unbounded trajectory.

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, $z_0 = .25-.54i$, 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 z_0 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, $z_0 = .25-.54i$, is in the Mandelbrot set, while the second value, $z_0 = .22-.54i$, which is nearby, is not.

The algorithm doesn't have to wait until z reaches floating point overflow. As soon as z satisfies

```
abs(z) >= 2
```

subsequent iterations will essentially 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.

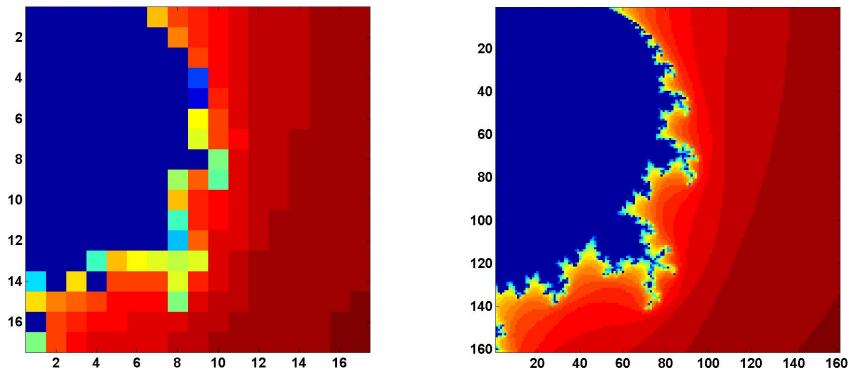


Figure 13.4. Improving resolution.

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`.

$$z_{k,j} = x_k + y_j i, \quad i = \sqrt{-1}, \quad k, j = 1, \dots, n$$

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 array 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 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 3 2 2 2
 32 32 32 32 32 32 32 32 27 5 4 3 3 3 3 2 2 2
 32 32 32 32 32 32 32 32 30 6 4 3 3 3 3 2 2 2
 32 32 32 32 32 32 32 32 13 7 4 3 3 3 3 2 2 2
 32 32 32 32 32 32 32 32 14 7 5 3 3 3 2 2 2 2
 32 32 32 32 32 32 32 32 32 17 4 3 3 2 2 2 2 2
 32 32 32 32 32 32 32 32 16 8 18 4 3 3 2 2 2 2
 32 32 32 32 32 32 32 32 11 6 5 4 3 3 2 2 2 2
 32 32 32 32 32 32 32 32 19 6 5 4 3 2 2 2 2 2
 32 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 2
 22 32 12 32 7 7 7 14 6 4 3 3 2 2 2 2 2 2 2
 12 9 8 7 5 5 5 17 4 3 3 2 2 2 2 2 2 1 1
 32 7 6 5 5 4 4 4 3 3 2 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 z_0 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 investigate 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

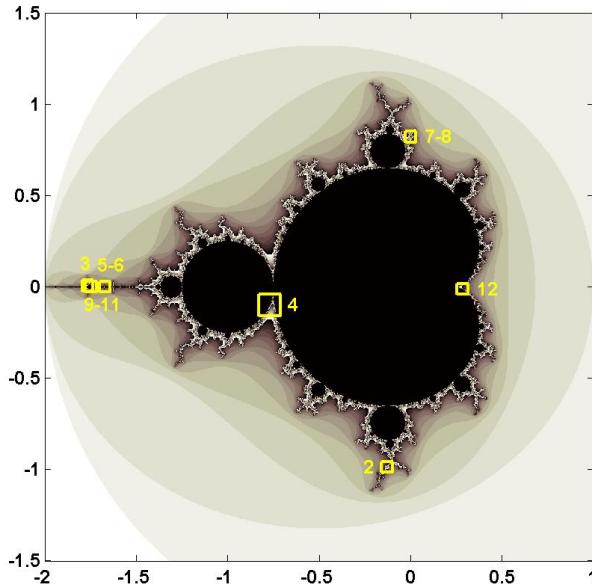


Figure 13.5. The figures in this chapter, and the predefined regions in our mandelbrot program, show these regions in the fringe just outside the Mandelbrot set.

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.

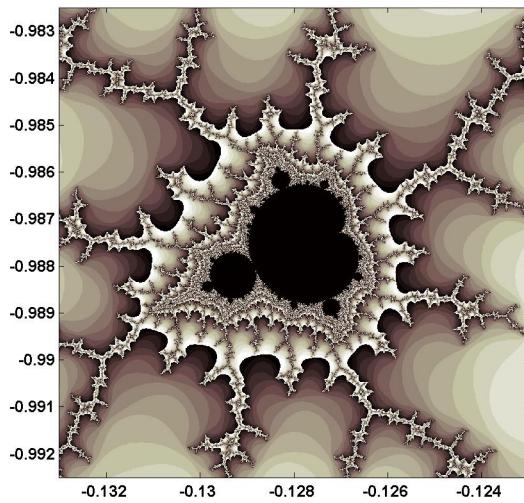


Figure 13.6. Region #2, “Mandelbrot Junior”. The fringe around the Mandelbrot set is self similar. Small versions of the set appear at all levels of magnification.

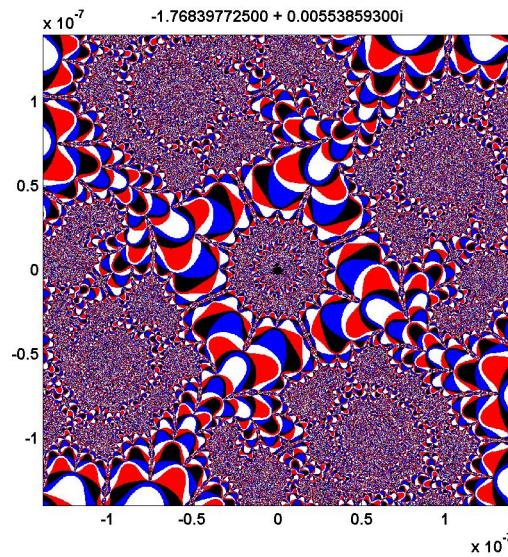


Figure 13.7. Region #3, “Plaza”, with the flag color map.

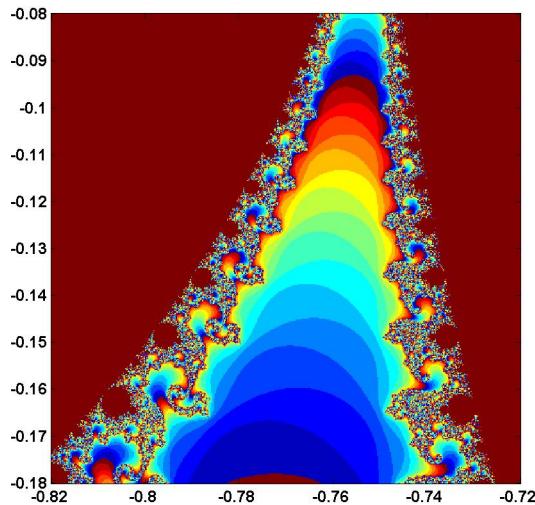


Figure 13.8. Region #4. “Valley of the Seahorses”.

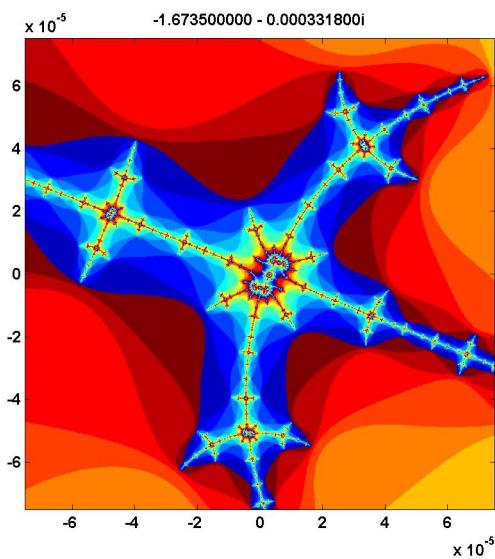


Figure 13.9. Region #5. Our “West Wing” is located just off the real axis in the thin far western portion of the set, near $\text{real}(z) = -1.67$.

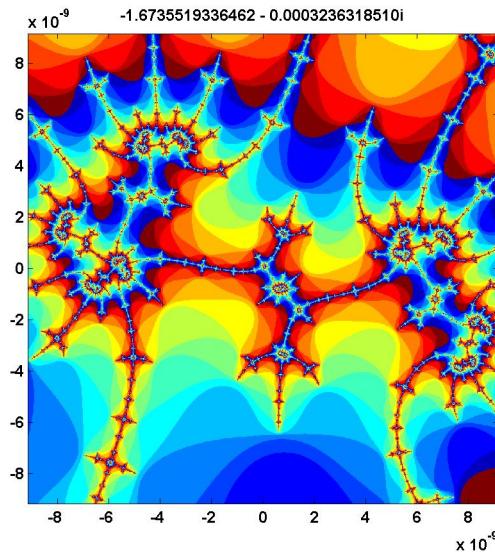


Figure 13.10. Region #6. “Dueling Dragons”.

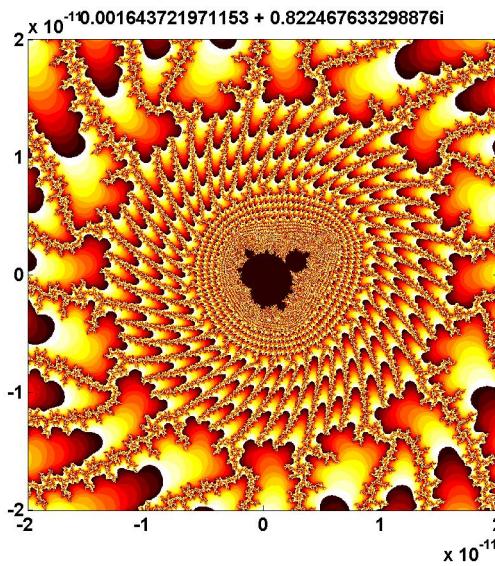


Figure 13.11. Region #7. The “Buzzsaw” requires a magnification factor of 10^{11} to reveal a tiny copy of the Mandelbrot set.

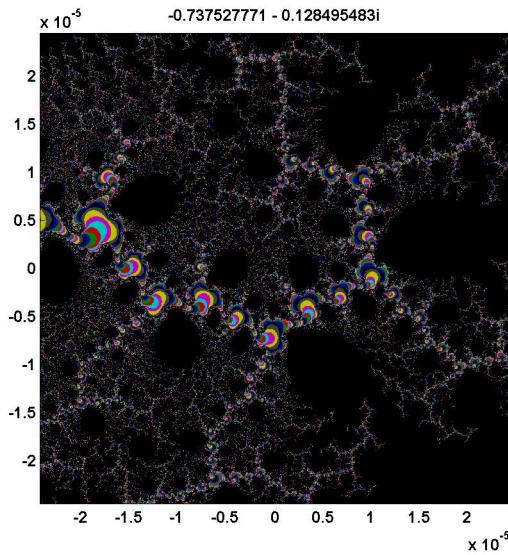


Figure 13.12. Region #8. “Nebula”. Interstellar dust.

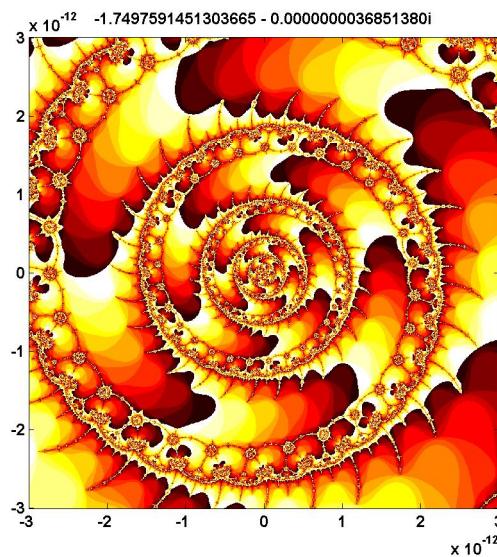


Figure 13.13. Region #9. A vortex, not far from the West Wing. Zoom in on one of the circular “microbugs” near the left edge.

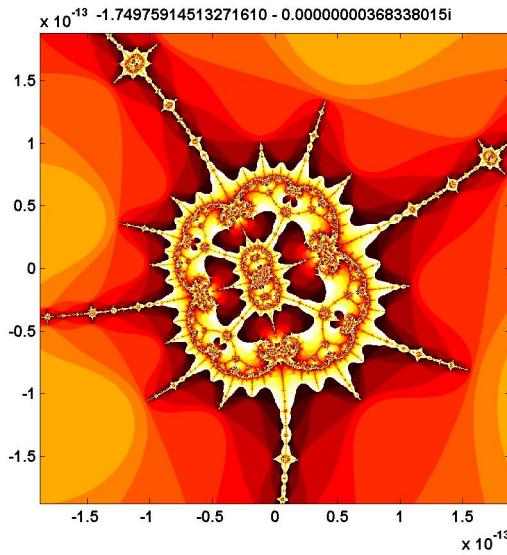


Figure 13.14. Region #10. A 10^{13} magnification factor reveals a “microbug” within the vortex.

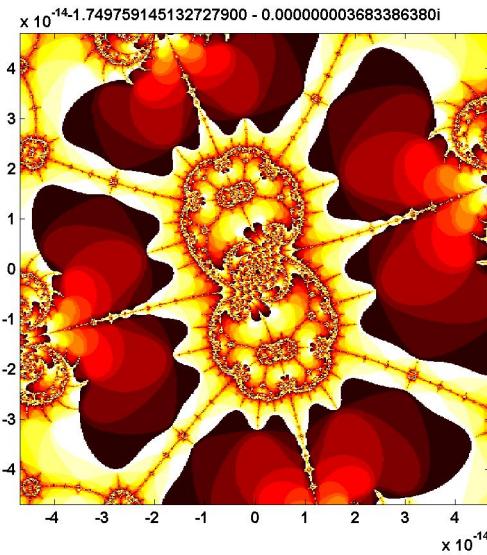


Figure 13.15. Region #11. The nucleus of the microbug.

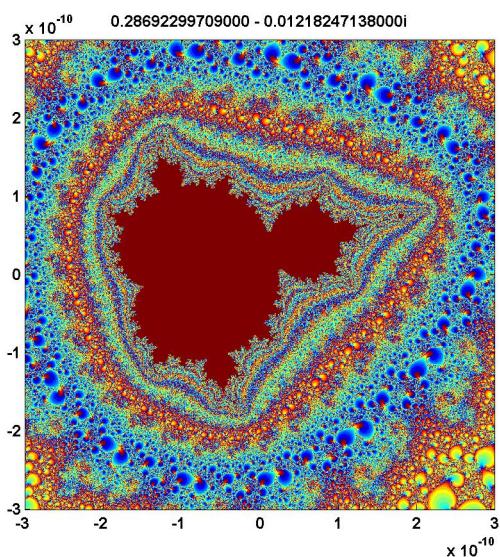


Figure 13.16. Region #12. “Geode”. This colorful image requires a 2048-by-2048 grid and depth = 8192.

Chapter 14

Sudoku

Humans look for patterns, but machines use backtracking.

The fascination that people have for solving *Sudoku* puzzles without a computer derives from the discovery and mastery of a myriad of subtle combinations and patterns that provide tips toward the solution. The Web has hundreds of sites describing these patterns, which have names like “hidden quads”, “X-wing” and “squirmbag”.

It is not our intention to describe a MATLAB program that duplicates these human pattern recognition capabilities. Instead, our program, like most other *Sudoku* computer codes, takes a very different approach, one that relies on the machine’s almost limitless capacity to carry out brute force trial and error. We use only one pattern, *singletons*, together with a fundamental computer science technique, recursive backtracking.

In the Sudoku world, backtracking is regarded as “guessing” and is considered bad form. But as a computational technique, it is easy to understand, straightforward to implement and guaranteed to succeed.

In this chapter, we will describe five MATLAB programs for experiments with *Sudoku*.

- **sudoku.** An interactive program that lets you follow the computer solution process.
- **sudoku_basic.** The same solution process as **sudoku**, but without the graphics interface. Much easier to read, and much faster if you just want the solution.
- **sudoku_all.** Enumerate all solutions to a Sudoku puzzle. A valid puzzle should have only one solution.

Copyright © 2011 Cleve Moler
MATLAB® is a registered trademark of MathWorks, Inc.™
October 4, 2011

- `sudoku_assist`. An interactive program that lets you control the solution process, as you would by hand on paper.
- `sudoku_puzzle`. A collection of interesting puzzles.

Sudoku is actually an American invention. It first appeared, with the name Number Place, in the Dell Puzzle Magazine in 1979. The creator was probably Howard Garns, an architect from Indianapolis. A Japanese publisher, Nikoli, took the puzzle to Japan in 1984 and eventually gave it the name *Sudoku*, which is a kind of kanji acronym for “numbers should be single, unmarried.” The Times of London began publishing the puzzle in the UK in 2004 and it was not long before it spread back to the US and around the world.

8	1	6						3
3	5	7						8
4	9	2	1					
		3	8	1	6			
			3	5	7			
			4	9	2	1		
				3	8	1	6	
	2				3	5	7	
1					4	9	2	

Figure 14.1. A *Sudoku* puzzle featuring Lo-Shu, the magic square of order 3.

You probably already know the rules of *Sudoku*. Figure 20 is an example of an initial 9-by-9 grid, with a few specified digits known as the *clues*. This particular puzzle reflects our interest in magic squares. It is the sum of the two matrices shown in figure 20. The matrix on the left is generated by the functions `kron`, which computes something known as the *Kronecker product*, and `magic`, which generates magic squares.

```
X = kron(eye(3),magic(3))
```

In this case, a Kronecker product involving the identity matrix produces a block matrix with three copies of `magic(3)` on the diagonal. By itself, this array is not a valid *Sudoku* puzzle, because there is more than one way to complete it. The solution is not unique. In order to get a puzzle with a unique solution, we have added the matrix on the right in figure 20.

Figure 14.3 is the final completed grid. Each row, each column, and each major 3-by-3 block, must contain exactly the digits 1 through 9. In contrast to

8	1	6						
3	5	7						
4	9	2						
			8	1	6			
			3	5	7			
			4	9	2			
				8	1	6		
				3	5	7		
				4	9	2		
							3	
							8	
						1		
				3				
					1			
						3		
						2		
						1		

Figure 14.2. Our Lo-Shu based puzzle is the sum of these two matrices.

8	1	6	5	7	9	2	4	3
3	5	7	6	2	4	9	8	1
4	9	2	1	3	8	7	6	5
2	4	3	8	1	6	5	7	9
9	8	1	3	5	7	6	2	4
7	6	5	4	9	2	1	3	8
5	7	9	2	4	3	8	1	6
6	2	4	9	8	1	3	5	7
1	3	8	7	6	5	4	9	2

Figure 14.3. The completed puzzle. The digits have been inserted so that each row, each column, and each major 3-by-3 block contains 1 through 9. The original clues are shown in blue.

magic squares and other numeric puzzles, no arithmetic is involved. The elements in a *Sudoku* grid could just as well be nine letters of the alphabet, or any other distinct symbols.

1			
		3	
	2		
			4

Figure 14.4. Shidoku

1	3 4	2 4	2
2 4	4	3	1 2
3 4	2	1	1 3
3	1 3	1 2	4

Figure 14.5. Candidates

1	3 4	2 4	2
2 4	4	3	1 2
4	2	1	1 3
3	1	1 2	4

Figure 14.6. Insert singleton

1	3	4	2
2	4	3	1
4	2	1	3
3	1	2	4

Figure 14.7. Solution

Shidoku

To see how our *sudoku* program works, we can use *Shidoku* instead of *Sudoku*. “Shi” is Japanese for “four”. The puzzles use a 4-by-4 grid and are almost trivial to solve by hand. Figure 20 is our first *Shidoku* puzzle and the next three figures show steps in its solution. In figure 20, the possible entries, or candidates, are shown by small digits. For example, row two contains a 3 and column one contains a 1 so the candidates in position (2,1) are 2 and 4. Four of the cells have only one candidate each. These are the *singletons*, shown in red. In figure 20, we have inserted the singleton 3 in the (4,1) cell and recomputed the candidates. In figure 14.7, we have inserted the remaining singletons as they are generated to complete the solution.

1			
	2		
		3	
			4

Figure 14.8. $\text{diag}(1:4)$

1	3 4	2 4	2 2
3 4	2	1 4	1 3
2 4	1 4	3	1 2
3	1 3	1 2	4

Figure 14.9. No singletons.

1			
3	2		
		3	
			4

Figure 14.10. Backtrack step.

1	4	2	3
3	2	4	1
4	1	3	2
2	3	1	4

Figure 14.11. Solution is not unique.

The input array for figure 14.8 is generated by the MATLAB statement

```
X = diag(1:4)
```

As figure 14.9 shows, there are no singletons. So, we employ a basic computer science technique, recursive backtracking. We select one of the empty cells and tentatively insert one of its candidates. All of the empty cells have two candidates, so we pick the first one and tentatively insert a 3 in cell (2,1). This creates a new puzzle, shown in figure 14.10. Our program is then called recursively. In this example, the new puzzle is easily solved and the result is shown in figure 14.11. However, the solution depends upon the choices that we made before the recursive call. Other choices can lead to different solutions. For this simple diagonal initial condition,

the solution is not unique. There are two possible solutions, which happen to be matrix transposes of each other.

```

Y = shidoku(diag(1:4))

Y =
    1     4     2     3
    3     2     4     1
    4     1     3     2
    2     3     1     4

Z = shidoku(diag(1:4)')

Z =
    1     3     4     2
    4     2     1     3
    2     4     3     1
    3     1     2     4

```

Existence and uniqueness

Mathematicians are always concerned about existence and uniqueness in the various problems that they encounter. For *Sudoku*, neither existence nor uniqueness can be determined easily from the initial clues. It would be very frustrating if a puzzle with no solution, or with more than one solution, were to show up in your daily newspaper.

Uniqueness is an elusive property. In fact, most descriptions of *Sudoku* do not specify that there has to be exactly one solution. The only way that I know to check uniqueness is to exhaustively enumerate all possibilities.

For our magic square puzzle in figure 20, if we were to replace the 1 in the (9,1) cell by a 5, 6 or 7, the row, column and block conditions would still be satisfied. It turns out that a 5 would produce another valid puzzle with a unique solution, a 6 would produce a puzzle with two possible solutions, and a 7 would produce a puzzle with no solution. An exercise asks you to verify these claims.

It takes `sudoku_all` a little over half an hour on my laptop to determine that the magic square puzzle generated by

```
X = kron(eye(3),magic(3))
```

without the additional entries on right in figure 20 has 283576 solutions.

The puzzle generated by

```
X = sudoku_puzzle(13)
```

has an interesting spiral pattern. But there are over 300 solutions. An exercise asks you to find out exactly how many.

A number of operations on a Sudoku grid can change its visual appearance without changing its essential characteristics. All of the variations are basically the same puzzle. These equivalence operations can be expressed as array operations in MATLAB. For example

```
p = randperm(9)
z = find(X > 0)
X(z) = p(X(z))
```

permutes the digits representing the elements. Other operations include

```
X'
rot90(X,k)
flipud(X)
fliplr(X)
X([4:9 1:3],:)
X(:,[randperm(3) 4:9])
```

The algorithm

If we do not count the comments and GUI, `sudoku.m` involves less than 40 lines of code. The outline of the main program is:

- Fill in all singletons.
- Exit if a cell has no candidates.
- Fill in a tentative value for an empty cell.
- Call the program recursively.

`sudoku` and `sudoku_basic`

Here is the header for `sudoku_basic`.

```
function [X,steps] = sudoku_basic(X,steps)
% SUDOKU_BASIC Solve the Sudoku puzzle using recursive backtracking.
%   sudoku_basic(X), for a 9-by-9 array X, solves the Sudoku puzzle for X
%   without providing the graphics user interface from sudoku.m
%   [X,steps] = sudoku_basic(X) also returns the number of steps.
%   See also sudoku, sudoku_all, sudoku_assist, sudoku_puzzle.
```

This first bit of code checks `nargin`, the number of input arguments, and initializes the step counter on the first entry. Any subsequent recursive calls will have a second argument.

```
if nargin < 2
    steps = 0;
end
```

These comments describe the primary variables in the program.

```
% C is the array of candidate values for each cell.
% N is the vector of the number of candidates for each cell.
% s is the index of the first cell with the fewest candidates.
```

The program fills in any singletons, one at a time. The candidates are recomputed after each step. This section of code, by itself, can solve puzzles that do not require backtracking. Such puzzles can be classified as “easy”.

```
[C,N] = candidates(X);
while all(N>0) & any(N==1)
    s = find(N==1,1);
    X(s) = C{s};
    steps = steps + 1;
    [C,N] = candidates(X);
end
```

If filling in the singletons does not solve the puzzle, we reach the following section of code that implements backtracking. The backtracking generates many impossible configurations. The recursion is terminated by encountering a puzzle with no solution.

```
if all(N>0)
    Y = X;
    s = find(N==min(N),1);
    for t = [C{s}]                                % Iterate over the candidates.
        X = Y;
        X(s) = t;                                 % Insert a tentative value.
        steps = steps + 1;

        [X,steps] = sudoku_basic(X,steps);      % Recursive call.

        if all(X(:) > 0)                         % Found a solution.
            break
        end
    end
end
```

All of the bookkeeping required by backtracking is handled by the recursive call mechanism in MATLAB and the underling operating system.

The key internal function for `sudoku` is `candidates`. Here is the preamble.

```
function [C,N] = candidates(X)
    % C = candidates(X) is a 9-by-9 cell array of vectors.
    % C{i,j} is the vector of allowable values for X(i,j).
    % N is a row vector of the number of candidates for each cell.
    % N(k) = Inf for cells that already have values.
```

The following statement creates an internal function named `tri` that computes the indices for blocks. For example, `tri(1)`, `tri(2)` and `tri(3)` are all equal to `1:3` because the block that includes a cell with row or column index equal to 1, 2 or 3 has row or column indices `1:3`.

```
tri = @(k) 3*ceil(k/3-1) + (1:3);
```

Here is the core of `candidates`.

```
C = cell(9,9);
for j = 1:9
    for i = 1:9
        if X(i,j)==0
            z = 1:9;
            z(nonzeros(X(:,j))) = 0;
            z(nonzeros(X(:,i))) = 0;
            z(nonzeros(X(tri(i),tri(j)))) = 0;
            C{i,j} = nonzeros(z)';
        end
    end
end
```

For each empty cell, this function starts with `z = 1:9` and uses the numeric values in the associated row, column and block to zero elements in `z`. The nonzeros that remain are the candidates. For example, consider the (1,1) cell in figure 20. We start with

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

The values in the first row change `z` to

```
z = 1 0 0 0 5 6 7 8 9
```

Then the first column changes `z` to

```
z = 1 0 0 0 5 0 7 0 9
```

The (1,1) block does not make any further changes, so the candidates for this cell are `C{1,1} = [1 5 7 9]`.

The `candidates` function concludes with the following statements. The number of candidates, `N(i,j)` for cell `(i,j)` is the length of `C{i,j}`. If cell `(i,j)` already has a value, then `X(i,j)` is nonzero, `C{i,j}` is empty and so `N(i,j)` is zero. But we make these `N(i,j)` infinite to distinguish them from the cells that signal an impossible puzzle.

```
N = cellfun(@length,C);
N(X>0) = Inf;
N = N(:)';
```

An example, puzzle number one

Figure 14.12 shows the key steps in the solution of our Lo-Shu puzzle. The initial candidates include just one singleton, the red 5 in position (3,9). Five steps with five singletons allow us to fill in the cells with green 5's and 8's. We now consider the first cell with two candidates, in position (6,1). The first step in the backtracking puts a tentative 6 in position (6,1). Blue values are the initial clues, cyan values are generated by the backtracking and green values are implied by the others. After five steps we reach an impossible situation because there are no candidates for position (6,2). Try again, with a 7 in position (6,1). This puzzle is easily completed by filling in singletons. The title on the final solutions shows that it took 50 steps, including the abortive ones in the first backtrack, to reach the solution.

References

- [1] Wikipedia's Mathematics of Sudoku.
http://en.wikipedia.org/wiki/Mathematics_of_Sudoku
- [2] Gordon Royle, Minimum Sudoku.
<http://mapleta.maths.uwa.edu.au/~gordon/sudokumin.php>
- [3] Nikoli.
<http://www.nikoli.co.jp/en>
- [4] Sudoku Squares and Chromatic Polynomials.
<http://www.ams.org/notices/200706/tx070600708p.pdf>
- [5] Sudoku web site modeled after Wikipedia.
<http://www.sudopedia.org/wiki>
- [6] Ruud's Guide, SudoCue.
<http://www.sudocue.net/guide.php>

Recap

```
%> Sudoku Chapter Recap
%> This is an executable program that illustrates the statements
%> introduced in the Sudoku Chapter of "Experiments in MATLAB".
%> You can access it with
%
%>     sudoku_recap
%>     edit sudoku_recap
%>     publish sudoku_recap
%
%> Related EXM programs
%
%>     sudoku
%>     sudoku_all
%>     sudoku_assist
%>     sudoku_basic
%>     sudoku_puzzle
```

```

%% Disclaimer
% Our Sudoku chapter and Sudoku program depend heavily
% upon recursion, which cannot be done by this script.

%% Sudoku puzzle incorporating the Lo-Shu magic square.
X = kron(eye(3),magic(3))
C = full(sparse([9 8 4 3 7 6 2 1], [1:4 6:9], [1 2 3 1 3 1 8 3]))
X = X + C

% Also available as
X = sudoku_puzzle(1);

%% Transforms of a Sudoku puzzle
T = X;
p = randperm(9);
z = find(X > 0);
T(z) = p(X(z))
X'
rot90(X,-1)
flipud(X)
fliplr(X)
X([4:9 1:3],:)
X(:,[randperm(3) 4:9])

%% Candidates

% C = candidates(X) is a cell array of vectors.
% C{i,j} is the set of allowable values for X(i,j).

C = cell(9,9);
tri = @k) 3*ceil(k/3-1) + (1:3);
for j = 1:9
    for i = 1:9
        if X(i,j)==0
            z = 1:9;
            z(nonzeros(X(i,:))) = 0;
            z(nonzeros(X(:,j))) = 0;
            z(nonzeros(X(tri(i),tri(j)))) = 0;
            C{i,j} = nonzeros(z)';
        end
    end
end
C

%% First singleton and first empty.

```

```
% N = number of candidates in each cell.
% s = first cell with only one candidate.
% e = first cell with no candidates.

N = cellfun(@length,C)
s = find(X==0 & N==1,1)
e = find(X==0 & N==0,1)

%% Sudoku puzzles

help sudoku_puzzle

for p = 1:16
    sudoku_puzzle(p)
end
```

Exercises

14.1 *xkcd*. Solve the binary Sudoku puzzle by Randal Munroe in the Web comic strip [xkcd](http://xkcd.com/74) at

<http://xkcd.com/74>

14.2 *Solve*. Use `sudoku` to solve a puzzle from a newspaper, magazine, or puzzle book. Use `sudoku_assist` to solve the same puzzle by hand.

14.3 *sudoku_puzzle*. The `exm` program `sudoku_puzzle` generates 16 different puzzles. The comments in the program describe the origins of the puzzles. How many steps are required by `sudoku_basic` to solve each of the puzzles?

14.4 *By hand*. Use `sudoku_assist` to solve our Lo-Shu based puzzle by hand.

```
X = sudoku_puzzle(1);
sudoku_assist(X)
```

14.5 *Modify puzzle #1*. The following program modifies the lower left hand clue in our Lo-Shu based puzzle. What happens with each of the resulting puzzles?

```
X = sudoku_puzzle(1);
for t = [0 1 5 6 7]
    X(9,1) = t;
    sudoku_all(X)
end
```

14.6 *Try this.* Try to use `sudoku_assist` to solve this puzzle by hand.

```
X = sudoku_puzzle(1);
X(9,1) = 7;
sudoku_assist(X)
```

14.7 *Puzzle #3.* How many solutions does `sudoku_puzzle(3)` have? How do they differ from each other?

14.8 *Puzzle #13.* How many solutions does `sudoku_puzzle(13)` have? In addition to the initial clues, what values are the same in all the solutions?

14.9 *All solutions.* What are the differences between `sudoku_basic` and `sudoku_all`?

14.10 *Combine.* Combine `sudoku_all` and `sudoku_basic` to create a program that returns all possible solutions and the number of steps required to find each one. Try your program on `sudoku_puzzle(13)`. What is the average number of steps required? Hints: if `S` is a cell array where each cell contains one numeric value, then `s = [S{:}]` creates the corresponding vector. And, `help mean`.

14.11 *Search strategy* In `sudoku_basic`, the statement

```
s = find(N==min(N),1);
```

determines the search strategy. What happens when you change this to

```
s = find(N==min(N),'last');
```

or

```
s = find(X==0,'first');
```

14.12 *Patterns.* Add some human puzzle solving techniques to `sudoku.m`. This will complicate the program and require more time for each step, but should result in fewer total steps.

14.13 *sudoku_alpha.* In `sudoku.m`, change `int2str(d)` to `char('A'+d-1)` so that the display uses the letters 'A' through 'I' instead of the digits 1 through 9. See figure 14.13. Does this make it easier or harder to solve puzzles by hand.

14.14 *sudoku16.* Modify `sudoku.m` to solve 16-by-16 puzzles with 4-by-4 blocks.

0

8	1	6	2	4	5	9	7	2	5	4	2	3
3	5	7	2	6	4	6	9	2	6	8	4	9
4	9	2	1	3	6	5	7	5	6	6	5	1
2	5	4	3	8	1	6	2	5	4	2	4	9
2	6	4	1	3	5	7	2	6	4	2	6	4
5	6	6	4	9	2	1	3	6	5	8	9	7
5	9	7	4	5	9	7	2	5	4	2	4	9
6	2	4	8	9	6	4	1	3	5	7	8	9
1	7	8	6	5	8	7	8	6	5	4	9	2

7

6

8	1	6	2	5	4	2	4	9	7	2	4	2	3
3	5	7	2	6	4	6	4	9	9	6	8	4	9
4	9	2	1	3	6	5	7	8	7	6	7	6	5
2	5	4	3	8	1	6	2	5	4	2	4	9	9
2	6	4	1	3	5	7	2	6	4	2	6	4	8
6	7	8	5	4	9	2	1	3	5	7	8	9	6
7	5	4	4	9	7	9	7	2	5	4	2	4	9
6	2	4	8	9	6	4	1	3	5	7	8	9	7
1	7	6	8	7	6	5	4	9	2	4	9	2	1

10

8	1	6										3
3	5	7										8
4	9	2	1		8							5
			3	8	1	6						
			3	5	7							
6	5	4	9	2	1							8
					3	8	1	6				
			2			3	5	7				
1	8			5	4	9	2					

11

8	1	6	2	5	4	2	4	9	7	2	4	2	3
3	5	7	2	6	4	6	4	9	9	6	8	4	9
4	9	2	1	3	6	5	7	8	7	6	7	6	5
7	4	3	8	1	6	2	5	4	2	4	9	9	6
2	4	1	4	9	2	3	5	7	6	4	6	4	9
6	5	4	9	2	1	3	8	7	6	7	6	5	8
5	4	4	7	9	7	9	7	2	5	4	2	4	9
9	2	4	8	9	6	4	6	4	8	3	5	7	6
1	7	6	8	7	6	5	4	9	2	4	9	2	1

50

8	1	6										3
3	5	7										8
4	9	2	1		8							5
			3	8	1	6						
			3	5	7							
7	5	4	9	2	1							8
					3	8	1	6				
2					3	5	7					
1	8			5	4	9	2					

8	1	6	5	7	9	2	4	3				
3	5	7	6	2	4	9	8	1				
4	9	2	1	3	8	7	6	5				
2	4	3	8	1	6	5	7	9				
9	8	1	3	5	7	6	2	4				
7	6	5	4	9	2	1	3	8				
5	7	9	2	4	3	8	1	6				
6	2	4	9	8	1	3	5	7				
1	3	8	7	6	5	4	9	2				

Figure 14.12. The key steps in the solution of our Lo-Shu puzzle.

H	A	F				C
C	E	G				H
D	I	B	A			
	C	H	A	F		
	C	E	G			
	D	I	B	A		
			C	H	A	F
	B			C	E	G
A				D	I	B

Figure 14.13. Use the letters 'A' through 'I' instead of the digits 1 through 9.

Chapter 15

Ordinary Differential Equations

Mathematical models in many different fields.

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

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

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

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

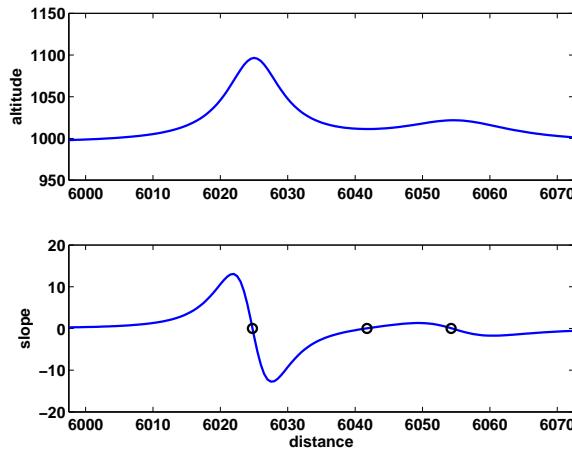


Figure 15.1. Altitude along a mountain road, and derivative of that altitude. The derivative is zero at the local maxima and minima of the altitude.

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

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

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

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

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

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

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

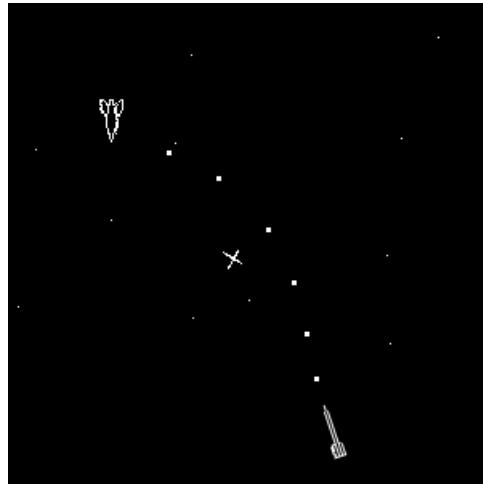


Figure 15.2. Spacewar, the world’s first video game. The gravitational pull of the central star causes the torpedo to move in an elliptical orbit.

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

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

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

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

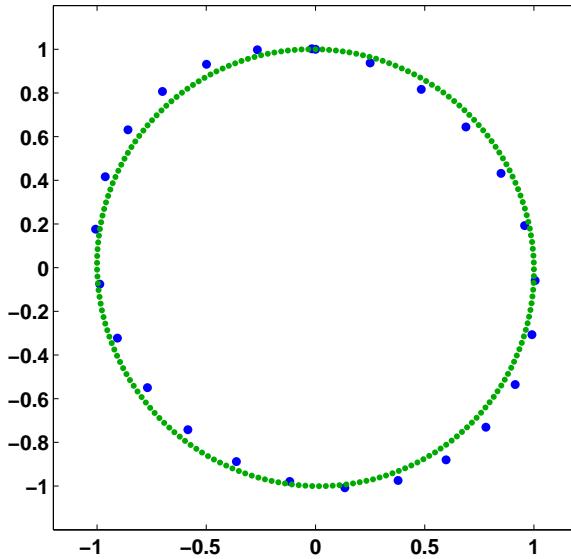


Figure 15.3. The 25 blue points are generated by the Spacewar orbit generator with a step size of $1/4$. The 201 green points are generated with a step size of $1/32$.

```

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

```

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

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

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

This can be rewritten as

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

You have probably noticed that the right hand side of this pair of equations involves

two different values of the time variable, t and $t + h$. That fact turns out to be important, but let's ignore it for now.

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

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

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

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

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

The initial values of x and y provide the initial conditions

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

The exact solution to the system is

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

To see why, recall the trig identities

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

For small values of h ,

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

Consequently

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

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

It turns out that the solution computed by the Spacewar orbit generator with a fixed step size h is an ellipse, not an exact circle. Decreasing h and taking more

steps generates a better approximation to a circle. Actually, the fact that $x(t + h)$ is used instead of $x(t)$ in the second half of the step means that the method is not quite as simple as it might seem. This subtle change is responsible for the fact that the method generates ellipses instead of spirals. One of the exercises asks you to verify this fact experimentally.

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

In MATLAB a system of odes takes the form

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

The function F always takes two arguments, the scalar independent variable, t , and the vector of dependent variables, y . A program that evaluates $F(t, y)$ should 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 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,

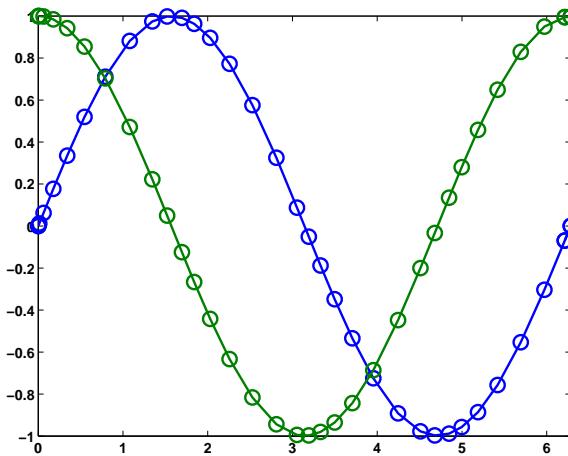


Figure 15.4. Graphs of sine and cosine generated by *ode23*.

- y_0 , 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 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')
```

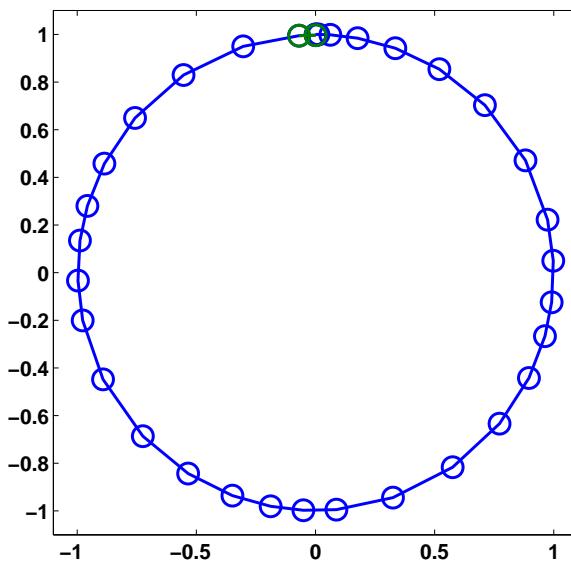


Figure 15.5. Graph of a circle generated by `ode23`.

```
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(@circle,tspan,y0)
```

is

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

We can see that the final value of `t` is 2π , 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,

$$\dot{\mathbf{y}} = A\mathbf{y}$$

where \mathbf{y} is a vector-valued function of time,

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

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

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

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

$$0 \leq t \leq T$$

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.

$$\dot{y} = k\left(1 - \frac{y}{\mu}\right)y$$

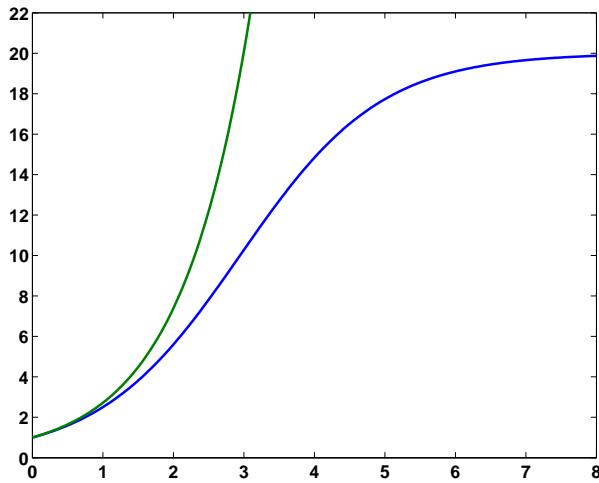


Figure 16.1. Exponential growth and logistic growth.

The new parameter μ is the *carrying capacity*. As $y(t)$ approaches μ 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 η and that as t approaches infinity, $y(t)$ approaches μ . 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 η 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.

$$\begin{aligned}\dot{y}_1 &= \left(1 - \frac{y_2}{\mu_2}\right)y_1 \\ \dot{y}_2 &= -\left(1 - \frac{y_1}{\mu_1}\right)y_2\end{aligned}$$

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 \mathbf{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, trigonometric, or any other elementary functions. It is necessary, but easy, to compute numerical solutions.

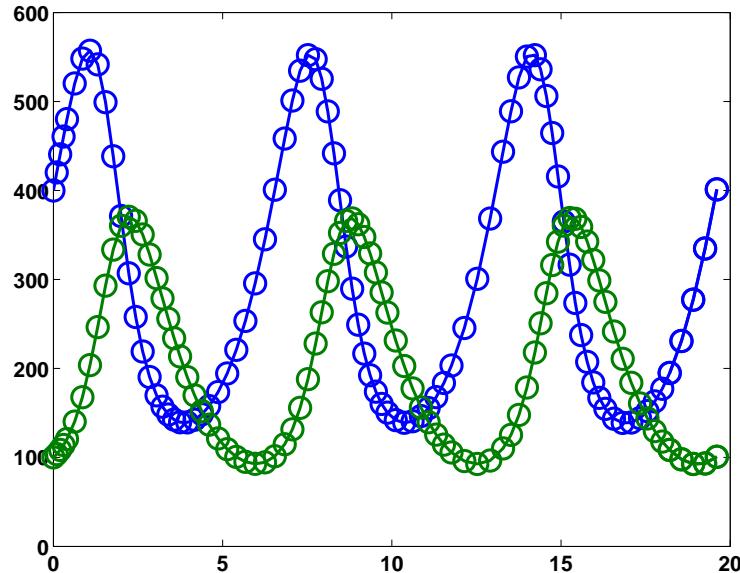


Figure 16.2. A typical solution of the predator-prey equations.

There are four parameters, the two constants μ_1 and μ_2 , and the two initial conditions,

$$\begin{aligned}\eta_1 &= y_1(0) \\ \eta_2 &= y_2(0)\end{aligned}$$

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 (μ_1, μ_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, than the curves near the maxima. The plot of $\sin t$ would look the same at the top as the bottom.

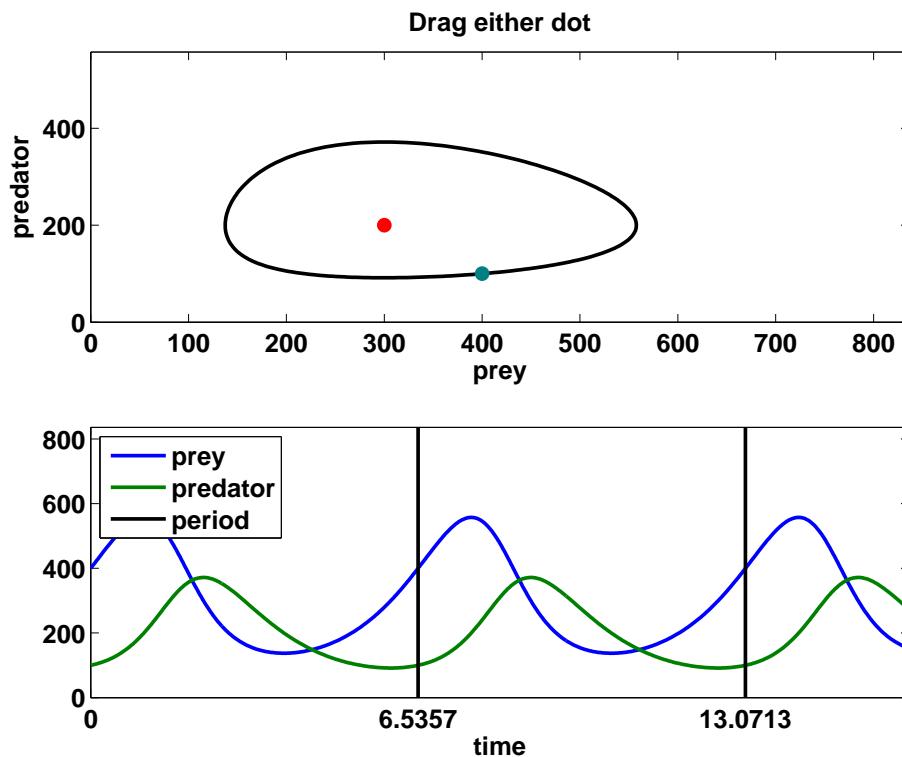


Figure 16.3. The predprey experiment.

Our MATLAB program `exm/predprey` shows a red dot at the equilibrium point, (μ_1, μ_2) , and a blue-green dot at the initial point, (η_1, η_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 (η_1, η_2) are close to the equilibrium point (μ_1, μ_2) , then the length of the period is close to a familiar 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 Predator 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 , η , and μ .

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 familiar 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

$$\begin{aligned}\dot{y}_1 &= \left(1 - \frac{y_1}{2\eta_1}\right)\left(1 - \frac{y_2}{\mu_2}\right)y_1 \\ \dot{y}_2 &= -(1 - \frac{y_1}{\mu_1})y_2\end{aligned}$$

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 δ 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 z_0 , 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, v_0 .

All the computation in `bouncer` is done within a doubly nested `while` loop. The outer loop involves the initial velocity v_0 .

```
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

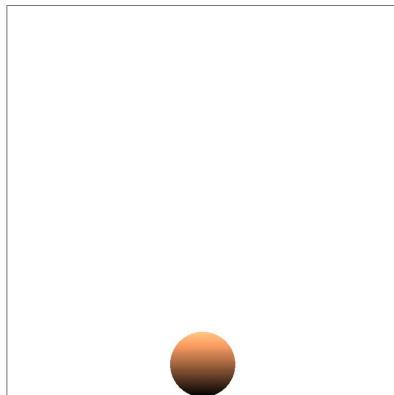


Figure 17.1. Initial, and final, position of a bouncing ball. To see what happens in between, run bouncer.

```
v = v0;
z = z0;
```

initialize the velocity v to v_0 and the height z to z_0 . 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, $\text{delta}*\text{g}$, 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 particles 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.

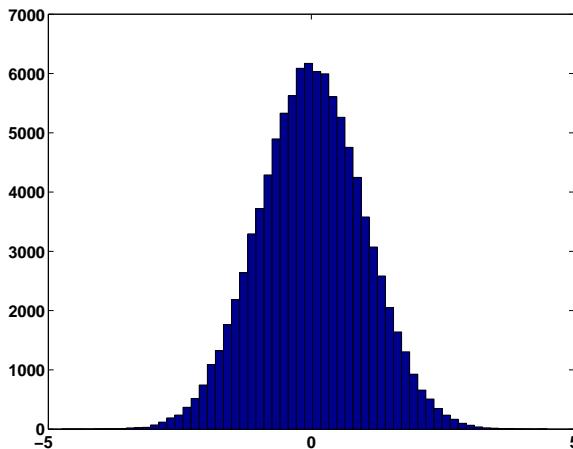


Figure 17.2. Histogram of the normal random number generator.

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]);
```

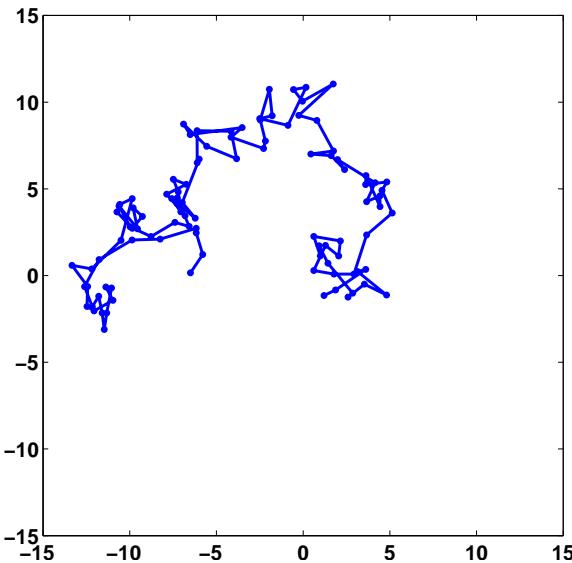


Figure 17.3. A simple example of Brownian motion.

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 $r = \text{randn}(m,1)$.

$$\begin{aligned}x_1 &= r_1 \\x_2 &= r_1 + r_2 \\x_3 &= r_1 + r_2 + r_3 \\&\dots\end{aligned}$$

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`.

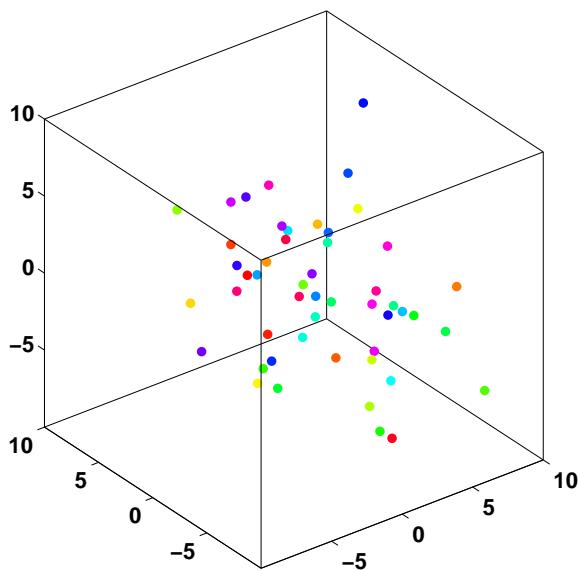


Figure 17.4. A snapshot of the output from `brownian3`, showing simultaneous random walks of many particles in three dimensions.

```

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 $i = 1, \dots, n$, 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,

$$\dot{p}_2 = (0, 0.72)$$

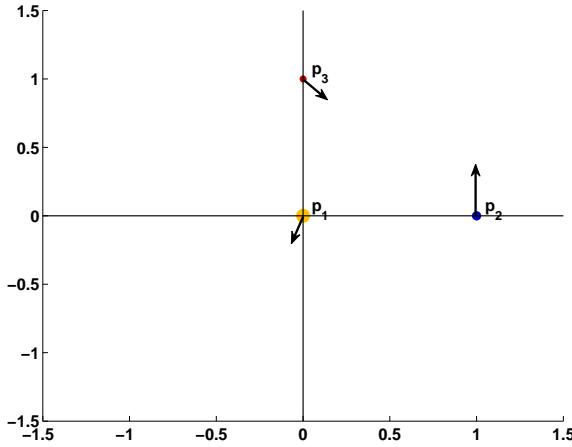


Figure 17.5. Initial positions and velocities of a small system with three bodies in two-dimensional space.

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

$$m_i \ddot{p}_i = \gamma \sum_{j \neq i} m_i m_j \frac{p_j - p_i}{\|p_j - p_i\|^3}, \quad i = 1, \dots, n$$

Here γ 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

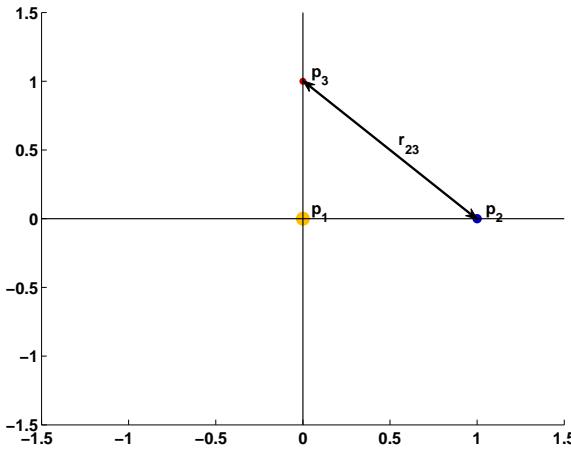


Figure 17.6. The double arrow depicts the vectors $r_{23} = p_3 - p_2$ and $-r_{32}$. The length of this arrow is the distance between p_2 and p_3 .

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

$$x(t) = \sin t, \quad y(t) = \cos t$$

The orbit is a perfect circle with a period equal to 2π .

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 δ 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.

$$\begin{aligned} P_{k+1} - \delta V_{k+1} &= P_k \\ V_{k+1} - \delta G(P_{k+1}) &= V_k \end{aligned}$$

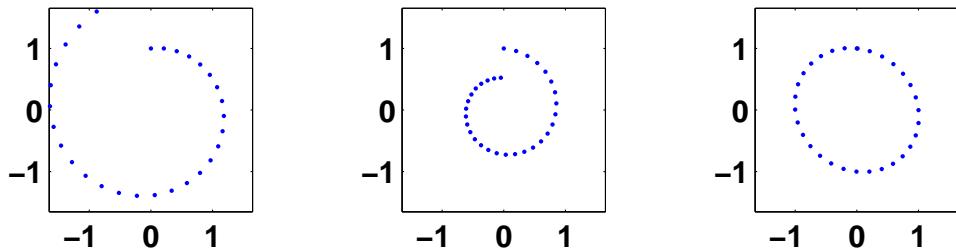


Figure 17.7. Three versions of Euler’s method for generating a circle. The first plot shows that the forward method is unstable. The second plot shows that the backward method has excessive damping. The third plot shows that symplectic method, which is a compromise between the first two methods, produces 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

$$\begin{aligned}x_{k+1} - \delta y_{k+1} &= x_k \\y_{k+1} + \delta x_{k+1} &= y_k\end{aligned}$$

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

$$\begin{aligned}x_{k+1} &= x_k + \delta y_k \\y_{k+1} &= y_k - \delta x_{k+1}\end{aligned}$$

The result is the third plot in figure 17.7. If you look carefully, you can see that the orbit is 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 2π . But the important fact is that the orbit is neither a growing nor a decaying spiral.

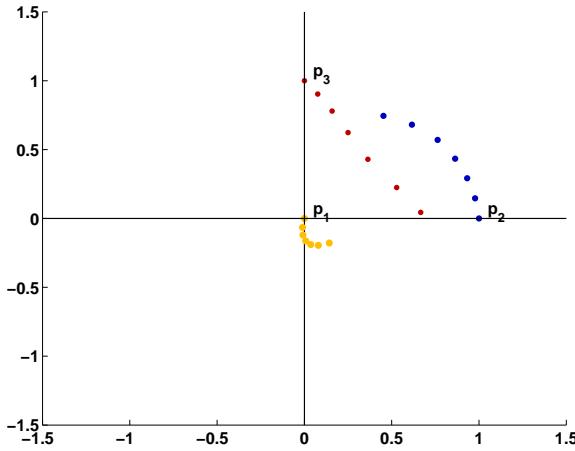


Figure 17.8. The first few steps of our example system.

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

$$\begin{aligned} P = \\ \begin{matrix} 0 & 0 \\ 1.0000 & 0 \\ 0 & 1.0000 \end{matrix} \end{aligned}$$

$$\begin{aligned} V = \\ \begin{matrix} -0.1200 & -0.3600 \\ 0 & 0.7200 \\ 0.3600 & -0.3600 \end{matrix} \end{aligned}$$

After one step with $\delta = 0.20$ we obtain the following values.

$$\begin{aligned} P = \\ \begin{matrix} -0.0107 & -0.0653 \\ 0.9776 & 0.1464 \\ 0.0767 & 0.9033 \end{matrix} \end{aligned}$$

$$\begin{aligned} V = \\ \begin{matrix} -0.0533 & -0.3267 \\ -0.1118 & 0.7318 \end{matrix} \end{aligned}$$

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 0.72δ . 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

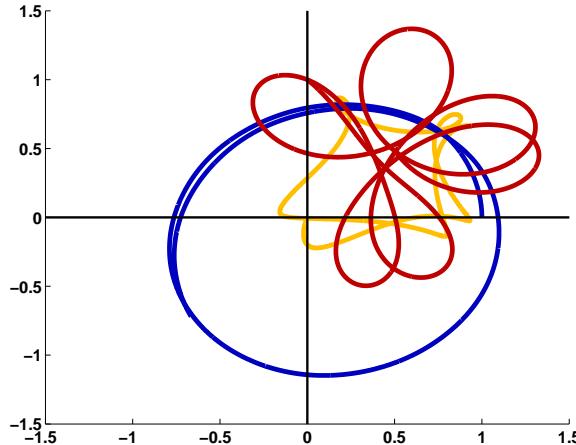


Figure 17.9. The initial trajectories of our example system.

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.

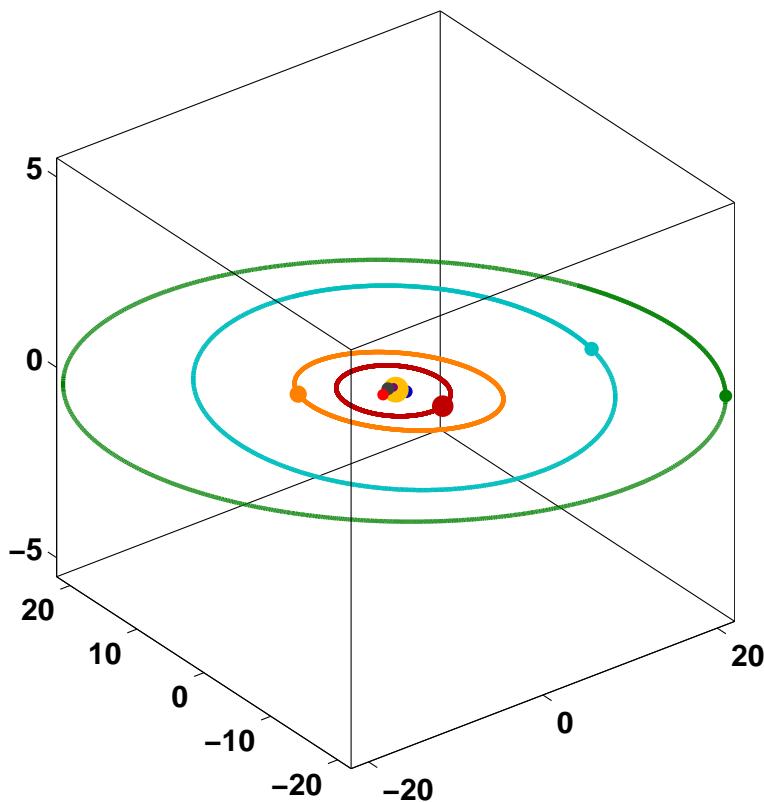


Figure 17.10. The solar system, with the initial positions of all the planets and the orbits of the outer planets, Jupiter, Saturn, Uranus, and Neptune.

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>

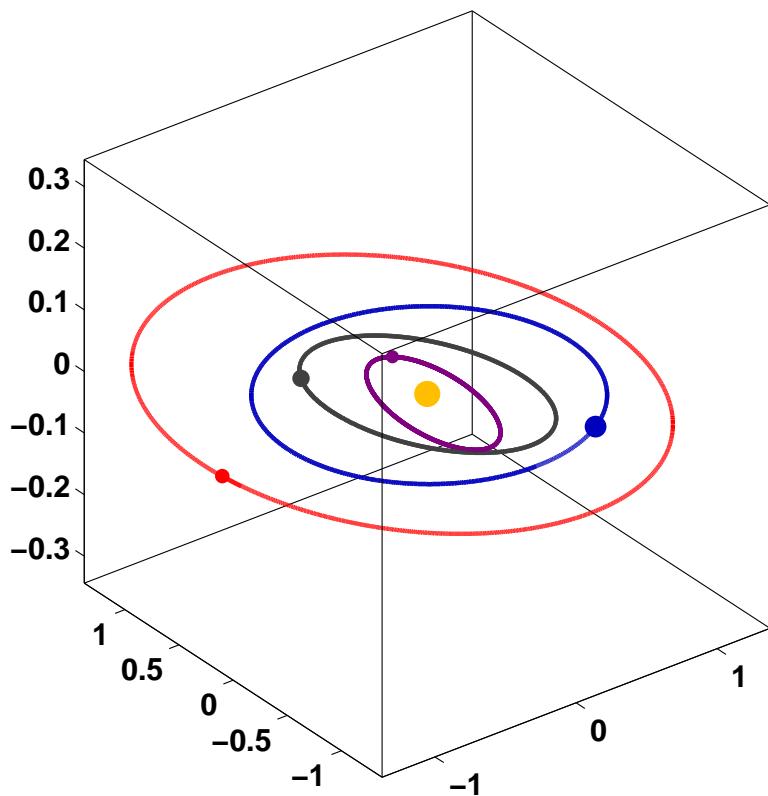


Figure 17.11. Zooming in by a factor of 16 reveals the orbits of the inner planets, Mercury, Venus, Earth and Mars.

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_\(dwarf_planet\)](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.

Chapter 18

Shallow Water Equations

The shallow water equations model tsunamis and waves in bathtubs.

This chapter is more advanced mathematically than earlier chapters, but you might still find it interesting even if you do not master the mathematical details.

The shallow water equations model the propagation of disturbances in water and other incompressible fluids. The underlying assumption is that the depth of the fluid is small compared to the wave length of the disturbance. For example, we do not ordinary think of the Indian Ocean as being shallow. The depth is two or three kilometers. But the devastating tsunami in the Indian Ocean on December 26, 2004 involved waves that were dozens or hundred of kilometers long. So the shallow water approximation provides a reasonable model in this situation.

The equations are derived from the principles of conservation of mass and conservation of momentum. The independent variables are time, t , and two space coordinates, x and y . The dependent variables are the fluid height or depth, h , and the two-dimensional fluid velocity field, u and v . With the proper choice of units, the conserved quantities are mass, which is proportional to h , and momentum, which is proportional to uh and vh . The force acting on the fluid is gravity, represented by the gravitational constant, g . The partial differential equations are:

$$\begin{aligned}\frac{\partial h}{\partial t} + \frac{\partial(uh)}{\partial x} + \frac{\partial(vh)}{\partial y} &= 0 \\ \frac{\partial(uh)}{\partial t} + \frac{\partial(u^2h + \frac{1}{2}gh^2)}{\partial x} + \frac{\partial(uvh)}{\partial y} &= 0 \\ \frac{\partial(vh)}{\partial t} + \frac{\partial(uvh)}{\partial x} + \frac{\partial(v^2h + \frac{1}{2}gh^2)}{\partial x} &= 0\end{aligned}$$

Copyright © 2011 Cleve Moler
MATLAB® is a registered trademark of MathWorks, Inc.™
October 4, 2011

In order to write the equations in a compact form, introduce three vectors.

$$U = \begin{pmatrix} h \\ uh \\ vh \end{pmatrix}$$

$$F(U) = \begin{pmatrix} h \\ u^2h + \frac{1}{2}gh^2 \\ uvh \end{pmatrix}$$

$$G(U) = \begin{pmatrix} vh \\ uvh \\ v^2h + \frac{1}{2}gh^2 \end{pmatrix}$$

With this notation, the shallow water equations are an instance of a *hyperbolic conservation law*.

$$\frac{\partial U}{\partial t} + \frac{\partial F(U)}{\partial x} + \frac{\partial G(U)}{\partial y} = 0$$

One delicate aspect of this model is the boundary conditions, especially if we intend to model a real world geometry such as the Indian Ocean. For our simple experiment, we confine ourselves to a square region and specify *reflective* boundary conditions, $u = 0$ on the vertical sides of the square and $v = 0$ on the horizontal sides. These conditions cause any waves that reach the boundary to be reflected back into the region.

More realistic models of oceans and tsunamis include terms that describe the topography of the ocean floor, the Coriolis force resulting from the earth's rotation, and possibly other external forces. But the equations we are considering here are still the basis of such models.

•	•	•	•
•	•	•	•
•	•	•	•
•	•	•	•

Figure 18.1. At the beginning of a time step, the variables represent the solution at the centers of the finite difference grid.

We will use the *Lax-Wendroff* method to compute a numerical approximation to the solution. Introduce a regular square finite difference grid with a vector-valued solution centered in the grid cells, as shown in figure 18.1. The quantity

$$U_{i,j}^n$$

represents a three component vector at each grid cell i, j that evolves with time step n .

Each time step involves two stages, something like a two-stage Runge Kutta method for ordinary differential equations. The first stage is a half step; it defines values of U at time step $n + \frac{1}{2}$ and the midpoints of the edges of the grid, as shown in figure 18.2.

$$U_{i+\frac{1}{2},j}^{n+\frac{1}{2}} = \frac{1}{2}(U_{i+1,j}^n + U_{i,j}^n) - \frac{\Delta t}{2\Delta x}(F_{i+1,j}^n - F_{i,j}^n)$$

$$U_{i,j+\frac{1}{2}}^{n+\frac{1}{2}} = \frac{1}{2}(U_{i,j+1}^n + U_{i,j}^n) - \frac{\Delta t}{2\Delta y}(G_{i,j+1}^n - G_{i,j}^n)$$

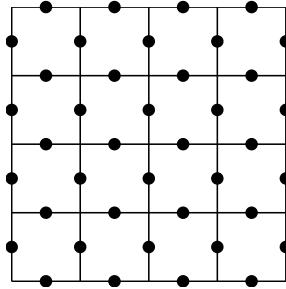


Figure 18.2. The first stage computes values that represent the solution at the midpoints of the edges in the finite difference grid.

The second stage completes the time step by using the values computed in the first stage to compute new values at the centers of the cells, returning to figure 18.1.

$$U_{i,j}^{n+1} = U_{i,j}^n - \frac{\Delta t}{\Delta x}(F_{i+\frac{1}{2},j}^{n+\frac{1}{2}} - F_{i-\frac{1}{2},j}^{n+\frac{1}{2}}) - \frac{\Delta t}{\Delta y}(G_{i,j+\frac{1}{2}}^{n+\frac{1}{2}} - G_{i,j-\frac{1}{2}}^{n+\frac{1}{2}})$$

Our MATLAB program, `exm/waterwave`, uses Lax-Wendroff to solve the shallow water equations on a square region with reflective boundary conditions. Initially, $h = 1, u = 0, v = 0$ over the entire region, so the solution is static. Then, at repeated intervals, a two dimensional Gaussian shaped peak is added to h , simulating an impulsive disturbance like a water drop hitting the surface. The resulting waves propagate back and forth over the region. A few snapshots of the dynamic graphic are shown in figure 18.3. The Lax-Wendroff scheme amplifies artificial, nonphysical oscillations. Eventually the numerical values overflow, producing floating point `Infs` and `Nans`, which cause the surface plot to disappear.

CLAWPACK, which stands for Conservation Law Package, is a large collection of Fortran subroutines developed by Randy LeVeque and his colleagues at the University of Washington. A version of the package specialized to modeling tsunamis has been developed by David George. See:

<http://www.clawpack.org>

<http://www.amath.washington.edu/~dgeorge/tsunamimodeling.html>

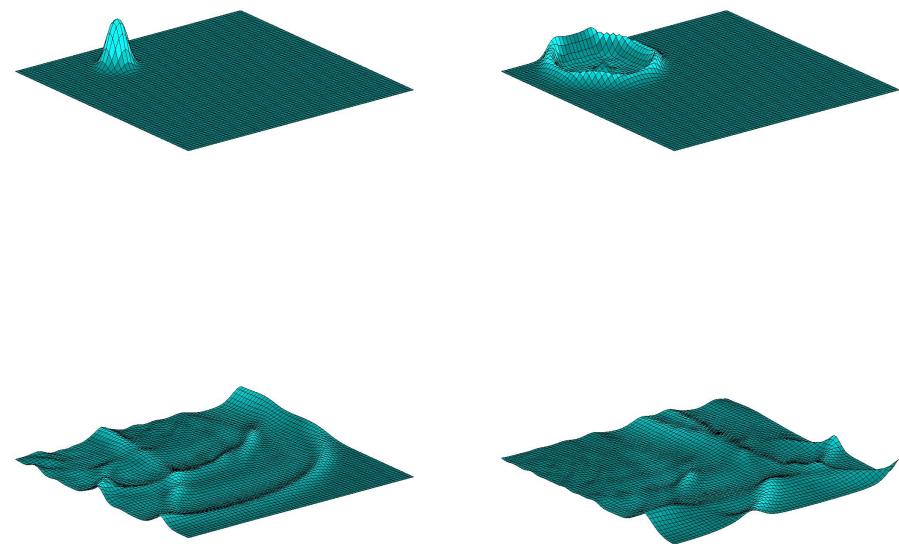


Figure 18.3. A water drop initiates a wave that reflects off the boundary.

Recap

```
%> Shallow Water Chapter Recap  
%> This is an executable program that illustrates the statements  
%> introduced in the Shallow Water Chapter of "Experiments in MATLAB".  
%> You can access it with  
%>  
%>     water_recap  
%>     edit water_recap  
%>     publish water_recap  
%>  
%> Related EXM programs  
%>  
%>     waterwave  
  
%> Finite Differences  
%> A simple example of the grid operations in waterwave.
```

```
%% Create a two dimensional grid.

m = 21;
[x,y] = ndgrid(-1: 2/(m-1): 1);

%% The water drop function from waterwave.

U = exp(-5*(x.^2+y.^2));

%% Surf plot of the function

clf
shg
h = surf(x,y,U);
axis off
ax = axis;

%% Colormap

c = (37:100)';
cyan = [0*c c c]/100;
colormap(cyan)
pause(1)

%% Indices in the four compass directions.

n = [2:m m];
e = n;
s = [1 1:m-1];
w = s;

%% A relation parameter. Try other values.
% Experiment with omega slightly greater than one.

omega = 1;

%% Relax.
% Repeatedly replace grid values by relaxed average of four neighbors.

tfinal = 500;
for t = 1:tfinal
    U = (1-omega)*U + omega*(U(n,:)+U(:,e)+U(s,:)+U(:,w))/4;
    set(h,'zdata',U);
    axis(ax)
    drawnow
end
```

Exercises

18.1 *Relax.* Modify `omega` in `water_recap`. Try values slightly larger than 1.0 .

18.2 *Boundary conditions.* Modify the boundary conditions used in `waterwave`.

Chapter 19

Morse Code

Morse code demonstrates binary trees and cell arrays.

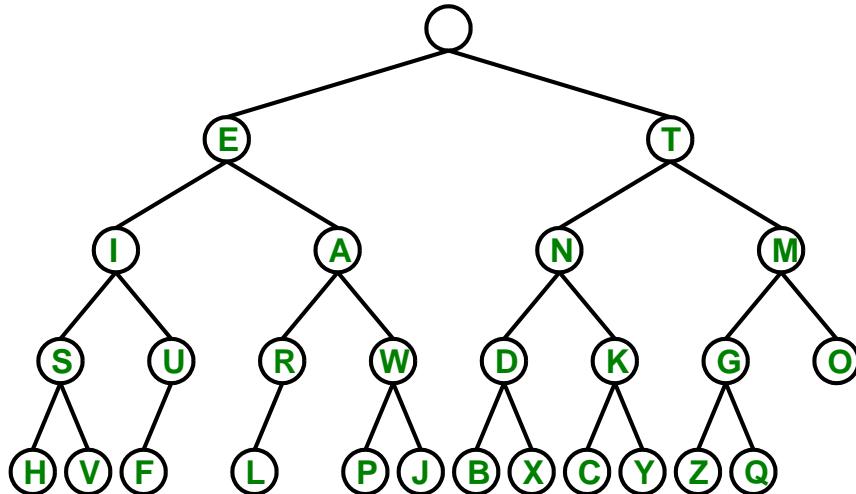


Figure 19.1. The binary tree defining Morse code. A branch to the left signifies a dot in the code and a branch to the right is a dash. In addition to the root node, there are 26 nodes containing the capital letters of the English alphabet.

This chapter brings together three disparate topics: Morse code, binary trees, and cell arrays. Morse code is no longer important commercially, but it still has

some avid fans among hobbyists. Binary trees are a fundamental data structure used throughout modern computing. And cell arrays, which are unique to MATLAB, are arrays whose elements are other arrays.

You can get a head start on our investigation by running the `exm` program

```
morse_gui
```

Experiment with the four buttons in various combinations, and the text and code box. This chapter will explain their operation.

Morse code was invented over 150 years ago, not by Samuel F. B. Morse, but by his colleague, Alfred Vail. It has been in widespread use ever since. The code consists of short *dots*, '.', and longer *dashes*, '—', separated by short and long spaces. You are certainly familiar with the international distress signal, '... —— ...', the code for "SOS", abbreviating "Save Our Ships" or perhaps "Save Our Souls". But did you notice that some modern cell phones signal '... -- ...', the code for "SMS", indicating activity of the "Short Message Service".

Until 2003, a license to operate an amateur radio required minimal proficiency in Morse code. (Full disclosure: When I was in junior high school, I learned Morse code to get my ham license, and I've never forgotten it.)

According to Wikipedia, in 2004, the International Telecommunication Union formally added a code for the ubiquitous email character, @, to the international Morse code standard. This was the first addition since World War I.

The Morse tree

We could provide a table showing that '.—' is the code for A, '—...—' the code for B, and so on. But we're not going to do that, and our MATLAB program does not start with such a table. Instead, we have figure 19.1. This is a *binary tree*, and for our purposes, it is the *definition* of Morse code. In contrast to nature, computer scientists put the root of a tree on top. Starting at this root, or any other node, and moving left along a link signifies a *dot*, while moving right is a *dash*. For example, starting at the root and moving one step to the left followed by one to the right gets us to A. So this fact, rather than a table, tells us '.—' is the code for A.

The length of a Morse code sequence for a particular character is determined by the frequency of that character in typical English text. The most frequent character is "E". Consequently, its Morse sequence is a single dot and it is linked directly to the root of our tree. The least frequent characters, such as "Z" and "X", have the longest Morse sequences and are far from the root. (We will consider the four missing nodes in the tree later.)

Binary trees are best implemented in MATLAB by *cell arrays*, which are arrays whose elements are themselves other MATLAB objects, including other arrays. Cell arrays have two kinds of indexing operations. Curly braces, { and }, are used for construction and for accessing individual cells. Conventional smooth parentheses, (and), are used for accessing subarrays. For example,

```
C = {'A','rolling','stone','gathers','momentum','.'}
```

produces a cell array that contains six strings of different lengths. This example is displayed as

```
C =
'A' 'rolling' 'stone' 'gathers' 'momentum' '..'
```

The third *element*, denoted with curly braces by $C\{3\}$, is the string '**stone**'. The third *subarray*, denoted with parentheses by $C(3)$, is another cell array containing a single element, the string '**stone**'. Now go back and read those last two sentences a few more times. The subtle distinction between them is both the key to the power of cell arrays and the source of pervasive confusion.

Think of a string of mailboxes along C street. Assume they are numbered consecutively. Then $C(3)$ is the third mailbox and $C\{3\}$ is the mail in that box. By itself, C is the entire array of mailboxes. The expression $C(1:3)$ is the subarray containing the first three mailboxes. And here is an unusual construction, with curly braces $C\{1:3\}$ is a *comma separated list*,

```
C\{1\}, C\{2\}, C\{3\}
```

By itself, on the command line, this will do three assignment statements, assigning the contents of each of the first three mailboxes, one at a time, to **ans**. With more curly braces, $\{C\{1:3\}\}$ is the same as $C(1:3)$.

In the computer hardware itself, there is a distinction between a memory location with a particular address and the contents of that location. This same distinction is preserved by indexing a cell array with parentheses and with braces.

Did you see the "Men in Black" movies? In one of them the MIB headquarters has a bank of storage lockers. It turns out that each locker contains an entire civilization, presumably with its own lockers. At the end of the movie it is revealed that the Earth itself is a storage locker in a larger civilization. It is possible that we are all living in one element of a huge cell array.

The binary tree defining Morse code is a cell array whose contents are characters and other cell arrays. Each cell represents a node in the tree. A cell, N , has three elements, The first element, $N\{1\}$, is a string with a single capital letter, X , designating the node. The second element, $N\{2\}$, is another cell array, the *dot* branch. The third element, $N\{3\}$, is the *dash* branch. There are a couple of exceptional cases. The root node does not have an associated letter, so its first element is an empty string. The U and R nodes, and the leaf nodes, have one or two empty cell arrays for branches.

In principle, we could create the entire Morse binary tree with a single gigantic, but unrealistic, assignment statement.

```
M = {'' ...
{'E' ...
{'I' {'S' {'H' {} {}} {'V' {} {}}} ...
{'U' {'F' {} {}} {}} ...
{'A' {'R' {'L' {} {}} {}} ...
{'W' {'P' {} {}} {'J' {} {}}} ...
{'T' ...}}
```

```
{'N' {'D' {'B' {} {}} {'X' {} {}}}} ...
{'K' {'C' {} {}} {'Y' {} {}}}} ...
{'M' {'G' {'Z' {} {}} {'Q' {} {}}}} ...
{'O' {} {}}}}}}
```

You can see the cell arrays within cell arrays and the many empty cell arrays at the leaves. This statement actually works, but it is unreasonable because it is error prone and nearly impossible to extend. Instead, our function `morse_tree` begins with a header

```
function M = morse_tree
```

This is followed by 27 assignment statements, twelve at level four.

```
h = {'H' {} {}};
v = {'V' {} {}};
f = {'F' {} {}};
l = {'L' {} {}};
p = {'P' {} {}};
j = {'J' {} {}};
b = {'B' {} {}};
x = {'X' {} {}};
c = {'C' {} {}};
y = {'Y' {} {}};
z = {'Z' {} {}};
q = {'Q' {} {}};
```

Eight at level three.

```
s = {'S' h v};
u = {'U' f {}};
r = {'R' l {}};
w = {'W' p j};
d = {'D' b x};
k = {'K' c y};
g = {'G' z q};
o = {'O' {} {}};
```

Four at level two.

```
i = {'I' s u};
a = {'A' r w};
n = {'N' d k};
m = {'M' g o};
```

Two at level one.

```
e = {'E' i a};
t = {'T' n m};
```

And finally one assignment statement at level zero to create the root node.

```
M = {'e': t};
```

This function is at the heart of all our Morse code software.

You can travel down this tree by first entering three commands.

```
M = morse_tree
M = M{2}
M = M{3}
```

You then can use the up-arrow on your keyboard to repeatedly select and reexecute these commands. Returning to the first command gives you a fresh tree. Executing the second command is a *dot* operation, moving down the tree to the left. Executing the third command is a *dash* operation, moving down the tree to the right. For example, the five commands

```
M = morse_tree
M = M{3}
M = M{2}
M = M{2}
M = M{3}
```

correspond to the Morse sequence '-...'. This brings you to the node

```
'X' {} {}
```

You have reached the X leaf of the tree.

Searching the tree

Returning to the “Men in Black” analogy, the Morse binary tree is a single locker. When you open that locker, you see an empty string (because the root does not have a name) and two more lockers. When you open the locker on the left, you see an ‘E’ and two more lockers. Now you have three choices. You can open either of the two lockers in the E locker, or you can go back to the root locker and open the locker on the right to visit the T locker.

Repeatedly choosing different lockers, or different branches in the tree, corresponds to traversing the tree in different orders. Among these many possible orders, two have standard names, “depth first search” and “breadth first search”. They are shown in figures 19.2 and 19.3 and you can see and hear animated versions with the *morse_gui* program.

Depth first search visits each branch as soon as it sees it. When it has visited both branches at a node, it backs up to the first node that has an available branch. Figure 19.2 shows the progress of depth first order up to node W.

```
E I S H V U F A R L W
```

Nothing to the right of W has yet been visited.

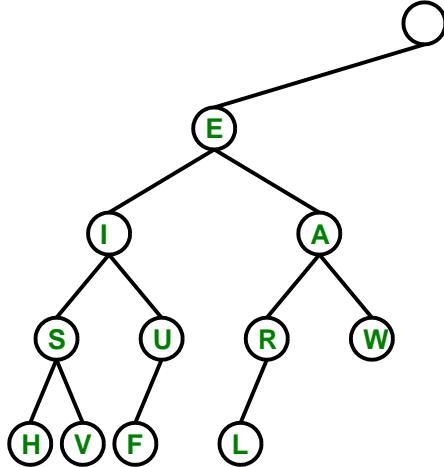


Figure 19.2. A depth first search in progress. The nodes are visited from left to right.

Breadth first search takes one step along each branch at a node before it continues. The result is a top to bottom search, much like reading English language text. Figure 19.3 shows the breadth first order up to node W.

E T I A N M S U R W

Nothing below W has yet been visited.

Depth first search uses a data structure known as a *stack*. Here is a code segment with a stack that simply displays the nodes of the tree in depth first order.

```

S = {morse_tree};
while ~isempty(S)
    N = S{1};
    S = S(2:end);
    if ~isempty(N)
        fprintf(' %s',N{1})
        S = {N{2} N{3}} S{1:1};
    end
end
fprintf('\n')
  
```

The stack **S** is a cell array. Initially, it has one cell containing the tree. The **while**

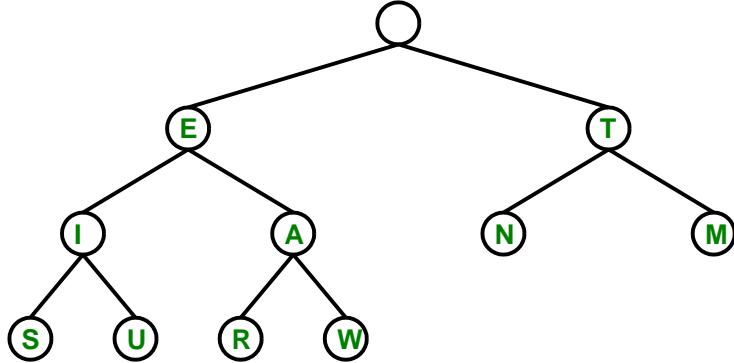


Figure 19.3. A breadth first search in progress. The nodes are visited from top to bottom.

loop continues as long as the stack is not empty. Within the body of the loop a node is removed from the top of the stack and the stack shortened by one element. If the node is not empty, the single character representing the node is displayed and two new nodes, the *dot* and *dash* branches, are inserted into the top of the stack. The traversal visits recently discovered nodes before it returns to older nodes.

Breadth first search uses a data structure known as a *queue*. Here is another code segment, this time with a queue, that displays the nodes of the tree in breadth first order.

```

Q = {morse_tree};
while ~isempty(Q)
    N = Q{1};
    Q = Q(2:end);
    if ~isempty(N)
        fprintf(' %s',N{1})
        Q = {Q{:} N{2} N{3}};
    end
end
fprintf('\n')
  
```

This code is similar to the stack code. The distinguishing feature is that new nodes

are inserted at the end, rather than the beginning of the queue.

A queue is employing a “First In, First Out”, or FIFO, strategy, while a stack is employing a “Last In, First Out”, or LIFO, strategy. A queue is like a line at a grocery store or ticket office. Customers at the beginning of the line are served first and new arrivals wait at the end of the line. With a stack, new arrivals crowd in at the start of the line.

Our stack and queue codes are not recursive. They simply loop until there is no more work to be done. Here is different approach that employs recursion to do a depth first search. Actually, this does involve a hidden stack because computer systems such as MATLAB use stacks to manage recursion.

```
function traverse(M)
    if nargin == 0
        M = morse_tree;      % Initial entry.
    end
    if ~isempty(M)
        disp(M{1})
        traverse(M{2})      % Recursive calls.
        traverse(M{3})
    end
end % traverse
```

Decode and encode

Decoding is the process of translating dots and dashes into text. Encoding is the reverse. With our binary tree, decoding is easier than encoding because the dots and dashes directly determine which links to follow. Here is a function that decodes one character’s worth of dots and dashes. The function returns an asterisk if the input does not correspond to one of the 26 letters in the tree.

```
function ch = decode(dd)
    M = morse_tree;
    for k = 1:length(dd)
        if dd(k) == '.'
            M = M{2};
        elseif dd(k) == '-'
            M = M{3};
        end
        if isempty(M)
            ch = '*';
            return
        end
    end
    ch = M{1};
end % decode
```

Encoding is a little more work because we have to search the tree until we find the desired letter. Here is a function that employs depth first search to encode one character. A stack of dots and dashes is built up during the search. Again, an asterisk is returned if the input character is not in the tree.

```

function dd = encode(ch)
    S = {morse_tree};
    D = {''};
    while ~isempty(S)
        N = S{1};
        dd = D{1};
        S = S(2:end);
        D = D(2:end);
        if ~isempty(N)
            if N{1} == ch;
                return
            else
                S = {N{2} N{3} S{:}};
                D = {[dd '.'] [dd '-'] D{:}};
            end
        end
    end
    dd = '*';
end % encode

```

These two functions are the core of the decoding and encoding aspect of `morse_gui`.

Extension

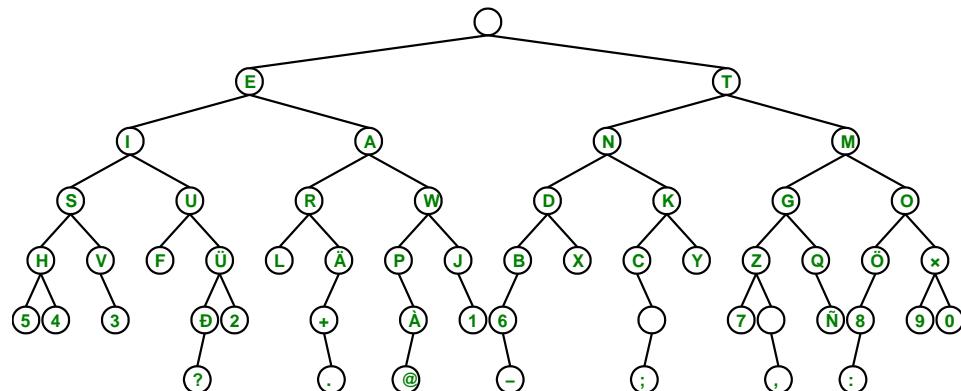


Figure 19.4. The Morse tree extended two levels to accommodate digits, punctuation, and non-English characters.

A binary tree with four levels has $2^5 - 1 = 31$ nodes, counting the root. Our `morse_tree` has a root and only 26 other nodes. So, there are four empty spots. You can see them on the dash branches of U and R and on both branches of O. So far, our definition of Morse code does not provide for the four sequences

```
..-- .-. ---. ----
```

How should these be decoded? We also want to add codes for digits and punctuation marks. And, it would be nice to provide for at least some of the non-English characters represented with umlauts and other diacritical marks.

Morse code was invented 100 years before modern computers. Today, alphabets, keyboards, character sets and fonts vary from country to country and even from computer to computer. Our function

```
morse_table_extended
```

extends the tree two more levels and adds 10 digits, 8 punctuation characters, and 7 non-English characters to the 26 characters in the original `morse_table`. Figure 19.4 shows the result. The `extend` button in `morse_gui` accesses this extension. Some of the issues involved in representing these additional characters are pursued in the exercises.

Morse code table

We promised that we would not use a table to define Morse code. Everything has been based on the binary tree. When an actual table is desired we can generate one from the tree. Our function is named `morse_code` and it employs the recursive algorithm `traverse` from the previous section. The recursion carries along the emerging table C and a growing string dd of dots and dashes.

Indexing into the table is based upon the ASCII code for characters. The ASCII standard is an 7-bit code that is the basis for computer representation of text. Seven bits provide for $2^7 = 128$ characters. The first 32 of these are nonprinting characters that were originally used for control of teletypes and are now largely obsolete. The remaining 96 are the 52 upper and lower case letters of the English alphabet, 10 digits, and 32 punctuation marks. In MATLAB the function `char` converts a numeric value to a character. For example

```
char(65)
ans =
'A'
```

The function `double` converts a character to a floating point number and the function `uint8` converts a character to an unsigned 8-bit integer. Either of these can be used as an index. For example

```
double('A')
ans =
65
```

A computer byte is 8 bits and so the ASCII standard is readily extended by another 128 characters. The actual graphic printed for some of these characters may vary from country to country and from font to font.

Our function `morse_code` produces a table of both ASCII and Morse code from either `morse_tree` or `morse_tree_extended`. The recursive `traverse` algorithm is used for the depth first search. ASCII codes are used as indices into a 256-element cell array of dots and dashes. The search inserts only 26 or 51 elements into the array, so a final step creates a printable table.

```
function C = morse_code(C,M,dd)
    % MORSE_CODE
    % C = morse_code
    % C = morse_code(morse_tree)
    % C = morse_code(morse_tree_extended)

    if nargin < 3           % Choose binary tree
        if nargin == 0
            M = morse_tree;
        else
            M = C;
        end
        C = cell(256,1);      % The temporary code table
        dd = '';              % dots and dashes
    end

    if ~isempty(M)          % Depth first search
        if ~isempty(M{1})
            C{double(M{1})} = dd;          % Use ASCII value as an index
        end
        C = morse_code(C,M{2},[dd '.']); % Recursive call
        C = morse_code(C,M{3},[dd '-']); % Recursive call
    end

    if nargin < 3           % Final processing, convert to char.
        c = char(C{:});
        k = find(c(:,1) ~= ' ');       % Find the nonblank entries.
        b = blanks(length(k))';
        C = [char(k) b b int2str(k) b b char(C{k})];
    end
```

The output from

```
morse_code
```

is the following 26 lines.

```
A   65  .-
B   66  -...
```

```
C    67  -.-.  
D    68  -..  
E    69  .  
F    70  ...-  
G    71  ---.  
H    72  ....  
I    73  ..  
J    74  .----  
K    75  --.  
L    76  .-..  
M    77  --  
N    78  -.  
O    79  ----  
P    80  .--.  
Q    81  ---.-  
R    82  .-.  
S    83  ...  
T    84  -  
U    85  ...  
V    86  ...-  
W    87  .--  
X    88  -..-  
Y    89  -.--  
Z    90  ---..
```

You should run

```
morse_code(morse_tree_extended)
```

to see what output is generated on your computer.

References

[1] Wikipedia article on Morse code. See the complete Morse code tree at the end of the article.

http://en.wikipedia.org/wiki/Morse_code

Recap

```
%% Morse Code Chapter Recap  
% This is an executable program that illustrates the statements  
% introduced in the Morse Code Chapter of "Experiments in MATLAB".  
% You can access it with  
%  
%     morse_recap  
%     edit morse_recap
```

```

%     publish morse_recap
%
% Related EXM programs
%
%     morse_gui
%     morse_tree
%     morse_tree_extended
%     morse_code

%% Cell Arrays

C = {'A','rolling','stone','gathers','momemtum','.'}
C{3}
C(3)
C(1:3)
C{1:3}
{C{1:3}}


%% Create a Morse Tree
% An absurd statement. For a better way, see morse_tree.

M = {'' ...
{'E' ...
{'I' {'S' {'H' {} {}} {'V' {} {}}} ...
{'U' {'F' {} {}} {}} ...
{'A' {'R' {'L' {} {}} {}} ...
{'W' {'P' {} {}} {'J' {} {}}} ...
{'T' ...
{'N' {'D' {'B' {} {}} {'X' {} {}}} ...
{'K' {'C' {} {}} {'Y' {} {}}} ...
{'M' {'G' {'Z' {} {}} {'Q' {} {}}} ...
{'O' {} {}}} ...

%% Follow '-...-
M = morse_tree
M = M{3}
M = M{2}
M = M{2}
M = M{3}

%% Depth first, with a stack.
S = {morse_tree};
while ~isempty(S)
    N = S{1};
    S = S(2:end);
    if ~isempty(N)

```

```

        fprintf(' %s',N{1})
        S = {N{2} N{3} S{:}};
    end
end
fprintf('\n')

%% Breadth first, with a queue.
Q = {morse_tree};
while ~isempty(Q)
    N = Q{1};
    Q = Q(2:end);
    if ~isempty(N)
        fprintf(' %s',N{1})
        Q = {Q{:} N{2} N{3}};
    end
end
fprintf('\n')

%% Recursive traversal.
% function traverse(M)
%     if nargin == 0
%         M = morse_tree;      % Initial entry.
%     end
%     if ~isempty(M)
%         disp(M{1})
%         traverse(M{2});    % Recursive calls.
%         traverse(M{3})
%     end
% end % traverse

%% ASCII character set
k = reshape([32:127 160:255],32,[]);
C = char(k)
txt = text(.25,.50,C,'interp','none');
set(txt,'fontname','Lucida Sans Typewriter')

```

Exercises

19.1 *Greetings.* What does this say?

.... . .-.. .-.. --- .-- --- .-. .-.. -..

19.2 *Email.* Use the `extend` button and translate box in `morse_gui` to translate your email address into Morse code.

19.3 *dash*. Why didn't I use an underscore, '_', instead of a minus sign, '−', to represent a dash?

19.4 *Note*. What musical note is used by the sound feature in `morse_gui`?

19.5 *Reverse order*. Find this statement in function `depth` in `morse_gui`.

```
S = {N{2} N{3} S{:}};
```

What happens if you interchange `N{2}` and `N{3}` like this?

```
S = {N{3} N{2} S{:}};
```

19.6 *Extend*. Using the tree at the end of the Wikipedia article, add more characters to `morse_code_extended`. If you live in a *locale* that has non-English characters in the alphabet, be sure to include them.

19.7 *Four dashes*. Why does the character with Morse code '----' cause a unique difficulty in `morse_code_extended`?

19.8 *YouTube*. Check out "Morse code" on YouTube. Be sure to listen to the "Morse code song". Who wins the Morse code versus texting contest on Jay Leno's Tonite show?

19.9 *Trinary*. What does this function do? Why is it named `trinary`. What determines how long it runs? What causes it to terminate? Why is the `if` statement necessary? Modify the program to make it use a depth first search. Modify the program to make it work with `morse_tree_extended`.

```
function trinary
    T = [0 0 0 0];
    Q = {morse_tree};
    while any(T(1,:) < 2)
        p = T(1,:);
        y = polyval(p,3);
        if ~isempty(Q{1})
            fprintf('%s %d%d%d%d %2d\n',Q{1}{1},p,y)
            Q = {Q{2:end} Q{1}{2} Q{1}{3}};
        else
            Q = {Q{2:end} {} {}};
        end
        T = [T(2:end,:); [T(1,2:end) 1]; [T(1,2:end) 2]];
    end
end % trinary
```

19.10 *Cell arrays*. Let

```
C = {'A' 'rolling' 'stone' 'gathers' 'momentum' '.'}
```

Explain why each of the following does what it does.

```
C'
char(C)
size(C)
size(char(C))
double(char(C))
upper(C)
C(end)
C{end}
C{3}(3)
fliplr(C)
[C{:}]
C(5:7) = {'no' 'moss' '.'}
```

19.11 *F*onts. Experiment with various fonts. See the Wikipedia article on the ASCII character set. You can generate a printable character table in MATLAB with

```
k = reshape([32:127 160:255],32,[]);
C = char(k)
```

The first half of this table is standard, but the second half depends upon the fonts you are using. You can change fonts in the command window by accessing the “File” menu, then selecting “Preferences” and “Fonts”. Highlight the desktop code font and use the up and down arrow keys. In my opinion the best font for the MATLAB command window is

Lucida Sans Typewriter

You can also display the character table in the figure window with

```
txt = text(.25,.50,C,'interp','none');
```

To change display fonts in the figure window, try commands like these work on your computer.

```
set(txt,'fontname','Lucida Sans Typewriter')
set(txt,'fontname','Courier New')
set(txt,'fontname','Comic Sans MS')
set(txt,'fontname','Wingdings')
set(txt,'fontname','GiGi')
```

Use the font names available under the command window preferences.

Chapter 20

Music

What does $\sqrt[12]{2}$ have to do with music?

In the theory of music, an *octave* is an interval with frequencies that range over a factor of two. In most Western music, an octave is divided into twelve *semitones* with equal frequency ratios. Since twelve semitones comprise a factor of two, one semitone is a factor of $\sqrt[12]{2}$. And because this quantity occurs so often in this chapter, let

$$\sigma = \sqrt[12]{2}$$

Our MATLAB programs use

```
sigma = 2^(1/12)
      = 1.059463094359295
```

Think of σ as an important mathematical constant, like π and ϕ .

Keyboard

Figure 20.1 shows our miniature piano keyboard with 25 keys. This keyboard has two octaves, with white keys labeled C D ... G A B, plus another C key. Counting both white and black, there are twelve keys in each octave. The frequency of each key is a semitone above and below its neighbors. Each black key can be regarded as either the *sharp* of the white below it or the *flat* of the white above it. So the black key between C and D is both C \sharp and D \flat . There is no E \sharp /F \flat or B \sharp /C \flat .

A conventional full piano keyboard has 88 keys. Seven complete octaves account for $7 \times 12 = 84$ keys. There are three additional keys at the lower left and one additional key at the upper end. If the octaves are numbered 0 through 8, then

Copyright © 2011 Cleve Moler
MATLAB® is a registered trademark of MathWorks, Inc.™
October 4, 2011

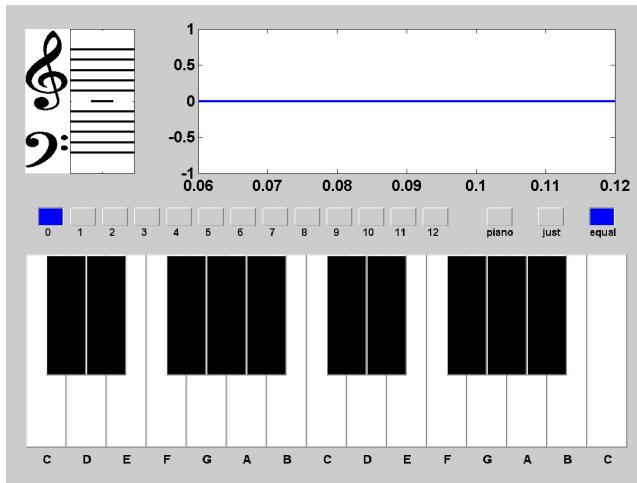


Figure 20.1. Miniature piano keyboard.

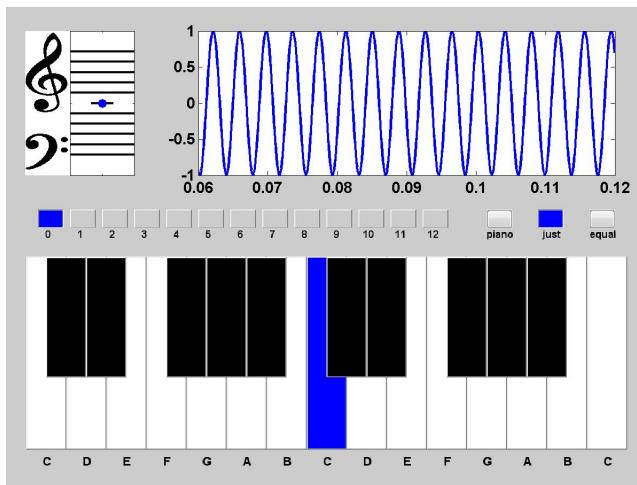


Figure 20.2. Middle C.

a key letter followed by an octave number specifies a unique key. In this notation, two important keys are C4 and A4. The C4 key is near the center of the keyboard and so is also known as *middle C*. A piano is usually tuned so that the frequency of the A4 key is 440 Hz. C4 is nine keys to the left of A4 so its frequency is

$$C4 = 440\sigma^{-9} \approx 261.6256 \text{ Hz}$$

Our EXM program `pianoex` uses C4 as the center of its 25 keys, so the number

range is `-12:12`. The statement

```
pianoex(0)
```

generates and plays the sound from a sine wave with frequency `C4`. The resulting visual display is shown in figure 20.2.

This `for` loop plays a two octave chromatic scale starting covering all 25 notes on our miniature keyboard.

```
for n = -12:12
    pianoex(n)
end
```

Do Re Mi

One of the first songs you learned to sing was

Do Re Mi Fa So La Ti Do

If you start at `C4`, you would be singing the *major scale* in the key of C. This scale is played on a piano using only the white keys. The notes are not equally spaced. Most of the steps skip over black keys and so are two semitones. But the steps between *Mi* and *Fa* and *Ti* and *Do* are the steps from E to F and B to C. There are no intervening black keys and so these steps are only one semitone. In terms of σ , the C-major scale is

$$\sigma^0 \sigma^2 \sigma^4 \sigma^5 \sigma^7 \sigma^9 \sigma^{11} \sigma^{12}$$

You can play this scale on our miniature keyboard with

```
for n = [0 2 4 5 7 9 11 12]
    pianoex(n)
end
```

The number of semitones between the notes is given by the vector

```
diff([0 2 4 5 7 9 11 12])
= [2 2 1 2 2 2 1]
```

The sequence of frequencies in our most common scale is surprising. Why are there 8 notes in the C-major scale? Why don't the notes in the scale have uniform frequency ratios? For that matter, why is the octave divided into 12 semitones? The notes in "*Do Re Me*" are so familiar that we don't even ask ourselves these questions. Are there mathematical explanations? I don't have definitive answers, but I can get some hints by looking at harmony, chords, and the ratios of small integers.

Vibrations and modes

Musical instruments create sound through the action of vibrating strings or vibrating columns of air that, in turn, produce vibrations in the body of the instrument

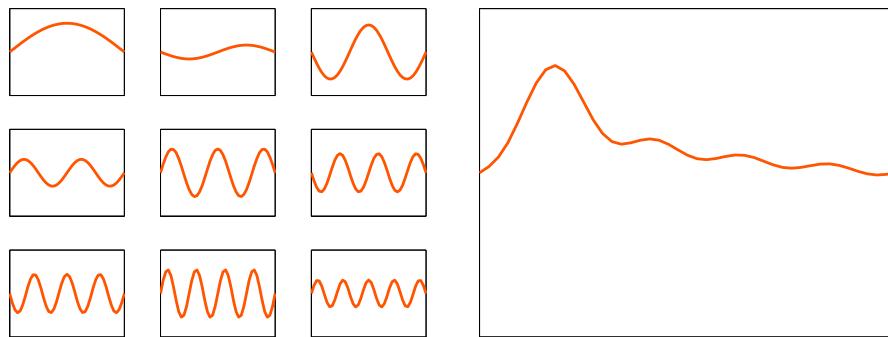


Figure 20.3. The first nine modes of a vibrating string, and their weighted sum.

and the surrounding air. Mathematically, vibrations can be modeled by weighted sums of characteristic functions known as *modes* or *eigenfunctions*. Different modes vibrate at different characteristic frequencies or *eigenvalues*. These frequencies are determined by physical parameters such as the length, thickness and tension in a string, or the geometry of the air cavity. Short, thin, tightly stretched strings have high frequencies, while long, thick, loosely stretched strings have low frequencies.

The simplest model is a one-dimensional vibrating string, held fixed at its ends. The units of the various physical parameters can be chosen so that the length of the string is 2π . The modes are then simply the functions

$$v_k(x) = \sin kx, \quad k = 1, 2, \dots$$

Each of these functions satisfy the fixed end point conditions

$$v_k(0) = v_k(2\pi) = 0$$

The time-dependent modal vibrations are

$$u_k(x, t) = \sin kx \sin kt, \quad k = 1, 2, \dots$$

and the frequency is simply the integer k . (Two- and three-dimensional models are much more complicated, but this one-dimensional model is all we need here.)

Our EXM program `vibrating_string` provides a dynamic view. Figure 20.3 is a snapshot showing the first nine modes and the resulting wave traveling along the string. An exercise asks you to change the coefficients in the weighted sum to produce different waves.

Lissajous figures

Lissajous figures provide some insight into the mathematical behavior of musical chords. Two dimensional Lissajous figures are plots of the parametric curves

$$x = \sin(at + \alpha), \quad y = \sin(bt + \beta)$$

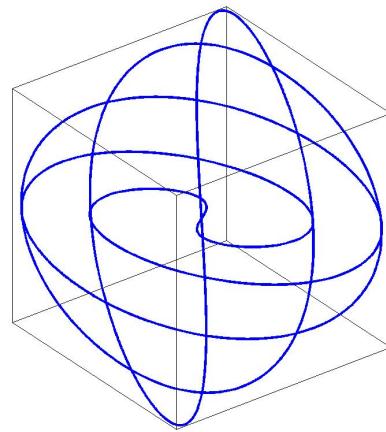


Figure 20.4. $x = \sin t, y = \sin 3/2t, z = \sin 5/4t$.

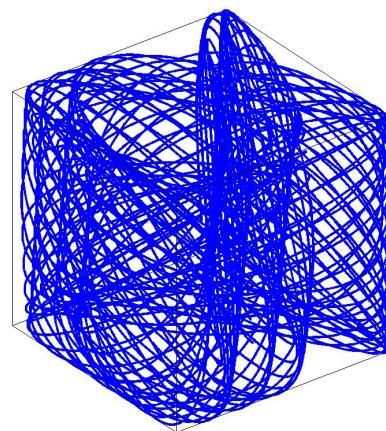


Figure 20.5. $x = \sin t, y = \sin \sigma^7 t, z = \sin \sigma^4 t$.

Three dimensional Lissajous figures are plots of the parametric curves

$$x = \sin(at + \alpha), y = \sin(bt + \beta), z = \sin(ct + \gamma)$$

We can simplify our graphics interface by just considering

$$x = \sin t, y = \sin at$$

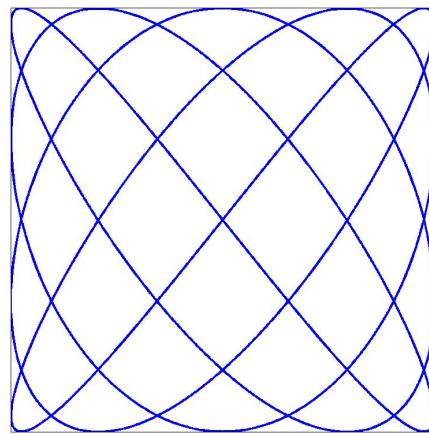


Figure 20.6. $x = \sin t, y = \sin 5/4t.$

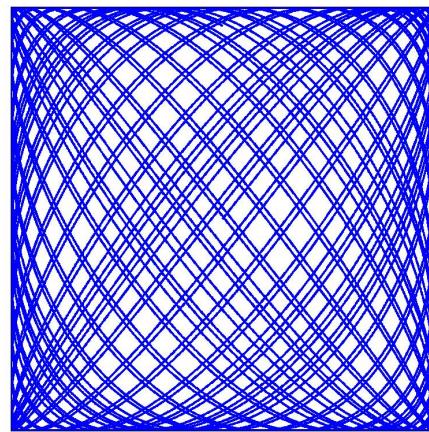


Figure 20.7. $x = \sin t, y = \sin \sigma^4 t.$

and

$$x = \sin t, y = \sin at, z = \sin bt$$

The example with $a = 3/2$ and $b = 5/4$ shown in figure 20.4. is produced by the default settings in our `exm` program `lissajous`. This program allows you to change

a and *b* by entering values in *edit boxes* on the figure. Entering $b = 0$ results in a two dimensional Lissajous figure like the one shown in figure 20.6.

The simplest, “cleanest” Lissajous figures result when the parameters *a* and *b* are ratios of small integers.

$$a = \frac{p}{q}, \quad b = \frac{r}{s}, \quad p, q, r, s = \text{small integers}$$

This is because the three functions

$$x = \sin t, \quad y = \sin \frac{p}{q}t, \quad z = \sin \frac{r}{s}t$$

all return to zero when

$$t = 2m\pi, \quad m = \text{lcm}(q, s)$$

where $\text{lcm}(q, s)$ is the *least common multiple* of *q* and *s*. When *a* and *b* are fractions with large numerators and denominators, the curves oscillate more rapidly and take longer to return to their starting values.

In the extreme situation when *a* and *b* are irrational, the curves never return to their starting values and, in fact, eventually fill up the entire square or cube.

We have seen that dividing the octave into 12 equal sized semitones results in frequencies that are powers of σ , an irrational value. The E key and G keys are four and seven semitones above C, so their frequencies are

```
sigma^4
= 1.259921049894873

sigma^7
= 1.498307076876682
```

The closest fractions with small numerator and denominator are

```
5/4
= 1.250000000000000

3/2
= 1.500000000000000
```

This is why we chose 5/4 and 3/2 for our default parameters. If the irrational powers of `sigma` are used instead, the results are figures 20.5 and 20.7. In fact, these figures are merely the initial snapshots. If we were to let the program keep running, the results would not be pleasant.

Harmony and Intonation

Harmony is an elusive attribute. Dictionary definitions involve terms like “pleasing”, “congruent”, “fitting together”. For the purposes of this chapter, let’s say that two or more musical notes are *harmonious* if the ratios of their frequencies are rational

numbers with small numerator and denominator. The human ear finds such notes fit together in a pleasing manner.

Strictly speaking, a musical chord is three or more notes sounded simultaneously, but the term can also apply to two notes. With these definitions, chords made from a scale with equal semitones are not exactly harmonious. The frequency ratios are powers of σ , which is irrational.

Tuning a musical instrument involves adjusting its physical parameters so that it plays harmonious music by itself and in conjunction with other instruments. Tuning a piano is a difficult process that is done infrequently. Tuning a violin or a guitar is relatively easy and can be done even during breaks in a performance. The human singing voice is an instrument that can undergo continuous retuning.

For hundreds of years, music theory has included the design of scales and the tuning of instruments to produce harmonious chords. Of the many possibilities, let's consider only two – *equal temperament* and *just intonation*.

Equal temperament is the scheme we've been describing so far in this chapter. The frequency ratio between the notes in a chord can be expressed in terms of σ . Tuning an instrument to have equal temperament is done once and for all, without regard to the music that will be played. A single base note is chosen, usually $A = 440$ Hz, and that determines the frequency of all the other notes. Pianos are almost always tuned to have equal temperament.

Just intonation modifies the frequencies slightly to obtain more strictly harmonious chords. The tuning anticipates the key of the music about to be played. Barbershop quartets and *a capella* choirs can obtain just intonation dynamically during a performance.

Here is a MATLAB code segment that compares equal temperament with just intonation from a strictly numerical point of view. Equal temperament is defined by repeated powers of σ . Just intonation is defined by a sequence of fractions.

```

sigma = 2^(1/12);
k = (0:12)';
equal = sigma.^k;

num = [1 16 9 6 5 4 7 3 8 5 7 15 2]';
den = [1 15 8 5 4 3 5 2 5 3 4 8 1]';
just = num./den;

delta = (equal - just)./equal;
T = [k equal num den just delta];
fprintf('%8d %12.6f %7d/%d %10.6f %10.4f\n',T')

k      equal          just        delta
0      1.000000      1/1      1.000000      0.0000
1      1.059463    16/15    1.066667     -0.0068
2      1.122462      9/8      1.125000     -0.0023
3      1.189207      6/5      1.200000     -0.0091
4      1.259921      5/4      1.250000      0.0079

```

5	1.334840	4/3	1.333333	0.0011
6	1.414214	7/5	1.400000	0.0101
7	1.498307	3/2	1.500000	-0.0011
8	1.587401	8/5	1.600000	-0.0079
9	1.681793	5/3	1.666667	0.0090
10	1.781797	7/4	1.750000	0.0178
11	1.887749	15/8	1.875000	0.0068
12	2.000000	2/1	2.000000	0.0000

The last column in the table, `delta`, is the relative difference between the two. We see that `delta` is less than one percent, except for one note. But the more important consideration is how the music sounds.

Chords

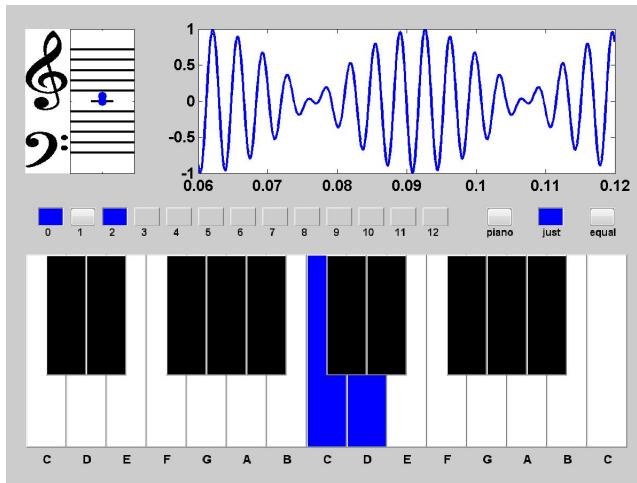


Figure 20.8. Dissonance and beats between two adjacent whole notes.

Chords are two or more notes played simultaneously. With a computer keyboard and mouse, we can't click on more than one key at a time. So chords are produced with `pianoex` by selecting the toggle switches labeled 1 through 12. The switch labeled 0 is always selected.

Figure 20.8 shows the visual output generated when `pianoex` plays a chord involving two adjacent white keys, in this case C and D. You can see, and hear, the phenomenon known as *beating*. This occurs when tones with nearly equal frequencies alternate between additive reinforcement and subtractive cancellation. The relevant trig identity is

$$\sin at + \sin bt = \sin \frac{a+b}{2}t \cos \frac{a-b}{2}t$$

The sum of two notes is a note with the average of the two frequencies, modulated by a cosine term involving the difference of the two frequencies. The players in an orchestra tune up by listening for beats between their instruments and a reference instrument.

The most important three-note chord, or *triad*, is the *C major fifth*. If C is the lowest, or root, note, the chord is C-E-G. In just intonation, the frequency ratios are

$$1 : \frac{5}{4} : \frac{3}{2}$$

These are the parameter values for our default Lissajous figure, shown in figure 20.4. Figures 20.9 and 20.10 show the visual output generated when `pianoex` plays a C major fifth with just intonation and with equal temperament. You can see that the wave forms in the oscilloscope are different, but can you hear any difference in the sound generated?

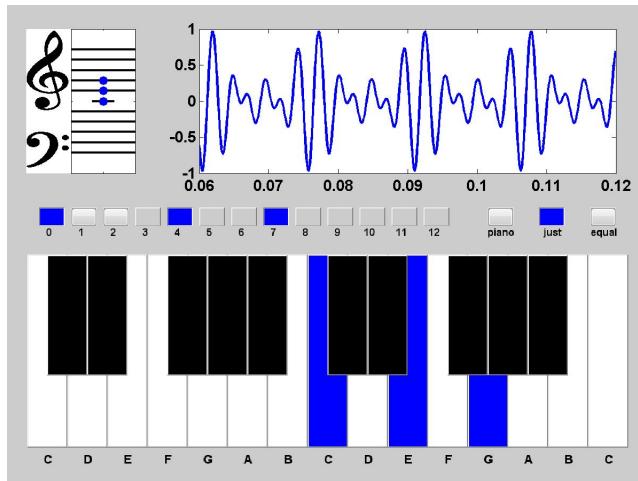


Figure 20.9. *C major fifth with just intonation.*

Synthesizing Music

Our `pianoex` program is not a powerful music synthesizer. Creating such a program is a huge undertaking, way beyond the scope of this chapter or this book. We merely want to illustrate a few of the interesting mathematical concepts involved in music.

The core of the `pianoex` is the code that generates a vector `y` representing the amplitude of sound as a function of time `t`. Here is the portion of the code that handles equal temperament. The quantity `chord` is either a single note number, or a vector like [0 4 7] with the settings of the chord toggles.

```
sigma = 2^(1/12);
```

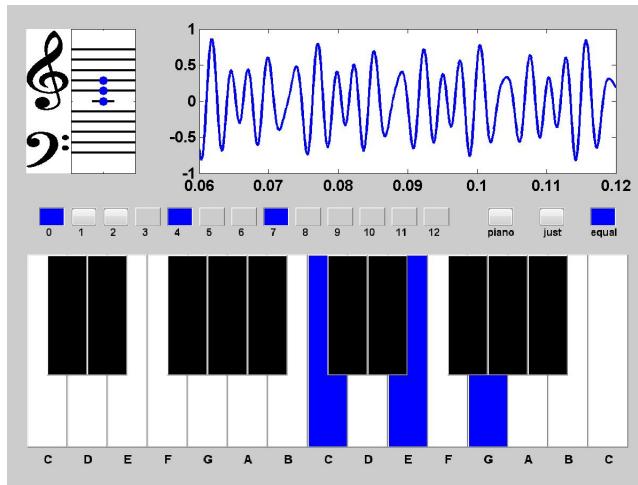


Figure 20.10. *C major fifth with equal temperament.*

```

C4 = 440*sigma^(-9);
fs = 44100;
t = 0:1/fs:T;
y = zeros(size(t));
for n = chord
    hz = C4 * sigma^n;
    y = y + sin(2*pi*hz*t);
end
y = y/length(chord);

```

Here is the corresponding portion of code for just intonation. The vector r of ratios is repeated a few times, scaled by powers of 2, to cover several octaves.

```

sigma = 2^(1/12);
C4 = 440*sigma^(-9);
fs = 44100;
t = 0:1/fs:T;
r = [1 16/15 9/8 6/5 5/4 4/3 7/5 3/2 8/5 5/3 7/4 15/8];
r = [r/2 r 2*r 4];
y = zeros(size(t));
for n = chord
    hz = C4 * r(n+13);
    y = y + sin(2*pi*hz*t);
end
y = y/length(chord);

```

A small example of what a full music synthesizer would sound like is provided by the “piano” toggle on `pianoex`. This synthesizer uses a single sample of an actual

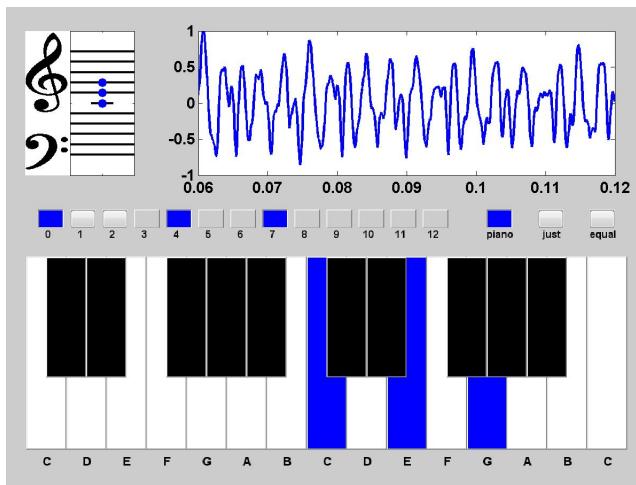


Figure 20.11. *C major fifth chord with simulated piano.*

piano playing middle C. This sample is loaded during the initialization of `pianoex`,

```
S = load('piano_c.mat');
middle_c = double(S.piano_c)/2^15;
set(gcf,'userdata',middle_c)
```

A function from the MATLAB Signal Processing Toolbox is then used to generate notes at different frequencies.

```
middle_c = get(gcf,'userdata');
fs = 44100;
t = 0:1/fs:T;
y = zeros(size(t));
for n = chord
    y = y + resamplex(middle_c,2^(n/12),length(y));
end
```

Figure 20.11 displays the piano simulation of the C major fifth chord. You can see that the waveform is much richer than the ones obtained from superposition of sine waves.

Further Reading, and Viewing

This wonderful video shows a performance of the “Do Re Mi” song from “The Sound of Music” in the Antwerp Central Railway Station. (I hope the URL persists.)

<http://www.youtube.com/watch?v=7EYAUazLI9k>

Wikipedia has hundreds of articles on various aspects of music theory. Here is one:

http://en.wikipedia.org/wiki/Music_and_mathematics

Wikipedia on Lissajous curves:

http://en.wikipedia.org/wiki/Lissajous_curve

Recap

```
%% Music Chapter Recap
% This is an executable program that illustrates the statements
% introduced in the Music Chapter of "Experiments in MATLAB".
% You can access it with
%
%     music_recap
%     edit music_recap
%     publish music_recap
%
% Related EXM programs
%
%     pianoex

%% Size of a semitone, one twelfth of an octave.

sigma = 2^(1/12)

%% Twelve pitch chromatic scale.

for n = 0:12
    pianoex(n)
end

%% C major scale

for n = [0 2 4 5 7 9 11 12]
    pianoex(n)
end

%% Semitones in C major scale

diff([0 2 4 5 7 9 11 12])

%% Equal temperament and just intonation

[sigma.^(0:12)
  1 16/15 9/8 6/5 5/4 4/3 7/5 3/2 8/5 5/3 7/4 15/8 2]'

%% C major fifth chord, equal temperament and just temperament
```

```
[sigma.^ [0 4 7]
 1 5/4 3/2],
```

Exercises

20.1 *Strings.* The Wikipedia page

http://en.wikipedia.org/wiki/Piano_key_frequencies/

has a table headed “Virtual Keyboard” that shows the frequencies of the piano keys as well as five string instruments. The open string violin frequencies are given by

```
v = [-14 -7 0 7],
440*sigma.^v
```

What are the corresponding vectors for the other four string instruments?

20.2 *Vibrating string.* In `vibrating_string.m`, find the statement

```
a = 1./(1:9)
```

Change it to

```
a = 1./(1:9).^2
```

or

```
a = 1./sqrt(1:9)
```

Also, change the loop control

```
for k = 1:9
```

to

```
for k = 1:2:9
```

or

```
for k = 1:3:9
```

What effect do these changes have on the resulting wave?

20.3 *Comet.* Try this:

```
a = 2/3;
b = 1/2;
```

```
tfinal = 12*pi;
t = 0:pi/512:tfinal;
x = sin(t);
y = sin(a*t);
z = sin(b*t);
comet3(x,y,z)
```

Why did I choose this particular value of `tfinal`? How does this `tfinal` depend upon `a` and `b`?

20.4 *Dissonant Lissajous.* What is the Lissajous figure corresponding to two or three adjacent keys, or adjacent white keys, on the piano keyboard?

20.5 *Irrational biorhythms.* The biorhythms described in our “Calendars and Clocks” chapter are based on the premise that our lives are governed by periodic functions of time with periods of 23, 28 and 33 days. What is the effect of revising `biorhythms.m` so that the periods are irrational values near these?

20.6 *Just intonation.* With just intonation, the ratios of frequencies of adjacent notes are no longer equal to σ . What are these ratios?

20.7 *Rational intonation.* MATLAB *almost* has the capability to discover just intonation. The MATLAB function `rat` computes rational approximations. For example, the following statement computes the numerator `n` and denominator `d` in a rational approximation of π .

```
[n,d] = rat(pi)
n =
    355
d =
    113
```

This gives us the approximation $\pi \approx 355/113$, which is accurate to 7 significant figures. For a less accurate approximation, specify a tolerance of 2 percent.

```
[n,d] = rat(pi,.02)
n =
    22
d =
     7
```

This gives us the familiar $\pi \approx 22/7$.

(a) Let’s have `rat`, with a tolerance of .02, generate rational approximations to the powers of σ . We can compare the result to the rational approximations used in just intonation. In our code that compares equal temperament with just intonation, change the statements that define just intonation from

```
num = [...]
```

```
den = [...]
to
[num,den] = rat(sigma.^k,.02);
```

You should see that the best rational approximation agrees with the one used by just intonation for most of the notes. Only notes near the ends of the scale are different.

- 18/17 vs. 16/15 for $k = 1$
- 9/5 vs. 7/4 for $k = 10$
- 17/9 vs. 15/8 for $k = 11$

The approximations from `rat` are more accurate, but the denominators are primes or powers of primes and so are less likely to be compatible with other denominators.

- (b) Change the tolerance involved in using `rat` to obtain rational approximations to powers of σ . Replace .02 by .01 or .10. You should find that .02 works best.
- (c) Modify `pianoex` to also use the rational approximations produced by `rat`. Can you detect any difference in the sound generated by `pianoex` if these `rat` approximations are incorporated.

20.8 Musical score. Our `pianoex` is able to process a MATLAB function that represents a musical score. The score is a cell array with two columns. The first column contains note numbers or chord vectors. The second column contains durations. If the second column is not present, all the notes have the same, default, duration. For example, here is a C-major scale.

```
s = {0 2 4 5 7 9 11 12}'  
pianoex(s)
```

A more comprehensive example is the portion of Vivaldi's "Four Seasons" in the EXM function `vivaldi`.

```
type vivaldi  
pianoex(vivaldi)
```

Frankly, I do not find this attempt to express music in a machine-readable form very satisfactory. Can you create something better?