Experiments with MATLAB®

Cleve Moler

August 11, 2009

Preface

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 introducing 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 the following figure. You can click on the icons to preview some of the experiments.

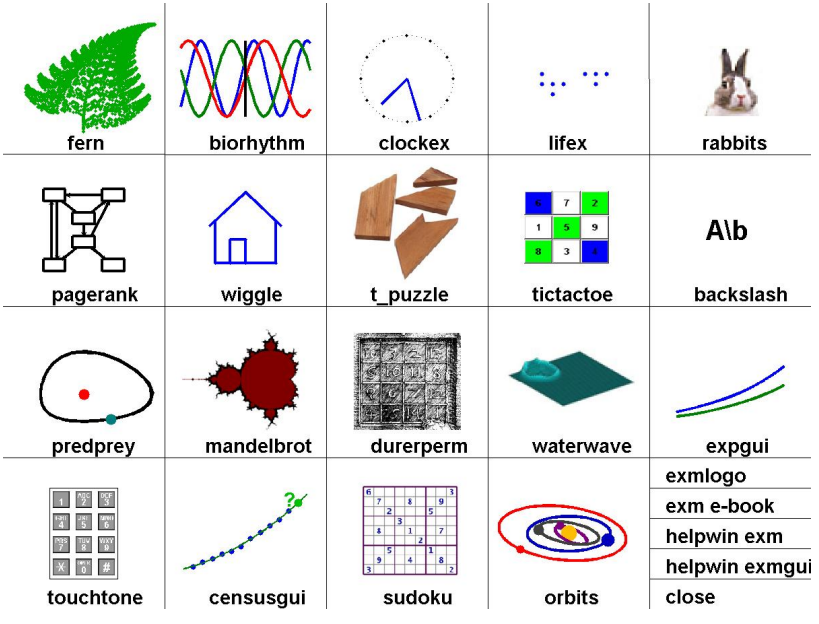

Figure 1. *exmgui provides a starting point for some of the experiments.*

Cleve Moler
August 11, 2009

Bibliography

[1] C. Moler, *Numerical Computing with MATLAB*,
Electronic edition: The MathWorks, Inc., Natick, MA, 2004.
`http://www.mathworks.com/moler`
Print edition: SIAM, Philadelphia, 2004.
`http://ec-securehost.com/SIAM/ot87.html`

Chapter 1

Iteration

An investigation of fixed point iterations introduces the assignment statement, `for` *and* `while` *loops, the* `plot` *function, and the Golden Ratio.*

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`. Now, repeatedly execute the statement by using the up-arrow key, followed by the enter or return key. 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. Starting with numbers less than -1 makes things more complicated, but you might want to try it anyway.)

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*.)

$$X = \sqrt{1+X}$$
$$X^2 = 1 + X$$
$$X^2 - X - 1 = 0$$
$$X = \frac{1 \pm \sqrt{1+4}}{2}$$
$$\varphi = \frac{1+\sqrt{5}}{2}$$

Figure 1.1. *Compute the fixed point by hand.*

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 use the quadratic formula. Take a look at the hand calculation shown in figure 1.1. Squaring the equation introduces a second, negative solution. The positive root 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
```

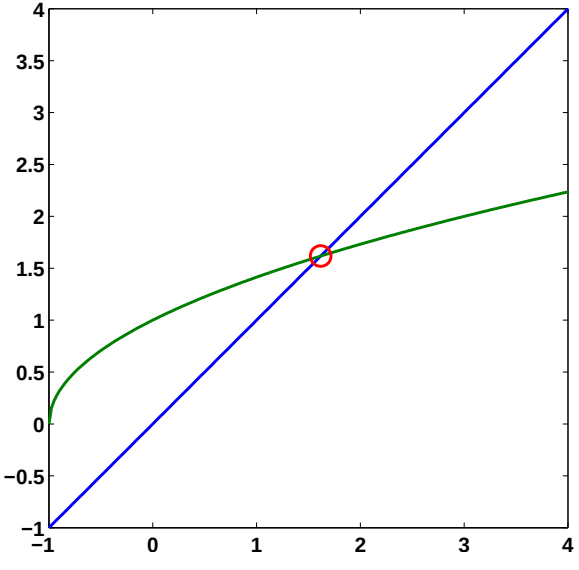

Figure 1.2. *A fixed point at $\phi = 1.6180$.*

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:.025:4;
```

generates a vector `x` containing the numbers from -1 to 4 in steps of .025. Then the statement

```
plot(x,x,'-',x,sqrt(1+x),'-',phi,phi,'o')
```

produces a figure that has three components. The first two components are graphs of $y = x$ and $y = \sqrt{1+x}$. The `'-'` argument tells the `plot` function to draw solid lines. The last component in the plot is a single point with both x and $y-$coordinates equal to ϕ. The `'o'` tells the `plot` function to draw a circle. The `plot` function has many variations, including specifying other colors and line types. You can see some of the possibilities with

```
help plot
```

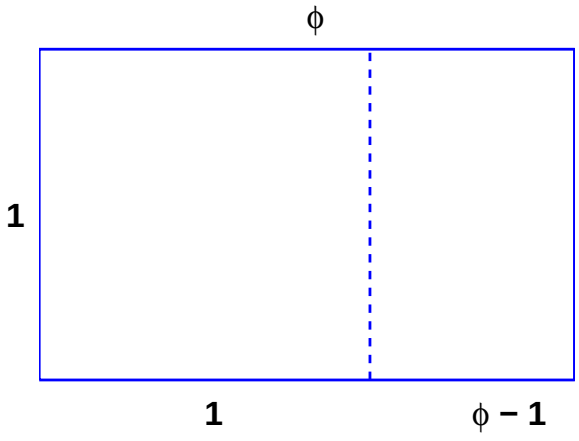

Figure 1.3. *The golden rectangle.*

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 with these fixed point iterations, 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 = Inf;
while abs(x-y) > eps(x)
   y = x;
   x = sqrt(1+x)
end
```

The quantity `Inf` is MATLAB abbreviation of *infinity*. 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.

Exercises

1.1 *Expressions.* Use MATLAB to evaluate each of these mathematical expressions.

43^2	-3^4	$\sin 1$
$4^{(3^2)}$	$(-3)^4$	$\sin 1^\circ$
$(4^3)^2$	$\sqrt[4]{-3}$	$\sin \frac{\pi}{3}$
$\sqrt[4]{32}$	$-2^{-4/3}$	$(\arcsin 1)/\pi$

You can get started with

```
help ^
help sin
```

1.2 *Temperature conversation.*
(a) Write a MATLAB statement that converts temperature in Fahrenheit, `f`, to Celius, `c`.

`c =` *something involving* `f`

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

`f =` *something involving* `c`

1.3 *Barn-megaparsec.* A *barn* is a unit of area employed by high energy physicists. Nuclear scattering experiments try to "hit the side of a barn". A *parsec* is a unit of length employed by astronomers. A star at a distance of one parsec exhibits a trigonmetric 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 *Comparison.* Which is larger, π^ϕ or ϕ^π?

1.5 *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)
```

1.6 *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?

1.7 *Continued fraction.* Verify that ϕ can be expressed as this infinite continued fraction.

$$\phi = 1 + \frac{1}{1 + \frac{1}{1 + \frac{1}{1 + \cdots}}}.$$

If this infinite expression is truncated after n terms, the result is a rational approximation to ϕ. For example, for $n = 4$, the approximation is

$$\phi \approx 1 + \frac{1}{1 + \frac{1}{1 + \frac{1}{1}}}.$$

This is this the same as

$$\phi \approx \frac{5}{3}$$

Verify that this MATLAB program computes two integers p and q whose ratio p/q is a conventional fraction with the same value as the continued fraction truncated after n terms.

```
n = ...
p = 1;
q = 0;
for k = 1:n
   s = p;
   p = p + q;
   q = s;
end
p/q
```

As n is increased, how well does p/q approximate ϕ ?

1.8 Find the numerical solution of the equation

$$x = \cos x$$

in the interval $[0, \frac{\pi}{2}]$, shown in figure 1.4.

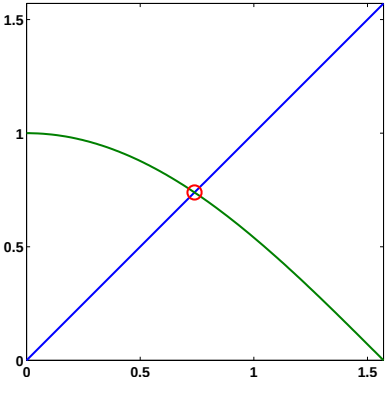

Figure 1.4. *Fixed point of x = cos(x).*

1.9 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 solution 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
```

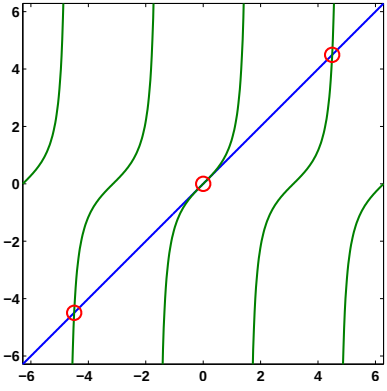

Figure 1.5. *Three fixed points of $x = tan(x)$*

```
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
```

1.10 *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
```

1.11 *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.

1.12 *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.

1.13 *Fixed point iterations.*

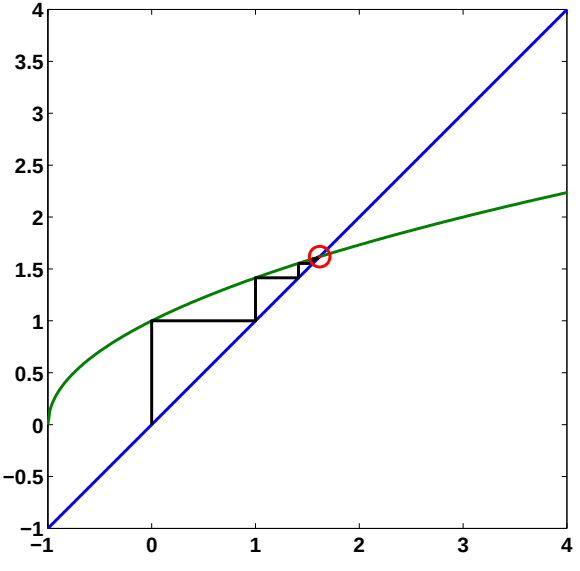

Figure 1.6. *The fixed point iteration.*

Let's examine fixed point iterations more carefully. Figure 1.6 is produced by entering the statement

```
fixedpoint
```

This executes the MATLAB program stored in the file `exm/fixedpoint.m`. In the figure, the straight blue line is the graph of $y = x$. The curved green line is the graph of the function $f(x) = \sqrt{1+x}$. The stairstep black line shows the steps of our iteration alternating back and forth between the two graphs. All three lines converge to a point in the red circle. Click on the magnifying glass with the plus sign in the figure header. Now you can zoom in with mouse clicks on the red circle. As you get closer and closer to the intersection point, the scale changes and the

green line gets straighter, but the overall picture doesn't change otherwise. (If you're reading the printed edition of this book, you can't zoom in on the graphic; you'll have to actually run MATLAB to get a live picture.)

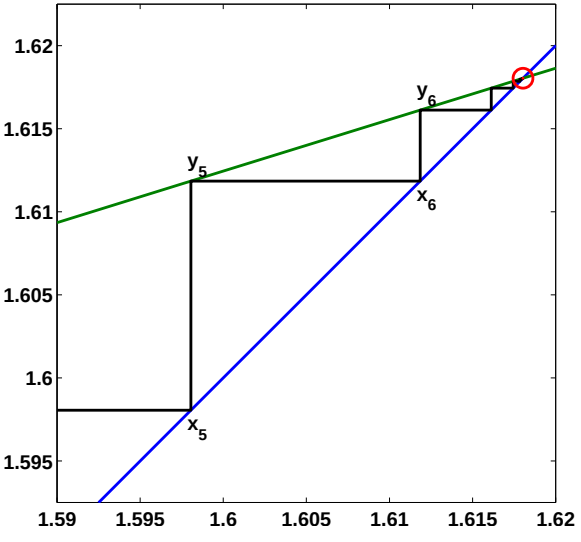

Figure 1.7. *The slope of the green curve is less than one.*

Run `fixedpoint` and zoom in, not on the red circle, but on one of the steps. Figure 1.7 shows the fifth step, from x_5 to x_6. The blue line is straight and so the object formed with a segment of the blue line as the hypoteneuse is actually an equilateral triangle. The green line is not straight, although it is hard to see the curvature at this level of magnification. The slope of one of the green segments is the ratio of the change in y to the change in x. In other words, the slope is the ratio of the length of successive steps of our iteration. The slope for the fifth step is

$$\frac{y_6 - y_5}{x_6 - x_5}$$

The key question is: what are the slopes of these green segments? Here is a MATLAB program that computes these slopes.

```
format short
x = 0;
y = 1;
while x ~= y
   delta = y - x;
   x = y;
   y = sqrt(1 + x);
   slope = (y - x)/delta
end
```

There are several important things to see in this program. The only assignment statement that is not terminated with a semicolon is the assignment to `delta`. Run this program andd verify that the output starts with

```
slope = 0.4142
slope = 0.3369
slope = 0.3173
slope = 0.3115
slope = 0.3098
     ...
```

Each time through the loop, our program computes `delta`, which is the length of the previous step. New values of `x` and `y` are then used to compute `slope`, the ratio of the length of the new step to the length of the previous step. The slopes are all less than one and soon reach a value that is constant to four decimal places. Call this value δ. What is the value of δ?

The blue line has slope one. The green curve has variable slope, but that slope approaches δ, which is significantly less than one. (In fact, δ is approxately $f'(\phi)$, the value of the derivative of $f(x)$ at the fixed point.)

The function

$$f(x) = \sqrt{1+x}$$

is a *contraction*. For points x_n and x_{n+1} near the fixed point, the function satisfies

$$|y_{n+1} - y_n| = |f(x_{n+1}) - f(x_n)| < \delta \ |x_{n+1} - x_n|$$

This implies that after n steps

$$|x_{n+1} - x_n| < \delta^n \ |x_1 - x_0|$$

Use MATLAB to compute δ^{10} and δ^{32}. The output uses an "`e`" to denote a power of 10 scale factor. For example `8e-6` is $8 \cdot 10^{-6}$. You should find that δ^{10} is less than 10^{-5}, which is the accuracy of `format short`, and that δ^{32} is less than 10^{-16}, which is the accuracy of `format long`.

Chapter 2

Fibonacci Numbers

Fibonacci numbers introduce vectors, functions and recursion. An exercise introduces graphics user interfaces.

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 put 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*.

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 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 the M-file `fibonacci.m`, 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 M-file, 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 nth 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. The relationship $f_n = f_{n-1} + f_{n-2}$ is known as a *recursion relation* and 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
```

Now compare the results produced by `goldfract(6)` 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 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

$$\frac{f_{n+1}}{f_n} = \phi_n.$$

In the infinite limit, the ratio of successive Fibonacci numbers approaches the golden ratio:

$$\lim_{n \to \infty} \frac{f_{n+1}}{f_n} = \phi.$$

To see this, compute 40 Fibonacci numbers.

```
n = 40;
f = fibonacci(n);
```

Then compute their ratios.

```
f(2:n)./f(1:n-1)
```

This takes the vector containing `f(2)` through `f(n)` and divides it, element by element, by the vector containing `f(1)` through `f(n-1)`. The output begins with

```
2.00000000000000
1.50000000000000
1.66666666666667
1.60000000000000
1.62500000000000
1.61538461538462
1.61904761904762
1.61764705882353
1.61818181818182
```

and ends with

```
1.61803398874990
1.61803398874989
1.61803398874990
1.61803398874989
1.61803398874989
```

Do you see why we chose `n = 40`? Use the up arrow key on your keyboard to bring back the previous expression. Change it to

```
f(2:n)./f(1:n-1) - phi
```

and then press the Enter key. What is the value of the last element?

The population of Fibonacci's rabbit pen doesn't double every month; it is multiplied by the golden ratio every month.

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

$$\rho^2 = \rho + 1.$$

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

$$\begin{aligned} f_0 &= c_1 + c_2 = 1, \\ f_1 &= c_1\phi + c_2(1-\phi) = 1. \end{aligned}$$

One of the exercises asks you to use the MATLAB backslash operator to solve this 2-by-2 system of simultaneous linear equations, but it is may be easier to solve the system by hand:

$$c_1 = \frac{\phi}{2\phi - 1},$$
$$c_2 = -\frac{(1-\phi)}{2\phi - 1}.$$

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 = round((phi.^(n+1) - (1-phi).^(n+1))/(2*phi-1))
```

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
```

Exercises

2.1 *Waltz.* Which Fibonacci numbers are even? Why?

2.2 *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
```

2.3 *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?

2.4 *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)`.

2.5 *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?

2.6 *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`.

2.7 *Mortality.* What if rabbits took one month to mature, but then died after six months. The sequence would be defined by

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

and, for $n > 2$,

$$d_n = d_{n-1} + d_{n-2} - d_{n-7}$$

(a) Modify `fibonacci.m` and `fibnum.m` to compute this sequence.
(b) How many pairs of rabbits are there after 12 months?
(c) $d_n \approx \delta^n$. What is δ?
(d) Estimate how long it would take your computer to compute `fibnum(50)` with this modified `fibnum`.

2.8 *Rabbits.* Simulate Fibonacci's rabbit pen. If you are not interested in the details of MATLAB graphical user interfaces, then just experiment with `rabbits` without trying to understand the program itself. However, if you are interested in MATLAB Handle Graphics, then `rabbits` is a good starting point.

Let me explain the program. First of all, if you have a running MATLAB handy, enter

```
type rabbits
```

so you can see the entire program. The preamble of `rabbits` is

```
function rabbits(action)
% RABBITS  Fibonacci's rabbit pen.
%   Create immature pushbuttons that age after one click.
```

The first line indicates that the function has no output arguments and at most one input argument. The input argument, if it exists, is named `action`. The two comment lines provide the response to `help rabbits`.

If you call `rabbits` with no arguments, then `nargin`, which stands for "number of input arguments", is zero and so you enter this section.

```
if nargin == 0
   clf
   shg
   uicontrol('style','text','fontsize',12,'fontweight','bold', ...
      'units','normal','position',[.47 .94 .06 .04])
   action = 'mature';
end
```

The commands `clf` and `shg` clear the current figure window and bring it forward if it is not already. The call to `uicontrol` creates a Handle Graphics `text` object and sets two of its font attributes. The coordinate system to be used within the

figure window is normalized, so the window is given unit height and width. The text object has its lower left corner at `(.47,.94)` and its size `.06`-by-`.04`, so it is centered horizontally near the top of the window and occupies a small fraction of the overall window. The final statement is this section sets the missing input argument named `action` to the string `'mature'`.

The function then switches on the value of `action`. If `action` is `'mature'`, which it is on the first entry, then the following section of `rabbits` is executed.

```
switch action
   case 'mature'
      f = get(gcf,'position');
      p = [.90*f(3:4).*rand(1,2) 24 24];
      g = [.7 .7 .7];
      c = 'rabbits(''immature'')';
      uicontrol('style','pushbutton','position',p,'background',g, ...
         'callback',c,'enable','off');
      set(gcbo,'enable','off')
```

The variable `f` is assigned a four-element vector giving the location and size of the current figure window. Since `units` is not specifed, the results are in pixels. The default figure window size, `f(3:4)`, is typically `560` pixels wide and `420` pixels high, but you can resize the window. Computers with large screens might have larger default figure windows. The variable `p` is assigned a four-element vector whose first two elements are random fractions of the window size and whose second two elements specify a 24-by-24 pixel portion of the window. The variable `g` is set to a 3-element color vector describing the shade of gray that results from having red, green and blue components equal to 70% of their full value. The variable `c` is assigned a string which, when subsequently evaluated, will result in a call to `rabbits` with an argument of `'immature'`. The `uicontrol` statement creates a new Handle Graphics object, this time a pushbutton with a gray background. The button is placed in position `p` within the figure window. Whenever the pushbutton is pushed, the callback string `c` will be evaluated, resulting in another call to `rabbits`. This is not exactly recursion, since `rabbits` does not call itself directly. Instead, `rabbits` posts a callback to the graphics system and eventually exits. Later, when a mouse click occurs, the graphics system calls `rabbits` again. The final statement disables `gcbo`, which stands for "get current callback object". Disabled buttons created are visible, but do not respond to mouse clicks.

If, on the other hand, `action` is `'immature'`, then the following section of `rabbits` is executed.

```
   case 'immature'
      R = imread('rabbit.jpg');
      g = get(gcbo,'position');
      p = [g(1:2)-12 48 48];
      c = 'rabbits(''mature'')';
      set(gcbo,'cdata',R,'position',p,'callback',c,'enable','off');
end
```

The first statement reads the jpeg image `rabbit.jpg` that is available in the `exm` toolbox. This size of this image is 47-by-45 pixels. The variable `g` is set to the position, in pixels within the figure, of `gcbo`, which is the button that initiated this callback. The variable `p` is set to a four-element vector obtained by moving `g` twelve pixels down and to the left and increasing its size to 48-by-48. The variable `c` is the callback string, `rabbits('mature')`. The final statement in this section sets the pushbutton's color data to be the image `R`, resets the button's position and callback string, and disables the button. In contrast to the `mature` section of program, this section does not create a new graphics object. It just modifies four of the properties of the button that was pushed.

The last three statements of `rabbits` are executed each time the function is called.

```
b = findobj(gcf,'style','pushbutton');
if ~any(any(char(get(b,'enable')) == 'n'))
   set(b,'background','w','enable','on')
end
set(findobj(gcf,'style','text'),'string',length(b))
```

The variable `b` is set to a vector of handles of all the pushbuttons that have been created. The `if` statement is pretty tricky. It checks the `enable` state of all the pushbuttons. If all of the buttons are disabled, they are all turned on. By this time you have probably forgotten that this exercise has something to do with Fibonacci numbers. The last line of `rabbits` counts the number of pushbuttons that have been created and updates the text object that was initialized a long time ago with the information needed to explain what this program is doing.

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`.

```
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
```

August 8, 2009

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 %9.3f\n'
```

I am writting this on May 22, 2007, at about 7:30pm, so

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

produces

```
2007      5     22     19     31    54.015
```

In other words,

```
year = 2007
month = 5
day = 22
hour = 19
minute = 31
seconds = 54.015
```

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 zeroes. 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, my current date number

```
datenum(c)
```

is

```
733184.848
```

This indicates that the current time is most of the way through day number `733184`. I get the same result from

```
datenum(now)
```

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

```
datenum(2007,5,22)
```

and

```
datenum('May 22, 2007')
```

return

```
733184
```

The same result is obtained from

```
fix(now)
```

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, 2008') - datenum(fix(now))
ans =
   224
```

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(2007,5,22))
```

and

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

return

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

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

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. This code counts the number of times that Friday occurs on the various weekdays in a 400 year calendar cycle and produces figure 3.1.

```
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
c
bar(c)
axis([0 8 680 690])
```

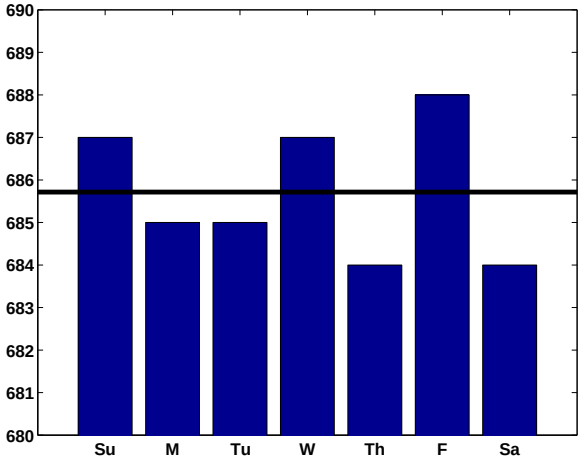

Figure 3.1. *The 13th is more likely to be on Friday than any other day.*

```
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 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 written, May 22, 2007. It shows that I must have been in pretty bad shape about a week ago, but that my physical strength can be expected to reach a peak a few days from now, and that my intellectual powers and emotional well-being should reach their peaks within a few hours of each other about a week after that.

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

`http://patft.uspto.gov`

The date and graphics functions in MATLAB make the computation and dis-

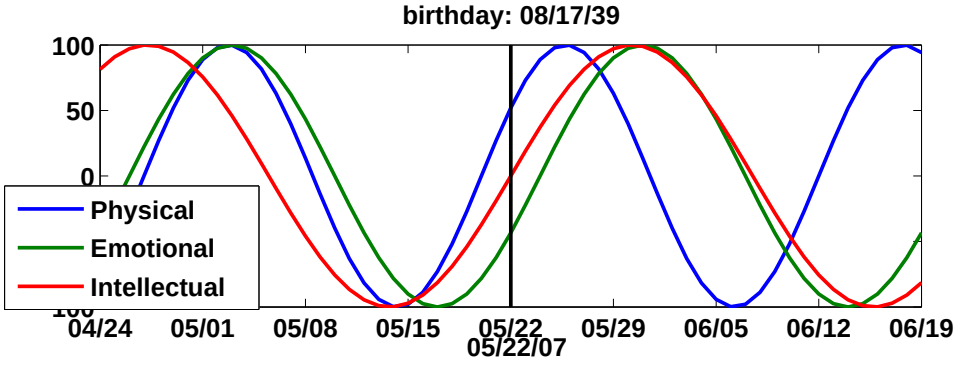

Figure 3.2. *My biorhythm.*

play 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.

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 astronomonical 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.

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 *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?

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

3.5 *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?

3.6 *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?

3.7 *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

```
JDN = datenum + 1721058.5
```

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

3.8 *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

```
time_t = 86400*(datenum(y,m,d) - 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.

3.9 *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?

3.10 `biorhythm`.
(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. How long does it take until all three simultaneously return to that initial condition? How old were you, or will you be, on that date? Plot your biorhythm near that date. You should find the `lcm` function helpful.
(c) Is it possible for all three biorhythm cycles to reach their maximum or minimum at exactly the same time? Why or why not?

3.11 `clockex`. This exercise is about the `clockex` program in our `exm` toolbox.
(a) Why does `clockex` use trig functions?
(b) Make `clockex` run counter-clockwise.
(c) Modify `clockex` to have a digital display of your own design.
(d) Why do the hour and minute hands in `clockex` move nearly continuously while the second hand moves in discrete steps.
(e) What happens if the statement

```
pause(1.0)
```

in `clockex.m` is changed to

```
pause(0.9)
```

or

```
pause(1.1)
```

(f) What happens if the statement

```
k = ceil(c(6))
```

in `clockex.m` is changed to

```
k = round(c(6))
```

(g) Instead of

```
k = ceil(c(6))
```

in `clockex.m`, I considered using

```
k = ceil(c(6)+realmin)
```

Why would that be a good idea?

Chapter 4

T Puzzle

A classic puzzle demonstrates complex arithmetic.

Figure 4.1. *The wooden T puzzle. Photo courtesy of Shop New Zeland,* `http://www.shopnewzealand.co.nz`.

I first saw the wooden T puzzle shown in figure 4.1 at Puzzling World in

August 8, 2009

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.

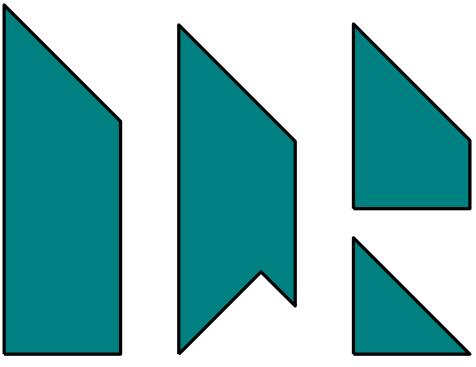

Figure 4.2. *The four pieces.*

Figure 4.2 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.

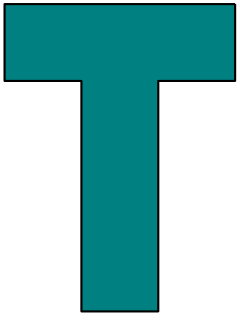

Figure 4.3. *The T.*

It turns out to be possible to arrange the four pieces to form the capital "T" shape shown in figure 4.3, 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

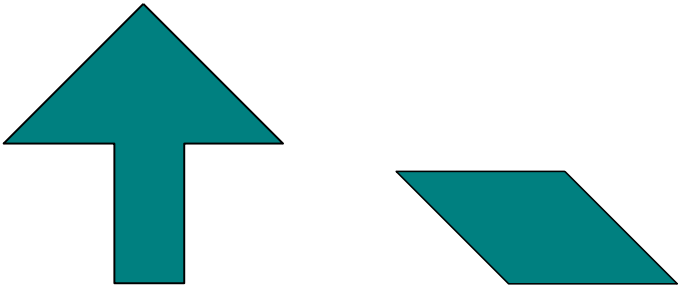

Figure 4.4. *The arrow and the rhombus.*

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 effecitive 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* represention 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` is 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
```

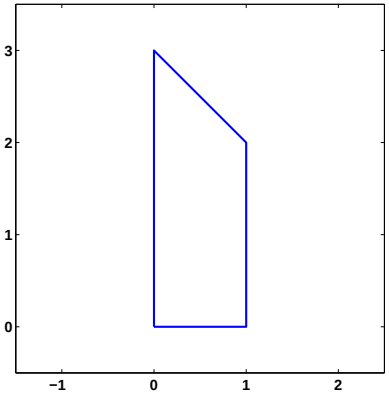

Figure 4.5. *The complex coordinates of the vertices are* $0 + 0i$, $1 + 0i$, $1 + 2i$, *and* $0 + 3i$.

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 entires. 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 4.5.

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 4.5 at one of the corners. Do you see which corner?

Rotations are also easily done. The statements

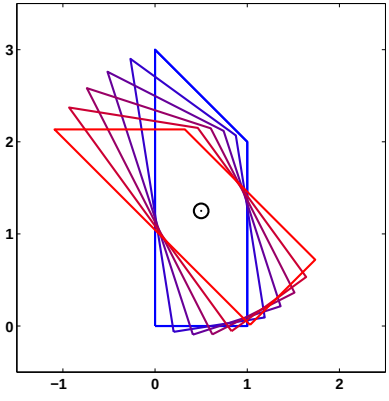

Figure 4.6. *Rotating through multiples of nine degrees.*

```
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 4.6 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.

Exercises

4.1 *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}

4.2 *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)
```

4.3 *Strip.* How many different ways are there to form the shape in figure 4.7 with the T-puzzle pieces?

Figure 4.7. *A strip.*

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

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

4.6 *Jumpy 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?

4.7 *Snappy traslation.* 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?

4.8 *Rotation.* Reproduce figure 4.6.

4.9 *Revelation.* Find `t_puzzle`'s hidden feature.

4.10 *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 5

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 5.1 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

$$A = \begin{pmatrix} a_{1,1} & a_{1,2}; & a_{2,1} & a_{2,2} \end{pmatrix}$$

For example

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

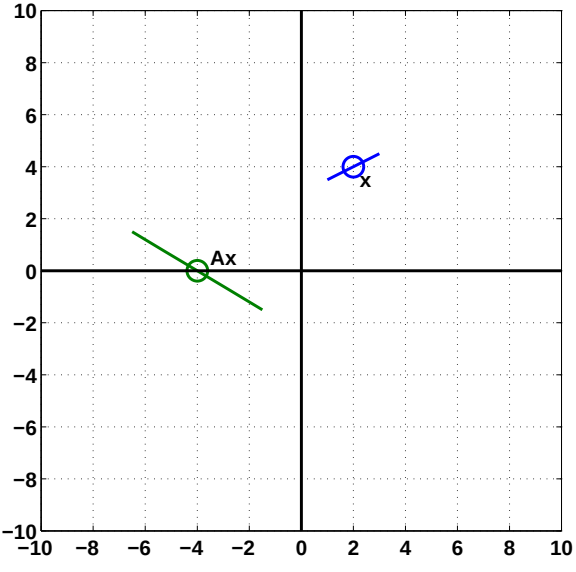

Figure 5.1. *Matrix multiplication transforms lines through x to lines through Ax.*

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 5.1 is transformed to the point labeled `Ax`. Matrix-vector multiplications produce *linear* transformations. This means that for scalars s and t and vectors x and y,

$$A(sx + ty) = sAx + tAy$$

This implies that points near x are transformed to points near Ax and that straight lines in the plane through x are transformed to straight lines through Ax.

Our definition of matrix-vector multiplication is the usual one involving the *dot product* of the *rows* of A, denoted $a_{i,:}$, with the vector x.

$$Ax = \begin{pmatrix} a_{1,:} \cdot x \\ a_{2,:} \cdot x \end{pmatrix}$$

An alternate, and sometimes more revealing, definition uses *linear combinations* of the *columns* of A, denoted by $a_{:,j}$.

$$Ax = x_1 a_{:,1} + x_2 a_{:,2}$$

For example

$$\begin{pmatrix} 4 & -3 \\ -2 & 1 \end{pmatrix} \begin{pmatrix} 2 \\ 4 \end{pmatrix} = 2 \begin{pmatrix} 4 \\ -2 \end{pmatrix} + 4 \begin{pmatrix} -3 \\ 1 \end{pmatrix} = \begin{pmatrix} -4 \\ 0 \end{pmatrix}$$

The *transpose* of a column vector is a row vector, denoted by x^T. The transpose of a matrix interchanges its rows and columns. For example,

$$x^T = \begin{pmatrix} 2 & 4 \end{pmatrix}$$

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

Vector-matrix multiplication can be defined by

$$x^T A = A^T x$$

That is pretty cryptic, so if you have never seen it before, you might have to ponder it a bit.

Matrix-matrix multiplication, AB, can be thought of as matrix-vector multiplication involving the matrix$A$ and the columns vectors from B, or as vector-matrix multiplication involving the row vectors from A and the matrix B. It is important to realize that AB is not the same matrix as BA.

MATLAB started its life as "Matrix Laboratory", so its very first capabilities involved matrices and matrix multiplication. The syntax follows the mathematical notation closely. We use square brackets instead of round parentheses, an asterisk to denote multiplication, and `x'` for the transpose of `x`. The foregoing example becomes

```
x = [2; 4]
A = [4 -3; -2 1]
A*x
```

This produces

```
x =
     2
     4
```

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

The matrices `A'*A` and `A*A'` are not the same.

```
A'*A =
    20   -14
   -14    10
```

while

```
A*A' =
    25   -11
   -11     5
```

The matrix

$$I = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$$

is the 2-by-2 *identity* matrix. It has the important property that for any 2-by-2 matrix A,

$$IA = AI = A$$

Originally, MATLAB variable names were not case sensitive, so `i` and `I` were the same variable. Since `i` is frequently used as a subscript, an iteration index, and `sqrt(-1)`, we could not use `I` for the identity matrix. Instead, we chose to use the sound-alike word `eye`. Today, MATLAB is case sensitive and has many users whose native language is not English, but we continue to use `eye(n,n)` to denote the `n`-by-`n` identity. (The Metro in Washington, DC, uses the same pun – "I street" is "eye street" on their maps.)

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 5.2. 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 5.2 is constructed from `X` by

```
dot_to_dot(X)
```

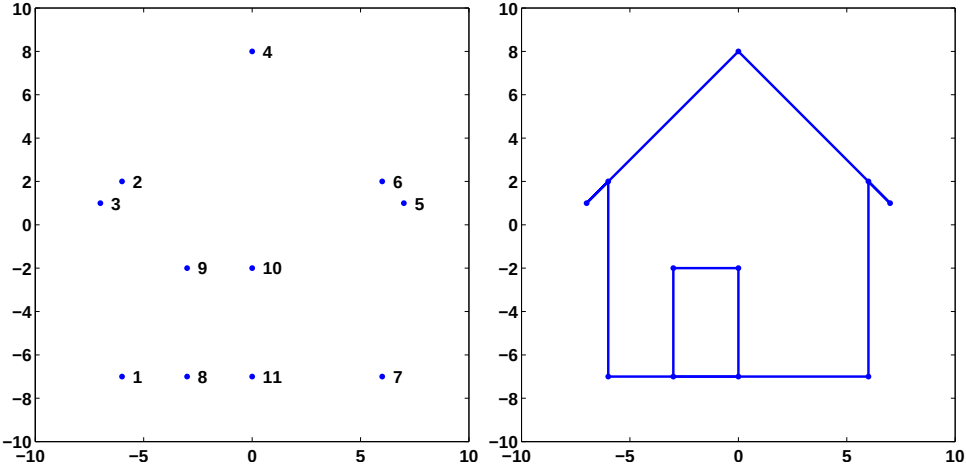

Figure 5.2. *Connect the dots.*

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 5.3 uses matrix multiplication `A*X` and `dot_to_dot(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 right angles to each other and parallel to the axes. The matrix `A1` shrinks the first

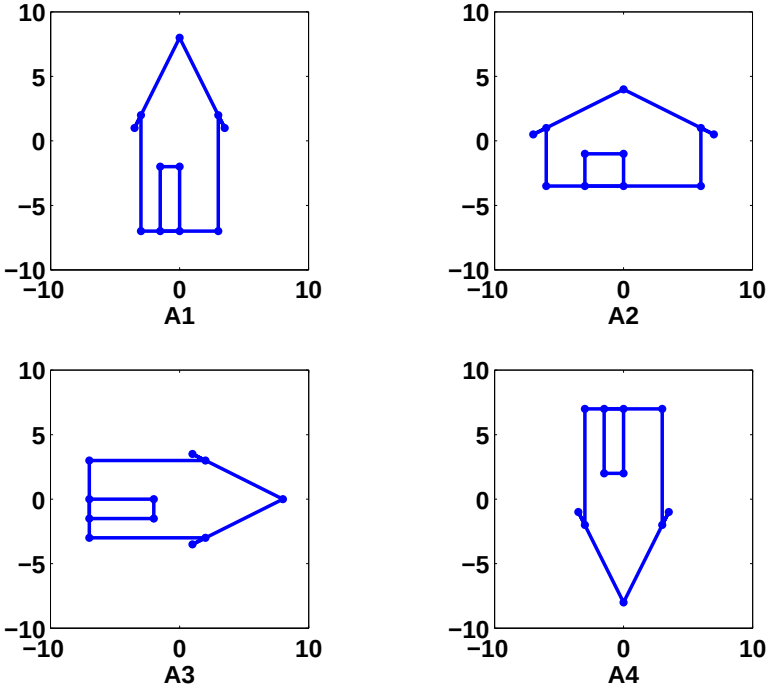

Figure 5.3. *The effect of multiplication by scaling matrices.*

coordinate to reduce the width of the house while the height remains unchanged. The matrix `A2` shrinks the second coordinate to reduce the height, but not the width. The matrix `A3` interchanges the two coordinates while shrinking one of them. The matrix `A4` shrinks the first coordinate and changes the sign of the second.

The *determinant* of a 2-by-2 matrix

$$A = \begin{pmatrix} a_{1,1} & a_{1,2} \\ a_{2,1} & a_{2,2} \end{pmatrix}$$

is the quantity

$$a_{1,1}a_{2,2} - a_{1,2}a_{2,1}$$

In general, determinants are not very useful in practical computation because they have atrocious scaling properties. But 2-by-2 determinants can be useful in understanding simple matrix properties. If the determinant of a matrix is positive, then multiplication by that matrix preserves left- or right-handedness. The first two of our four matrices have positive determinants, so the door remains on the left side of the house. The other two matrices have negative determinants, so the door is transformed to the other side of the house.

The MATLAB function `rand(m,n)` generates an m-by-n matrix with random entries between 0 and 1. So the statement

```
R = 2*rand(2,2) - 1
```

generates a 2-by-2 matrix with random entries between -1 and 1. Here are four of them.

```
R1 =
    0.0323   -0.6327
   -0.5495   -0.5674

R2 =
    0.7277   -0.5997
    0.8124    0.7188

R3 =
    0.1021    0.1777
   -0.3633   -0.5178

R4 =
   -0.8682    0.9330
    0.7992   -0.4821
```

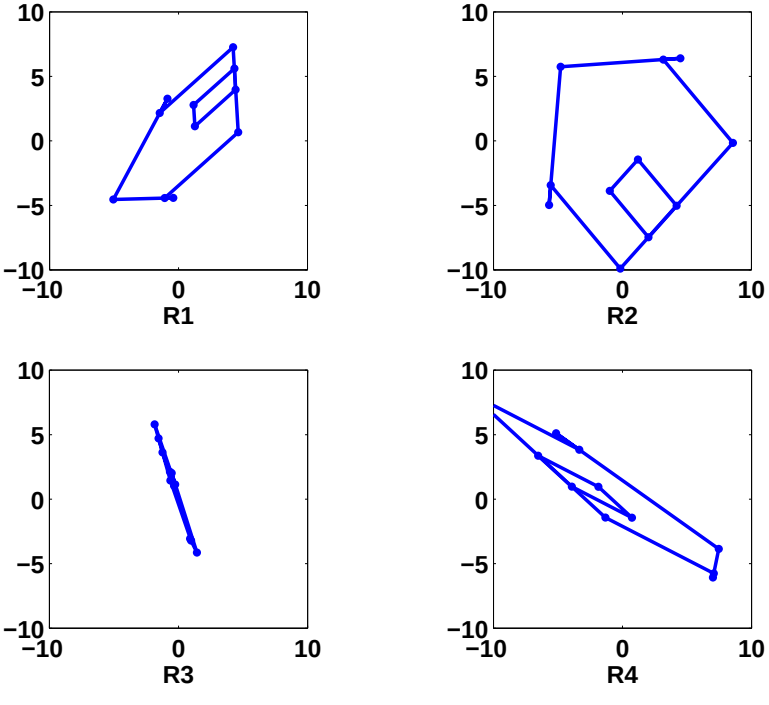

Figure 5.4. *The effect of multiplication by random matrices.*

Figure 5.4 shows the effect of multiplication by these four matrices on the house.

Matrices `R1` and `R4` have large off-diagonal entries and negative determinants, so they distort the house quite a bit and flip the door to the right side. The lines are still straight, but the walls are not perpendicular to the floor. Linear transformations preserve straight lines, but they do not necessarily preserve the angles between those lines. Matrix `R2` is close to a rotation, which we will discuss shortly. Matrix `R3` is nearly *singular*; its determinant is equal to 0.0117. If the determinant were exactly zero, the house would be flattened to a one-dimensional straight line.

The following matrix is a *plane rotation*.

$$G(\theta) = \begin{pmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{pmatrix}$$

We use the letter G because Wallace Givens pioneered the use of plane rotations in matrix computation in the 1950s. Multiplication by $G(\theta)$ rotates points in the plane through an angle θ. Figure 5.5 shows the effect of multiplication by the plane rotations with $\theta = 15°$, $45°$, $90°$, and $215°$.

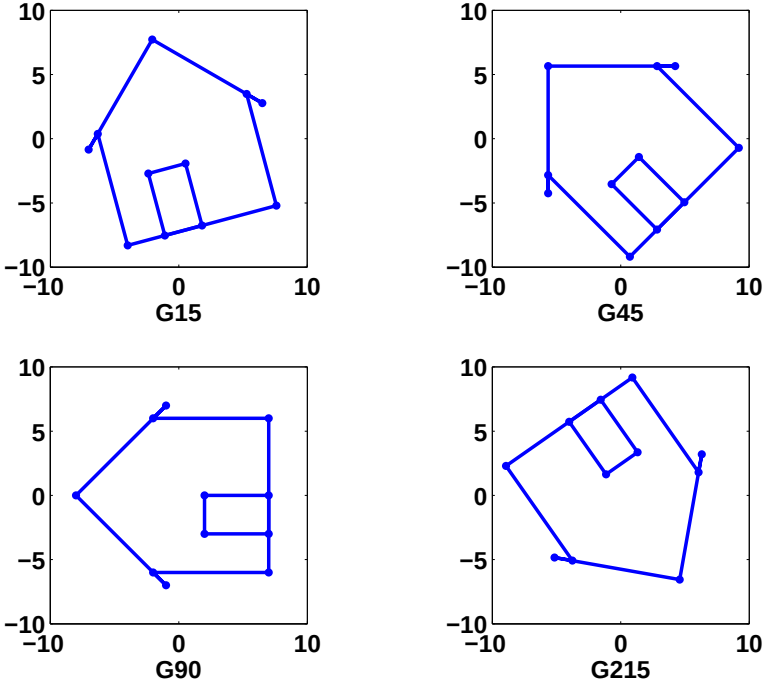

Figure 5.5. *The affect of multiplication by plane rotations though* 15°, 45°, 90°, *and* 215°.

```
G15 =
    0.9659   -0.2588
    0.2588    0.9659
```

```
G45 =
    0.7071   -0.7071
    0.7071    0.7071

G90 =
     0    -1
     1     0

G215 =
   -0.8192    0.5736
   -0.5736   -0.8192
```

You can see that `G45` is fairly close to the random matrix `R2` seen earlier and that its effect on the house is similar.

MATLAB generates a plane rotation for angles measured in radians with

```
G = [cos(theta) -sin(theta); sin(theta) cos(theta)]
```

and for angles measured in degrees with

```
G = [cosd(theta) -sind(theta); sind(theta) cosd(theta)]
```

Our `exm` toolbox function `wiggle` uses `dot_to_dot` 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)]
   dot_to_dot(G*X);
   t = t + delta;
end
```

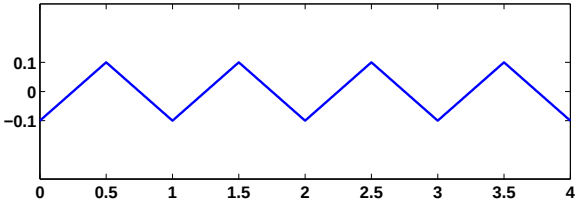

Figure 5.6. *Wiggle angle θ*

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 5.6. Each value of θ produces a corresponding plane rotation $G(\theta)$. Then

```
dot_to_dot(G*X)
```

applies the rotation to the input matrix `X` and plots the wiggling result.

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 5.4. 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.

Exercises

5.1 *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 $A^T A = AA^T$?

5.2 *Inverse.* Let

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

Find a matrix X so that $AX = I$.

5.3 *Powers.* Let

$$A = \begin{pmatrix} 0.99 & 0.01 \\ -0.01 & 1.01 \end{pmatrix}$$

What is A^n?

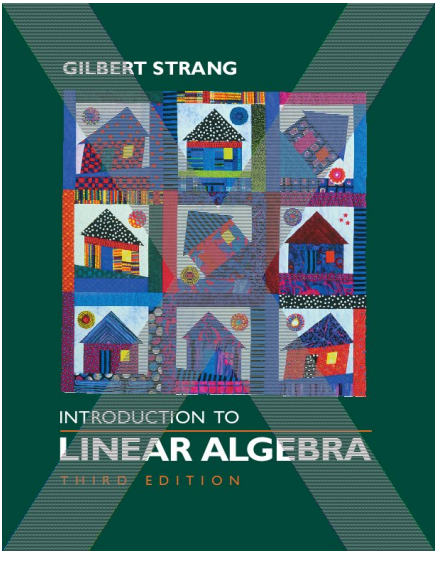

Figure 5.7. **** Update to third edition and give credit for the quilt. ****

5.4 *Powers.* Let

$$A = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}$$

What is A^n?

5.5 *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 $A = A^T$?

5.6 *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.

5.7 *Givens rotations.*
(a) What is the determinant of $G(\theta)$?
(b) Explain why $G(\theta)^2 = G(2\theta)$.
(c) Explain why $G(\theta)^n = G(n\theta)$.

5.8 X^8. Find a real 2-by-2 matrix X so that $X^8 = -I$.

5.9 G^T. What is the effect on points in the plane of multiplication by $G(\theta)^T$?

5.10 $\widehat{G}$. (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?

5.11 *Goldie.* What does the function `goldie` in the `exm` toolbox do?

5.12 *Transform a hand.* Repeat the experiments in this chapter with

```
X = hand
```

instead of

```
X = house
```

Figure 5.8 shows

```
dot_to_dot(hand)
```

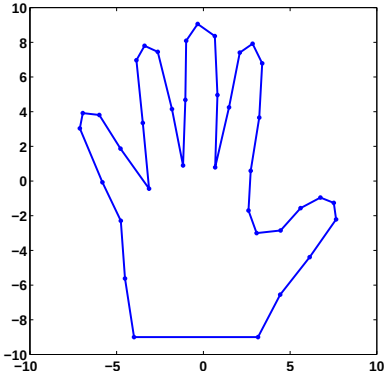

Figure 5.8. *A hand.*

5.13 *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

```
dot_to_dot(H)
dot_to_dot(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
```

5.14 *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 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)
```


August 8, 2009

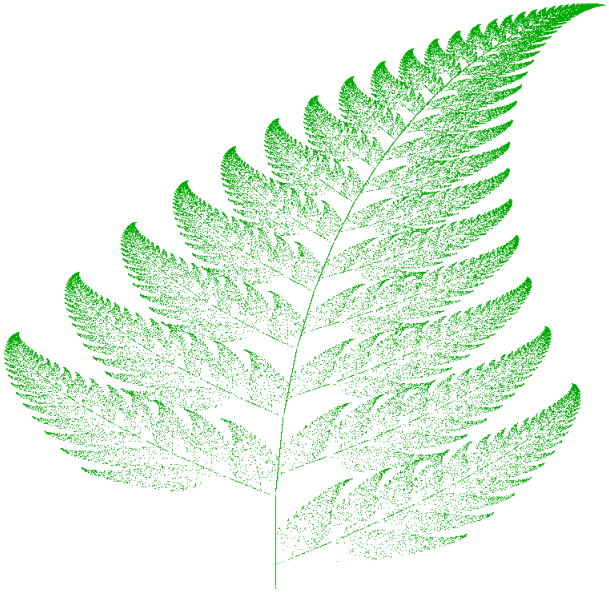

Figure 6.1. *Fractal fern.*

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
```

```
t = toc;
s = sprintf('%8.0f points in %6.3f seconds',cnt,t);
text(-1.5,-0.5,s,'fontweight','bold');
set(stop,'style','pushbutton','string','close', ...
  'callback','close(gcf)')
```

Let's examine this program a few statements at a time.

```
shg
```

stands for "show graph window." It brings an existing graphics window forward, or creates a new one if necessary.

```
clf reset
```

resets most of the figure properties to their default values.

```
set(gcf,'color','white','menubar','none', ...
   'numbertitle','off','name','Fractal Fern')
```

changes the background color of the figure window from the default gray to white and provides a customized title for the window.

```
x = [.5; .5];
```

provides the initial coordinates of the point.

```
h = plot(x(1),x(2),'.');
```

plots a single dot in the plane and saves a *handle*, `h`, so that we can later modify the properties of the plot.

```
darkgreen = [0 2/3 0];
```

defines a color where the red and blue components are zero and the green component is two-thirds of its full intensity.

```
set(h,'markersize',1,'color',darkgreen,'erasemode','none');
```

makes the dot referenced by `h` smaller, changes its color, and specifies that the image of the dot on the screen should not be erased when its coordinates are changed. A record of these old points is kept by the computer's graphics hardware (until the figure is reset), but MATLAB itself does not remember them.

```
axis([-3 3 0 10])
axis off
```

specifies that the plot should cover the region

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

but that the axes should not be drawn.

```
stop = uicontrol('style','toggle','string','stop', ...
   'background','white');
```

creates a toggle user interface control, labeled `'stop'` and colored white, in the default position near the lower left corner of the figure. The handle for the control is saved in the variable `stop`.

```
drawnow
```

causes the initial figure, including the initial point, to actually be plotted on the computer screen.

The statement

```
p  = [ .85  .92  .99  1.00];
```

sets up a vector of probabilities. The statements

```
A1 = [ .85  .04; -.04  .85];  b1 = [0; 1.6];
A2 = [ .20 -.26;  .23  .22];  b2 = [0; 1.6];
A3 = [-.15  .28;  .26  .24];  b3 = [0; .44];
A4 = [  0    0 ;   0   .16];
```

define the four affine transformations. The statement

```
cnt = 1;
```

initializes a counter that keeps track of the number of points plotted. The statement

```
tic
```

initializes a stopwatch timer. The statement

```
while ~get(stop,'value')
```

begins a `while` loop that runs as long as the `'value'` property of the `stop` toggle is equal to 0. Clicking the `stop` toggle changes the value from 0 to 1 and terminates the loop.

```
r = rand;
```

generates a *pseudorandom* value between 0 and 1. The compound `if` statement

```
if r < p(1)
   x = A1*x + b1;
elseif r < p(2)
   x = A2*x + b2;
elseif r < p(3)
   x = A3*x + b3;
else
   x = A4*x;
end
```

picks one of the four affine transformations. Because `p(1)` is `0.85`, the first transformation is chosen 85% of the time. The other three transformations are chosen relatively infrequently.

```
set(h,'xdata',x(1),'ydata',x(2));
```

changes the coordinates of the point `h` to the new (x_1, x_2) and plots this new point. But `get(h,'erasemode')` is `'none'`, so the old point also remains on the screen.

```
cnt = cnt + 1;
```

counts one more point.

```
drawnow
```

tells MATLAB to take the time to redraw the figure, showing the new point along with all the old ones. Without this command, nothing would be plotted until `stop` is toggled.

```
end
```

matches the `while` at the beginning of the loop. Finally,

```
t = toc;
```

reads the timer.

```
s = sprintf('%8.0f points in %6.3f seconds',cnt,t);
text(-1.5,-0.5,s,'fontweight','bold');
```

displays the elapsed time since `tic` was called and the final count of the number of points plotted. Finally,

```
set(stop,'style','pushbutton','string','close', ...
   'callback','close(gcf)')
```

changes the control to a push button that closes the window.

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 M-file `finitefern.m` 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 kth 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 *JPEG.* Use the following code, available in `fernjpeg.m`, to make your own JPEG file from the fern. Then compare your image with one obtained from `exm/fern.jpg`.

```
bg = [0 0 85];      % Dark blue background
fg = [255 255 255]; % White dots
sz = get(0,'screensize');
rand('state',0)
X = finitefern(500000,sz(4),sz(3));
d = fg - bg;
R = uint8(bg(1) + d(1)*X);
G = uint8(bg(2) + d(2)*X);
B = uint8(bg(3) + d(3)*X);
F = cat(3,R,G,B);
imwrite(F,'myfern.jpg','jpg');
```

6.8 *Sierpinski's triangle.* Modify `fern.m` or `finitefern.m` so that it produces *Sier-*

pinski's triangle. Start at

$$x = \begin{pmatrix} 0 \\ 0 \end{pmatrix}.$$

At each iterative step, the current point x is replaced with $Ax+b$, where the matrix A is always

$$A = \begin{pmatrix} 1/2 & 0 \\ 0 & 1/2 \end{pmatrix}$$

and the vector b is chosen at random with equal probability from among the three vectors

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

Chapter 7

Magic Squares

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

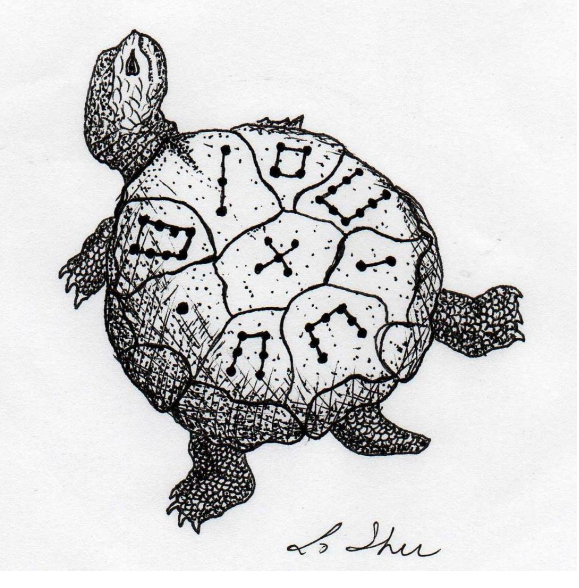

Figure 7.1. *Lo Shu. (Thanks to Byerly Wiser Cline.)*

August 8, 2009

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 7.1. 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 MATLAB's 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
```

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.

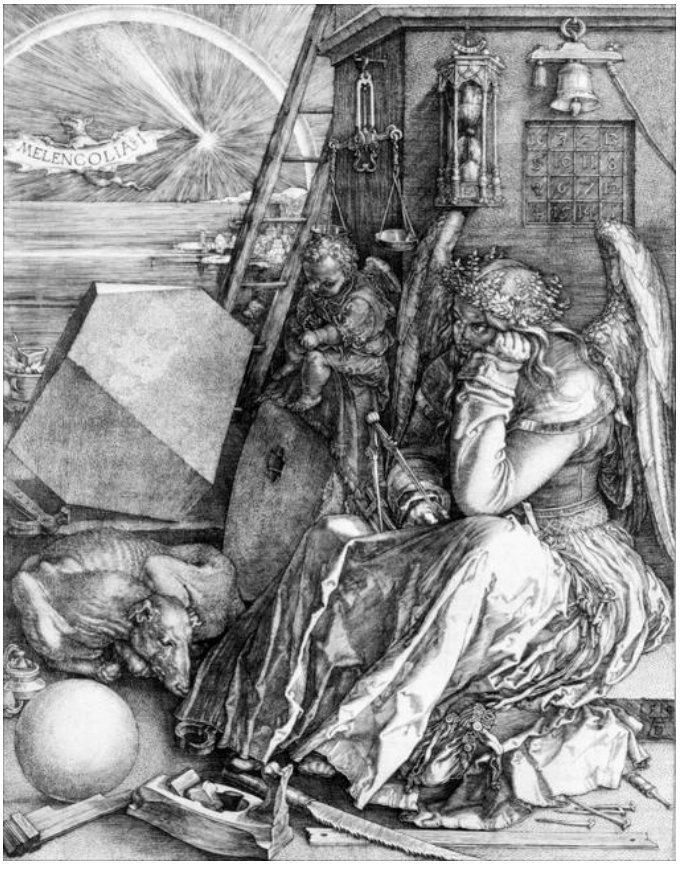

Figure 7.2. *Albrect Dürer's Melencolia, 1514.*

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 7.2. Better yet, issue these MATLAB commands

```
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

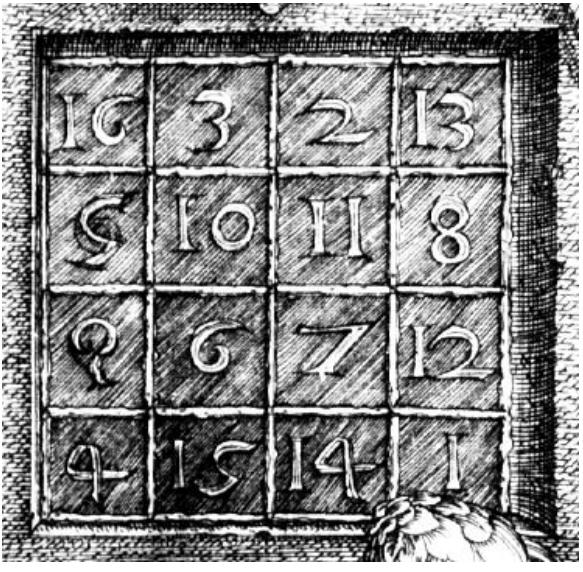

Figure 7.3. *Detail from Melencolia.*

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

display the higher resolution scan of the area around the magic square that we have in figure 7.3.

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 sequentually 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 northeasternly 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.

```
[  A      A+50
  A+75    A+25 ]
```

The column sums are all equal because `sum(A) + sum(A+75)` equals `sum(A+50) + sum(A`
But the rows 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
```

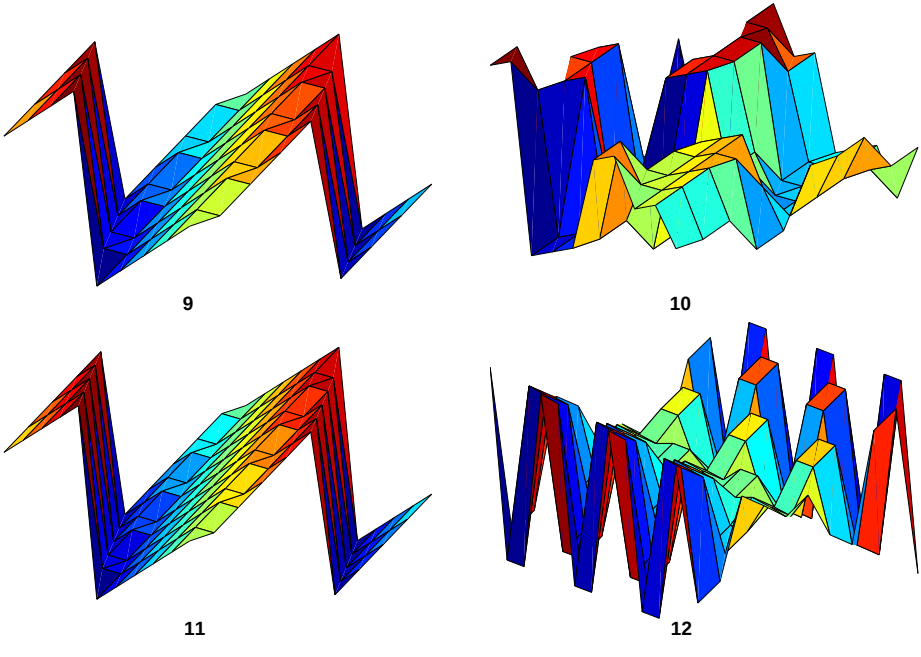

Figure 7.4. *Surf plots of magic squares of order 9, 10, 11,12.*

Let's conclude this chapter with some graphics. Figure 7.4 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 of 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.

Holger Danielsson
`http://www.magic-squares.de/magic.html`
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`

Exercises

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

7.2 *Magic sum.* Show that

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

7.3 *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?

7.4 *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.

7.5 *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.

7.6 *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 oppositve 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 $\mu(5) = 65$.

```
     .     .     .    14     .
     .     .     .     .     3
    17     .     .     .     .
     .     6     .     .     .
     .     .    25     .     .
```

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 `a0` and `b0`,

```
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 `a0` and `b0`,

```
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;
```

7.7 *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.

7.8 *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 7.5 will help. You can see

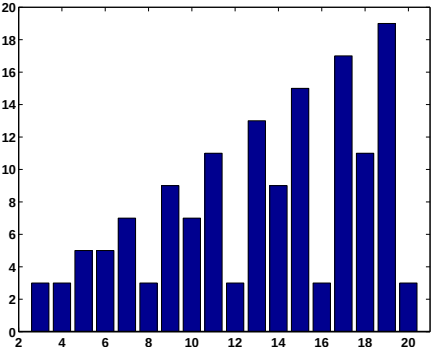

Figure 7.5. *Rank of magic squares.*

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 8

TicTacToe Magic

Three simple games are related in a surprising way.

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

August 8, 2009

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	:	2 6 7 8
Blue	:	1 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 8.1 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 8.1. *Starting position for Pick15.*

Figure 8.2 shows the position for our example game after two moves from each player. Green now has to choose 6 to block Blue.

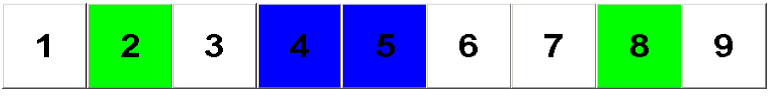

Figure 8.2. *Position after two moves from each player.*

Figure 8.3 shows the final position for our example game. Green has won with 2+6+7 = 15.

Figure 8.3. *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 8.6 shows typical output.

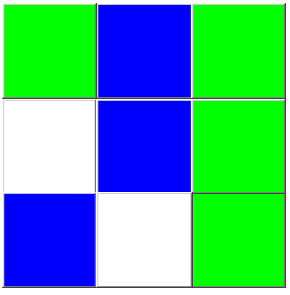

Figure 8.4. *Typical output from a game of TicTacToe. Green has won with the third column.*

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 TicTacToe played on a magic square. The rows, columns and main diagonals of the magic square provide all possible ways of having three distict numbers that sum to 15. Winning moves in Pick15 correspond to winning moves in TicTacToe. All three games are actually the same game with different displays.

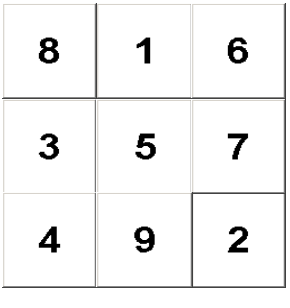

Figure 8.5. *Initial configuration for a game of Magic3.*

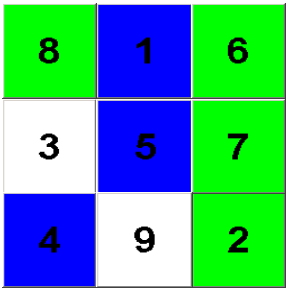

Figure 8.6. *Green has won with 6+7+2 = 15 in the third column.*

Exercises

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

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

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

8.4 *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?

8.5 *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 9

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 neigbhors, 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

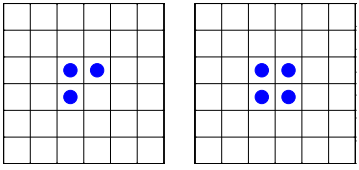

Figure 9.1. *A pre-block and a block.*

and reflexive symmetries, there are only two different possibilities – the population is either L-shaped or I-shaped. The left half of figure 9.1 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 9.1. 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.

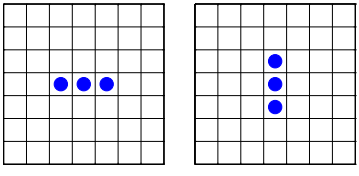

Figure 9.2. *A blinker blinking.*

The other three-cell initial population is I-shaped. The two possible orientations are shown in each half of figure 9.2. 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 other half of the figure. 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 9.3. 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 9.4 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.

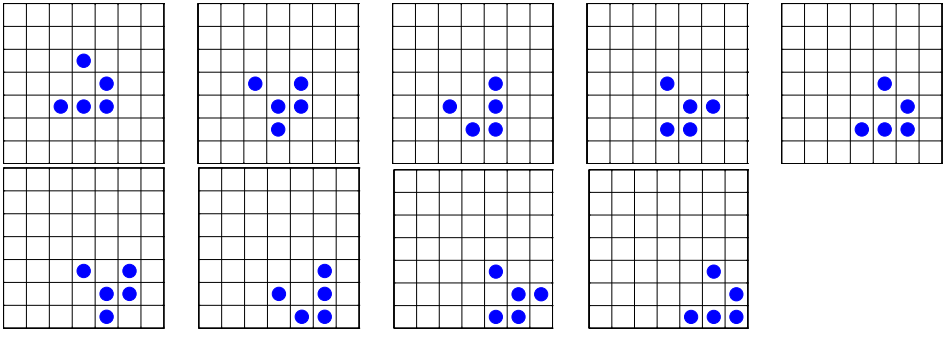

Figure 9.3. *A glider gliding.*

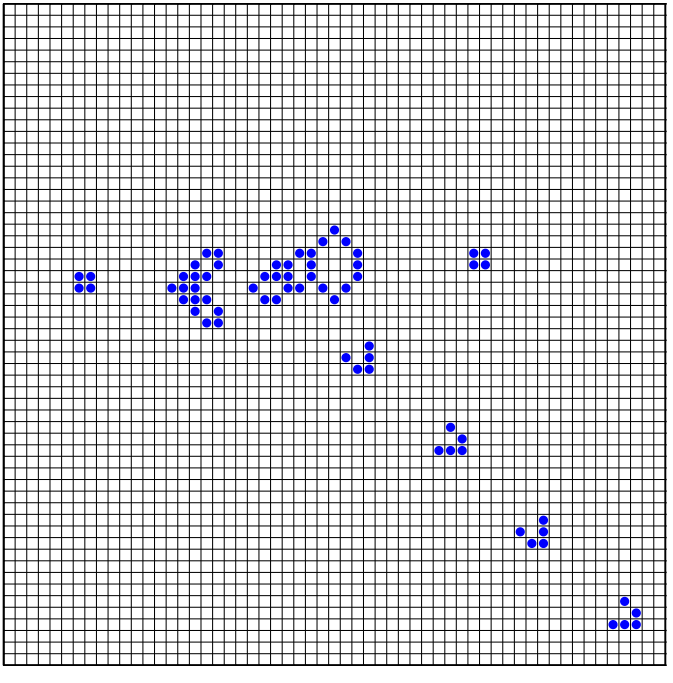

Figure 9.4. *Gosper's glider gun.*

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 `nnz(X)` – the number of nonzero

elements in `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 9.4 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 `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

```
p = [n 1:n-1];
q = [2:n 1];
```

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

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

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
```

Exercises

9.1 *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)
```

9.2 *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
```

9.3 *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')`.

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

9.5 *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 9.5 and can be generated by

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

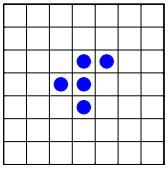

Figure 9.5. *The R-pentomino.*

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?

9.6 *Exection time.* Display actual computer execution time by adding `tic` and `toc` to `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.

9.7 *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?

9.8 *Houses.* Check out

```
lifex(houses)
```

9.9 *Checkerboards.* This code generates an n-by-n checkboard 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 checkboard initial populations with $3 <= n <= 30$?

Chapter 10

Mandelbrot Set

Fractals, topology, complex arithmetic and fascinating computer graphics.

Benoit Mandelbrot is 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 first 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 consisting of the values z_0 for which the trajectories defined by

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

remain bounded at $k \to \infty$. It is easy to see that this set is simply the unit disc, $|z_0| <= 1$, shown in figure 10.1. If $|z_0| <= 1$, the repeated squares remain bounded. If $|z_0| > 1$, the repeated squares are unbounded. The boundary of the unit disc is the unit circle, $|z_0| = 1$. There is nothing very difficult or exciting here.

The definition is the Mandelbrot set is only slightly more complicated. It involves repeatedly adding in the initial point. The Mandelbrot set is the region in the complex plane consisting of the values z_0 for which the trajectories defined by

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

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

August 8, 2009

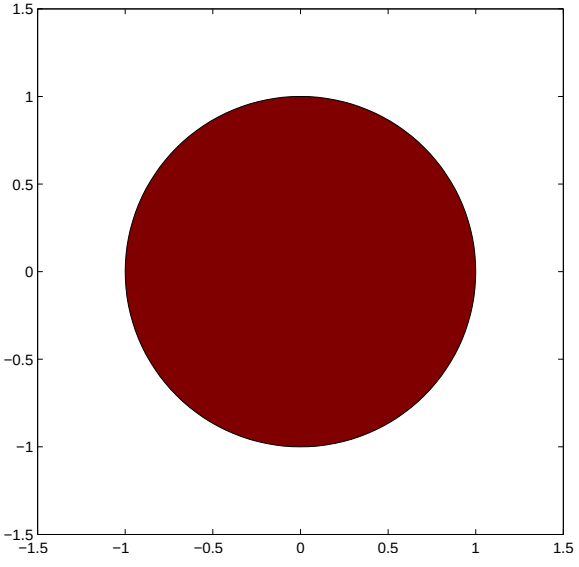

Figure 10.1. *The unit disc is shown in red. The boundary is simply the unit circle. There is no intricate fringe.*

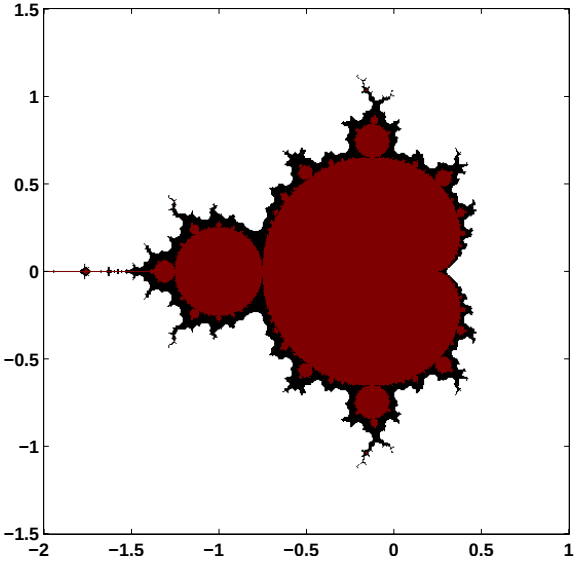

Figure 10.2. *The Mandelbrot set is shown in red. The fringe just outside the set, shown in black, is a region of rich structure.*

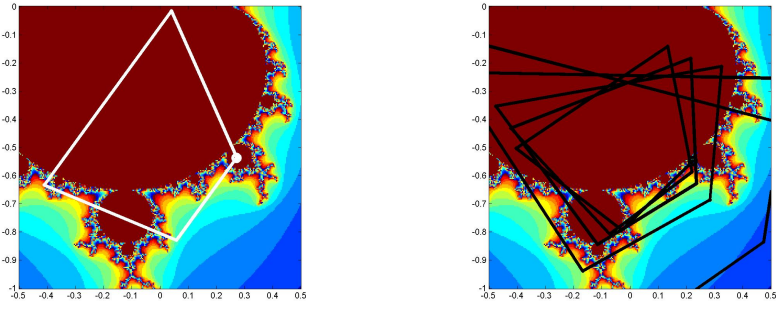

Figure 10.3. *Two trajectories. z0 = .25-.54i generates a cycle of length four, while nearby z0 = .22-.54i generates an unbounded trajectory.*

Figure 10.2 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
```

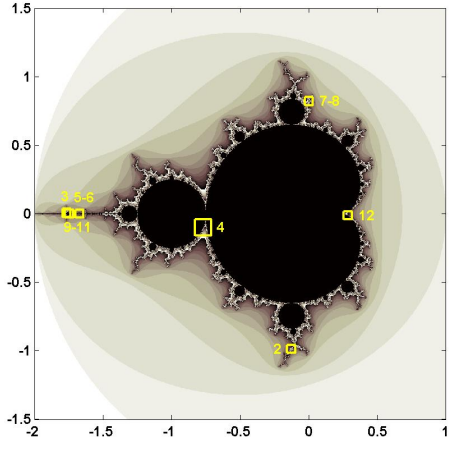

Figure 10.4. *The figures in this chapter, and the predefined regions in our mandelbrot program, show these regions in the fringe just outside the Mandelbrot set.*

```
-0.4377 - 0.5659i
 0.1213 - 0.0446i
 0.2627 - 0.5508i
   ...
```

This cycle repeats forever. The trajectory remains bounded. This tells us that the starting value value, `z0 = .25-.54i`, is in the Mandelbrot set. The same cycle is shown in the left half of figure 10.3.

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
```

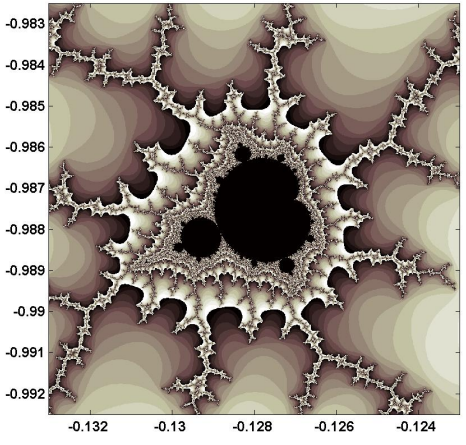

Figure 10.5. *Region #2, "Mini Mandelbrot". The fringe around the Mandelbrot set in self similar. Small versions of the set appear at all levels of magnification.*

```
 -0.3841 - 0.5039i
  0.1136 - 0.1529i
  0.2095 - 0.5747i
...
```

Then, after 24 iterations,

```
    ...
  1.5708 - 1.1300i
  1.4107 - 4.0899i
-14.5174 -12.0794i
 6.5064e+001 +3.5018e+002i
-1.1840e+005 +4.5568e+004i
```

The trajectory is blowing up rapidly. After a few more iterations, the floating point numbers overflow. So this `z0` is not in the Mandelbrot set. The same unbounded trajectory is shown in the right half of figure 10.3. We see that the first value, `z0 = .25-.54i`, is in the Mandelbrot set, while the second value, `z0 = .22-.54i`, which is nearby, is not.

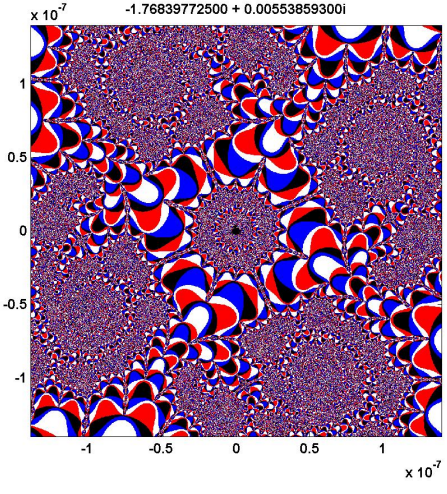

Figure 10.6. *Region #3, "Plaza", with the "flag" color map.*

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

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

subsequent iterations will essential square the value of $|z|$ and it will behave like 2^{2^k}.

Try it yourself. Put these statements on one line.

```
z0 = ...
z = 0;
while abs(z) < 2
  z = z^2+z0;
  disp(z),
end
```

Use the up arrow and backspace keys to retrieve the statement and change `z0` to different values near `.25-.54i`. If you have to hit `<ctrl>-c` to break out of an infinite loop, then `z0` is in the Mandelbrot set. If the `while` condition is eventually `false` and the loop terminates without your help, then `z0` is not in the set.

The number of iterations required for z to escape the disc of radius 2 provides the basis for showing the detail in the fringe. Let's add an iteration counter to the

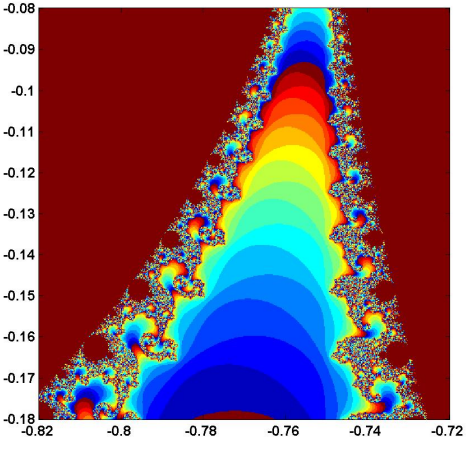

Figure 10.7. *Region #4. "Valley of the Seahorses".*

loop and make the code into a function, `M(z0)` 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.

```
function k = M(z0)
z = 0;
k = 0;
while abs(z) < 2 && k < depth
  z = z^2+z0;
  k = k + 1;
end
```

The maximum value of `M(z0)` is `depth`. If the value of `M(z0)` is less than depth, then `z0` is outside the set. Large values of `M(z0)` indicate that `z0` is in the fringe, close to the boundary. If `M(z0)` reaches `depth` then `z0` is declared to be inside the Mandelbrot set.

MATLAB array operations allow us to run this loop on an large grid of complex numbers simultaneously and accumulate an array of iteration counters. Most of the time is spent squaring complex numbers and then counting the number whose absolute value exceeds two.

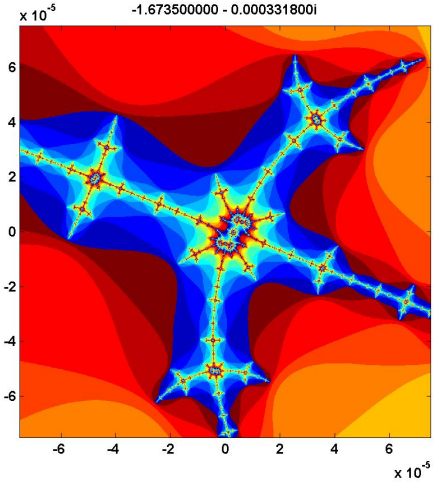

Figure 10.8. *Region #5. Our "West Wing" is located just off the real axis in the thin far western portion of the set, near real(z) = -1.67.*

Here is a small table of `M(z0)` as `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

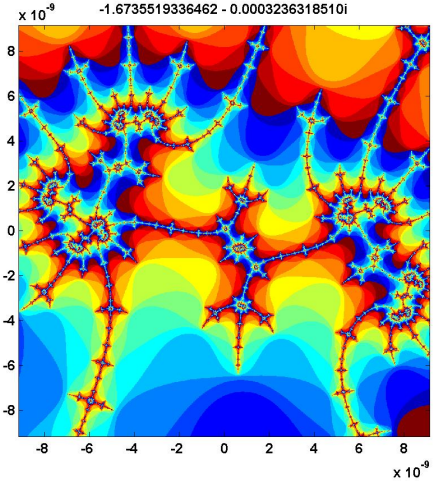

Figure 10.9. *Region #6. "Dueling Dragons".*

are now capped at `512` might increase.

The function values `M(z0)` 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 10.2 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.

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

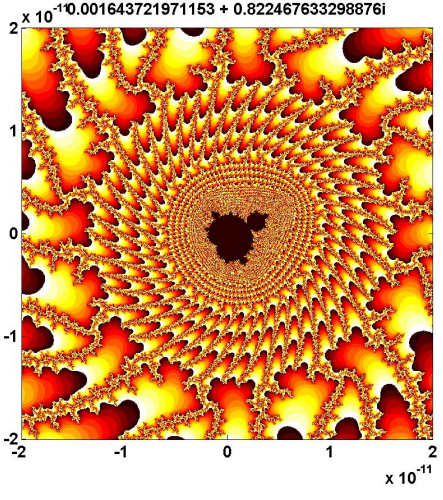

Figure 10.10. *Region #7. The "Buzzsaw" requires a magnification factor of* 10^{11} *to reveal a miniature copy of the Mandelbrot set.*

```
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 with index `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)
```

This generates figure 10.2, with the `jets` color map. On my laptop, the computation takes less than one second.

A simple estimate of the execution time is proportional to

```
grid^2 * depth
```

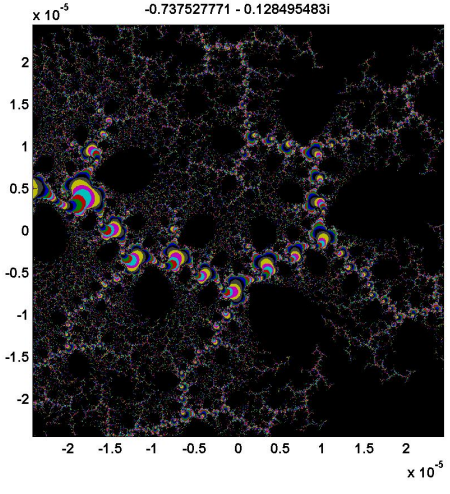

Figure 10.11. *Region #8. "Nebula". Interstellar dust.*

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 half of that.

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.

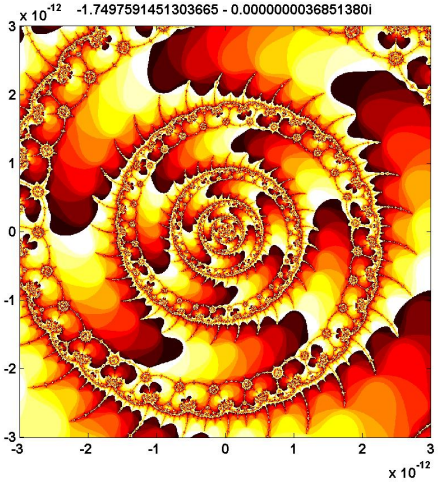

Figure 10.12. *Region #9. A vortex, not far from the West Wing. Zoom in on one of the circular "microbugs" near the left edge.*

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. A `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`, and

`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 10.2.

`exit`. Close the gui.

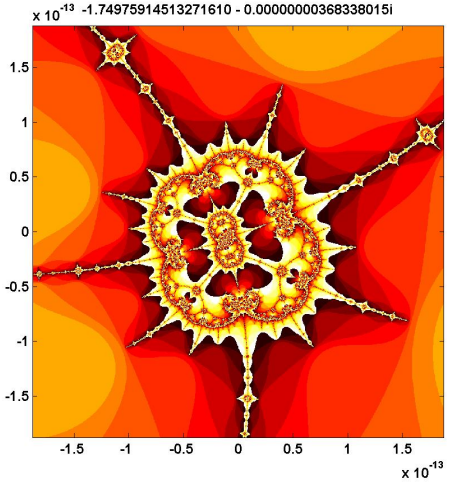

Figure 10.13. *Region #10. A* 10^{13} *magnification factor reveals a "microbug" within the vortex.*

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 10.7 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 10.8 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)
```

The Mandelbrot set is *self similar*. Small regions in the fringe reveal features that are similar to the original set. One of the best known examples, shown in figure 10.10, is called the "Buzzsaw". It can be generated with

```
mandelbrot(0.001643721971153+0.822467633298876i, ...
   4.0e-11,1024,2048,2)
```

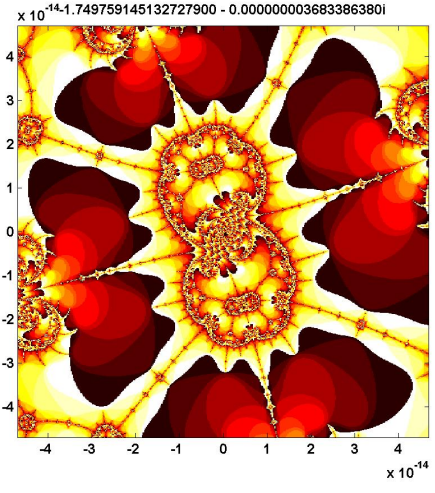

Figure 10.14. *Region #11. The nucleus of the microbug.*

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 of your computer, the Buzzsaw is smaller than the individual transistors in your machine's microprocessor.

We call figure 10.11 "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 10.15, the "Geode". It involves a fine grid and a large value of `depth` and consequently requires a few minutes to compute.

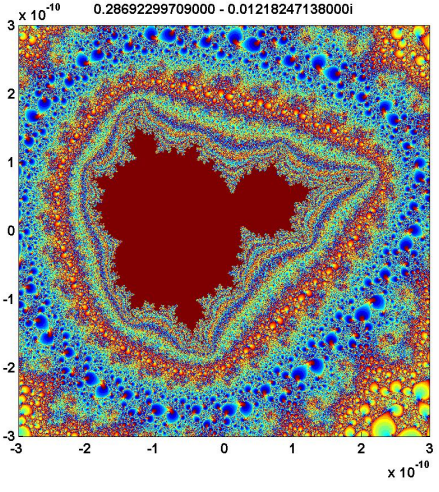

Figure 10.15. *Region #12. "Geode". This colorful image requires a 2048-by-2048 grid and depth = 8192.*

```
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.

Exercises

10.1 *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.

10.2 *M(z0).* Reproduce our table of `M(z0)` for `z0` for `z0 = x+i*y` where `x = .205:.005:` and `y = -.520:-.005:-.560`. First, use `depth = 512`. Then use larger values of `depth` and see which table entries change.

10.3 *p-th power.* Change the iteration to

$$z_{k+1} = z_k^p + z_0$$

for some fixed $p \neq 2$. Better yet, add a button that lets you set p.

10.4 *Color maps.* Add more color maps.

10.5 When the width of the region gets to be smaller than about 10^{-15}, our `mandelbrot` gui does not work very well. Why?

10.6 *Spin the color map.* 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.

Chapter 11

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

$$0x = b$$

August 8, 2009

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 give us the solution to our story problem – apples cost 29 cents each, bananas are a nickel each, and cantaloupes are 89 cents each.

Very rarely, systems of linear equations come in the form

$$xA = b$$

where b and x are row vectors. In this case, the solution is found using the forward slash operator.

```
x = b/A
```

The two operations `A\b` and `b/A` are sometimes called *left* and *right* matrix division. In both cases, the coefficient matrix is in the "denominator". For scalars, left and right division are the same thing. The quantities `7\21` and `21/7` are both equal to `3`.

Let's change our story problem a bit. Assume now that Carol buys six apples and one cantaloupe for $2.77. The coefficient matrix and right hand side become

```
A =
     3    12     1
    12     0     2
     6     0     1
```

and

```
b =
    2.36
    5.26
    2.77
```

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

```
x = A\b
```

produces

```
Warning: Matrix is singular to working precision.
x =
  NaN
 -Inf
  Inf
```

`Inf` and `-Inf` stand for plus and minus infinity and result from division of nonzero numbers by zero. `NaN` stands for "Not-a-Number" and results from doing arithmetic involving infinities.

The source of the difficulty is that the new information about Carol's purchase is inconsistent with the earlier information about Alice's and Bob's purchases. We have said that Carol bought exactly half as much fruit as Bob. But she paid 2.77 when half of Bob's payment would have been only 2.63. The third row of `A` is equal to one-half of the second row, but b(3) is not equal to one-half of b(2). For this particular matrix A and vector b, the solution to the linear system of equations $Ax = b$ does not exist.

What if we make Carol's purchase price consistent with Bob's? We leave `A` unchanged and revise `b` with

```
b(3) = 2.63
```

so

```
b =
    2.36
    5.26
    2.63
```

Now we do not have enough information. Our last two equations are scalar multiples of each other.

$$\begin{aligned} 12x_1 + 2x_3 &= 5.26 \\ 6x_1 + x_3 &= 2.63 \end{aligned}$$

We can pick an arbitrary value for x_1 or x_3, use either of these equations to compute the value we didn't pick, and then use the first equation to compute x_2. The result is a solution to all three equations. One possible solution is the solution to the original problem.

```
x =
    0.29
    0.05
    0.89
```

But another solution is

```
x =
    0.09
       0
    2.09
```

There are infinitely many more. In this case, for this particular matrix A and vector b, the solution to $Ax = b$ is not unique.

This illustrates two fundamental facts about technical computing.

- The hardest things to compute are things that do not exist.
- The next hardest things to compute are things that are not unique.

Exercises

11.1 *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)

$$\begin{pmatrix} 1 & 2 \\ 3 & 6 \end{pmatrix} x = \begin{pmatrix} 1 \\ 3 \end{pmatrix}$$

11.2 *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)

$$\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix} x = \begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix}$$

11.3 *Null vector.* Find a nonzero solution x to

$$\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix} x = \begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix}$$

11.4 Matrix equations. Backslash can be used to solve matrix equations of the form

$$AX = B$$

where B has several colums. Do you recognize the solution to the following equation?

$$\begin{pmatrix} 53 & -52 & 23 \\ -22 & 8 & 38 \\ -7 & 68 & -37 \end{pmatrix} X = \begin{pmatrix} 360 & 0 & 0 \\ 0 & 360 & 0 \\ 0 & 0 & 360 \end{pmatrix}$$

Chapter 12

Google PageRank

The world's largest matrix computation.

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 jth column shows the links on the jth page. The number of nonzeros in G is the total number of hyperlinks in W.

August 8, 2009

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 jth 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

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

Notice that A comes from scaling the connectivity matrix by its column sums. The jth column is the probability of jumping from the jth page to the other pages on the Web. If the jth 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,

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

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

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

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)\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 12.1 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 *URL*s. 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.

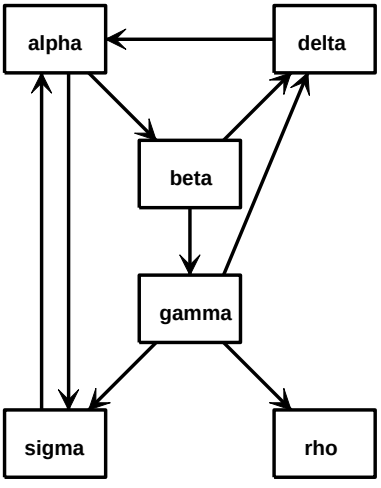

Figure 12.1. *A tiny Web.*

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
```

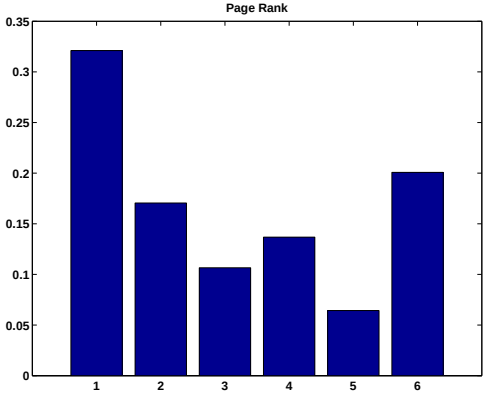

Figure 12.2. *Page Rank for the tiny Web*

The `bar` graph of `x` is shown in figure 12.2. 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.xxx.zzz',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.

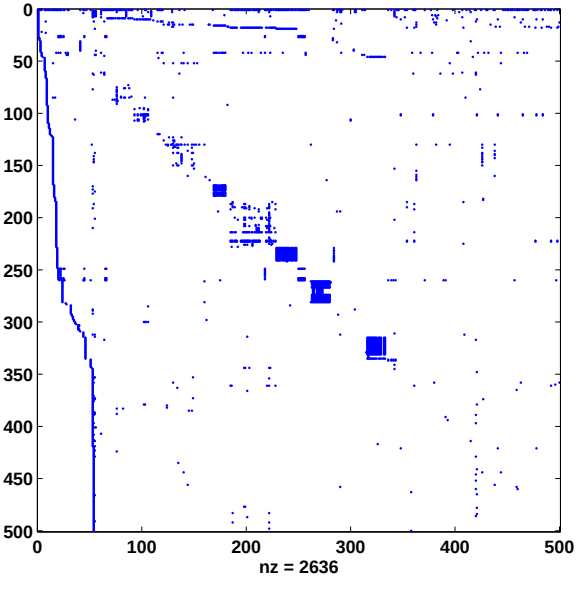

Figure 12.3. *Spy plot of the* `harvard500` *graph.*

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 12.3) that shows the nonzero structure of the connectivity matrix. The statement

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

computes page ranks, produces a bar graph (Figure 12.4) of the ranks, and prints the most highly ranked URLs in PageRank order.

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

```
     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 and institutes, including the Kennedy School of Government, the 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.

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 [**?**].

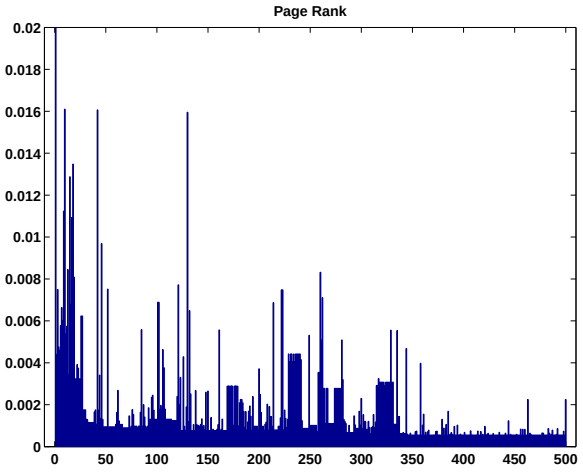

Figure 12.4. *PageRank of the* `harvard500` *graph.*

Exercises

12.1 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?

12.2 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.

12.3 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.

12.4 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?

12.5 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.

12.6 Figure 12.5 is the graph of another six-node subset of the Web. In this example, there are two disjoint subgraphs.

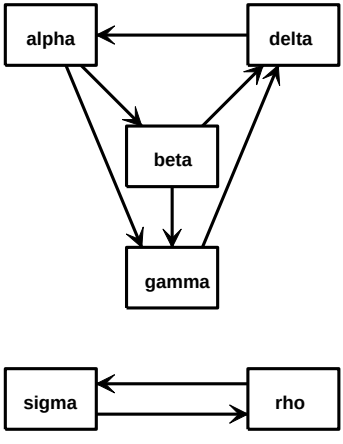

Figure 12.5. *Another tiny Web.*

(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 $p \rightarrow 1$.

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

12.8 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 13

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 travelling 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 13.1.

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

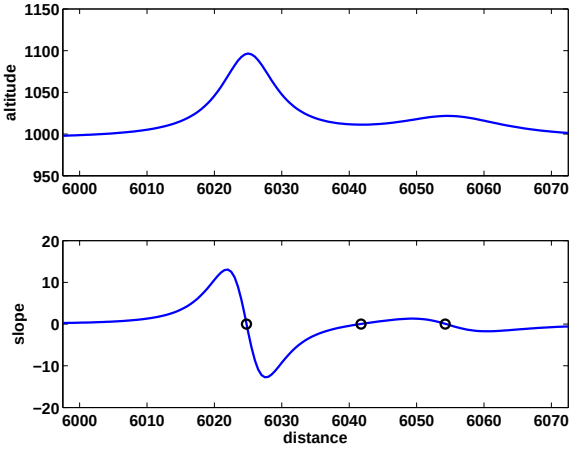

Figure 13.1. *Altitude along a mountain road, and derivative of that altitude. The derivative is zero at the local maxima and minima of the altitude.*

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 travelled, This is the derivative dy/dx. You could plot a graph of dy/dx, like the second plot in figure 13.1, 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 13.2 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
```

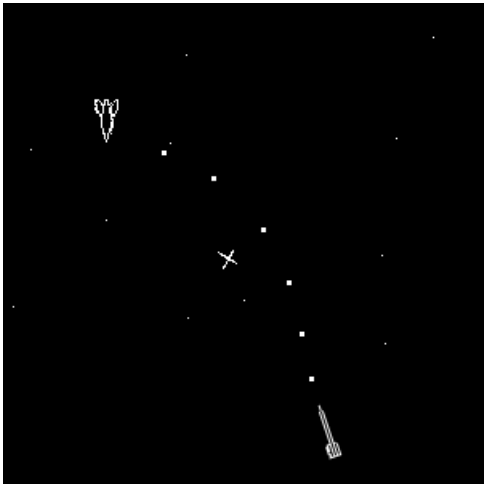

Figure 13.2. *Spacewar, the world's first video game. The gravitational pull of the central star causes the torpedo to move in an elliptical orbit.*

```
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
   x = x + h*y;
```

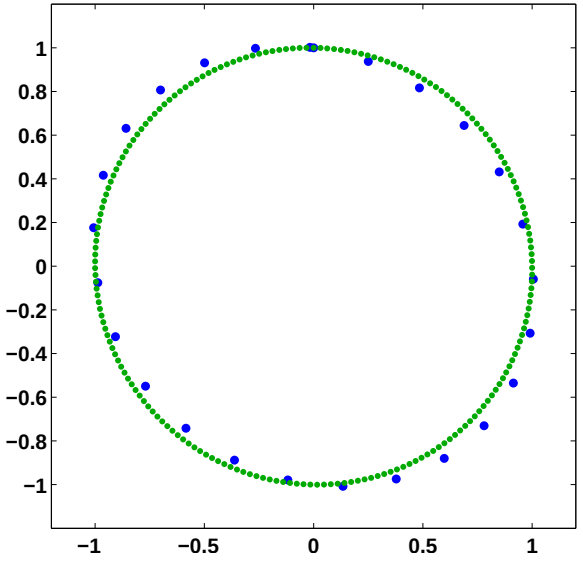

Figure 13.3. *The 25 blue points are generated by the Spacewar orbit generator with a step size of 1/4. The 201 green points are generated with a step size of 1/32.*

```
    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 13.3. 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 familar 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 s. 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

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

The function F always takes two arguments, the scalar independent variable, t, and the vector of dependent variables, s. A program that evaluates $F(t, s)$ 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.

$$\begin{aligned} s(t) &= \begin{pmatrix} x(t) \\ y(t) \end{pmatrix} \\ &= \begin{pmatrix} s_1(t) \\ s_2(t) \end{pmatrix} \end{aligned}$$

The function F defines the velocity.

$$\begin{aligned} \dot{s}(t) &= \begin{pmatrix} \dot{s}_1(t) \\ \dot{s}_2(t) \end{pmatrix} \\ &= \begin{pmatrix} \dot{x}(t) \\ \dot{y}(t) \end{pmatrix} \\ &= \begin{pmatrix} y(t) \\ -x(t) \end{pmatrix} \\ &= \begin{pmatrix} s_2(t) \\ -s_1(t) \end{pmatrix} \end{aligned}$$

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

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

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

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

- `F`, the function defining the differential equations,
- `tspan`, the vector specifying the integration interval,
- `s0`, the vector of initial conditions.

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

```
function sdot = circle(t,s)
x = s(1);
y = s(2);
xdot = y;
ydot = -x;
sdot = [xdot; ydot];
```

Here is a shorter, but less descriptive, version of `circle.m`.

```
function sdot = circle(t,s)
sdot = [s(2); -s(1)];
```

Notice that both of these functions have two input arguments, `t` and `s`, even though the output in this example does not depend upon `t`.

If either of these function definitions is stored in `circle.m`, the following code calls `ode23` to compute the solution over the interval $0 \le t \le 2\pi$, starting with $x(0) = 0$ and $y(0) = 1$.

```
tspan = [0 2*pi];
s0 = [0; 1];
ode23(@circle,tspan,s0)
```

With no output arguments, the ode solvers automatically plot the solutions. Figure 13.4 is the result for our example. The small circles in the plot are not equally spaced. They show the points chosen by the step size algorithm.

To produce the phase plot plot shown in figure 13.5, capture the output and plot it yourself.

```
tspan = [0 2*pi];
s0 = [0; 1];
[t,s] = ode23(@circle,tspan,s0)
plot(s(:,1),s(:,2))
axis square
```

The circle generator example is so simple that we can bypass the creation of an M-file and write the function in one line.

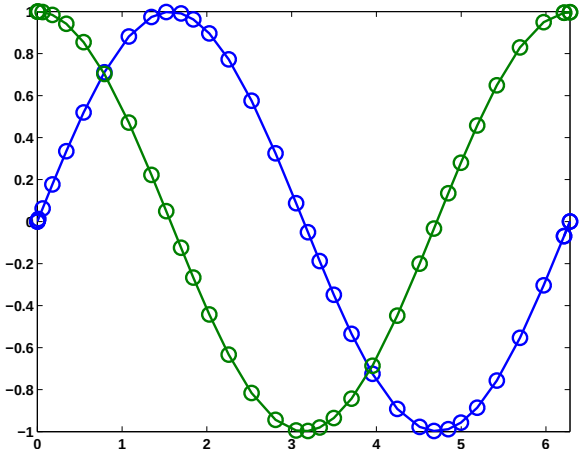

Figure 13.4. *Graphs of sine and cosine generated by ode23.*

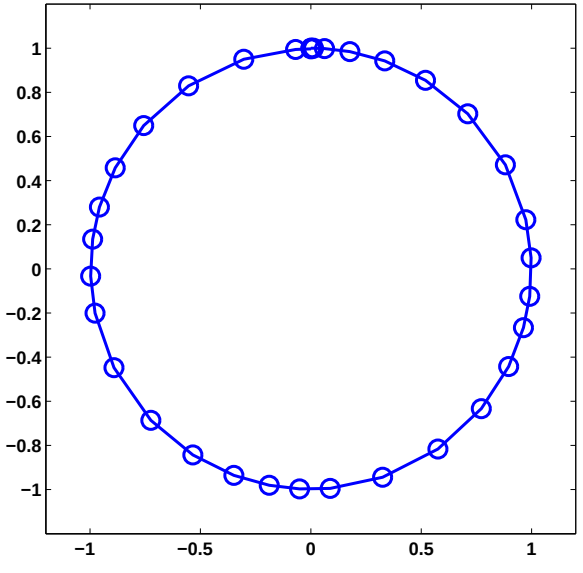

Figure 13.5. *Graph of a circle generated by ode23.*

```
F = @(t,s) [s(2); -s(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 `F`. Since the "`@`" sign is included in the definition of `F`, you don't need it when you call an ode solver.

```
[t,s] = ode23(F,tspan,s0)
```

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 1024 equal sized pieces. This is 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 ode1(F,tspan,s0)
t0 = tspan(1);
tfinal = tspan(end);
h = (tfinal - t0)/1024;
t = t0
s = s0
while t < tfinal
   sdot = F(t,s);
   t = t + h
   s = s + h*sdot
end
```

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`.

Exercises

13.1 *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.

13.2 *Divided differences.* Create your own graphic like our figure 13.1. 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
```

13.3 *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
```

13.4 *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
```

13.5 $\dot{s} = As$. Write the system of differential equations

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

in matrix-vector form,

$$\dot{s} = As$$

where s is a vector-valued function of time,

$$s(t) = \begin{pmatrix} x(t) \\ y(t) \end{pmatrix}$$

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

13.6 *Example from ode23.* Using our notation, the first example in the documentation for `ode23` is

$$\begin{aligned} \dot{s}_1 &= s_2 s_3 \\ \dot{s}_2 &= -s_1 s_3 \\ \dot{s}_3 &= -0.51 s_1 s_2 \end{aligned}$$

with initial conditions

$$\begin{aligned} s_1(0) &= 0 \\ s_2(0) &= 1 \\ s_3(0) &= 1 \end{aligned}$$

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

```
doc ode23
```

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

$$\begin{aligned} \dot{x} &= y^3 \\ \dot{y} &= -x^3 \end{aligned}$$

with initial conditions

$$x(0) = 0$$
$$y(0) = 1$$

on the interval

$$0 \le t \le 7.4163$$

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

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

$$\dot{x} = y^5$$
$$\dot{y} = -x^5$$

with initial conditions

$$x(0) = 0$$
$$y(0) = 1$$

on an interval

$$0 \le t \le T$$

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

$$x(T) = 0$$
$$y(T) = 1$$

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

$$\dot{x} = y^2$$
$$\dot{y} = -x^2$$

Why do solutions of

$$\dot{x} = y^p$$
$$\dot{y} = -x^p$$

have such different behavior if p is odd or even?

Chapter 14

Exponential Function

The function e^t.

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

$$y = 2^t$$

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

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

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.

$$2^{1/2} = \sqrt{2} = 1.4142...,$$

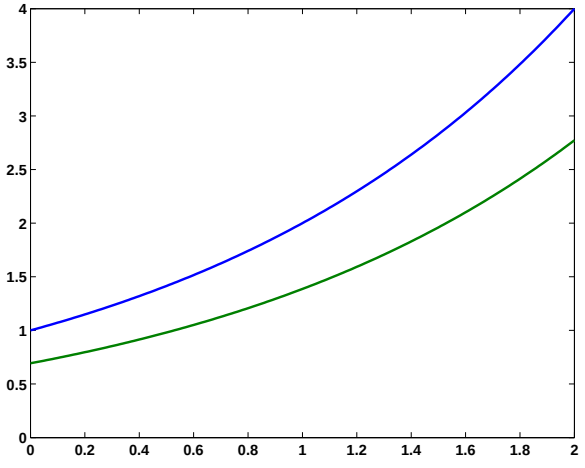

Figure 14.1. *The blue curve is the graph of* $y = 2^t$. *The green curve is the graph of the rate of growth,* $\dot{y} = dy/dt$.

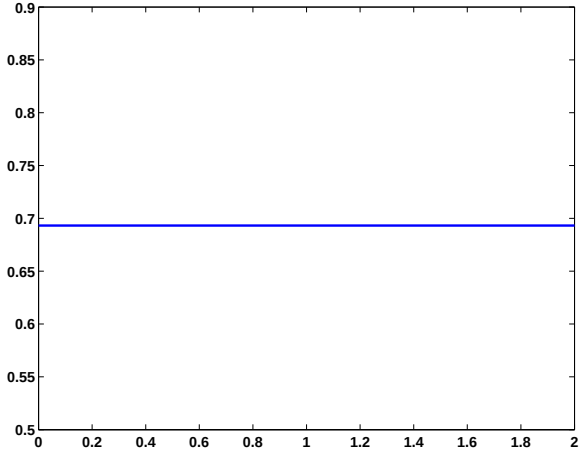

Figure 14.2. *The ratio,* $\dot{y}/y$.

$$2^{5/3} = \sqrt[3]{2^5} = 3.1748...,$$
$$2^{355/113} = \sqrt[113]{2^{355}} = 8.8250...$$

In principal, for floating point arithmetic, this is all we need to know. All floating point numbers are ratios of two integers. We do not have to be concerned yet about the definition of 2^t for irrational t. If MATLAB can compute powers and roots, we can plot the graph of 2^t, the blue curve in figure 14.1

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

```
plot(t,ydot./y)
```

We see that the ratio of the derivative to the function, shown in figure 14.2, has a constant value, $\dot{y}/y = 0.6932...$, that does not depend upon t .

Now, if you are following along with a live MATLAB, repeat the preceeding calculations with $y = 3^t$ instead of $y = 2^t$. You should find that the ratio is again independent of t. This time $\dot{y}/y = 1.0986....$ Better yet, experiment with `expgui`.

If we take any value a and look at $y = a^t$, we find that, numerically at least, the ratio $\dot{y}/y$ is constant. In other words, $\dot{y}$ is proportional to y. If $a = 2$, the proportionality constant is less than one. If $a = 3$, the proportionality constant is greater than one. Can we find an a so that $\dot{y}/y$ is actually equal to one? If so, we have found a function that is equal to its own derivative.

The approximate derivative of the function $y(t) = a^t$ is

$$\dot{y}(t) = \frac{a^{t+h} - a^t}{h}$$

This can be factored and written

$$\dot{y}(t) = \frac{a^h - 1}{h} a^t$$

So the ratio of the derivative to the function is

$$\frac{\dot{y}(t)}{y(t)} = \frac{a^h - 1}{h}$$

The ratio depends upon h, but not upon t. If we want the ratio to be equal to 1, we need to find a so that

$$\frac{a^h - 1}{h} = 1$$

Solving this equation for a, we find

$$a = (1 + h)^{1/h}$$

The approximate derivative becomes more accurate as h goes to zero, so we are interested in the value of

$$(1 + h)^{1/h}$$

as h approaches zero. This involves taking numbers very close to 1 and raising them to very large powers. The surprising fact is that this limiting process defines a number that turns out to be one of the most important quantities in mathematics

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

Here is the beginning and end of a table of values generated by repeatedly cutting h in half.

```
format long
format compact
h = 1;
while h > 2*eps
    h = h/2;
    e = (1 + h)^(1/h);
    disp([h e])
end
```

```
0.500000000000000   2.250000000000000
0.250000000000000   2.441406250000000
0.125000000000000   2.565784513950348
0.062500000000000   2.637928497366600
0.031250000000000   2.676990129378183
0.015625000000000   2.697344952565099
      ...                 ...
0.000000000000014   2.718281828459026
0.000000000000007   2.718281828459036
0.000000000000004   2.718281828459040
0.000000000000002   2.718281828459043
0.000000000000001   2.718281828459044
0.000000000000000   2.718281828459045
```

The last line of output involves a value of `h` that is not zero, but is so small that it prints as a string of zeros. We are actually computing

$$(1 + 2^{-51})^{2^{51}}$$

which is

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

The result gives us the numerical value of e correct to 16 significant decimal digits. It's easy to remember the repeating pattern of the first 10 significant digits.

$$e = 2.718281828...$$

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

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

Here is the *Binomial Theorem*.

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

If n is an integer, this terminates after $n+1$ terms with b^n. But if n is not an integer, the expansion is an infinite series. Apply the binonimial theorem with $a = 1$, $b = h$ and $n = t/h$.

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

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

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

This series is a rigorous mathematical definition that applies to any t, positive or negative, rational or irrational, real or complex. The $n+1$-st term is $t^n/n!$. As n increases, the t^n in the numerator is eventually overwhelmed by the $n!$ in the denominator, so the terms go to zero fast enough that the infinite series converges.

It is almost possible to use this power series for actual computation of e^t. Here is an experimental MATLAB program.

```
function s = expex(t)
% EXPEX  Experimental version of EXP(T)
s = 1;
term = 1;
n = 0;
r = 0;
while r ~= s
   r = s;
   n = n + 1;
   term = (t/n)*term;
   s = s + term;
end
```

Notice that there are no powers or factorials. Each term is obtained from the previous one using the fact that

$$\frac{t^n}{n!} = \frac{t}{n}\frac{t^{n-1}}{(n-1)!}$$

The potentially infinite loop is terminated when `r == s`, that is when the floating point values of two successive partial sums are equal.

There are "only" two things wrong with this program – its speed and its accuracy. The terms in the series increase as long as $|t/n| \geq 1$, then decrease after n reaches the point where $|t/n| < 1$. So if $|t|$ is not too large, say $|t| < 2$, everything is OK; only a few terms are required and the sum is computed accurately. But

larger values of t require more terms and the program requires more time. This is not a very serious defect if t is real and positive. The series converges so rapidly that the extra time is hardly noticeable.

However, if t is real and negative the computed result may be inaccurate. The terms alternate in sign and cancel each other in the sum to produce a small value for e^t. Take, for example, $t = -20$. The true value of e^{-20} is roughly $2 \cdot 10^{-9}$. Unfortunately, the largest terms in the series are $(-20)^{19}/19!$ and $(-20)^{20}/20!$, which are opposite in sign and both of size $4 \cdot 10^7$. There is 16 orders of magnitude difference between the size of the largest terms and the size of the final sum. With only 16 digits of accuracy, we lose everything. The computed value obtained from `expex(-20)` is completely wrong.

For real, negative t it is possible to get an accurate result from the power series by using the fact that

$$e^t = \frac{1}{e^{-t}}$$

For complex t, there is no such easy fix for the accuracy difficulties of the power series.

In contrast to its more famous cousin, 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 = expex(1)
```

to compute an approximate value for e. Only seventeen terms are required to get floating point accuracy.

```
e = 2.718281828459045
```

If we revise the code to compute the numerator and denominator separately using integer arithmetic, we find a rational approximation to e with 17! in the denominator.

$$\mathrm{e} = \frac{966858672404690}{355687428096000}$$

After computing `e`, you could then use `e^t`, but `exp(t)` is preferable.

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 $\log_2(y)$.

The term *exponential growth* is often used informally to describe any kind of rapid growth. Mathematically, the term refers to any time evolution, $y(t)$, where the rate of growth is proportional to the quantity itself.

$$\dot{y} = ky$$

The solution to this equation is determined for all t by specifying the value of y at one particular t, usually $t = 0$.

$$y(0) = y_0$$

Then

$$y(t) = y_0 e^{kt}$$

Suppose, at time $t = 0$, we have a million *E. coli* bacteria in a test tube under ideal laboratory conditions. Twenty minutes later each bacterium has fissioned to produce another one. So at $t = 20$, the population is two million. Every 20 minutes the population doubles. At $t = 40$, it's four million. At $t = 60$, it's eight million. And so on. The population, measured in millions of cells, $y(t)$, is

$$y(t) = 2^{t/20}$$

Let $k = \ln 2/20 = .0347$. Then, with t measured in minutes and the population $y(t)$ measured in millions, we have

$$\dot{y} = ky, \quad y(0) = 1$$

Consequently

$$y(t) = e^{kt}$$

This is exponential growth, but it cannot go on forever. Eventually, the growth rate is affected by the size of the container. Initially at least, the size of the population is modelled by the exponential function.

Suppose, at time $t = 0$, you invest \$1000 in a savings account that pays 5% interest, compounded yearly. A year later, at $t = 1$, the bank adds 5% of \$1000 to your account, giving you $y(1) = 1050$. Another year later you get 5% of 1050, which is 52.50, giving $y(2) = 1102.50$. If $y(0) = 1000$ is your initial investment, $r = 0.05$ is the yearly interest rate, t is measured in years, and h is the step size for the compound interest calculation, we have

$$y(t + h) = y(t) + rhy(t)$$

What if the interest is compounded monthly instead of yearly? At the end of the each month, you get .05/12 times your current balance added to your account. The

same equation applies, but now with $h = 1/12$ instead of $h = 1$. Rewrite the equation as

$$\frac{y(t+h) - y(t)}{h} = ry(t)$$

and let h tend to zero. We get

$$\dot{y}(t) = ry(t)$$

This defines interest compounded *continuously*. The evolution of your investment is described by

$$y(t) = y(0)e^{rt}$$

Here is a MATLAB program that tabulates the growth of \$1000 invested at 5% over a 20 year period , with interest compounded yearly, monthly, and continuously.

```
format bank
r = 0.05;
y0 = 1000;
for t = 0:20
   y1 = (1+r)^t*y0;
   y2 = (1+r/12)^(12*t)*y0;
   y3 = exp(r*t)*y0;
   disp([t y1 y2 y3])
end
```

The first few and last few lines of output are

```
 t        yearly       monthly     continuous
 0       1000.00       1000.00       1000.00
 1       1050.00       1051.16       1051.27
 2       1102.50       1104.94       1105.17
 3       1157.63       1161.47       1161.83
 4       1215.51       1220.90       1221.40
 5       1276.28       1283.36       1284.03
..       .......       .......       .......
16       2182.87       2221.85       2225.54
17       2292.02       2335.52       2339.65
18       2406.62       2455.01       2459.60
19       2526.95       2580.61       2585.71
20       2653.30       2712.64       2718.28
```

Compound interest actually qualifies as exponential growth, although with modest interest rates, most people would not use that term.

Let's borrow money to buy a car. We'll take out a \$20,000 car loan at 10% per year interest, make monthly payments, and plan to pay off the loan in 3 years. What is our monthly payment, p? Each monthly transaction adds interest to our

current balance and subtracts the monthly payment.

$$\begin{aligned} y(t+h) &= y(t) + rhy(t) - p \\ &= (1+rh)y(t) - p \end{aligned}$$

Apply this repeatedly for two, three, then n months.

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

Solve for p

$$p = (1+rh)^n/((1+rh)^n - 1)rhy_0$$

Use MATLAB to evaluate this for our car loan.

```
y0 = 20000
r = .10
h = 1/12
n = 36
p = (1+r*h)^n/((1+r*h)^n-1)*r*h*y0
```

We find the monthly payment would be

```
p = 645.34
```

If we didn't have to pay interest on the loan and just made 36 monthly payments, they would be

```
y0/n

  = 555.56
```

It's hard to think about continuous compounding for a loan because we would have to figure out how to make infinitely many infinitely small payments.

Exercises

14.1 *e cubed.* The value of e^3 is close to 20. How close? What is the percentage error?

14.2 *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`?

14.3 *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?

14.4 *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)
```

14.5 *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 detioriates for large t, particularly negative t.

14.6 *Rational expex.* Make a version of `expex` that does not do any divisions, but returns two large integers that form the numerator and denominator of a rational approximation to e. Reproduce the result

$$\mathrm{e} = \frac{966858672404690}{355687428096000}$$

Chapter 15

Predators and Prey

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 a the population results a proportional increase or decrease in the number of births and deaths. Mathematically, this is described by the differential equation

$$\dot{y} = ky$$

The proportionality constant k relates the size of the population, $y(t)$, to its rate of growth, $\dot{y}(t)$. If k is positive, the population increases; if k is negative, the population decreases.

As we know, the solution to this equation is a function $y(t)$ that is proportional to the exponential function

$$y(t) = \eta e^{kt}$$

where $\eta = y(0)$.

This simple model is appropriate in the initial stages of growth when there are no restrictions or constraints on the population. A small sample of bacteria in a large Petri dish, for example. But in more realistic situations there are limits to growth, such as finite space or food supply. A more realistic model says that the population competes with itself. As the population increases, its growth rate decreases linearly. The differential equation is sometimes called the *logistic* equation.

$$\dot{y} = k(1 - \frac{y}{\mu})y$$

August 8, 2009

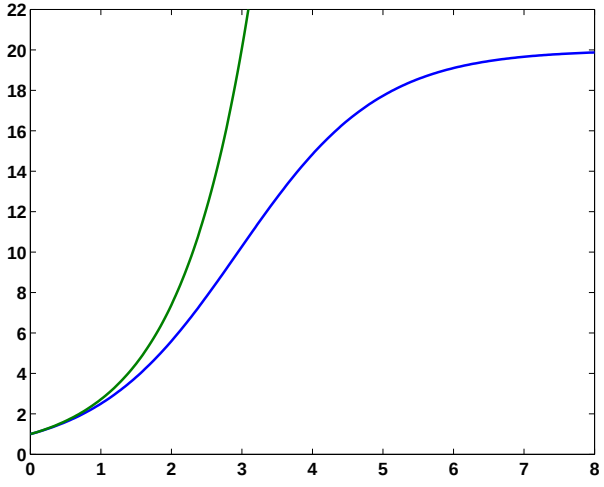

Figure 15.1. *Exponential growth and logistic growth.*

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 15.1 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 15.1 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 15.1.

```
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 isolatation. Everybody has a few enemies here and there. The Lotka-Volterra predator-prey model is the simplest description of competion between two species. Think of rabbits and foxes, or zebras and lions, or little fish and big fish.

The idea is that, if left to themselves with an infinite food supply, the rabbits or zebras would live happily and experience exponential population growth. On the other hand, if the foxes or lions were left with no prey to eat, they would die faster than they could reproduce, and would experience exponential population decline.

The predator-prey model is a pair of differential equations involving a pair of competing populations, $y_1(t)$ and $y_2(t)$. The growth rate for y_1 is a linear function of y_2 and vice versa.

$$\begin{aligned} \dot{y}_1 &= (1 - \frac{y_2}{\mu_2})y_1 \\ \dot{y}_2 &= -(1 - \frac{y_1}{\mu_1})y_2 \end{aligned}$$

We are using notation $y_1(t)$ and $y_2(t)$ instead of, say, $r(t)$ for rabbits and $f(t)$ for foxes, because our MATLAB program uses a two-component vector `y`.

The extra minus sign in the second equation distinguishes the predators from the prey. Note that if y_1 ever becomes zero, then

$$\dot{y}_2 = -y_2$$

and the predators are in trouble. But if y_2 ever becomes zero, then

$$\dot{y}_1 = y_1$$

and the prey population grows exponentially.

We have a formula for the solution of the single species logistic model. However it is not possible to express the solution to this predator-prey model in terms of exponential, trigonmetric, or any other elementary functions. It is necessary, but easy, to compute numerical solutions.

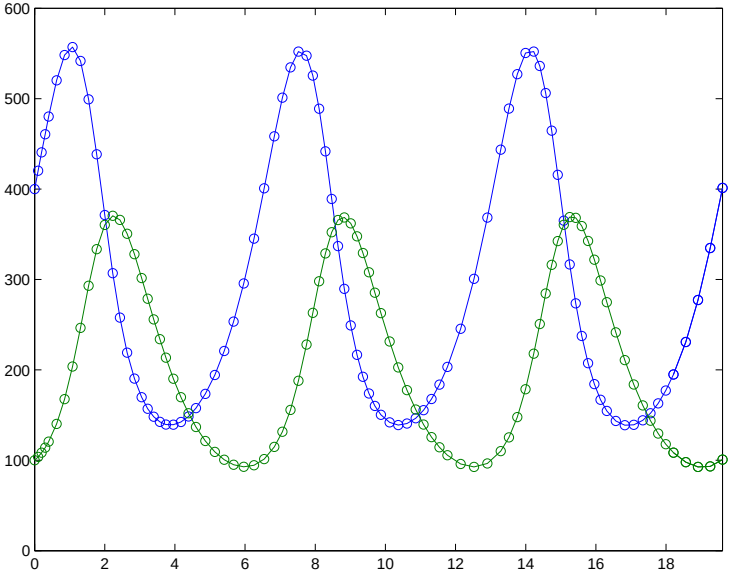

Figure 15.2. *A typical solution of the predator-prey equations.*

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 15.2.

```
mu = [300 200]'
eta = [400 100]'
sig = [1 -1]'
ppode = @(t,y) sig.*flipud(1-y./mu).*y
pit = 6.5357
ode45(ppode,[0 3*pit],eta)
```

There are two tricky parts of this code. MATLAB vector operations are used to define `ppode`, the predator-prey differential equations, in one line. And, the calculation that generates figure 15.3 provides the value assigned to `pit`. 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 15.2 show the points where `ode45` computes the solution. The plots look something like trig functions, but they're not. Notice that the curves near the minima are broader, and require more steps to compute, then the curves near the maxima. The plot of $\sin t$ would look the same at the top as the bottom.

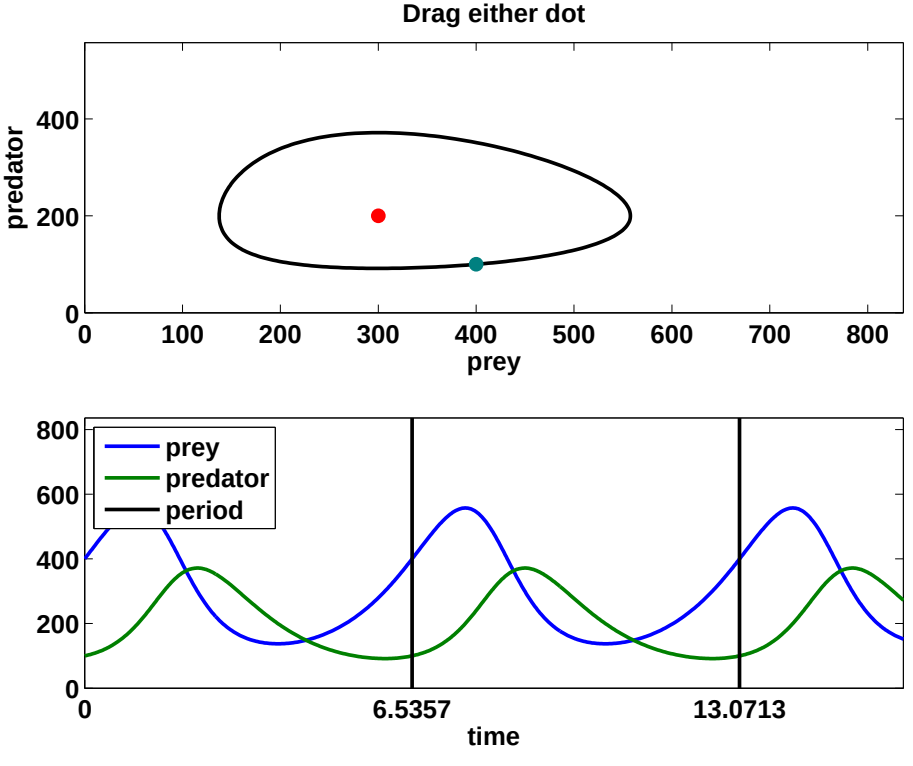

Figure 15.3. *The predprey experiment.*

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 15.3 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 15.2 and 15.3 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 `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 familar value. An exercise asks you to discover that value experimentally.

Exercises

15.1 *Plot.* Make a more few plots like figures 15.1 and 15.2, but with different values of the parameters k, η, and μ.

15.2 *Decay.* Compare exponential and logistic decay. Make a plot like figure 15.1 with negative k.

15.3 *Differentiate.* Verify that our formula for $y(t)$ actually satisfies the logistic differential equations.

15.4 *Easy as pie.* In `predprey`, if the red and blue-green dots are close to each other, then the length of the period is close to a familar value. What is that value? Does that value depend upon the actual location of the dots, or just their relative closeness?

15.5 *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?

15.6 *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 minamum some fraction of the period later. What is that fraction?

15.7 *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`.

15.8 *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?

15.9 *Limit growth.* Modify `predprey` to include a growth limiting term for the prey, similar to one in the logistic equation. Avoid another parameter by making the carrying capacity twice the initial value. The equations become

$$\begin{aligned} \dot{y}_1 &= (1 - \frac{y_1}{2\,\eta_1})(1 - \frac{y_2}{\mu_2})y_1 \\ \dot{y}_2 &= -(1 - \frac{y_1}{\mu_1})y_2 \end{aligned}$$

What happens to the shape of the solution curves? Are the solutions still periodic? What happens to the length of the period?

Chapter 16

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 propogation 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 momemtum. 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:

$$\frac{\partial h}{\partial t} + \frac{\partial (uh)}{\partial x} + \frac{\partial (vh)}{\partial y} = 0$$

$$\frac{\partial (uh)}{\partial t} + \frac{\partial (u^2 h + \frac{1}{2} g h^2)}{\partial x} + \frac{\partial (uvh)}{\partial y} = 0$$

$$\frac{\partial (vh)}{\partial t} + \frac{\partial (uvh)}{\partial x} + \frac{\partial (v^2 h + \frac{1}{2} g h^2)}{\partial x} = 0$$

August 8, 2009

In order to write the equations in a compact form, introduce three vectors.

$$U = \begin{pmatrix} h \\ uh \\ vh \end{pmatrix}$$

$$F(U) = \begin{pmatrix} uh \\ u^2h + \frac{1}{2}gh^2 \\ uvh \end{pmatrix}$$

$$G(U) = \begin{pmatrix} vh \\ uvh \\ v^2h + \frac{1}{2}gh^2 \end{pmatrix}$$

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 the earth's rotation, and possibly other external forces. But the equations we are considering here are still the basis of such models.

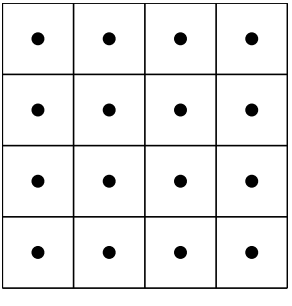

Figure 16.1. *At the beginning of a time step, the variables represent the solution at the centers of the finite difference grid.*

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 16.1. The quantity

$$U^n_{i,j}$$

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 16.2.

$$U^{n+\frac{1}{2}}_{i+\frac{1}{2},j} = \frac{1}{2}(U^n_{i+1,j} + U^n_{i,j}) - \frac{\Delta t}{2\Delta x}(F^n_{i+1,j} - F^n_{i,j})$$

$$U^{n+\frac{1}{2}}_{i,j+\frac{1}{2}} = \frac{1}{2}(U^n_{i,j+1} + U^n_{i,j}) - \frac{\Delta t}{2\Delta y}(G^n_{i,j+1} - G^n_{i,j})$$

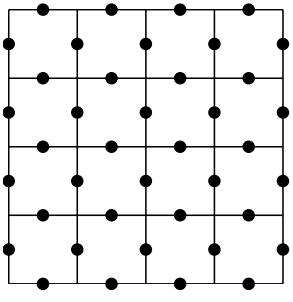

Figure 16.2. *The first stage computes values that represent the solution at the midpoints of the edges in the finite difference grid.*

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 16.1.

$$U^{n+1}_{i,j} = U^n_{i,j} - \frac{\Delta t}{\Delta x}(F^{n+\frac{1}{2}}_{i+\frac{1}{2},j} - F^{n+\frac{1}{2}}_{i-\frac{1}{2},j}) - \frac{\Delta t}{\Delta y}(G^{n+\frac{1}{2}}_{i,j+\frac{1}{2}} - G^{n+\frac{1}{2}}_{i,j-\frac{1}{2}})$$

Our MATLAB program, `exm/shallow_water`, 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 propogate back and forth over the region. A few snapshots of the dynamic graphic are shown in figure 16.3. The Lax-Wendroff scheme amplifies artificial, nonphysical oscillations. Eventually the numerical values overflow, producing floating point `Inf`s and `NaN`s, 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
```

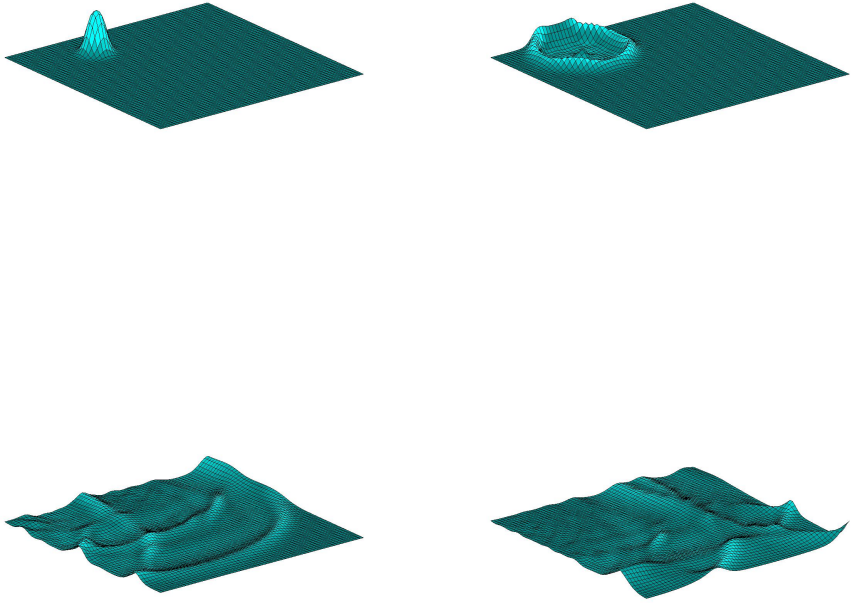

Figure 16.3. *A water drop initiates a wave that reflects off the boundary.*

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`.

```
[z0,h] = initialize_bouncer;
g = 9.8;
c = 0.75;
delta = 0.005;
v0 = 21;
while v0 >= 1
   v = v0;
   z = z0;
   while all(z >= 0)
      set(h,'zdata',z)
      drawnow
      v = v - delta*g;
      z = z + delta*v;
   end
   v0 = c*v0;
end
finalize_bouncer
```

The first statement

```
[z0,h] = initialize_bouncer;
```

generates the plot of a sphere shown in figure 17.1 and returns `z0`, the z-coordinates of the sphere, and `h`, the Handle Graphics "handle" for the plot. One of the exercises has you investigate the details of `initialize_bouncer`. The figure shows the situation at both the start and the end of the simulation. The ball is at rest and so the picture is pretty boring. To see what happens during the simulation, you have to actually run `bouncer`.

The next four statements in `bouncer.m` are

```
g = 9.8;
c = 0.75;
delta = 0.005;
v0 = 21;
```

These statrements set the values of the acceleration of gravity `g`, an elasticity coefficient `c`, the small time step `delta`, and the initial velocity for the ball, `v0`.

All the computation in `bouncer` is done within a doubly nested `while` loop. The outer loop involves the initial velocity `v0`.

```
while v0 >= 1
   ...
   v0 = c*v0;
end
```

To achieve the bouncing affect, the initial velocity is repeatedly multiplied by $c = 0.75$ until it is less than 1. Each bounce starts with a velocity equal to 3/4 of the previous one.

Within the outer loop, the statements

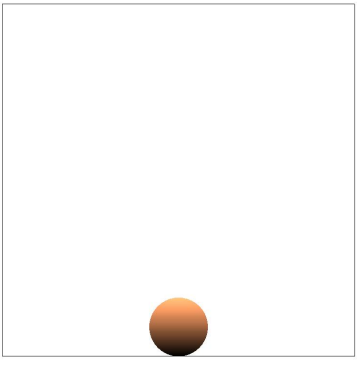

Figure 17.1. *Initial, and final, position of a bouncing ball. To see what happens in between, run bouncer.*

```
v = v0;
z = z0;
```

initialize the velocity `v` to `v0` and the height `z` to `z0`. Then the inner loop

```
while all(z >= 0)
   set(h,'zdata',z)
   drawnow
   v = v - delta*g;
   z = z + delta*v;
end
```

proceeds until the height goes negative. The plot is repeatedly updated to reflect the current height. At each step, the velocity `v` is decreased by a constant amount, `delta*g`, thereby affecting the gravitational deceleration. This velocity is then used to compute the change in the height `z`. As long as `v` is positive, the `z` increases with each step. When `v` reaches zero, the ball has reached its maximum height. Then `v` becomes negative and `z` decreases until the ball returns to height zero, terminating the inner loop.

After both loops are complete, the statement

```
finalize_bouncer
```

activates a pushbutton that offers you the possibility of repeating the simulation.

Brownian motion is not as obvious as gravity in our daily lives, but we do encounter it frequently. Albert Einstein's first important scientific paper was about Brownian motion. Think of particples of dust suspended in the air and illuminated

by a beam of sunlight. Or, diffusion of odors throughout a room. Or, a beach ball being tossed around a stadium by the spectators.

In Brownian motion an object – a dust particle, a molecule, or a ball – reacts to surrounding random forces. Our simulation of these forces uses the built-in MATLAB function `randn` to generate normally distributed random numbers. Each time the statement

```
randn
```

is executed a new, unpredictable, value is produced. The statement

```
randn(m,n)
```

produces an m-by-n array of random values. Each time the statement

```
hist(randn(100000,1),60)
```

is executed a histogram plot like the one in figure 17.2 is produced. Try executing this statement several times. You will see that different histograms are produced each time, but they all have the same shape. You might recognize the "bell-shaped curve" that is known more formally as the Gaussian or normal distribution. The histogram shows that positive and negative random numbers are equally likely and that small values are more likely than large ones. This distribution is the mathematical heart of Brownian motion.

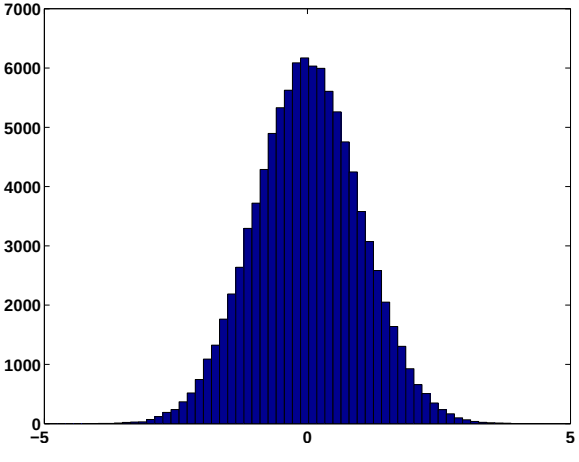

Figure 17.2. *Histogram of the normal random number generator.*

A simple example of Brownian motion known as a random walk is shown in figure 17.3. This is produced by the following code fragment.

```
m = 100;
x = cumsum(randn(m,1));
y = cumsum(randn(m,1));
```

```
plot(x,y,'.-')
s = 2*sqrt(m);
axis([-s s -s s]);
```

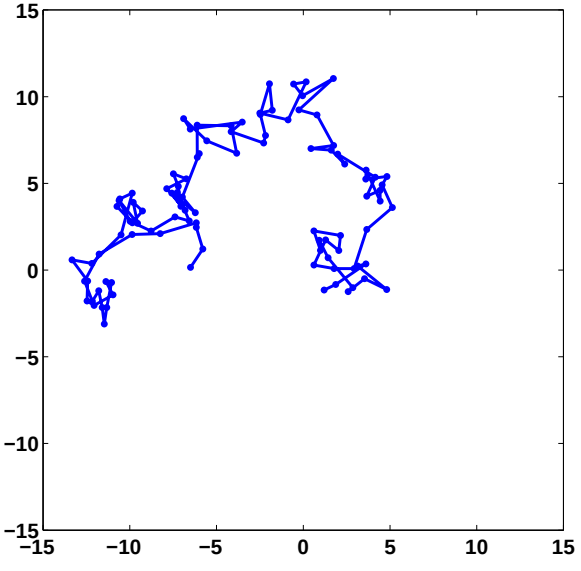

Figure 17.3. *A simple example of Brownian motion.*

The key statement is

```
x = cumsum(randn(m,1));
```

This statement generates the x-coordinates of the walk by forming the successive cumulative partial sums of the elements of the vector `r = randn(m,1)`.

$$
\begin{aligned}
x_1 &= r_1 \\
x_2 &= r_1 + r_2 \\
x_3 &= r_1 + r_2 + r_3 \\
&\ldots
\end{aligned}
$$

A similiar 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 particules in three dimensions, is available in `brownian3.m`. A snapshot of the evolving motion is shown in figure 17.4. Here is the core of `brownian3.m`.

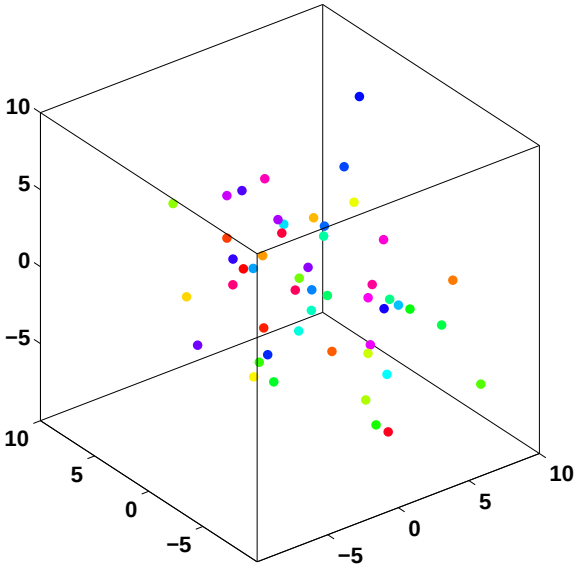

Figure 17.4. *A snapshot of the output from brownian3, showing simultaneous random walks of many particules in three dimensions.*

```
n = 50;  % Default number of particles
P = zeros(n,3);
H = initialize_graphics(P);

while ~get(H.stop,'value')

   % Obtain step size from slider.
   delta = get(H.speed,'value');

   % Normally distributed random velocities.
   V = randn(n,3);

   % Update positions.
   P = P + delta*V;

   update_plot(P,H);

end
```

The variable `n` is the number of particles. It is usually equal to 50, but some other number is possible with `brownian3(n)`. The array `P` contains the positions of `n` particles in three dimensions. Initially, all the particles are located at the origin, $(0, 0, 0)$. The variable `H` is a MATLAB structure containing handles for all the user

interface controls. In particular, `H.stop` refers to a toggle that terminates the `while` loop and `H.speed` refers to a slider that controls the speed through the value of the time step `delta`. The array `V` is an `n`-by-3 array of normally distributed random numbers that serve as the particle velocities in the random walks. Most of the complexity of `brownian3` is contained in the subfunction `initialize_graphics`. In addition to the speed slider and the stop button, the GUI has pushbuttons or toggles to turn on a trace, zoom in and out, and change the view point.

We are now ready to tackle the *n-body problem* in celestial mechanics. This is a model of a system of planets and their interaction described by Newton's laws of motion and gravitational attraction. Over five hundred years ago, Johannes Kepler realized that if there are only two planets in the model, the orbits are ellipses with a common focus at the center of mass of the system. This provides a fair description of the moon's orbit around the earth, or of the earth's orbit around the sun. But if you are planning a trip to the moon or a mission to Mars, you need more accuracy. You have to realize that the sun affects the moon's orbit around the earth and that Jupiter affects the orbits of both the earth and Mars. Furthermore, if you wish to model more than two planets, an analytic solution to the equations of motion is not possible. It is necessary to compute numerical approximations.

Our notation uses vectors and arrays. Let n be the number of bodies and, for $i = 1, \ldots, n$, let p_i be the vector denoting the position of the i-th body. For two-dimensional motion the i-th position vector has components (x_i, y_i). For three-dimensional motion its components are (x_i, y_i, z_i). The small system shown in figure 17.5 illustrates this notation. There are three bodies moving in two dimensions. The coordinate system and units are chosen so that initially the first body, which is gold if you have color, is at the origin,

$$p_1 = (0, 0)$$

The second body, which is blue, is one unit away from the first body in the x direction, so

$$p_2 = (1, 0)$$

The third body, which is red, is one unit away from the first body in the y direction, so

$$p_3 = (0, 1)$$

We wish to model how the position vectors p_i vary with time, t. The *velocity* of a body is the rate of change of its position and the *acceleration* is the rate of change of its velocity. We use one and two dots over p_i to denote the velocity and acceleration vectors, $\dot{p}_i$ and $\ddot{p}_i$. If you are familiar with calculus, you realize that the dot means differentiation with respect to t. For our three body example, the first body is initially heading away from the other two bodies, so its velocity vector has two negative components,

$$\dot{p}_1 = (-0.12, -0.36)$$

The initial velocity of the second body is all in the y direction,

$$\dot{p}_2 = (0, 0.72)$$

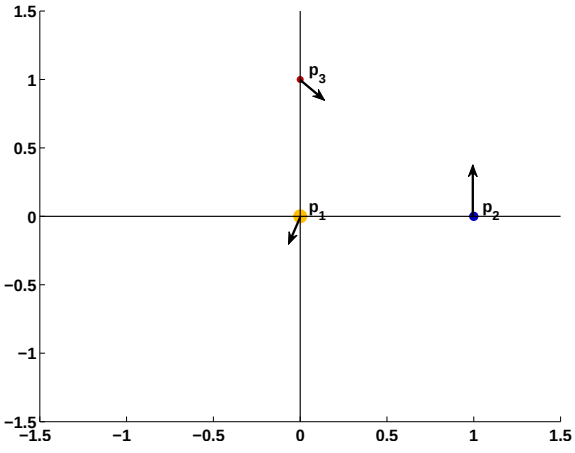

Figure 17.5. *Initial positions and velocities of a small system with three bodies in two-dimensional space.*

and the initial velocity of the third body is sending it towards the second body,

$$\dot{p}_3 = (0.36, -0.36)$$

Newton's law of motion, the famous $F = ma$, says that the mass of a body times its acceleration is proportional to the sum of the forces acting on it. Newton's law of gravitational says that the force between any two bodies is proportional to the product of their masses and inversely proportional to the square of the distance between them. So, the equations of motion are

$$m_i \ddot{p}_i = \gamma \sum_{j \neq i} m_i m_j \frac{p_j - p_i}{||p_j - p_i||^3}, \quad i = 1, \ldots, n$$

Here γ is the gravitional 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 instaneous change in position of this body is given by its velocity vector, denoted by $\dot{p}_i$. The instaneous change in the velocity is given by its acceleration vector, denoted by $\ddot{p}_i$. The acceleration is determined from the position and masses of all the bodies by Newton's laws of motion and gravitation.

The following notation simplifies the discussion of numerical methods. Stack the position vectors on top of each other to produce an n-by-d array where n is the

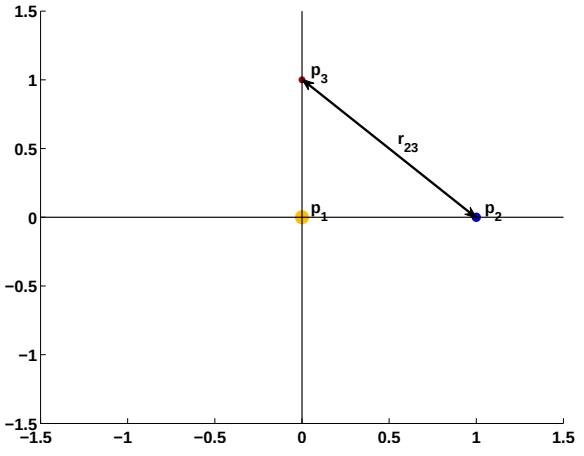

Figure 17.6. *The double arrow depicts the vectors* $r_{23} = p_3 - p_2$ *and* $-r_{32}$. *The length of this arrow is the distance between* p_2 *and* p_3.

number of bodies and $d = 2$ or 3 is the number of spatial dimensions..

$$P = \begin{pmatrix} p_1 \\ p_2 \\ \vdots \\ p_n \end{pmatrix}$$

Let V denote a similar array of velocity vectors.

$$V = \begin{pmatrix} \dot{p}_1 \\ \dot{p}_2 \\ \vdots \\ \dot{p}_n \end{pmatrix}$$

And, let $G(P)$ denote the array of gravitation forces.

$$G(P) = \begin{pmatrix} g_1 \\ g_2 \\ \vdots \\ g_n \end{pmatrix}$$

where

$$g_i = \gamma \sum_{j \neq i} m_j \frac{p_j - p_i}{||p_j - p_i||^3}$$

With this notation, the equations of motion can be written

$$\begin{aligned} \dot{P} &= V \\ \dot{V} &= G(P) \end{aligned}$$

For our three body example, the initial values of P and V are

$$P = \begin{pmatrix} 0 & 0 \\ 1 & 0 \\ 0 & 1 \end{pmatrix}$$

and

$$V = \begin{pmatrix} -0.12 & -0.36 \\ 0 & 0.72 \\ 0.36 & -0.36 \end{pmatrix}$$

The masses in our three body example are

$$m_1 = 1/2, \quad m_2 = 1/3, \quad m_3 = 1/6$$

From these quantities, we can compute the initial value of the gravitation forces, $G(P)$.

We will illustrate our numerical methods by trying to generate a circle. The differential equations are

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

With initial conditions $x(0) = 0, y(0) = 1$, the exact solution is

$$x(t) = \sin t, \;\; y(t) = \cos t$$

The orbit is a perfect circle with a period equal to 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.

$$\begin{aligned} P_{k+1} - \delta\, V_{k+1} &= P_k \\ V_{k+1} - \delta\, G(P_{k+1}) &= V_k \end{aligned}$$

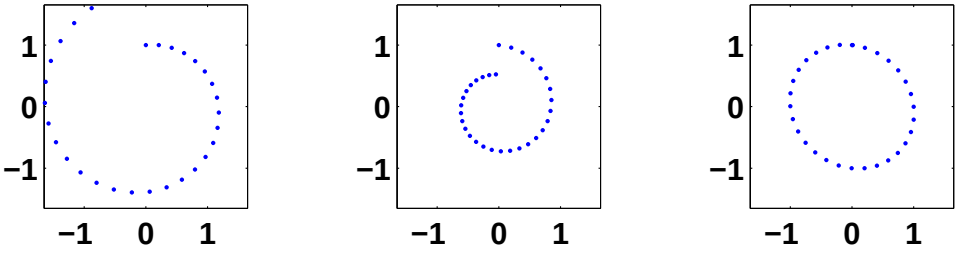

Figure 17.7. *Three versions of Euler's method for generating a circle. The first plot shows that the forward method is unstable. The second plot shows that the backward method has excessive damping. The third plot shows that symplectic method, which is a compromise between the first two methods, produces 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 in not quite a circle. It's actually a nearly circular ellipse. And the final value does not quite return to the initial value, so the period is not exactly 2π . But the important fact is that the orbit is neither a growing nor a decaying spiral.

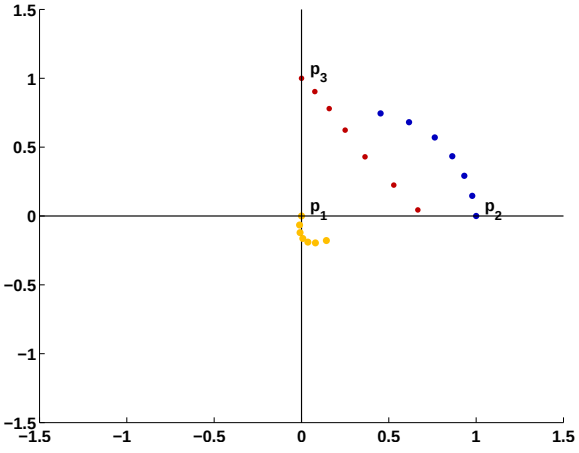

Figure 17.8. *The first few steps of our example system.*

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

```
P =
         0         0
    1.0000         0
         0    1.0000

V =
   -0.1200   -0.3600
         0    0.7200
    0.3600   -0.3600
```

After one step with $\delta = 0.20$ we obtain the following values.

```
P =
   -0.0107   -0.0653
    0.9776    0.1464
    0.0767    0.9033

V =
   -0.0533   -0.3267
   -0.1118    0.7318
```

```
    0.3836   -0.4836
```

The three masses, 1/2, 1/3, and 1/6, are not equal, but are comparable, so all three bodies have significant affects on each other and all three move noticable 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-coorinate has changed by roughly $0.72\,\delta$. The third body moves in the direction indicated by the velocity vector in figure 17.5.

After a second step we have the following values. As expected, all the trends noted in the first step continue.

```
P =
   -0.0079   -0.1209
    0.9325    0.2917
    0.1589    0.7793

V =
    0.0136   -0.2779
   -0.2259    0.7268
    0.4109   -0.6198
```

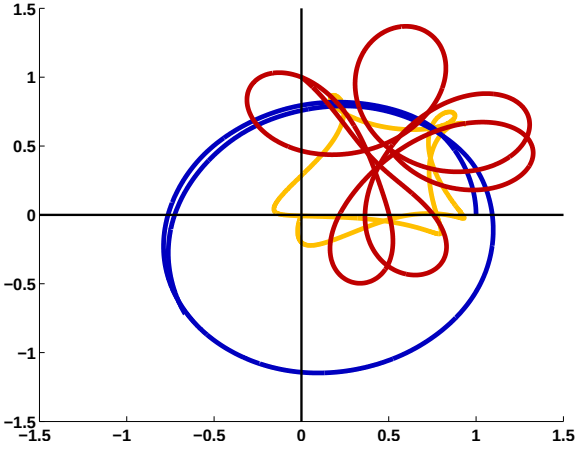

Figure 17.9. *The initial trajectories of our example system.*

Figure 17.9 shows an initial section of the trajectories. You should run our *Experiments* program `orbits(3)` to see the three bodies in motion. The small body and the large body orbit in a clockwise direction around each other while the medium-size body orbits in a counter-clockwise direction around the other two.

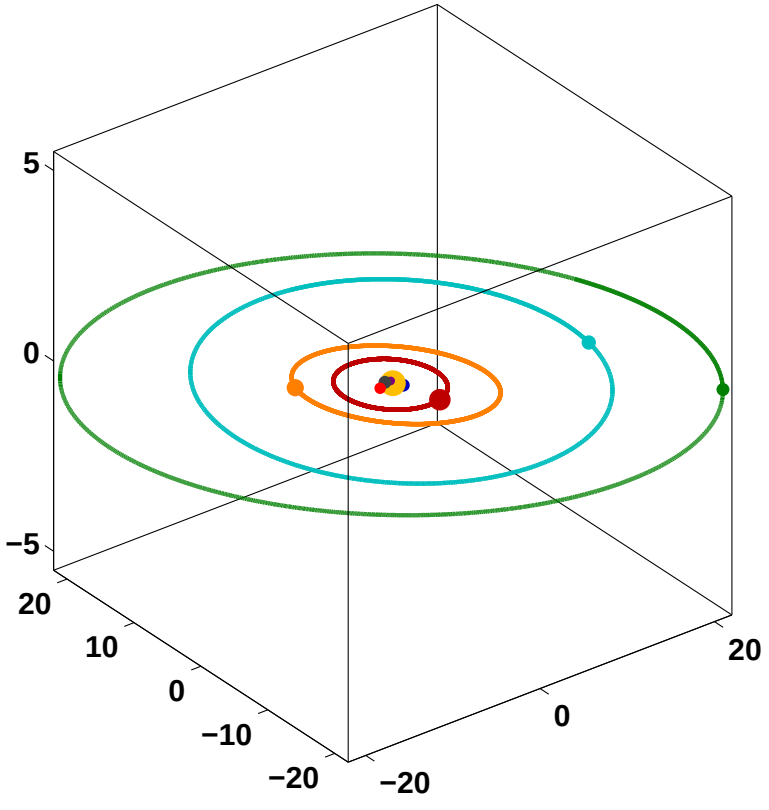

Figure 17.10. *The solar system, with the initial positions of all the planets and the orbits of the outer planets, Jupiter, Saturn, Uranus, and Neptune.*

Our *Experiments* program `orbits` models nine bodies in the solar system, namely the sun and eight planets. Figures 17.10 and 17.11 show snapshots of the output from `orbits` with two different zoom factors that are necessary to span the scale of the system. The orbits for all the planets are in the proper proportion. But, obviously, the symbols for the sun and the planets do not have the same scale. Web sources for information about the solar system are provided by the University Corporation for Atmospheric Research, the Jet Propulsion Laboratory, and the US National Air and Space Museum,

```
http://www.windows.ucar.edu
http://www.jpl.nasa.gov/solar_system
http://airandspace.si.edu:80/etp/ss/index.htm
```

Here is the core portion of our *Experiments* program `orbits`. The subfunction `initialize_orbits` provides initial positions and velocities and thereby specifies the number of bodies, n, and the number of spacial dimensions, d. The other

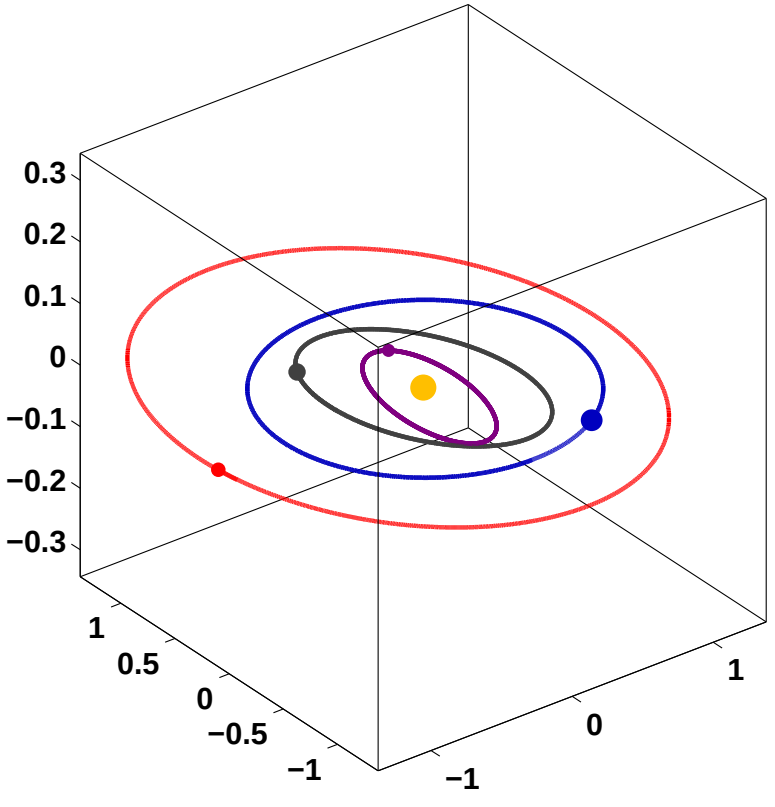

Figure 17.11. *Zooming in by a factor of 16 reveals the orbits of the inner planets, Mercury, Venus, Earth and Mars.*

subfunctions control the plot and the graphics user interface. An exercise asks you to explain how the doubly nested `for` loops evaluate the n sums involved in Newton's law of gravitation.

```
function orbits
% ORBITS  Motion of n bodies under mutual gravitional attraction.
%    ORBITS The solar system with n = 9 for one sun and 8 planets.

   % P = n-by-d array of position coordinates, d = 2 or 3.
   % V = n-by-d array of velocities.
   % M = n-by-1 array of masses.
   % H = graphics and user interface handles.

   [P,V,M] = initialize_orbits;
   H = initialize_graphics(P);
```

```
    n = size(P,1);  % Number of bodies
    steps = 20;     % Number of steps between plots
    t = 0;          % time

    while ~get(H.stop,'value')

       % Obtain step size from slider.
       dt = get(H.speed,'value')/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 + dt*G;

          % Update positions using updated velocities.
          P = P + dt*V;

       end

       t = t + steps*dt;
       update_plot(P,H,t)
    end

    finalize_graphics
end
```

Exercises

17.1 *Bouncing ball.*
(a) What is the maximum height of the bouncing ball?
(b) How many times does the ball bounce?
(c) What is the effect of changing each of the four `bouncer` values `g`, `c`, `delta`, and `v0`.

17.2 *Pluto and Ceres.* Change `orbits` to `orbits11` by adding the erstwhile planet Pluto and the recently promoted dwarf planet Ceres. See Wikipedia:

```
http://en.wikipedia.org/wiki/Planet
http://en.wikipedia.org/wiki/Ceres_(dwarf_planet)
```

and

```
http://orbitsimulator.com/gravity/articles/ceres.html
```

17.3 *Comet.* Add a comet to `orbits`. Find initial conditions so that the comet has a stable, but highly elliptical orbit that extends well beyond the orbits of the planets.

17.4 *Twin suns.* Turn the sun in `orbits` into a twin star system, with two suns orbiting each other out of the plane of the planets. What eventually happens to the planetary orbits? For example, try

```
sun1.p = [1 0 0];
sun1.v = [0 0.25 0.25];
sun1.m = 0.5;
sun2.p = [-1 0 0];
sun2.v = [0 -0.25 -0.25];
sun2.m = 0.5;
```

Try other values as well.

Chapter 18

Sudoku

The remarkably popular puzzle demonstrates man versus machine, backtraking and recursion, and the mathematics of symmetry.

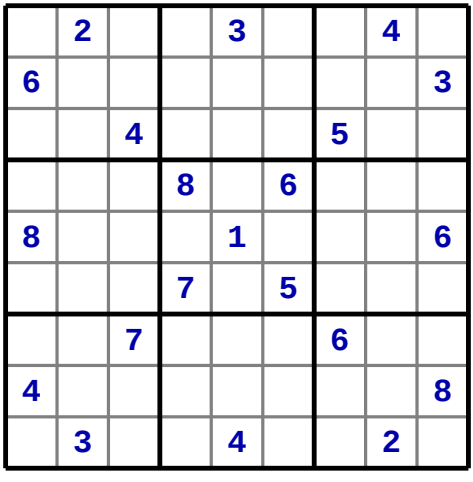

Figure 18.1. *A Sudoku puzzle with especially pleasing symmetry. The clues are shown in blue.*

You probably already know the rules of *Sudoku*, but figures 18.1 and 18.2 illustrate them. Figure 18.1 is the initial 9-by-9 grid, with a specified few digits

known as the *clues*. I especially like the symmetry in this example, which is due to Gordon Royle of the University of Western Australia [1]. Figure 18.2 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 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.

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

Figure 18.2. *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.*

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.

The fascination with solving *Sudoku* by hand 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 easy to program a computer to duplicate human pattern recognition capabilities. Most *Sudoku* computer codes take a very different approach, relying on the machine's almost limitless capacity to carry out brute force trial and error. Our MATLAB program, `sudoku.m`, uses only one pattern, singletons, together with recursive backtracking.

To see how our `sudoku` program works, we can use *Shidoku* instead of *Sudoku*. "Shi" is Japanese for "four". The puzzles, which are almost trivial to solve by

1			
		3	
	2		
			4

Figure 18.3. *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 18.4. *Candidates*

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

Figure 18.5. *Insert singleton*

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

Figure 18.6. *Solution*

hand, use a 4-by-4 grid. Figure 18.3 is our first *Shidoku* puzzle and the next three figures show steps in its solution. In figure 18.4, 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 18.5, we have inserted the singleton "3" in the (4,1) cell and recomputed the candidates. In figure 18.6, we have inserted the remaining singletons as they are generated to complete the solution.

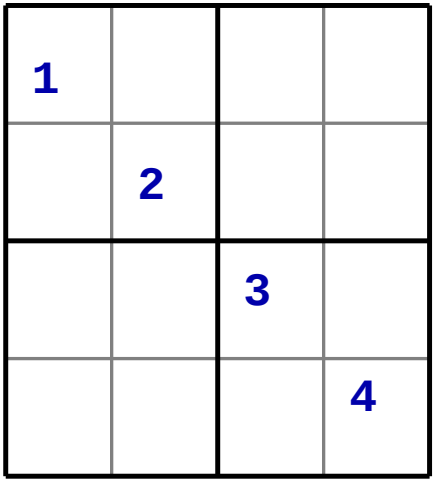

Figure 18.7. *diag(1:4)*

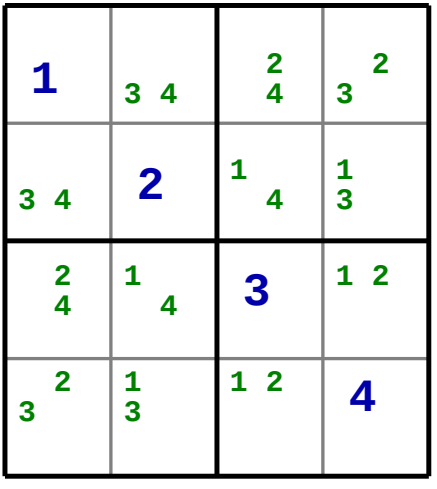

Figure 18.8. *No singletons.*

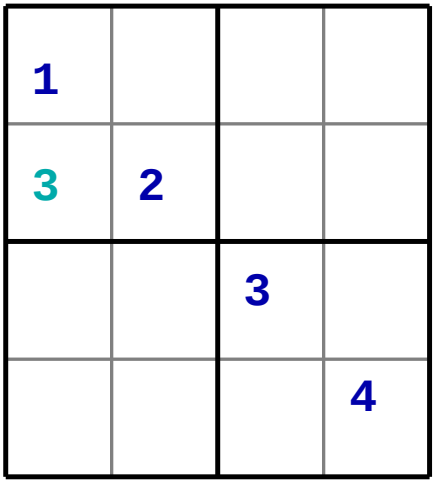

Figure 18.9. *Backtrack step.*

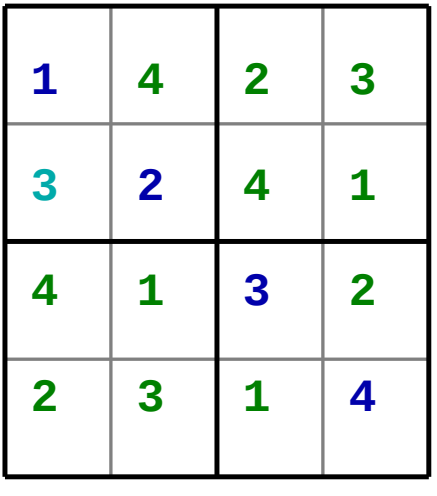

Figure 18.10. *Solution is not unique.*

The input array for figure 18.7 is generated by the MATLAB statement

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

As figure 18.8 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. We have chosen to consider the cells in

the order implied by MATLAB one-dimensional subscripting, `X(:)`, and consider the candidates in numerical order, so we tentatively insert a "3" in cell (2,1). This creates a new puzzle, shown in figure 18.9. Our program is then called recursively. In this example, the new puzzle is easily solved and the result is shown in figure 18.10. 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
```

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. With the puzzle in figure 18.1, if we were to insert a "1", "5" or "7" in the (1,1) cell, the row, column and block conditions would still be satisfied, but it turns out that the resulting puzzle has no solution. It would be very frustrating if such a puzzle were to show up in your daily newspaper. Backtracking generates many impossible configurations. The recursion is terminated by encountering a puzzle with no solution.

Uniqueness is also a elusive property. In fact, most descriptions of *Sudoku* do not specify that there has to be exactly one solution. Again, it would be frustrating to find a different solution from the one given by your newspaper. The only way that I know to check uniqueness is to exhaustively enumerate all possibilities.

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])
```

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.

Here is the code for the main program. All of the bookkeeping required by backtracing is handled by the recursive call mechanism in MATLAB and the underling operating system.

```
function X = sudoku(X)
% SUDOKU  Solve Sudoku using recursive backtracking.
%   sudoku(X), expects a 9-by-9 array X.

   % Fill in all "singletons".
   % C is a cell array of candidate vectors for each cell.
   % s is the first cell, if any, with one candidate.
   % e is the first cell, if any, with no candidates.

   [C,s,e] = candidates(X);
   while ~isempty(s) && isempty(e)
      X(s) = C{s};
      [C,s,e] = candidates(X);
   end

   % Return for impossible puzzles.

   if ~isempty(e)
      return
   end

   % Recursive backtracking.

   if any(X(:) == 0)
      Y = X;
      z = find(X(:) == 0,1);       % The first unfilled cell.
```

```
      for r = [C{z}]                % Iterate over candidates.
         X = Y;
         X(z) = r;                  % Insert a tentative value.
         X = sudoku(X);             % Recursive call.
         if all(X(:) > 0)           % Found a solution.
            return
         end
      end
   end
```

The key internal function is `candidates`.

```
   function [C,s,e] = candidates(X)
      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
      L = cellfun(@length,C);   % Number of candidates.
      s = find(X==0 & L==1,1);
      e = find(X==0 & L==0,1);

   end % candidates
```

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 18.1. 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]`.

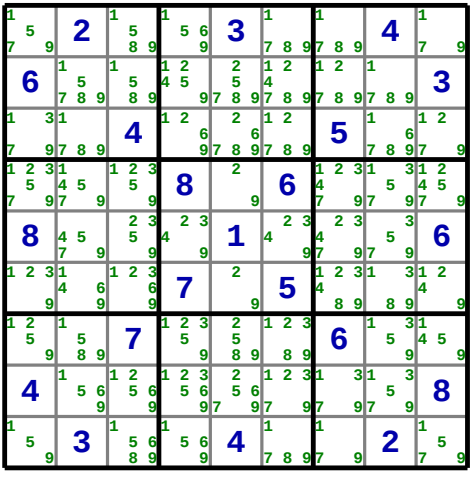

Figure 18.11. *The initial candidates for the puzzle in figure 18.1. There are no singletons.*

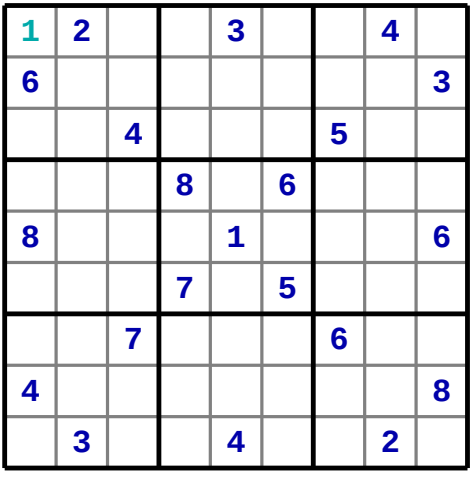

Figure 18.12. *The first step in the backtracking.*

Figure 18.1 is actually a very difficult puzzle, either by hand or by machine. Figures 18.11 through 18.14 are a few snapshots of the solution process. The initial candidates are shown in figure 18.11. There are no singletons, so the first recursive step, shown in figure 18.12, happens immediately. (We know that this puzzle with a "1" in the (1,1) cell has no solution, but the program needs to rediscover that fact every time.) Figure 18.13 shows how the first column is filled in for the first time at

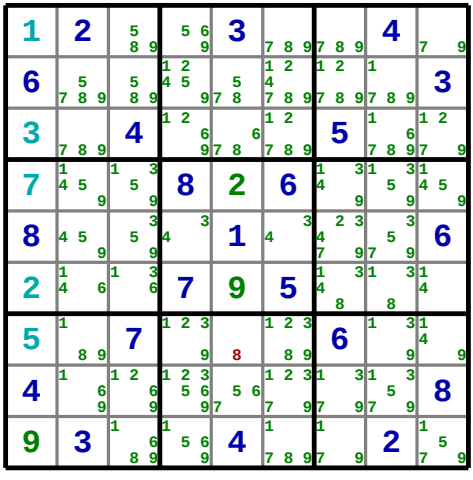

Figure 18.13. *After 22 steps the first column has been filled with possible values, but this track will eventually fail. The cyan values are generated by the backtracking and the green values are implied by the others.*

Figure 18.14. *After 14,781 steps, we appear to be close to a solution, but it is impossible to continue. The eventual solution is in figure 18.2.*

step 22. The elements in cyan are the tentative values from the backtracking and the elements in green are implied by those choices. But we're still a long way from the solution. After 3,114 steps the recursion puts a "5" in the (1,1) cell and after 8,172 steps it tries a "7". Figure 18.14 shows the situation after 14,781 steps. We appear

to be close to a solution because 73 of the 81 cells have been assigned values. But the first row and last column already contain all the of the digits from 1 through 9, so there are no values left for the (1,9) cell in the upper right corner. The candidate list for this cell is empty and the recursion terminates. (There are also two cells in the third row showing the same singleton "3". This could be another reason to terminate the recursion, but it is not necessary to check.) Finally, after 19,229 steps backtracing finally tries a "9" in cell (1,1). The "9" is a good idea because less than 200 steps later, after 19,422 steps, the program reaches the solution shown in figure 18.2. This is many more steps than most puzzles require.

References

[1] Gordon Royle, Symmetry in Sudoku,
`http://people.csse.uwa.edu.au/gordon/sudoku/sudoku-symmetry.pdf`
[2] Sudoku Squares and Chromatic Polynomials,
`http://www.ams.org/notices/200706/tx070600708p.pdf`
[3] Strategy families,
`http://www.scanraid.com/Strategy\_Families`
[4] Nikoli,
`http://www.nikoli.co.jp/en/misc/2008speech\_from\_ussc.htm`

Exercises

18.1 *Solve.* Solve a Sudoku puzzle by hand.

18.2 *sudoku_puzzle.* The EXM program `sudoku_puzzle` generates 15 different puzzles. The comments in the program describe the origins of the puzzles. How many steps are required by `sudoku.m` to solve each of the puzzles?

18.3 *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.

18.4 *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 18.15. Does this make it easier or harder to solve puzzles by hand.

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

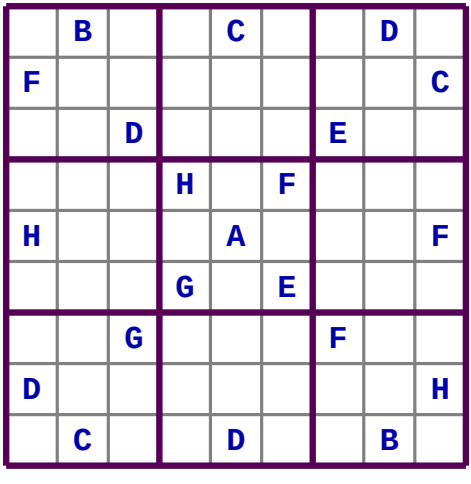

Figure 18.15. *Use the letters 'A' through 'I' instead of the digits 1 through 9.*