Chapter 1

Introduction to MATLAB

This book is an introduction to two subjects, numerical computing and MATLAB. This first chapter introduces MATLAB by presenting several sample programs that investigate elementary, but hopefully interesting, mathematical problems. We hope that you can see how MATLAB works by studying these sample programs.

You should have a copy of MATLAB close at hand so you can run the sample programs as you read about them. All of the programs used in this book have been collected in a directory (or folder) named

```
NCM
```

(The directory name is an acronym for the book title). You can either start MATLAB in this directory, or add the directory to the MATLAB path.

1.1 The Golden Ratio

What is the world's most interesting number? Perhaps you like 17, or π, or e? Many people would vote for ϕ, the *golden ratio*, computed here by our first MATLAB statement

```
phi = (1 + sqrt(5))/2
```

This produces

```
phi =
    1.6180
```

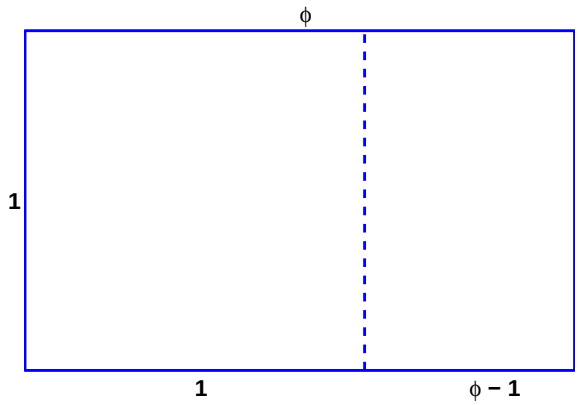
Let's see more digits.

```
format long
phi

phi =
   1.61803398874989
```

This didn't recompute ϕ, it just displayed 15 significant digits instead of five.

The golden ratio shows up in many places in mathematics; we'll see several in this book. It gets its name from the golden rectangle, the rectangle with the perfect aspect ratio. It has the property that removing a square leaves a smaller rectangle with the same shape. Here it is.



Equating the aspect ratios of the rectangles gives a defining equation for ϕ.

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

This equation says that you can compute the reciprocal of ϕ by simply subtracting one. How many numbers have that property?

Multiplying the aspect ratio equation by ϕ produces a polynomial equation

$$\phi^2 - \phi - 1 = 0$$

The roots of this equation are given by the quadratic formula.

$$\phi = \frac{1 \pm \sqrt{5}}{2}$$

The positive root is the golden ratio. The negative root, which is actually equal to $1 - \phi$, has no meaning in the golden rectangle, but it will show up later in other contexts.

If you have forgotten the quadratic formula, you can ask MATLAB to find the roots of the polynomial. MATLAB represents a polynomial by the vector of its coefficients, in descending order. So the vector

```
p = [1 -1 -1]
```

represents the polynomial

$$p(x) = x^2 - x - 1$$

The roots are computed by the `roots` function.

```
r = roots(p)
```

produces

```
r =
  -0.61803398874989
   1.61803398874989
```

These two numbers are the only numbers whose reciprocal can be computed by subtracting one.

You can use the Symbolic Toolbox, which connects MATLAB to Maple, to solve the aspect ratio equation without converting it to a polynomial. The equation is represented by a character string. The `solve` function finds two solutions.

```
r = solve('1/x = x-1')
```

produces

```
r =
[ 1/2*5^(1/2)+1/2]
[ 1/2-1/2*5^(1/2)]
```

The `pretty` function displays the results in a way that resembles typeset mathematics.

```
pretty(r)
```

produces

```
        [     1/2      ]
        [1/2 5    + 1/2]
        [              ]
        [           1/2]
        [1/2 - 1/2 5   ]
```

The variable `r` is a vector with two components, the symbolic forms of the two solutions. You can pick off the first component with

```
phi = r(1)
```

which produces

```
phi =
1/2*5^(1/2)+1/2
```

This expression can be converted to a numerical value in a couple of different ways. It can be evaluated to any number of digits using variable-precision arithmetic with the `vpa` function.

```
vpa(phi,50)
```

produces 50 digits

```
1.6180339887498948482045868343656381177203091798058
```

It can also be converted to double-precision floating-point, which is the principal way that MATLAB represents numbers, with the `double` function.

```
phi = double(phi)
```

produces

```
phi =
   1.61803398874989
```

The aspect ratio equation is simple enough to have closed form symbolic solutions. More complicated equations have to be solved approximately. The `inline` function is a quick way to convert character strings to objects that can be arguments to the MATLAB functions that operate on other functions.

```
f = inline('1/x - (x-1)');
```

defines $f(x) = 1/x - (x-1)$ and produces

```
f =
     Inline function:
     f(x) = 1/x - (x-1)
```

A graph of $f(x)$ over the interval $0 \le x \le 4$ is obtained with

```
ezplot(f,0,4)
```

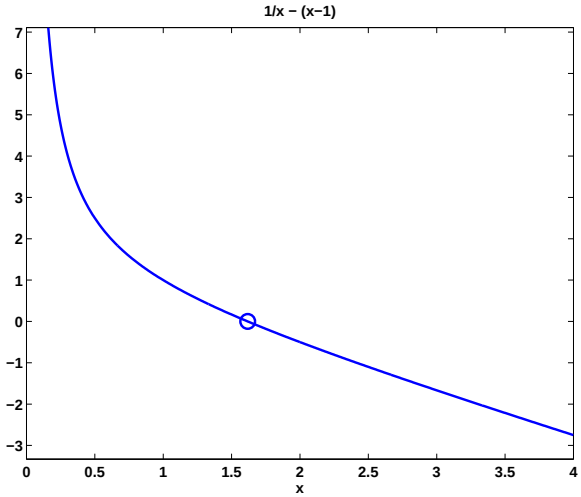
The name `ezplot` stands for *easy plot*, although some of the English-speaking world would pronounce it as "e-zed plot". Even though $f(x)$ becomes infinite as $x \to 0$, `ezplot` automatically picks a reasonable vertical scale.

The statement

```
phi = fzero(f,1)
```

looks for a zero of $f(x)$ near x = 1. It produces an approximation to ϕ that is accurate to almost full precision. The result can be inserted in the `ezplot` graph with

```
hold on
plot(phi,0,'o')
```



Here is the MATLAB program that produces the picture of the golden rectangle shown at the beginning of this section. The program is contained in an M-file named `goldrect.m`, so issuing the command

```
goldrect
```

runs the script and creates the picture.

```
% GOLDRECT  Plot the golden rectangle

phi = (1+sqrt(5))/2;
x = [0 phi phi 0 0];
y = [0 0 1 1 0];
u = [1 1];
v = [0 1];
plot(x,y,'b',u,v,'b--')
text(phi/2,1.05,'\phi')
text((1+phi)/2,-.05,'\phi - 1')
text(-.05,.5,'1')
text(.5,-.05,'1')
axis equal
axis off
set(gcf,'color','white')
```

The vectors `x` and `y` each contain five elements. Connecting consecutive (x_k, y_k) pairs with straight lines produces the outside rectangle. The vectors `u` and `v` each contain two elements. The line connecting (u_1, v_1) with (u_2, v_2) separates the rectangle into the square and the smaller rectangle. The `plot` command draws these lines, the $x - y$ lines in solid blue and the $u - v$ line in dashed blue. The next four statements place text at various points; the string '`\phi`' denotes the Greek letter. The two `axis` statements cause the scaling in the x and y directions

to be equal and then turn off the display of the axes. The last statement sets the background color of `gcf`, which stands for *get current figure*, to white.

A *continued fraction* is an infinite expression of the form

$$a_0 + \cfrac{1}{a_1 + \cfrac{1}{a_2 + \cfrac{1}{a_3 + \cdots}}}$$

If all the a_k's are equal to one, the continued fraction is another representation of the golden ratio.

$$\phi = 1 + \cfrac{1}{1 + \cfrac{1}{1 + \cfrac{1}{1 + \cdots}}}$$

Here is a MATLAB function that generates and evaluates truncated continued fraction approximations to ϕ. The code is stored in an M-file named `goldfract.m`.

```
function goldfract(n)
% GOLDFRACT    Golden ratio truncated continued fraction.
% GOLDFRACT(n) displays n terms.

g = '1';
for k = 1:n
   g = ['1+1/(' g ')'];
end
g

g = 1;
f = 1;
for k = 1:n
   s = g;
   g = g + f;
   f = s;
end
g = sprintf('%d/%d',g,f)

format long
g = eval(g)

format short
err = (1+sqrt(5))/2 - g
```

The statement

```
goldfract(6)
```

produces

```
g =
1+1/(1+1/(1+1/(1+1/(1+1/(1+1/(1))))))
```

```
g =
21/13

g =
   1.61538461538462

err =
    0.0026
```

The three `g`'s are all different representations of the same approximation to ϕ.

The first `g` is the continued fraction truncated to six terms. There are six right parentheses. This `g` is a string generated by starting with a single '`1`' (that's `goldfract(0)`) and repeatedly inserting the string '`1+1/(`' in front and the string '`)`' in back. No matter how long this string becomes, it is a valid MATLAB expression.

The second `g` is an "ordinary" fraction with a single integer numerator and denominator obtained by collapsing the first `g`. The basis for the reformulation is

$$1 + \frac{1}{\frac{g}{f}} = \frac{g+f}{g}$$

So the iteration starts with

$$\frac{1}{1}$$

and repeatedly replaces the fraction

$$\frac{g}{f}$$

by

$$\frac{g+f}{g}$$

The statement

```
g = sprintf('%d/%d',g,f)
```

prints the final fraction by formatting `g` and `f` as decimal integers and placing a '`/`' between them.

The third `g` is the same number as the first two `g`'s, but is represented as a conventional decimal expansion, obtained by having the MATLAB `eval` function actually do the division expressed in the second `g`.

The final quantity `err` is the difference between `g` and ϕ. With only six terms, the approximation is accurate to less than three digits. How many terms does it take to get 10 digits of accuracy?

As the number of terms `n` increases, the truncated continued fraction generated by `goldfract(n)` theoretically approaches ϕ. But limitations on the size of the integers in the numerator and denominator, as well as roundoff error in the actual floating-point division, eventually intervene. One of the exercises asks you to investigate the limiting accuracy of `goldfract(n)`.

1.2 Fibonacci Numbers

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 still 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.

$$f_1 = 1, \quad f_2 = 2$$

Here is a MATLAB function, stored in the M-file `fibonacci.m`, that 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 this is a `function` M-file, 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.
```

Here 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

```
g = g + f;
```

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 before. 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

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

An exercise will ask you to use the MATLAB backslash operator to solve this 2-by-2 system of simultaneous linear equations, but it is actually easier to solve the system by hand.

$$\begin{aligned} c_1 &= \frac{\phi}{2\phi - 1} \\ c_2 &= -\frac{(1-\phi)}{2\phi - 1} \end{aligned}$$

Inserting these in the general solution gives

$$f_n = \frac{1}{2\phi - 1}(\phi^{n+1} - (1-\phi)^{n+1})$$

This is an amazing equation. The right-hand side involves powers and quotients of irrational numbers, but the result is a sequence of integers. You can check this with MATLAB, displaying the results in scientific notation.

```
format long e
n = (1:40)';
f = (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. The computed result starts with

```
f =
     1.000000000000000e+000
     2.000000000000000e+000
     3.000000000000000e+000
     5.000000000000001e+000
     8.000000000000002e+000
     1.300000000000000e+001
     2.100000000000000e+001
     3.400000000000001e+001
```

and ends with

```
     5.702887000000007e+006
     9.227465000000011e+006
     1.493035200000002e+007
     2.415781700000003e+007
     3.908816900000005e+007
     6.324598600000007e+007
     1.023341550000001e+008
     1.655801410000002e+008
```

Roundoff error prevents the results from being exact integers, but

```
f = round(f)
```

finishes the job.

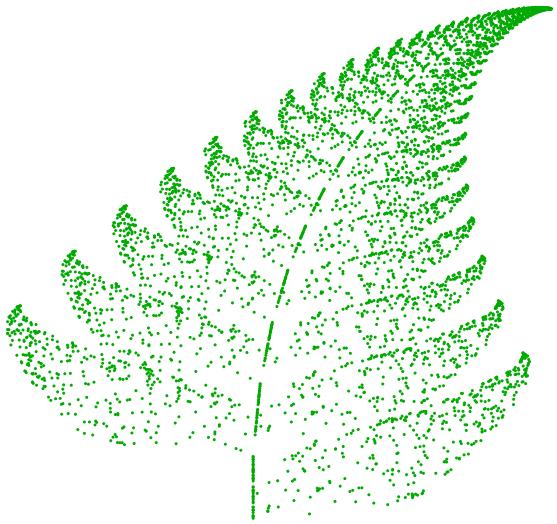
1.3 Fractal Fern

The M-file `fern.m` produces the "Fractal Fern" described by Michael Barnsley in his book *Fractals Everywhere*, published by Academic Press in 1993. It generates and plots a potentially infinite sequence of random, but carefully choreographed, points in the plane. The command

```
fern
```

runs forever, producing an increasingly dense picture.

An accurate plot of the fractal fern contains at least a few hundred thousand points and it is impractical to reproduce high resolution output in this book. Here is a low resolution plot with 5000 points.



We have included a compressed version of a 1024-by-768 pixel plot containing 250,000 points in the NCM collection. Find the file

```
fern.bmp.gz
```

Use `WinZip` or `gunzip` to uncompress the file. This produces a 2.25 megabyte bitmap file, `fern.bmp`, that you can view with a browser or a paint program. If you have access to the MATLAB Image Processing Toolbox, you can view the file with

```
imshow(imread('fern.bmp'))
```

If you like the image, you might even choose to make it your desktop background "wallpaper". 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

$$A = \begin{pmatrix} .85 & .04 \\ -.04 & .85 \end{pmatrix}, \quad b = \begin{pmatrix} 0 \\ 1.6 \end{pmatrix}$$

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 towards 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 statements

```
cnt = 1;
```

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

```
tic
```

initializes a stopwatch timer.

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

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

```
r = rand;
```

generates a *pseudorandom* value between zero and one. 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 eighty-five percent 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 uicontrol to a pushbutton that closes the window.

1.4 Magic Squares

MATLAB stands for *Matrix Laboratory*. Over the years MATLAB has evolved into a general purpose technical computing environment, but operations involving vectors, matrices, and linear algebra continue to be its most distinguishing feature.

Magic squares provide an interesting set of sample matrices. The commands `help magic` or `helpwin magic` tell us that

```
MAGIC(N) is an N-by-N matrix constructed from the integers
1 through N^2 with equal row, column, and diagonal sums.
Produces valid magic squares for all N > 0 except N = 2.
```

Magic squares were known in China over two thousand years before the birth of Christ. The 3-by-3 magic square is known as *lo-shu*. Legend has it that *lo-shu* was discovered on the shell of a turtle that crawled out of the Lo River in the twenty-third century B.C. *Lo-shu* provides a mathematical basis for *feng shui*, the ancient Chinese philosophy of balance and harmony. 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, that 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 "up-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`. When these integers are allocated to three 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
```

This is all the magic squares of order three.

Now for some linear algebra. The determinant of our magic square,

```
det(A)
```

is

```
    -360
```

The inverse,

```
X = inv(A)
```

is

```
X =
    0.1472   -0.1444    0.0639
   -0.0611    0.0222    0.1056
   -0.0194    0.1889   -0.1028
```

The inverse looks better when it is displayed with a rational format.

```
format rat
X
```

shows that the elements of `X` are fractions with `det(A)` in the denominator.

```
X =
    53/360      -13/90        23/360
   -11/180        1/45        19/180
    -7/360       17/90       -37/360
```

The statement

```
format short
```

restores the output format to its default.

Three other important quantities in computational linear algebra are *matrix norms*, *eigenvalues*, and *singular values*. The statements

```
r = norm(A)
e = eig(A)
s = svd(A)
```

produce

```
r =
    15

e =
   15.0000
    4.8990
   -4.8990

s =
   15.0000
    6.9282
    3.4641
```

The magic sum occurs in all three because the vector of all ones is an eigenvector, and is also a left and right singular vector.

So far, all the computations in this section have been done using floating-point arithmetic. This is the arithmetic used for almost all scientific and engineering computation, especially for large matrices. But for a 3-by-3 matrix, it is easy to repeat the computations using symbolic arithmetic and the Symbolic Toolbox connection to Maple. The statement

```
A = sym(A)
```

changes the internal representation of `A` to a symbolic form that displays as

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

Now commands like

```
sum(A), sum(A')', det(A), inv(A), eig(A), svd(A)
```

produce symbolic results. In particular, the eigenvalue problem for this matrix can be solved exactly, and

```
e =
[         15]
[  2*6^(1/2)]
[ -2*6^(1/2)]
```

A 4-by-4 magic square is one of several mathematical objects on display in *Melancolia*, a Renaissance etching by Albrect Durer. An electronic copy of the etching is available in a MATLAB data file.

```
load durer
whos
```

produces

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

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

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

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

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

display a higher resolution scan of the area around the magic square.

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)))
```

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

The 4-by-4 magic square produced by MATLAB is not the same as Durer'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 Durer's etching. Durer probably chose this particular 4-by-4 square because the date he did the work, 1514, occurs in the middle of the bottom row.

We have seen two different 4-by-4 magic squares. It turns out that there are 880 different magic squares of order four and 275305224 different magic squares of order five. Nobody knows how many different magic squares of order six or larger there are.

The determinant of our 4-by-4 magic square, `det(A)`, is zero. If we try to compute its inverse

```
inv(A)
```

we get

```
Warning: Matrix is close to singular or badly scaled.
         Results may be inaccurate.
```

So, some magic squares represent singular matrices. Which ones? The *rank* of a square matrix is the number of linearly independent rows or columns. An n-by-n matrix is singular if and only if its rank is less than n.

The statements

```
for n = 1:24, r(n) = rank(magic(n)); end
[(1:24)' r']
```

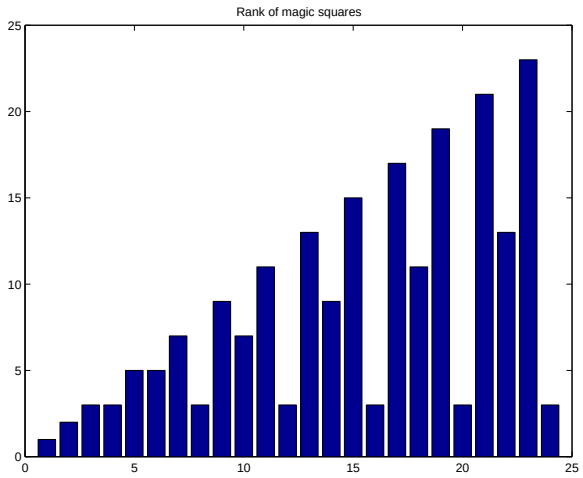
produce a table of order versus rank.

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

Look carefully at this table. Ignore $n = 2$ because `magic(2)` is not really a magic square. What patterns do you see? A bar graph makes the patterns easier to see.

```
bar(r)
title('Rank of magic squares')
```

produces



This graph shows that there are three different kinds of magic squares:

- *Odd* order; n is odd.
- *Singly even* order; n is a multiple of 2, but not 4.
- *Doubly even* order; n is a multiple of 4.

Odd-ordered magic squares, $n = 3, 5, 7, ...$, have full rank n. They are nonsingular and have inverses. Doubly even magic squares, $n = 4, 8, 12, ...$, have rank three no matter how large n is. They might be called *very singular*. Singly even magic squares, $n = 6, 10, 14, ...$, have rank $n/2 + 2$. They are also singular, but have fewer row and column dependencies than the doubly even squares.

If you have MATLAB Version 6 or later, you can look at the M-file that generates magic squares with

```
edit magic.m
```

or

```
type magic.m
```

You will see the three different cases in the code.

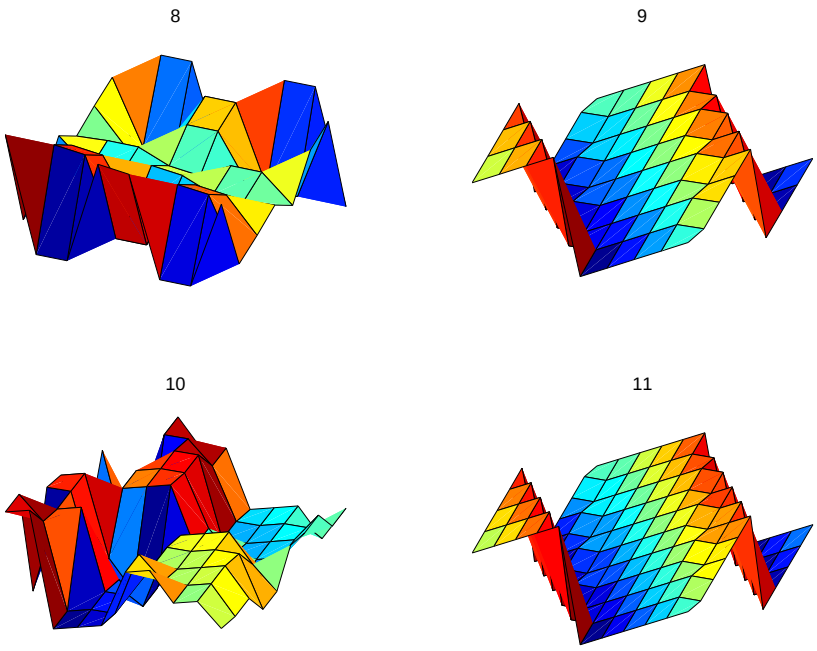
The different kinds of magic squares also produce different three-dimensional surface plots. Try the following for various values of `n`.

```
surf(magic(n))
axis off
set(gcf,'doublebuffer','on')
cameratoolbar
```

Double buffering prevents flicker when you use the various camera tools to move the viewpoint. Here is a slightly different way to see these magic surface plots.

```
for n = 8:11
   subplot(2,2,n-7)
   surf(magic(n))
   title(num2str(n))
   axis off
   view(30,45)
   axis tight
end
```

produces



1.5 Cryptography

This section uses a cryptography example to show how MATLAB deals with text and character strings. The cryptographic technique, which is known as a *Hill cipher*, involves arithmetic in a *finite field*.

Almost all modern computers use the ASCII character set to store basic text. ASCII stands for *American Standard Code for Information Interchange*. The character set uses seven of the eight bits in a byte to encode 128 characters. The first 32 characters are nonprinting control characters, such as tab, backspace and end-of-line. The 128th character is another nonprinting character that corresponds to the delete key on your keyboard. In between these control characters are 95 printable characters, including a space, 10 digits, 26 lowercase letters, 26 uppercase letters and 32 punctuation marks.

MATLAB can easily display all the printable characters, in the order determined by their ASCII encoding. Start with

```
x = reshape(32:127,32,3)'
```

This produces a 3-by-32 matrix.

```
x =
    32    33    34   ...   61    62    63
    64    65    66   ...   93    94    95
    96    97    98   ...  125   126   127
```

The `char` function converts numbers to characters. The statement

```
c = char(x)
```

produces

```
c =
 !"#$%&'()*+,-./0123456789:;<=>?
@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_
`abcdefghijklmnopqrstuvwxyz{|}~
```

We have cheated a little bit because the last element of `x` is `127`, which corresponds to the nonprinting delete character, and we have not shown the last character in `c`. You can try this on your computer and see what is actually displayed.

The first character in `c` is blank, indicating that

```
char(32)
```

is the same as

```
' '
```

The last printable character in `c` is the *tilde*, indicating that

```
char(126)
```

is the same as

```
'~'
```

The characters representing digits are in the first line of `c`. In fact

```
d = char(48:57)
```

displays a ten-character string

```
d =
0123456789
```

This string can be converted to the corresponding numerical values with `double` or `real`. The statement

```
double(d) - '0'
```

produces

```
     0     1     2     3     4     5     6     7     8     9
```

Comparing the second and third line of `c`, we see that the ASCII encoding of the lowercase letters is obtained by adding 32 to the ASCII encoding of the uppercase letters. Understanding this encoding allows us to use vector and matrix operations in MATLAB to manipulate text.

The ASCII standard is often extended to make use of all eight bits in a byte, but the characters that are displayed depend on the computer and operating system you are using, the font you have chosen, and even the country you live in. Try

```
char(reshape(160:255,32,3)')
```

and see what happens on your machine.

Our encryption technique involves *modular arithmetic*. All the quantities involved are integers and the result of any arithmetic operation is reduced by taking the remainder or *modulus* with respect to a prime number, p. The functions `rem(x,y)` and `mod(x,y)` both compute the remainder when `x` is divided by `y`. They produce the same result when `x` and `y` have the same sign; the result also has that sign. But if `x` and `y` have opposite signs, then `rem(x,y)` has the same sign as `x`, while `mod(x,y)` has the same sign as `y`. Here is a table.

```
x = [37 -37 37 -37]';
y = [10 10 -10 -10]';
r = [ x  y  rem(x,y) mod(x,y)]
```

produces

```
    37    10     7     7
   -37    10    -7     3
    37   -10     7    -3
   -37   -10    -7    -7
```

We have chosen to encrypt text that uses the entire ASCII character set, not just the letters. There are 95 such characters. The next larger prime number is $p = 97$, so we represent the p characters by the integers `0:p-1` and do arithmetic mod p.

The characters are encoded two at a time. Each pair of characters is represented by a 2-vector, x. For example, suppose the text contains the pair of letters `'TV'`. The ASCII values for this pair of letters are 84 and 86. Subtracting 32 to make the representation start at 0 produces the column vector

$$x = \begin{pmatrix} 52 \\ 54 \end{pmatrix}$$

The encryption is done with a 2-by-2 matrix-vector multiplication over the integers mod p. The symbol $\equiv$ is used to indicate that two integers have the same remainder, modulo the specified prime.

$$y \equiv Ax, \mod p$$

where A is the matrix

$$A = \begin{pmatrix} 71 & 2 \\ 2 & 26 \end{pmatrix}$$

For our example, the product Ax is

$$Ax = \begin{pmatrix} 3800 \\ 1508 \end{pmatrix}$$

When this is reduced mod p the result is

$$y = \begin{pmatrix} 17 \\ 53 \end{pmatrix}$$

Converting this back to characters by adding 32 produces `'1U'`.

Now comes the interesting part. Over the integers modulo p, the matrix A is its own inverse. If

$$y \equiv Ax, \mod p$$

then

$$x \equiv Ay, \mod p$$

In other words, in arithmetic mod p, A^2 is the identity matrix. You can check this with MATLAB.

```
p = 97;
A = [71 2; 2 26]
I = mod(A^2,p)
```

produces

```
A =
    71     2
     2    26

I =
     1     0
     0     1
```

This means that the encryption process is its own inverse. One function can be used to both encrypt and decrypt a message.

Here is the preamble of the M-file `crypto.m`.

```
function y = crypto(x)
% CRYPTO Cryptography example.
% y = crypto(x) converts an ASCII text string into another
% coded string.  The function is its own inverse, so
%   crypto(crypto(x)) gives x back.
% See also: ENCRYPT.

% Use a two-character Hill cipher with arithmetic
% modulo 97, a prime.
p = 97;
```

The conversion from characters to numerical values is done by

```
% Convert printable ASCII text to integers mod p.
space = 32;
delete = 127;
k = find(x >= delete);
x(k) = x(k)-delete;
x = mod(real(x-space),p);
```

Prepare for the matrix-vector product by forming a matrix with two rows and lots of columns.

```
% Reshape into a matrix with 2 rows and
% floor(length(x)/2) columns.
n = 2*floor(length(x)/2);
X = reshape(x(1:n),2,n/2);
```

All this preparation has been so that we can do the actual finite field arithmetic quickly and easily.

```
% Encode with matrix multiplication modulo p.
A = [71 2; 2 26];
Y = mod(A*X,p);
```

Finally, convert the numbers back to printable characters.

```
% Reshape into a single row.
y = reshape(Y,1,n);

% If length(x) is odd, encode the last character.
if length(x) > n
   y(n+1) = mod((p-1)*x(n+1),p);
end

% Convert to printable ASCII characters.
y = char(y+space);
k = find(y >= delete);
y(k) = y(k)+delete;
```

Let's follow the computation of `y = crypto('Hello world')`. We begin with a character string.

```
x = 'Hello world'
```

This is converted to an integer vector.

```
x =
    40   69   76   76   79    0   87   79   82   76   68
```

The `length(x)` is odd, so the reshaping temporarily ignores the last element.

```
X =
    40   76   79   87   82
    69   76    0   79   76
```

A conventional matrix-vector multiplication `A*X` produces an intermediate matrix.

```
  2978    5548    5609    6335    5974
  1874    2128     158    2228    2140
```

Then the `mod(.,p)` operation produces

```
Y =
    68   19   80   30   57
    31   91   61   94    6
```

This is rearranged to a row vector.

```
y =
    68   31   19   91   80   61   30   94   57    6
```

Now the last element of `x` is encoded by itself and attached to the end of `y`.

```
y =
    68   31   19   91   80   61   30   94   57    6   29
```

Finally, `y` is converted back to a character string to produce the encrypted result.

```
y = 'd?3{p]>~Y&='
```

If we now compute `crypto(y)`, we get back our original `'Hello world'`.

1.6 The $3n + 1$ sequence

Here is a famous unsolved problem in number theory. Start with any positive integer n. Repeat the following steps:

- If $n = 1$, stop.
- If n is even, replace it with $n/2$.
- If n is odd, replace it with $3n + 1$.

For example, starting with $n = 7$ produces

$$7, 22, 11, 34, 17, 52, 26, 13, 40, 20, 10, 5, 16, 8, 4, 2, 1$$

The sequence terminates after 17 steps. Note that whenever n reaches a power of 2, the sequence terminates in $\log_2 n$ more steps.

The unanswered question is, does the process *always* terminate? Or, is there some starting value that causes the process to go on forever, either because the numbers get larger and larger, or because some periodic cycle is generated?

This problem is known as the $3n + 1$ problem. It has been studied by many eminent mathematicians, including Collatz, Ulam, and Kakatani. A survey paper by Jeffrey Lagarias was published in the *American Mathematical Monthly,* Vol. 92 (1985), pp. 3-23. The paper is also available on the Web at

```
http://www.cecm.sfu.ca/organics/papers/lagarias
```

Here is a MATLAB code fragment that generates the sequence starting with any specified `n`.

```
y = n;
while n > 1
   if rem(n,2)==0
      n = n/2;
   else
      n = 3*n+1;
   end
   y = [y n];
end
```

We don't know ahead of time how long the resulting vector `y` is going to be. But the statement

```
y = [y n];
```

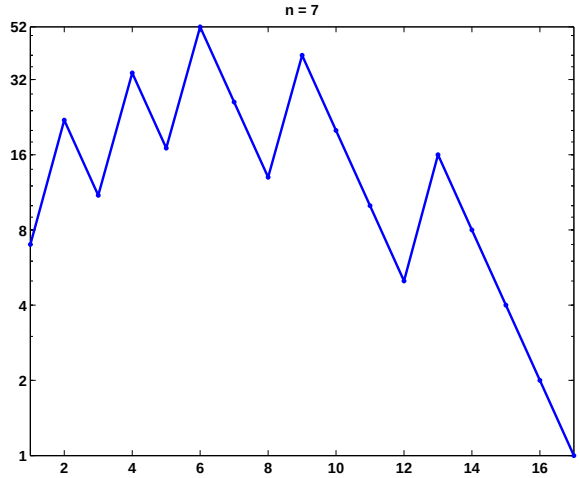
automatically increases `length(y)` each time it is executed.

In principle, the unsolved mathematical problem is: can this code fragment run forever? In actual fact, floating-point roundoff error causes the calculation to misbehave whenever $3n + 1$ becomes greater than 2^{53}, but it is still interesting to investigate modest values of n.

Let's embed our code fragment in a GUI. The complete function is in M-file `threenplus1.m`. For example, the statement

```
threenplus1(7)
```

produces



The M-file begins with a preamble containing the function header and the `help` information.

```
function threenplus1(n)
% ‘‘Three n plus 1’’.
% Study the 3n+1 sequence.
% threenplus1(n) plots the sequence starting with n.
```

```
% threenplus1 with no arguments starts with n = 1.
% uicontrols decrement or increment the starting n.
% Is it possible for this to run forever?
```

The next section of code brings the current graphics window forward and resets it. Two pushbuttons, which are the default `uicontrol`s, are positioned near the bottom center of the figure at pixel coordinates `[260,5]` and `[300,5]`. Their size is 25-by-22 pixels and they are labeled with `'<'` and `'>'`. When either button is subsequently pushed, the `'callback'` string is executed, calling the function recursively with a corresponding `'-1'` or `'+1'` string argument. The `'tag'` property of the current figure, `gcf`, is set to a characteristic string that prevents this section of code from being reexecuted on subsequent calls.

```
if ~isequal(get(gcf,'tag'),'3n+1')
   shg
   clf reset
   uicontrol( ...
      'position',[260 5 25 22], ...
      'string','<', ...
      'callback','threenplus1(''-1'')');
   uicontrol( ...
      'position',[300 5 25 22], ...
      'string','>', ...
      'callback','threenplus1(''+1'')');
   set(gcf,'tag','3n+1');
end
```

The next section of code sets `n`. If `nargin`, the number of input arguments, is zero, then `n` is set to `1`. If the input argument is either of the strings from the pushbutton callbacks, then `n` is retrieved from the `'userdata'` field of the `figure` and decremented or incremented. If the input argument is not a string, then it is the desired `n`. In all situations, `n` is saved in `'userdata'` for use on subsequent calls.

```
if nargin == 0
   n = 1;
elseif isequal(n,'-1')
   n = get(gcf,'userdata') - 1;
elseif isequal(n,'+1')
   n = get(gcf,'userdata') + 1;
end
if n < 1, n = 1; end
set(gcf,'userdata',n)
```

We've seen the next section of code before; it does the actual computation.

```
y = n;
while n > 1
```

```
    if rem(n,2)==0
       n = n/2;
    else
       n = 3*n+1;
    end
    y = [y n];
end
```

The final section of code plots the generated sequence with dots connected by straight lines, using a logarithmic vertical scale and customized tick labels.

```
semilogy(y,'.-')
axis tight
ymax = max(y);
ytick = [2.^(0:ceil(log2(ymax))-1) ymax];
if length(ytick) > 8, ytick(end-1) = []; end
set(gca,'ytick',ytick)
title(['n = ' num2str(y(1))]);
```

1.7 Circle Generator

Here is an algorithm that was used to plot circles on some of the first computers with graphical displays. At the time, there was no MATLAB and no floating-point arithmetic. Programs were written in machine language and arithmetic was done on scaled integers. The circle generating program looked something like this:

```
      x = 32768
      y = 0
   L: load y
      shift right 5 bits
      add x
      store in x
      change sign
      shift right 5 bits
      add y
      store in y
      plot x y
      go to L
```

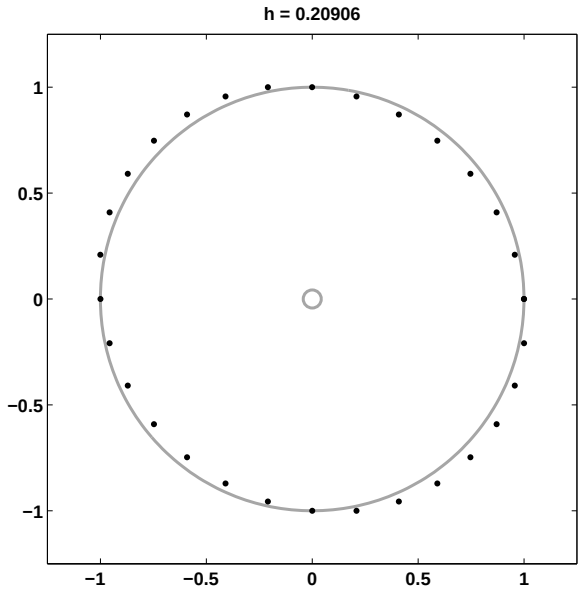
Why does this generate a circle? In fact, does it actually generate a circle? There are no trig functions, no square roots, no multiplications or divisions. It's all done with shifts and additions.

The key to this algorithm is the fact that the new `x` is used in the computation of the new `y`. This was convenient on computers at the time because it meant you needed only two storage locations, one for `x` and one for `y`. But, as we shall see, it is also why the algorithm comes close to working at all.

Here is a MATLAB version of the same algorithm.

```
h = 1/32;
x = 1;
y = 0;
while 1
   x = x + h*y;
   y = y - h*x;
   plot(x,y,'.')
   drawnow
end
```

The M-file `circlegen` lets you experiment with various values of the step size `h`. It provides an actual circle in the background. Here is the output for the carefully chosen default value, `h = .20906`. It's not quite a circle. However, `circlegen(h)` generates better circles with smaller values of `h`. Try `circlegen(h)` for various `h` yourself.



If we let (x_n, y_n) denote the nth point generated, then the iteration is

$$\begin{aligned} x_{n+1} &= x_n + hy_n \\ y_{n+1} &= y_n - hx_{n+1} \end{aligned}$$

The key is the fact that x_{n+1} appears on the right in the second equation. Substituting the first equation in the second gives

$$\begin{aligned} x_{n+1} &= x_n + hy_n \\ y_{n+1} &= -hx_n + (1 - h^2)y_n \end{aligned}$$

Let's switch to matrix-vector notation. Let x_n now denote the two-vector specifying the nth point and let A be the *circle generator* matrix

$$A = \begin{pmatrix} 1 & h \\ -h & 1-h^2 \end{pmatrix}$$

With this notation, the iteration is simply

$$x_{n+1} = Ax_n$$

This immediately leads to

$$x_n = A^n x_0$$

So, the question is, for various values of h, how do powers of the circle generator matrix behave?

For most matrices A, the behavior of A^n is determined by the its *eigenvalues*. The MATLAB statement

```
[V,E] = eig(A)
```

produces a *diagonal* eigenvalue matrix E and a corresponding eigenvector matrix V so that

$$AV = VE$$

If V^{-1} exists, then

$$A = VEV^{-1}$$

and

$$A^n = VE^nV^{-1}$$

Consequently, the powers A^n remain bounded if the eigenvector matrix is nonsingular and the eigenvalues λ_k, which are the diagonal elements of E, satisfy

$$|\lambda_k| \leq 1$$

Here is an easy experiment. Enter the line

```
h = 2*rand, A = [1 h; -h 1-h^2], lambda = eig(A), abs(lambda)
```

Repeatedly press the up arrow key, then the Enter key. You should eventually become convinced, at least experimentally, that

> For any h in the interval $0 < h < 2$, the eigenvalues of the circle generator matrix A are complex numbers with absolute value 1.

The Symbolic Toolbox provides some assistance in actually proving this fact.

```
syms h
A = [1 h; -h 1-h^2]
lambda = eig(A)
```

creates a symbolic version of the iteration matrix and finds its eigenvalues.

```
A =
[     1,     h]
[    -h, 1-h^2]

lambda =
[ 1-1/2*h^2+1/2*(-4*h^2+h^4)^(1/2)]
[ 1-1/2*h^2-1/2*(-4*h^2+h^4)^(1/2)]
```

The statement

```
abs(lambda)
```

does not do anything useful, in part because we have not yet made any assumptions about the symbolic variable `h`.

We note that the eigenvalues will be complex whenever the quantity involved in the square root is negative, that is, when $|h| < 2$. The determinant of a matrix should be the product of its eigenvalues. This is confirmed with

```
d = det(A)
```

or

```
d = simple(prod(lambda))
```

Both produce

```
d =
1
```

Consequently, when $|h| < 2$, the eigenvalues, λ, are complex and their product is 1, so they must satisfy $|\lambda| = 1$.

Because

$$\lambda = 1 - h^2/2 \pm h\sqrt{-1 + h^2/4}$$

it is plausible that, if we define θ by

$$\cos\theta = 1 - h^2/2$$

or

$$\sin\theta = h\sqrt{1 - h^2/4}$$

then

$$\lambda = \cos\theta \pm i\sin\theta$$

The Symbolic Toolbox confirms this with

```
theta = acos(1-h^2/2);
Lambda = [cos(theta)-i*sin(theta); cos(theta)+i*sin(theta)]
diff = simple(lambda-Lambda)
```

which produces

```
Lambda =
[ 1-1/2*h^2-1/2*i*(4*h^2-h^4)^(1/2)]
[ 1-1/2*h^2+1/2*i*(4*h^2-h^4)^(1/2)]

diff =
[ 0]
[ 0]
```

In summary, this proves that, if $|h| < 2$, the eigenvalues of the circle generator matrix are

$$\lambda = e^{\pm i\theta}$$

The eigenvalues are distinct, hence V must be nonsingular and

$$A^n = V \begin{pmatrix} e^{in\theta} & 0 \\ 0 & e^{-in\theta} \end{pmatrix} V^{-1}$$

If the step size h happens to correspond to a value of θ that is $2\pi/p$ where p is an integer, then the algorithm generates only p discrete points before it repeats itself.

How close does our circle generator come to actually generating circles? In fact, it generates ellipses. As the step size h gets smaller, the ellipses get closer to circles. The *aspect ratio* of an ellipse is the ratio of its major axis to its minor axis. It turns out that the aspect ratio of the ellipse produced by the generator is equal to the *condition number* of the matrix of eigenvectors, V. The condition number of a matrix is computed by the MATLAB function `cond(V)` and is discussed in more detail in the chapter on linear equations.

The solution to the 2-by-2 system of ordinary differential equations

$$\dot{x} = Qx$$

where

$$Q = \begin{pmatrix} 0 & 1 \\ -1 & 0 \end{pmatrix}$$

is a circle

$$x(t) = \begin{pmatrix} \cos t & \sin t \\ -\sin t & \cos t \end{pmatrix} x(0)$$

So, the iteration matrix

$$\begin{pmatrix} \cos h & \sin h \\ -\sin h & \cos h \end{pmatrix}$$

generates perfect circles. The Taylor series for $\cos h$ and $\sin h$ show that the iteration matrix for our circle generator

$$A = \begin{pmatrix} 1 & h \\ -h & 1 - h^2 \end{pmatrix}$$

approaches the perfect iterator as h gets small.

1.8 Floating Points

Some people believe that

- Numerical analysis is the study of floating-point arithmetic.
- Floating-point arithmetic is unpredictable and hard to understand.

We intend to convince you that both of these assertions are false. Very little of this book is actually about floating-point arithmetic. But when the subject does arise, we hope you will find floating-point arithmetic is not only computationally powerful, but also mathematically elegant.

If you look carefully at the definitions of fundamental arithmetic operations like addition and multiplication, you soon encounter the mathematical abstraction known as the real numbers. But actual computation with real numbers is not very practical because it involves limits and infinities. Instead, MATLAB and most other technical computing environments use floating-point arithmetic, which involves a finite set of numbers with finite precision. This leads to the phenomena of *roundoff error*, *underflow*, and *overflow*. Most of the time, it is possible to use MATLAB effectively without worrying about these details, but every once in a while, it pays to know something about the properties and limitations of floating-point numbers.

Twenty years ago, the situation was far more complicated than it is today. Each computer had its own floating-point number system. Some were binary; some were decimal. There was even a Russian computer that used trinary arithmetic. Among the binary computers, some used 2 as the base; others used 8 or 16. And everybody had a different precision. In 1985, the IEEE Standards Board and the American National Standards Institute adopted the ANSI/IEEE Standard 754-1985 for Binary Floating-Point Arithmetic. This was the culmination of almost a decade of work by a 92-person working group of mathematicians, computer scientists, and engineers from universities, computer manufacturers, and microprocessor companies.

All computers designed since 1985 use IEEE floating-point arithmetic. This doesn't mean that they all get exactly the same results, because there is some flexibility within the standard. But it does mean that we now have a machine-independent model of how floating-point arithmetic behaves.

MATLAB uses the IEEE double-precision format. There is also a single-precision format that saves space but isn't much faster on modern machines. And, there is an extended precision format, which is optional and therefore is one of the reasons for lack of uniformity among different machines.

Most nonzero floating-point numbers are normalized. This means they can be expressed as

$$x = \pm(1 + f) \cdot 2^e$$

where f is the fraction or mantissa and e is the exponent. The fraction satisfies

$$0 \le f < 1$$

and must be representable in binary using at most 52 bits. In other words, $2^{52}f$ is an integer in the interval

$$0 \le 2^{52}f < 2^{52}$$

The exponent e is an integer in the interval

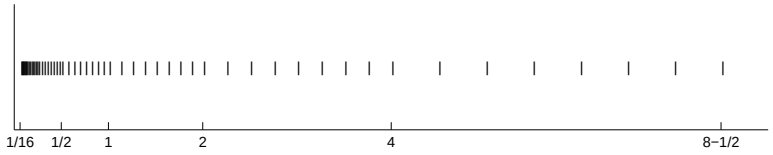
$$-1022 \le e \le 1023$$

The finiteness of f is a limitation on *precision*. The finiteness of e is a limitation on *range*. Any numbers that don't meet these limitations must be approximated by ones that do.

Double precision floating-point numbers can be stored in a 64 bit word, with 52 bits for f, 11 bits for e, and one bit for the sign of the number. The sign of e is accommodated by storing $e + 1023$, which is between 1 and $2^{11} - 2$. The two extreme values for the exponent field, 0 and $2^{11} - 1$, are reserved for exceptional floating-point numbers that we will describe later.

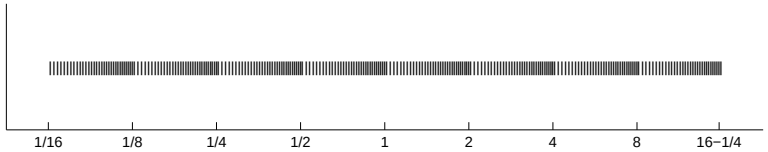
The entire fractional part of a floating-point number is not f, but $1 + f$, which has 53 bits. However the leading 1 doesn't need to be stored. In effect, the IEEE format packs 65 bits of information into a 64 bit word.

The program `floatgui` shows the distribution of the positive numbers in a model floating-point system with variable parameters. The parameter t specifies the number of bits used to store f. In other words, $2^t f$ is an integer. The parameters e_{min} and e_{max} specify the range of the exponent, so $e_{min} \le e \le e_{max}$. Initially, `floatgui` sets $t = 3$, $e_{min} = -4$, and $e_{max} = 3$ and produces this distribution



Within each binary interval, $2^e \le x \le 2^{e+1}$, the numbers are equally spaced, with an increment of 2^{e-t}. When $e = 0$ and $t = 3$, for example, the spacing of the numbers between 1 and 2 is 1/8. As e increases, the spacing increases.

It is also instructive to display the floating-point numbers with a logarithmic scale. Here is `floatgui` with `logscale` checked and $t = 5$, $e_{min} = -4$ and $e_{max} = 3$. With this logarithmic scale, it is more apparent that the distribution in each binary interval is the same.



A very important quantity associated with floating-point arithmetic is highlighted in red by `floatgui`. MATLAB calls this quantity `eps`, which is short for *machine epsilon*.

`eps` *is the distance from* 1 *to the next larger floating-point number.*

For the `floatgui` model floating-point system, `eps = 2^(-t)`.

Before the IEEE standard, different machines had different values of `eps`. Now, for IEEE double-precision,

```
eps = 2^(-52)
```

The approximate decimal value of `eps` is $2.2204 \cdot 10^{-16}$. Either `eps/2` or `eps` can be called the roundoff level. The maximum relative error incurred when the result of an arithmetic operation is rounded to the nearest floating-point number is `eps/2`. The maximum relative spacing between numbers is `eps`. In either case, you can say that the roundoff level is about 16 decimal digits.

A frequent instance of roundoff error occurs with the simple MATLAB statement

```
t = 0.1
```

The value stored in `t` is not exactly 0.1 because expressing the decimal fraction 1/10 in binary requires an infinite series. In fact,

$$\frac{1}{10} = \frac{1}{2^4} + \frac{1}{2^5} + \frac{0}{2^6} + \frac{0}{2^7} + \frac{1}{2^8} + \frac{1}{2^9} + \frac{0}{2^{10}} + \frac{0}{2^{11}} + \frac{1}{2^{12}} + ...$$

After the first term, the sequence of coefficients $1, 0, 0, 1$ is repeated infinitely often. The floating-point number nearest 0.1 is obtained by terminating this series after 53 terms. The infinite series beginning with the 54th term is larger than half a unit in the 53rd place, so the the last four coefficients are rounded up to binary 1010. Grouping the resulting terms together four at a time expresses the approximation as a base 16, or hexadecimal, series. So the resulting value of `t` is actually

$$t = (1 + \frac{9}{16} + \frac{9}{16^2} + \frac{9}{16^3} + ... + \frac{9}{16^{12}} + \frac{10}{16^{13}}) \cdot 2^{-4}$$

The MATLAB command

```
format hex
```

causes `t` to be displayed as

```
3fb999999999999a
```

The characters `a` through `f` represent the hexadecimal "digits" 10 through 15. The first three characters, `3fb`, give the hexadecimal representation of decimal 1019, which is the value of the biased exponent, $e + 1023$, when e is -4. The other 13 characters are the hex representation of the fraction f.

In summary, the value stored in `t` is very close to, but not exactly equal to, 0.1. The distinction is occasionally important. For example, the quantity

```
0.3/0.1
```

is not exactly equal to 3 because the actual numerator is a little less than 0.3 and the actual denominator is a little greater than 0.1.

Ten steps of length `t` are not precisely the same as one step of length 1. MATLAB is careful to arrange that the last element of the vector

```
0:0.1:1
```

is exactly equal to 1, but if you form this vector yourself by repeated additions of 0.1, you will miss hitting the final 1 exactly.

What does the floating-point approximation to the golden ratio look like?

```
format hex
phi = (1 + sqrt(5))/2
```

produces

```
phi =
   3ff9e3779b97f4a8
```

The first hex digit, 3, is 0011 in binary. The first bit is the sign of the floating-point number; 0 is positive, 1 is negative. So `phi` is positive. The remaining bits of the first three hex digits contain $e + 1023$. In this example, `3ff` in base 16 is $3 \cdot 16^2 + 15 \cdot 16 + 15 = 1023$ in decimal. So

$$e = 0$$

In fact, any floating-point number between 1.0 and 2.0 has $e = 0$, so its `hex` output begins with `3ff`. The other 13 hex digits contain f. In this example,

$$f = \frac{9}{16} + \frac{14}{16^2} + \frac{3}{16^3} + ... + \frac{10}{16^{12}} + \frac{8}{16^{13}}$$

With these values of f and e

$$(1 + f)2^e \approx \phi$$

Another example is provided by the following code segment.

```
format long
x = 4/3 - 1
y = 3*x
z = 1 - y
```

The output produced is

```
x =
   0.33333333333333
y =
   1.00000000000000
z =
   2.220446049250313e-016
```

With exact computation, `z` would be 0. But in floating-point, the computed `z` is actually equal to `eps`. It turns out that the only roundoff error occurs in division in the first statement. The quotient cannot be exactly 4/3, except on that Russian trinary computer. Consequently the value stored in `x` is close to, but not exactly equal to, 1/3. Moreover, its last bit is zero. This means that the multiplication `3*x` can be done without any roundoff error. The value stored in `y` is not exactly equal to 1 and so the value stored in `z` is not 0. Before the IEEE standard, this code was used as a quick way to estimate the roundoff level on various computers.

The roundoff level `eps` is sometimes called *floating-point zero*, but that's a misnomer. There are many floating-point numbers much smaller than `eps`. The smallest positive normalized floating-point number has $f = 0$ and $e = -1022$. The largest floating-point number has f a little less than 1 and $e = 1023$. MATLAB calls these numbers `realmin` and `realmax`. Together with `eps`, they characterize the standard system.

```
              Binary            Decimal
eps           2^(-52)           2.2204e-16
realmin       2^(-1022)         2.2251e-308
realmax       (2-eps)*2^1023    1.7977e+308
```

When any computation tries to produce a value larger than `realmax`, it is said to *overflow*. The result is an exceptional floating-point value called *infinity* or `Inf`. It is represented by taking $e = 1024$ and $f = 0$ and satisfies relations like `1/Inf = 0` and `Inf+Inf = Inf`.

When any computation tries to produce a value that is undefined even in the real number system, the result is an exceptional value known as Not-a-Number, or `NaN`. Examples include `0/0` and `Inf-Inf`. `NaN` is represented by taking $e = 1024$ and f nonzero.

When any computation tries to produce a value smaller than `realmin`, it is said to *underflow*. This involves one of the optional, and controversial, aspects of the IEEE standard. Many, but not all, machines allow exceptional denormal or subnormal floating-point numbers in the interval between `realmin` and `eps*realmin`. The smallest positive subnormal number is about `0.494e-323`. Any results smaller than this are set to zero. On machines without subnormals, any results less than `realmin` are set to zero. The subnormal numbers fill in the gap you can see in the `floatgui` model system between zero and the smallest positive number. They do provide an elegant way to handle underflow, but their practical importance for MATLAB style computation is very rare. Denormal numbers are represented by taking $e = -1023$, so the biased exponent $e + 1023$ is zero.

MATLAB uses the floating-point system to handle integers. Mathematically, the numbers 3 and 3.0 are the same, but many programming languages would use different representations for the two. MATLAB does not distinguish between them. We sometimes use the term *flint* to describe a floating-point number whose value is an integer. Floating-point operations on flints do not introduce any roundoff error, as long as the results are not too large. Addition, subtraction, and multiplication of flints produce the exact flint result, if it is not larger than 2^{53}. Division and square root involving flints also produce a flint when the result is an integer. For example,

`sqrt(363/3)` produces `11`, with no roundoff error.

Two MATLAB functions that take apart and put together floating point numbers are `log2` and `pow2`.

```
help log2
help pow2
```

produces

```
[F,E] = LOG2(X) for each element of the real array X,
returns an array F of real numbers, usually in the range
0.5 <= abs(F) < 1, and an array E of integers, so that
X = F .* 2.^E.  Any zeros in X produce F = 0 and E = 0.

X = POW2(F,E) for each element of the real array F and
an integer array E computes X = F .* (2 .^ E).  The result
is computed quickly by simply adding E to the floating
point exponent of F.
```

The quantities `F` and `E` used by `log2` and `pow2` predate the IEEE floating-point standard and so are slightly different from the f and e we are using in this section. In fact, f = `2*F-1` and e = `E-1`.

```
[F,E] = log2(phi)
```

produces

```
F =
   0.80901699437495
E =
     1
```

Then

```
phi = pow2(F,E)
```

gives back

```
phi =
   1.61803398874989
```

As an example of how roundoff error affects matrix computations, consider the two-by-two set of linear equations

$$\begin{aligned} 17x_1 + 5x_2 &= 22 \\ 1.7x_1 + 0.5x_2 &= 2.2 \end{aligned}$$

The obvious solution is $x_1 = 1, x_2 = 1$. But the MATLAB statements

```
A = [17 5; 1.7 0.5]
b = [22; 2.2]
x = A\b
```

produce

```
x =
  -1.0588
   8.0000
```

Where did this come from? Well, the equations are singular, but consistent. The second equation is just 0.1 times the first. The computed x is one of infinitely many possible solutions. But the floating-point representation of the matrix A is not exactly singular because `A(2,1)` is not exactly 17/10.

The solution process subtracts a multiple of the first equation from the second. The multiplier is `mu = 1.7/17`, which turns out to be the floating-point number obtained by truncating, rather than rounding, the binary expansion of 1/10. The matrix `A` and the right-hand side `b` are modified by

```
A(2,:) = A(2,:) - mu*A(1,:)
b(2) = b(2) - mu*b(1)
```

With exact computation, both `A(2,2)` and `b(2)` would become zero, but with floating-point arithmetic, they both become nonzero multiples of `eps`.

```
A(2,2) = (1/4)*eps
       = 5.5511e-17
b(2) = 2*eps
     = 4.4408e-16
```

MATLAB notices the tiny value of the new `A(2,2)` and displays a message warning that the matrix is close to singular. It then computes the solution of the modified second equation by dividing one roundoff error by another.

```
x(2) = b(2)/A(2,2)
     = 8
```

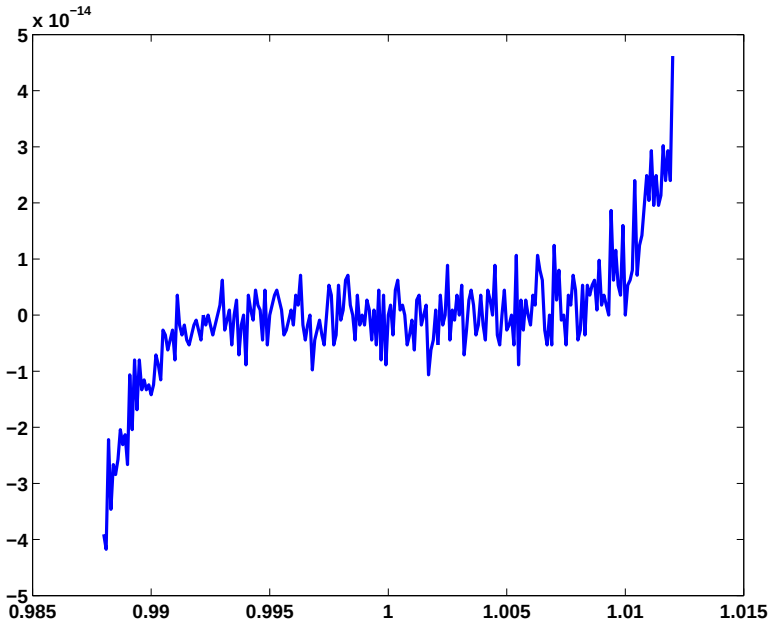
This value is substituted back into the first equation to give

```
x(1) = (22 - 5*x(2))/17
     = -1.0588
```

The details of the roundoff error lead MATLAB to pick out one particular solution from among the infinitely many possible solutions to the singular system.

Our final example plots a seventh degree polynomial.

```
x = 0.988:.0001:1.012;
y = x.^7-7*x.^6+21*x.^5-35*x.^4+35*x.^3-21*x.^2+7*x-1;
plot(x,y)
```



The resulting plot doesn't look anything like a polynomial. It isn't smooth. You are seeing roundoff error in action. The y axis scale factor is tiny, 10^{-14}. The tiny values of y are being computed by taking sums and differences of numbers as large as $35 \cdot 1.012^4$. There is severe subtractive cancellation. The example was contrived by using the Symbolic Toolbox to expand $(x-1)^7$ and carefully choosing the range for the x axis to be near x = 1. If the values of `y` are computed instead by

```
y = (x-1).^7;
```

then a smooth (but very flat) plot results.

Exercises

1.1. Which of these familiar rectangles is closest to a golden rectangle? Use MATLAB to do the calculations with an element-by-element vector division, `w./h`.

- 3-by-5 inch index card
- 8.5-by-11 inch US letter paper
- 8.5-by-14 inch US legal paper
- 9-by-12 foot rug
- 9:16 "letterbox" TV picture
- 768-by-1024 pixel computer monitor

1.2. ISO standard A4 paper is commonly used throughout the world, except in the United States. Its dimensions are 210 by 297 millimeters. This is not a golden rectangle, but the aspect ratio is close to another familiar irrational mathematical quantity. What is that quantity? If you fold a piece of A4 paper in half, what is the aspect ratio of each of the halves? Modify the M-file `goldrect.m` to illustrate this property.

1.3. How many terms in the truncated continued fraction does it take to approximate ϕ with an error less than 10^{-10}? As the number of terms increases beyond this, roundoff error eventually intervenes. What is the best accuracy you can hope to achieve with double-precision floating-point arithmetic and how many terms does it take?

1.4. 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 next chapter of this book, or with the commands

```
help \
help slash
```

1.5. 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?

1.6. 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)`.

1.7. 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?

1.8. Enter the statements

```
A = [1 1; 1 0]
X = [1 0; 0 1]
```

Then enter the statement

```
X = A*X
```

Now repeatedly press the up arrow key, followed by the Enter key. What happens? Do you recognize the matrix elements being generated? How many times would you have to repeat this iteration before X overflows?

1.9. Change the fern color scheme to use pink on a black background. Don't forget the stop button.

1.10. What happens if you resize the figure window while the fern is being generated? Why?

1.11. Flip the fern by interchanging its x and y coordinates.

1.12. What happens to the fern if you change the only nonzero element in the matrix `A4`?

1.13. In the following statements, change `how_often` to some reasonable value and insert the statements in appropriate places in `fern.m`.

```
htitle = title('0');

if rem(cnt,how_often) == 0
   set(htitle,'string',num2str(cnt))
end
```

1.14. What are the coordinates of the lower end of the fern's stem?

1.15. 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?

1.16. 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`.

1.17. The fern algorithm involves repeated random choices from four different formulas for advancing the point. When the kth formula is used repeatedly by itself, without random choices, it defines a deterministic trajectory, $T_k,\ k = 1, ..., 4$, 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. Each trajectory starts at $(\frac{1}{2}, \frac{1}{2})$ and approaches some limit point, z_k. Compute these limit points and plot a `'o'` there. You can superimpose several plots with

```
plot(...)
hold on
plot(...)
plot(...)
hold off
```

1.18. Modify `fern.m` or `finitefern.m` so that it produces *Sierpinski's triangle*. Start at

$$x = \begin{pmatrix} 1/2 \\ 1/2 \end{pmatrix}$$

At each iterative step the current point x is replaced by $Ax + b$ 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

$$b = \begin{pmatrix} 1/2 \\ 0 \end{pmatrix}, \quad b = \begin{pmatrix} 0 \\ 1/2 \end{pmatrix}, \quad \text{or } b = \begin{pmatrix} 0 \\ 0 \end{pmatrix}$$

1.19. `A = magic(4)` is singular. Its columns are linearly dependent. What do `null(A)`, `null(A,'r')`, `null(sym(A))`, and `rref(A)` tell you about that dependence?

1.20. Let `A = magic(n)` for `n = 3, 4,` or `5`. What does

```
p = randperm(n); q = randperm(n); A = A(p,q);
```

do to

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

1.21. The character `char(7)` is a control character. What does it do?

1.22. What does `char([169 174])` display on your machine?

1.23. What fundamental physical law is hidden in this string?

```
s = '/b_t3{$H~MO6JTQI>v~#3Gieyjl(p,nF'
```

You should be able to enter most of the string with a cut and paste operation. Then set

```
s(25) = char(255)
```

1.24. Find the two files `encrypt.m` and `gettysburg.dat`. Use `encrypt` to encrypt `gettysburg.dat`. Then decrypt the result. Use `encrypt` to encrypt itself.

1.25. If `x` is the character string consisting of just two blanks,

```
x = '  '
```

then `crypto(x)` is actually equal to `x`. Why does this happen? Are there any other two-character strings that `crypto` does not change?

1.26. Find another 2-by-2 integer matrix `A` for which

```
mod(A*A,97)
```

is the identity matrix. Replace the matrix in `crypto.m` with your matrix and verify that the function still works correctly.

1.27. The function `crypto` works with 97 characters instead of 95, so it can produce output, and correctly handle input, that contains two characters with ASCII values greater than 127. What are these characters? Choose two other characters in the extended set and change `crypto` to use these two instead.

1.28. Create a new `crypto` function that works with just 29 characters, the 26 lowercase letters, plus blank, period, and comma. You will need to find a 2-by-2 integer matrix `A` for which `mod(A*A,29)` is the identity matrix.

1.29. The graph of the $3n+1$ sequence has a particular characteristic shape if the starting n is $5, 10, 20, 40, \ldots$, that is, n is five times a power of two. What is this shape and why does it happen?

1.30. The graphs of the $3n+1$ sequences starting at $n = 108, 109$, and 110 are very similar to each other. Why?

1.31. Let $L(n)$ be the number of terms in the $3n+1$ sequence that starts with n. Write a MATLAB function that computes $L(n)$ *without* using any vectors or unpredictable amounts of storage. Plot $L(n)$ for $1 \le n \le 1000$. What is the maximum value of $L(n)$ for n in this range, and for what value of n does it occur? Use `threenplus1` to plot the sequence that starts with this particular value of n.

1.32. How was the default value of the step size `h` for `circlegen` chosen?

1.33. Modify `circlegen` so that both components of the new point are determined from the old point, that is,

$$\begin{aligned} x_{n+1} &= x_n + hy_n \\ y_{n+1} &= y_n - hx_n \end{aligned}$$

(This is the *explicit Euler's method* for solving the circle ordinary differential equation.) What happens to the "circles"? What is the iteration matrix? What are its eigenvalues?
Modify `circlegen` so that the new point is determined by solving a 2-by-2 system of simultaneous equations.

$$\begin{aligned} x_{n+1} - hy_{n+1} &= x_n \\ y_{n+1} + hx_{n+1} &= y_n \end{aligned}$$

(This is the *implicit Euler's method* for solving the circle ordinary differential equation.) What happens to the "circles"? What is the iteration matrix? What are its eigenvalues?

1.34. Modify `circlegen` so that it keeps track of the maximum and minimum radius during the iteration and returns the ratio of these two radii as the value of the function. Compare this computed aspect ratio with the eigenvector condition number, `cond(V)`, for various values of `h`.

1.35. What happens with `circlegen(sqrt(2))`? (Don't answer too quickly.)

1.36. Modify `circlegen` so that you can investigate the behavior of the circle-generating algorithm for $\sqrt{2} < h < 2$.

1.37. What is the circle generator iteration matrix A when $h = 2$? How does A^n behave? Can you explain this behavior using just the eigenvalues of A? What is the condition number of the eigenvector matrix, V?

1.38. What happens with the circle-generating algorithm when $h > 2$?

1.39. Modify `floatgui.m` by changing its last line from a comment to an executable statement and changing the question mark to a simple expression that counts

the number of floating-point numbers in the model system.

1.40. Explain the output produced by

```
t = 0.1
n = 1:10
e = n/10 - n*t
```

1.41. What does each of these programs do? How many lines out output does each program produce? What are the last two values of `x` printed?

```
x = 1; while 1+x > 1, x = x/2, pause(.02), end

x = 1; while x+x > x, x = 2*x, pause(.02), end

x = 1; while x+x > x, x = x/2, pause(.02), end
```

1.42. Which familiar real numbers are approximated by floating-point numbers that display the following values with `format hex`?

```
4059000000000000
3f847ae147ae147b
3fe921fb54442d18
```

1.43. Let F be the set of all IEEE double precision floating point numbers, except `NaN`s and `Inf`s, which have biased exponent `7ff`(hex), and denormals, which have biased exponent `000`(hex).
(a) How many elements are there in F?
(b) What fraction of the elements of F are in the interval $1 <= x < 2$?
(c) What fraction of the elements of F are in the interval $1/64 <= x < 1/32$?
(d) Determine by random sampling approximately what fraction of the elements `x` of F satisfy the MATLAB logical relation

```
x*(1/x) == 1
```

1.44. The classic quadratic formula says that the two roots of the quadratic equation

$$ax^2 + bx + c = 0$$

are

$$x_1, x_2 = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

Use this formula in MATLAB to compute both roots when

$$a = 1, b = -100000000, c = 1$$

Compare your computed results with

```
roots([a b c])
```

What happens if you try to compute the roots by hand or with a hand calculator?
You should find that the classic formula is good for computing one root, but not the other. So use it to compute one root accurately and then use the fact that

$$x_1 x_2 = \frac{c}{a}$$

to compute the other.

1.45. The power series for $\sin x$ is

$$\sin x = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + ...$$

Here is a MATLAB function that uses this series to compute $\sin x$.

```
function s = powersin(x)
% POWERSIN.  Power series for sin(x).
% POWERSIN(x) tries to compute sin(x) from a power series
s = 0;
t = x;
n = 1;
while s+t ~= s;
   s = s + t;
   t = -x.^2/((n+1)*(n+2)).*t;
   n = n + 2;
end
```

What causes the `while` loop to terminate?
Answer each of the following questions for $x = \pi/2$, $11\pi/2$, $21\pi/2$, and $31\pi/2$:

How accurate is the computed result?

How many terms are required?

What is the largest term in the series?

What do you conclude about the use of floating-point arithmetic and power series to evaluate functions?

1.46. In the Gregorian calendar, a year `y` is a *leap year* if and only if

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

Thus, 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 MATLAB functions `datenum`, `datevec`, `datestr`, and `weekday` use these facts to facilitate computations involving calendar dates. For example, either of the statements

```
[d,w] = weekday('Aug. 17, 2002')
```

or

```
[d,w] = weekday(datenum([2002 8 17]))
```

tells me that my birthday was on a Saturday in 2002..
Use MATLAB to answer the following questions.
(a) On what day of the week were you born?
(b) In a 400 year Gregorian calendar cycle, which week day is the most likely for your birthday?
(c) What is the probability that the 13th of any month falls on a Friday? The answer is close to, but not exactly equal to, 1/7.