Chapter 1

Power Tools of the Trade

MATLAB is a matrix-vector-oriented system that supports a wide range of activity that is crucial to the computational scientist. In this chapter we get acquainted with this system through a collection of examples that sets the stage for the proper study of numerical computation. The MATLAB environment is very easy to use and you might start right now by running `demo`. Our introduction in this chapter previews the central themes that occur with regularity in the following chapters.

We start with the exercise of plotting. MATLAB has an extensive array of visualization tools. But even the simplest plot requires setting up a vector of function values, and so very quickly we are led to the many vector-level operations that MATLAB supports. Our mission is to build up a linear algebra sense to the extent that vector-level thinking becomes as natural as scalar-level thinking. MATLAB encourages this in many ways, and plotting is the perfect start-up topic. The treatment is spread over two sections.

Building environments that can be used to explore mathematical and algorithmic ideas is the theme of §1.3. A pair of random simulations is used to illustrate how MATLAB can be used in this capacity.

In §1.4 we learn how to think and reason about error. Error is a fact of life in computational science, and our examples are designed to build an appreciation for two very important types of error. Mathematical errors result when we take what is infinite or continuous and make it finite or discrete. Rounding errors arise because floating-point representation and arithmetic is inexact.

§1.5 is devoted to the art of designing effective functions. The user-defined function is a fundamental building block in scientific computation. More complicated data structures are discussed in §1.6, while in the last section we point to various techniques that can be used to enrich the display of visual data.

1.1 Vectors and Plotting

Suppose we want to plot the function $f(x) = \sin(2\pi x)$ across the interval $[0, 1]$. In MATLAB there are three components to this task.

- A vector of x-values that range across the interval must be set up:

$$0 = x_1 < x_2 < \cdots < x_n = 1.$$

- The function must be evaluated at each x-value:

$$y_k = f(x_k), \qquad k = 1, \ldots, n.$$

- A polygonal line that connects the points $(x_1, y_1), \ldots, (x_n, y_n)$ must be displayed.

If we take 21 equally spaced x-values, then the result looks like the plot shown in Figure 1.1. The plot is

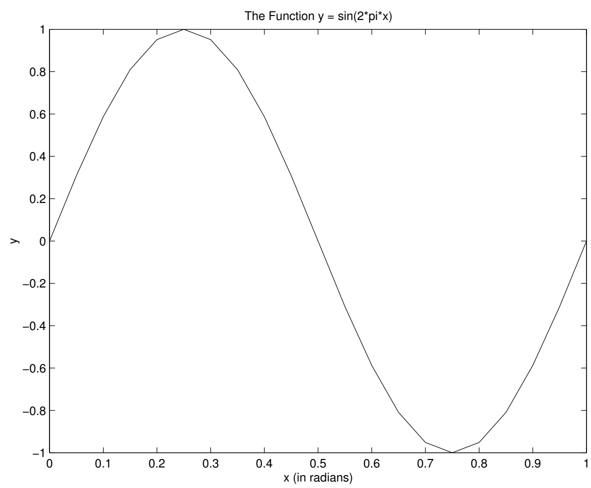


FIGURE 1.1 *A crude plot of* $\sin(2\pi x)$

"crude" because the polygonal effect is noticeable in regions where the function is changing rapidly. But otherwise the graph looks quite good. Our introduction to MATLAB begins with the details of the plotting process and the vector computations that go along with it. The $\sin(2\pi x)$ example is used throughout because it is simple and structured. Exploiting that structure leads naturally to some vector operations that are well supported in the MATLAB environment.

1.1.1 Setting Up Vectors

When you invoke the MATLAB system, you enter the *command window* and are prompted to enter commands with the symbol ">>". For example,

```
>> x = [10.1 20.2 30.3]
```

MATLAB is an interactive environment and it responds with

```
x =

   10.1000   20.2000   30.3000

>>
```

This establishes `x` as a length-3 row vector. Square brackets delineate the vector and spaces separate the components. On the other hand, the exchange

```
>> x = [ 10.1; 20.2; 30.3]
x =
   10.1000
   20.2000
   30.3000
```

establishes `x` as a length-3 column vector. Again, square brackets define the vector being set up. But this time semicolons separate the component entries and a column vector is produced.

In general, MATLAB displays the consequence of a command unless it is terminated with a semicolon. Thus,

```
>> x = [ 10.1; 20.2; 30.3];
```

sets up the same column 3-vector as in the previous example, but there is no echo that displays the result. However, the dialog

```
x = [10.1; 20.2; 30.3];
x

x =
   10.1000
   20.2000
   30.3000
```

shows that the contents of a vector can be displayed merely by entering the name of the vector. Even if one component in a vector is changed with no terminating semicolon, MATLAB displays the whole vector:

```
x = [10.1; 20.2; 30.3];
x(2) = 21

x =

   10.1000
   21.0000
   30.3000
```

It is clear that when dealing with large vectors, a single forgotten semicolon can result in a deluge of displayed output.

To change the orientation of a vector from row to column or column to row, use an apostrophe. Thus,

```
x = [10.1 20.2 30.3]'
```

establishes `x` as a length-3 column vector. Placing an apostrophe after a vector effectively takes its transpose.

The plot shown in Figure 1.1 involves the equal spacing of $n = 21$ x-values across $[0, 1]$; that is

```
x = [0 .05 .10 .15 .20 .25 .30 .35 .40 .45 .50 ...
           .55 .60 .65 .70 .75 .80 .85 .90 .95 1.0 ]
```

The ellipsis symbol "`...`" permits the entry of commands that occupy more than one line.

It is clear that for even modest values of n, we need other mechanisms for setting up vectors. Naturally enough, a `for`-loop can be used:

```
n = 21;
h = 1/(n-1);
for k=1:n
   x(k) = (k-1)*h;
end
```

This is a MATLAB *script*. It assigns the same length-21 vector to `x` as before and it brings up an important point.

> *In* MATLAB*, variables are not declared by the user but are created on a need-to-use basis by a memory manager. Moreover, from* MATLAB*'s point of view, every simple variable is a complex matrix indexed from unity.*

Scalars are 1-by-1 matrices. Vectors are "skinny" matrices with either one row or one column. We have much more to say about "genuine" matrices later. Our initial focus is on real vectors and scalars.

In the preceding script, `n`, `h`, `k`, and `x` are variables. It is instructive to trace how `x` "turns into" a vector during the execution of the `for`-loop. After one pass through the loop, `x` is a length-1 vector (i.e., a scalar). During the second pass, the reference `x(2)` prompts the memory manager to make `x` a 2-vector. During the third pass, the reference `x(3)` prompts the memory manager to make `x` a 3-vector. And so it goes until by the end of the loop, `x` has length 21. It is a convention in MATLAB that this kind of vector construction yields row vectors.

The MATLAB `zeros` function is handy for setting up the shape and size of a vector prior to a loop that assigns it values. Thus,

```
n = 21;
h = 1/(n-1);
x = zeros(1,n);
for k=1:n;
   x(k) = (k-1)*h;
end
```

computes `x` as row vector of length-21 and initializes the values to zero. It then proceeds to assign the appropriate value to each of the 21 components. Replacing `x = zeros(1,n)` with the command `x = zeros(n,1)` sets up a length-21 column vector. This style of vector set-up is recommended for two reasons. First, it forces you to think explicitly about the orientation and length of the vectors that you are working with. This reduces the chance for "dimension mismatch" errors when vectors are combined. Second, it is more efficient because the memory manager does not have to "work" so hard with each pass through the loop.

MATLAB supplies a `length` function that can be used to probe the length of any vector. To illustrate its use, the script

```
u = [10 20 30];
n = length(u);
v = [10;20;30;40];
m = length(v);
u = [50 60];
p = length(u);
```

assigns the values of 3, 4, and 2 to `n`, `m`, and `p`, respectively.

This brings up another important feature of MATLAB. It supports a very extensive `help` facility. For example, if we enter

```
help length
```

then MATLAB responds with

```
 LENGTH Number of components of a vector.
  LENGTH(X) returns the length of vector X.  It is equivalent
  to MAX(SIZE(X)).
```

So extensive and well structured is the help facility that it obviates the need for us to go into excessive detail when discussing many of MATLAB's capabilities. Get in the habit of playing around with each new MATLAB feature that you learn, exploring the details via the `help` facility. Start right now by trying

```
help help
```

Here in Chapter 1 there are many occasions to use the help facility as we proceed to acquire enough familiarity with the system to get started. Before continuing, you are well advised to try

```
 help who
 help whos
 help clear
```

to learn more about the management of memory. We have already met a number of MATLAB language features and functions. You can organize your own mini-review by entering

```
help for
help zeros
help ;
help []
```

1.1.2 Regular Vectors

Regular vectors arise so frequently that MATLAB has a number of features that support their construction. With the *colon notation* it is possible to establish row vectors whose components are equally spaced. The command

```
x  = 20:24
```

is equivalent to

```
x = [ 20 21 22 23 24]
```

The spacing between the component values is called the *stride* and the vector `x` has unit stride. Nonunit strides can also be specified. For example,

```
x = 20:2:29;
```

This stride-2 vector is the same as

```
x = [20 22 24 26 28]
```

Negative strides are also permissible. The assignment

```
x = 10:-1:1
```

is equivalent to

```
x = [ 10 9 8 7 6 5 4 3 2 1]
```

As seen from the examples, the general use of the colon notation has the form

$$\langle \textit{Starting Index} \rangle : \langle \textit{Stride} \rangle : \langle \textit{Bounding Index} \rangle$$

If the starting index is beyond the bounding index, then the *empty vector* is produced:

```
x = 3:2

x =
     []
```

The empty vector has length zero and is denoted with a square bracket pair with nothing in between.

The colon notation also works with nonintegral values. The command

```
x = 0:.05:1
```

sets up a length-21 row vector with the property that $x_i = (i-1)/20,\ i = 1, \ldots, 21$. Alternatively, we could multiply the vector `0:20` by the scalar `.05`:

```
x = .05*(0:20)
```

However, if nonintegral strides are involved, then it is preferable to use the `linspace` function. If `a` and `b` are real scalars, then

```
x = linspace(a,b,n)
```

returns a row vector of length n whose kth entry is given by

$$x_k = a + (k-1) * (b-a)/(n-1).$$

For example,

```
x = linspace(0,1,21)
```

is equivalent to

```
x = [0 .05 .10 .15 .20 .25 .30 .35 .40 .45 .50 ...
           .55 .60 .65 .70 .75 .80 .85 .90 .95 1.0 ]
```

In general, a reference to `linspace` has the form

`linspace`(⟨*Left Endpoint*⟩,⟨*Right Endpoint*⟩,⟨*Number of Points*⟩)

Logarithmic spacing is also possible. The assignment

```
x = logspace(-2,3,6);
```

is the same as `x = [ .01 .1 1 10 100 1000]`. More generally, `x = logspace(a,b,n)` sets

$$x_k = 10^{a+(b-a)(k-1)/(n-1)}, \qquad k = 1, \ldots, n$$

and is equivalent to

```
m = linspace(a,b,n);
for k=1:n
   x(k) = 10^m(k);
end
```

The `linspace` and `logspace` functions bring up an important detail. Many of MATLAB's functions can be called with a reduced parameter list that is often useful in simple, canonical situations. For example, `linspace(a,b)` is equivalent to `linspace(a,b,100)` and `logspace(a,b)` is equivalent to `logspace(a,b,50)`. Make a note of these shortcuts as you become acquainted with MATLAB's many features.

So far we have not talked about how MATLAB displays results except to say that if a semicolon is left off the end of a statement, then the consequences of that statement are displayed. Thus, if we enter

```
x = .123456789012345*logspace(1,5,5)'
```

then the vector `x` is displayed according to the active *format*. For example,

```
x =
   1.0e+04 *

    0.0001
    0.0012
    0.0123
    0.1235
    1.2346
```

The preceding is the `short` format. The `long`, `short e`, and `long e` formats are also handy as depicted in Figure 1.2. The `short` format is active when you first enter MATLAB. The `format` command is used to switch formats. For example,

```
format long
```

It is important to remember that the display of a vector is independent of its internal floating point representation, something that we will discuss in §1.4.4.

short	long	short e	long e
1.0e+14 *	1.0e+14 *		
0.0000	0.00000000000001	1.2346e+00	1.234567890123450e+00
0.0000	0.00000000000012	1.2346e+01	1.234567890123450e+01
0.0000	0.00000000000123	1.2346e+02	1.234567890123450e+02
0.0000	0.00000000001235	1.2346e+03	1.234567890123450e+03
0.0000	0.00000000012346	1.2346e+04	1.234567890123450e+04
0.0000	0.00000000123457	1.2346e+05	1.234567890123450e+05
0.0000	0.00000001234568	1.2346e+06	1.234567890123450e+06
0.0000	0.00000012345679	1.2346e+07	1.234567890123450e+07
0.0000	0.00000123456789	1.2346e+08	1.234567890123450e+08
0.0000	0.00001234567890	1.2346e+09	1.234567890123450e+09
0.0001	0.00012345678901	1.2346e+10	1.234567890123450e+10
0.0012	0.00123456789012	1.2346e+11	1.234567890123450e+11
0.0123	0.01234567890123	1.2346e+12	1.234567890123450e+12
0.1235	0.12345678901234	1.2346e+13	1.234567890123450e+13
1.2346	1.23456789012345	1.2346e+14	1.234567890123450e+14

FIGURE 1.2 *The display of* `.123456789012345*logspace(1,15,15)'`

1.1.3 Evaluating Functions

We return to the task of plotting $\sin(2\pi x)$. MATLAB comes equipped with a host of built-in functions including `sin`. (Enter `help elfun` to see the available elementary functions.) The script

```
n = 21;
x = linspace(0,1,n);
y = zeros(1,n);
for k=1:n
   y(k) = sin(2*pi*x(k));
end
```

sets up a vector of sine values that correspond to the values in `x`. But many of the built-in functions like `sin` accept vector arguments, and the preceding loop can be replaced with a single reference as follows:

```
n = 21;
x = linspace(0,1,n);
y = sin(2*pi*x);
```

The act of replacing a loop in MATLAB with a single vector-level operation will be referred to as *vectorization* and has three fringe benefits:

- *Speed.* Many of the built-in MATLAB functions provide the results of several calls faster if called once with the corresponding vector argument(s).
- *Clarity.* It is often easier to read a vectorized MATLAB script than its scalar-level counterpart.
- *Education.* Scientific computing on advanced machines requires that one be able to think at the vector level. MATLAB encourages this and, as the title of this book indicates, we have every intention of fostering this style of algorithmic thinking.

As a demonstration of the vector-level manipulation that MATLAB supports, we dissect the following script:

```
m = 5; n = 4*m+1;
x = linspace(0,1,n); a = x(1:m+1);
y = zeros(1,n);
y(1:m+1) = sin(2*pi*a);
y(2*m+1:-1:m+2) = y(1:m);
y(2*m+2:n) = -y(2:2*m+1);
```

which sets up the same vector `y` as before but with one-fourth the number of scalar sine evaluations. The idea is to exploit symmetries in the table shown in Figure 1.3. The script starts by assigning to `a` a *subvector*

k	x_k	$\sin(x_k)$
1	0	0.000
2	18	0.309
3	36	0.588
4	54	0.809
5	72	0.951
6	90	1.000
7	108	0.951
8	126	0.809
9	144	0.588
10	162	0.309
11	180	0.000
12	198	-0.309
13	216	-0.588
14	234	-0.809
15	252	-0.951
16	270	-1.000
17	288	-0.951
18	306	-0.809
19	324	-0.588
20	342	-0.309
21	360	-0.000

FIGURE 1.3 *Selected values of the sine function (x_k in degrees)*

of `x`. In particular, the assignment to `a` is equivalent to

```
a = [0.00  0.05  0.10  0.15  0.20  0.25]
```

In general, if `v` is a vector of integers that are valid subscripts for a row vector `z`, then

```
w = z(v);
```

is equivalent to

```
for k=1:length(v)
   w(k) = z(v(k));
end
```

The same idea applies to column vectors. Extracted subvectors have the same orientation as the parent vector.

Assignment to a subvector is also legal provided the named subscript range is valid. Thus,

```
y(1:m+1) = sin(2*pi*a);
```

is equivalent to

```
for k=1:m+1
   y(k) = sin(2*pi*a(k));
end
```

Now comes the first of two mathematical exploitations. The sine function has the property that

$$\sin\left(\frac{\pi}{2}+x\right) = \sin\left(\frac{\pi}{2}-x\right).$$

Thus,

$$\begin{bmatrix} \sin(10h) \\ \sin(9h) \\ \sin(8h) \\ \sin(7h) \\ \sin(6h) \end{bmatrix} = \begin{bmatrix} \sin(0h) \\ \sin(h) \\ \sin(2h) \\ \sin(3h) \\ \sin(4h) \end{bmatrix} \qquad h = 2\pi/20.$$

Note that the components on the left should be stored in reverse order in `y(7:11)`, while the components on the right have already been computed and are housed in `y(1:5)`. (See Figure 1.3.) The assignment

```
y(m+1:2*m+1) =  y(m:-1:1);
```

establishes the necessary values in `y(7:11)`.

At this stage, `y(1:2*m+1)` contains the sine values from $[0, \pi]$ that are required. To obtain the remaining values, we exploit a second trigonometric identity:

$$\sin(\pi + x) = -\sin(x).$$

We see that this implies

$$\begin{bmatrix} \sin(11h) \\ \sin(12h) \\ \sin(13h) \\ \sin(14h) \\ \sin(15h) \\ \sin(16h) \\ \sin(17h) \\ \sin(18h) \\ \sin(19h) \\ \sin(20h) \end{bmatrix} = - \begin{bmatrix} \sin(h) \\ \sin(2h) \\ \sin(3h) \\ \sin(4h) \\ \sin(5h) \\ \sin(6h) \\ \sin(7h) \\ \sin(8h) \\ \sin(9h) \\ \sin(10h) \end{bmatrix} \qquad h = 2\pi/20.$$

The sine values on the left belong in `y(12:21)` while those on the right have already been computed and occupy `y(2:11)`. Hence, the construction of `y(1:21)` is completed with the assignment

```
y(2*m+2:n) = -y(2:2*m+1);
```

(See Figure 1.3.)

Why go though such contortions when `y = sin(2*pi*linspace(0,1,21))` is so much simpler? The reason is that more often than not, function evaluations are expensive and one should always be searching for relationships that reduce their number. Of course, `sin` is not expensive. But the vector computations detailed in this subsection above are instructive because we must learn to be sparing when it comes to the evaluation of functions.

1.1.4 Displaying Tables

Any vector can be displayed by merely typing its name and leaving off the semicolon. However, sometimes a more customized output is preferred, and for that a facility with the `disp` and `sprintf` functions is required.

But before we can go any further we must introduce the concept of a *script file.* Already, our scripts are getting too long and too complicated to assemble line-by-line in the command window. The time has come to enlist the services of a text editor and to store the command sequence in a file that can then be executed.

To illustrate the idea, we set up a script file that can be used to display the table in Figure 1.3. We start by entering the following into a file named `SineTable.m`:

```
% Script File: SineTable
% Prints a short table of sine evaluations.
clc
n = 21;
x = linspace(0,1,n);
y = sin(2*pi*x);
disp(' ')
disp('  k     x(k)   sin(x(k))')
disp('------------------------')
for k=1:21
   degrees = (k-1)*360/(n-1);
   disp(sprintf(' %2.0f     %3.0f     %6.3f    ',k,degrees,y(k)));
end
disp( ' ');
disp('x(k) is given in degrees.')
disp(sprintf('One Degree = %5.3e Radians',pi/180))
```

The `.m` suffix is crucial, for then the preceding command sequence is executed merely by entering `SineTable` at the prompt:

```
>> SineTable
```

This displays the table shown in Figure 1.3, assuming that MATLAB can find `SineTable.m`. This is assured if the file is in the current working directory or if `path` is properly set. Review what you must know about key file organization by entering `help dir cd ls lookfor`.

Focusing on `SineTable` itself, there are a number of new features that we must explain. The script begins with a sequence of *comments* indicating what happens when it is run. Comments in MATLAB begin with the percent symbol "`%`". Aside from enhancing readability, the lead comments are important because they are displayed in response to a `help` enquiry. That is,

```
help SineTable
```

Use `type` to list the entire contents of a file, e.g.,

```
type SineTable
```

The `clc` command clears the command window and places the cursor in the home position. (This is usually a good way to start a script that is to generate command window output.) The `disp` command has the form

$$\texttt{disp}(\langle \textit{string} \rangle)$$

Strings in MATLAB are enclosed by single quotes. The commands

```
disp(' ')
disp('  k     x(k)   sin(x(k))')
disp('------------------------')
```

are used to print a blank line, a heading, and a dashed line.

The `sprintf` command is used to produce a string that includes the values of named variables. It has the form

$$\texttt{sprintf}(\langle \textit{String with Format Specifications} \rangle,\langle \textit{List-of-Variables} \rangle)$$

A variable must be listed for each format. Sample format insertions include `%5.0f`, `%8.3f`, and `%10.6e`. The first integer in a format specification is the total width of the field. The second number specifies how many places are allocated to the fractional part. In the script, the command

```
disp(sprintf(' %2d     %3.0f     %6.3f    ',k,degrees,y(k)));
```

prints a line with three numbers. The three numbers are stored in `k`, `degrees`, and `y(k)`. The value of `k` is printed as an integer while `degrees` is printed with a decimal point but with no digits to the right of the decimal point. On the other hand, `y(k)` is printed with three decimal places. The `e` format is used to specify mantissa/exponent style. For example,

```
disp(sprintf('One Degree = %5.3e Radians',pi/180))
```

This produces the output of the form

```
One Degree = 1.745e-02 Radians
```

If `x` is a vector then

```
disp(sprintf('  %5.3e  ',x))
```

displays all the components of `x` on a single line, each with `5.3e` format.

1.1.5 A Note About `fprintf`

It is sometimes handy to use `fprintf` instead of the combinations of `disp` and `sprintf`. Consider the fragement

```
disp(' ')
disp('  k     x(k)   sin(x(k))')
disp('------------------------')
for k=1:21
   degrees = (k-1)*360/(n-1);
   disp(sprintf(' %2.0f     %3.0f     %6.3f    ',k,degrees,y(k)));
end
disp( ' ');
disp('x(k) is given in degrees.')
disp(sprintf('One Degree = %5.3e Radians',pi/180))
```

taken from the script `SinePlot` above. This is equivalent to

```
fprintf('\n  k     x(k)   sin(x(k))\n------------------------\n')
for k=1:21
   degrees = (k-1)*360/(n-1);
   fprintf(' %2.0f     %3.0f     %6.3f    \n',k,degrees,y(k));
end
fprintf( ' \nx(k) is given in degrees.\nOne Degree = %5.3e Radians',pi/180)
```

The carriage return command “\n” effecively says “start a new line of output”.

1.1.6 A Simple Plot

We are now in a position to solve the plotting problem posed at the beginning of this section. The script

```
n = 21; x = linspace(0,1,n); y = sin(2*pi*x);
plot(x,y)
title('The Function  y = sin(2*pi*x)')
xlabel('x (in radians)')
ylabel('y')
```

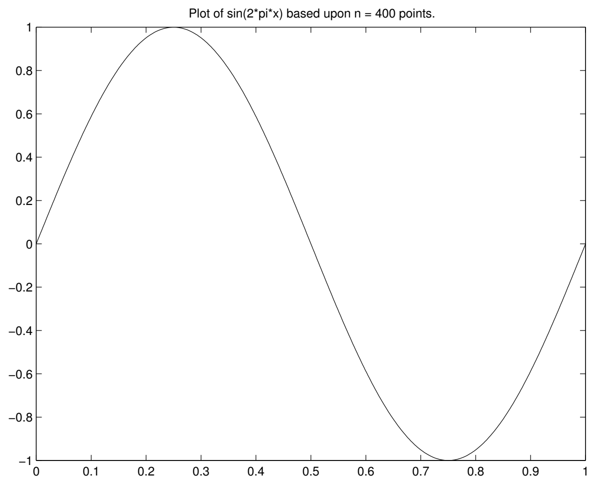


FIGURE 1.4 *A smooth plot of* $\sin(2\pi x)$

reproduces Figure 1.1. It draws a polygonal line in a *figure* that connects the vertices (x_k, y_k), $k = 1{:}21$ in order. In its most simple form, `plot` takes two vectors of equal size and plots the second versus the first. The scaling of the axes is done automatically. The `title`, `xlabel`, and `ylabel` functions enable us to "comment" the plot. Each requires a string argument.

To produce a better plot with no "corners," we increase n so that the line segments that make up the graph are sufficiently short, thereby rendering the impression of a genuine curve. For example,

```
n = 200;
x = linspace(0,1,n);
y = sin(2*pi*x);
plot(x,y)
title('The function y = sin(2*pi*x)')
xlabel('x (in radians)')
ylabel('y')
```

produces the plot displayed in Figure 1.4. In general, the smoothness of a displayed curve depends on the spacing of the underlying sample points, screen granularity, and the vision of the observer. Here is a script file that produces a sequence of increasingly refined plots:

```
% Script File: SinePlot
% Displays increasingly smooth plots of sin(2*pi*x).
close all
for n = [4 8 12 16 20 50 100 200 400]
   x = linspace(0,1,n);
   y = sin(2*pi*x);
   plot(x,y)
   title(sprintf('Plot of sin(2*pi*x) based upon n = %3.0f points.',n))
   pause(1)
end
```

There are four new features to discuss. The `close all` command closes all windows. It is a good idea to begin script files that draw figures with this command so as to start with a "clean slate." Second, notice the use of a general vector in the specification of the `for`-loop. The count variable `n` takes on the values in the specified vector one at a time. Third, observe the use of `sprintf` in the reference to `title`. This enables us to report the number of points associated with each plot. Finally, the fragment makes use of the `pause` function. In general, a reference of the form `pause(s)` holds up execution for approximately `s` seconds. Because a sequence of plots is produced in the preceding example, the `pause(1)` command permits a 1-second viewing of each plot.

Problems

P1.1.1 The built-in functions like `sin` accept vector arguments and return vectors of values. If `x` is an n vector, then

$$\texttt{y = }\left\{\begin{array}{l}\texttt{abs(x)}\\ \texttt{sqrt(x)}\\ \texttt{exp(x)}\\ \texttt{log(x)}\\ \texttt{sin(x)}\\ \texttt{cos(x)}\\ \texttt{asin(x)}\\ \texttt{acos(x)}\\ \texttt{atan(x)}\end{array}\right\} \Rightarrow y_i = \left\{\begin{array}{l}|x_i|\\ \sqrt{x_i},\ x_i \geq 0\\ e^{x_i}\\ \log(x_i),\ x_i > 0\\ \sin(x_i)\\ \cos(x_i)\\ \arcsin(x_i),\ -1 \leq x_i \leq 1\\ \arccos(x_i),\ -1 \leq x_i \leq 1\\ \arctan(x_i)\end{array}\right\},\ i = 1{:}n.$$

The vector `x` can be either a row vector or a column vector and `y` has the same shape. Write a script file that plots these functions in succession with two-second pauses in between the plots.

P1.1.2 Define the function

$$f(x) = \left\{\begin{array}{ll}\sqrt{1-(x-1)^2} & 0 \leq x \leq 2\\ \sqrt{1-(x-3)^2} & 2 < x \leq 4\\ \sqrt{1-(x-5)^2} & 4 < x \leq 6\\ \sqrt{1-(x-7)^2} & 6 < x \leq 8\end{array}\right. .$$

Set up a length-201 vector `y` with the property that $y_i = f(8*(i-1)/200)$ for $i = 1{:}201$.

1.2 More Vectors, More Plotting, and Now Matrices

We continue to refine our vector intuition by considering several additional plotting situations. New control structures are introduced and some of MATLAB's matrix algebra capabilities are presented.

1.2.1 Vectorizing Function Evaluations

Consider the problem of plotting the rational function

$$f(x) = \left(\frac{1+\dfrac{x}{24}}{1-\dfrac{x}{12}+\dfrac{x^2}{384}}\right)^8$$

across the interval $[0,1]$. (This happens to be an approximation to the function e^x.) Here is a scalar approach:

```
n = 200;
x = linspace(0,1,n);
y = zeros(1,n);
for k=1:n
   y(k) = ((1 + x(k)/24)/(1 - x(k)/12 + (x(k)/384)*x(k)))^8;
end
plot(x,y)
```

However, by using vector operations that are available in MATLAB, it is possible to replace the loop with a single, vector-level command:

```
% Script File: ExpPlot
% Examines the function f(x) = ((1 + x/24)/(1 - x/12 + x^2/384))^8
% as an approximation to exp(z) across [0,1].
close all
x  = linspace(0,1,200);
num   = 1 + x/24;
denom = 1 - x/12 + (x/384).*x;
quot = num./denom;
y = quot.^8;
plot(x,y,x,exp(x))
```

The assignment to `y` involves the familiar operations of *vector scale*, *vector add*, and *vector subtract*, and the not-so-familiar operations of *pointwise vector multiply*, *pointwise vector divide*, and *pointwise vector exponentiation*. To clarify each of these operations, we break the script down into more elemental steps:

```
z = (1/24)*x;
num = 1 + z;
w = x/384;
q = w.*x;
denom = 1 - z/2 + q;
quot = num./denom;
y = quot.^8;
```

Matlab supports scalar-vector multiplication. The command

```
z = (1/24)*x;
```

multiplies every component in `x` by (1/24) and stores the result in `z`. The vector `z` has exactly the same length and orientation as `x`. The command

```
num = 1 + z;
```

adds 1 to every component of `z` and stores the result in `num`. Thus `num = 1 + [20 30 40]` and `num = [21 31 41]` are equivalent. Strictly speaking, scalar-plus-vector is not a legal vector space operation, but it is a very handy Matlab feature.

Now let us produce the vector of denominator values. The command

```
w = x/384
```

is equivalent to

```
w = (1/384)*x
```

It is also the same as `w = x*(1/384)`. The command

```
q = w.*x
```

makes use of pointwise vector multiplication and produces a vector `q` with the property that each component is equal to the product of the corresponding components in `w` and `x`. Thus

```
q = [2 3 4].*[20 30 50]
```

is equivalent to

```
q = [40 90 200]
```

The same rules apply when the two operands are column vectors. The key is that the vectors to be multiplied have to be identical in length and orientation. The command

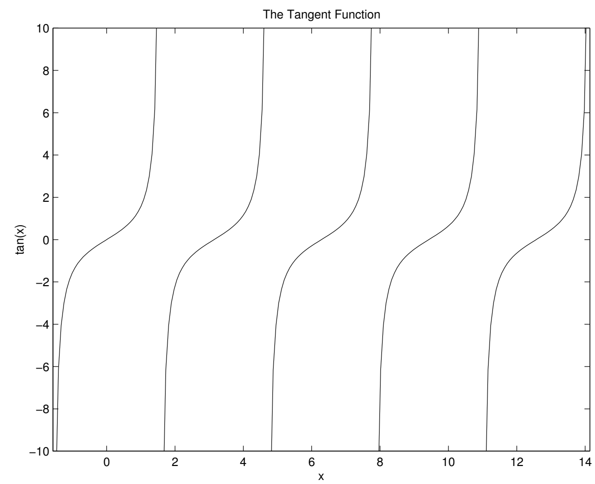


FIGURE 1.5 *A plot of* $\tan(x)$

```
denom = 1 - z/2 + q
```

sets `denom(i)` to `1 - z(i)/2 + q(i)` for all i. Vector addition, like vector subtraction, requires both operands to have the same length and orientation.

The pointwise division `quotient = num./denom` performs as expected. The ith component of `quotient` is set to `num(i)/denom(i)`. Lastly, the command

```
y = quotient.^8
```

raises each component in `quotient` to the 8th power and assembles the results in the vector `y`.

1.2.2 Scaling and Superpositioning

Consider the plotting of the function $\tan(x) = \sin(x)/\cos(x)$ across the interval $[-\pi/2, 11\pi/2]$. This is interesting because the function has poles at points where the cosine is zero. The script

```
x = linspace(-pi/2,11*pi/2,200);
y = tan(x);
plot(x,y)
```

produces a plot with minimum information because the autoscaling feature of the `plot` function must deal with an essentially infinite range of y-values. This can be corrected by using the `axis` function:

```
x = linspace(-pi/2,11*pi/2,200);
y = tan(x);
plot(x,y)
axis([-pi/2 9*pi/2 -10 10])
```

The `axis` function is used to scale manually the axes in the current plot, and it requires a 4-vector whose values define the x and y ranges. In particular,

```
axis([xmin xmax ymin ymax])
```

imposes the x-axis range `xmin` $\le x \le$ `xmax` and a y-axis range `ymin` $\le y \le$ `ymax`. In our example, the $[-10, 10]$ range in the y-direction is somewhat arbitrary. Other values would work. The idea is to choose the range so that the function's poles are dramatized without sacrificing the quality of the plot in domains where it is nicely behaved. (See Figure 1.5.) We mention that the command `axis` by itself returns the system to the original autoscaling mode.

Another way to produce the same graph is to plot the first branch and then to reuse the function evaluations for the remaining branches:

```
% Script File: TangentPlot
% Plots the function tan(x), -pi/2 <= x <= 9pi/2
close all
x = linspace(-pi/2,pi/2,40); y = tan(x); plot(x,y)
ymax = 10;
axis([-pi/2 9*pi/2 -ymax ymax])
title('The Tangent Function'), xlabel('x'), ylabel('tan(x)')
hold on
for k=1:4
   xnew = x+ k*pi;
   plot(xnew,y);
end
hold off
```

This script has a number of new features that require explanation. The `hold on` command effectively tells MATLAB to superimpose all subsequent plots on the current figure window. Each time through the `for`-loop, a different branch is plotted. The axis scaling is frozen during these computations. The `xnew` calculation produces the required x-domain for each branch plot. During the kth pass through the loop, the expression `xnew + k*pi` establishes a vector of equally spaced values across the interval

$$[-\pi/2 + k\pi, -\pi/2 + (k+1)\pi].$$

The same vector of tan-evaluations is used in each branch plot. Observe that with superpositioning we produce a plot with only one-fifth the number of `tan` evaluations that our initial solution required.

The `hold off` command shuts down the superpositioning feature and sets the stage for "normal" plotting thereafter.

Another way that different graphs can be superimposed in the same plot is by calling `plot` with an extended parameter list. Suppose we want to plot the functions $\sin(2\pi x)$ and $\cos(2\pi x)$ across the interval $[0, 1]$ and to mark the point where they intersect. The script

```
x = linspace(0,1,200); y1 = sin(2*pi*x); y2 = cos(2*pi*x);
plot(x,y1)
hold on
plot(x,y2,'-')
plot([1/8 5/8],[1/sqrt(2) -1/sqrt(2)],'*')
hold off
```

accomplishes this task. (See Figure 1.6.) The first three-argument call to `plot` uses a dashed line to produce the graph of $\cos(2\pi x)$. Other line designations are possible (e.g., '–','-.'). The second three-argument call to plot places an asterisk at the intersection points $(1/8, 1/\sqrt{2})$ and $(5/8, -1/\sqrt{2})$. Other point designations are possible (e.g., '+', '.', 'o'.) The key idea is that when `plot` is used to draw a graph, an optional third parameter can be included that specifies the line style. This parameter is a string that specifies the "nature of the pen" that is doing the drawing. Colors may also be specified. (See §1.7.6.) The superpositioning can also be achieved as follows:

```
% Script File: SineAndCosPlot
% Plots the functions sin(2*pi*x) and cos(2*pi*x) across [0,1]
% and marks their intersection.
close all
x = linspace(0,1,200); y1 = sin(2*pi*x); y2 = cos(2*pi*x);
plot(x,y1,x,y2,'--',[1/8 5/8],[1/sqrt(2) -1/sqrt(2)],'*')
```

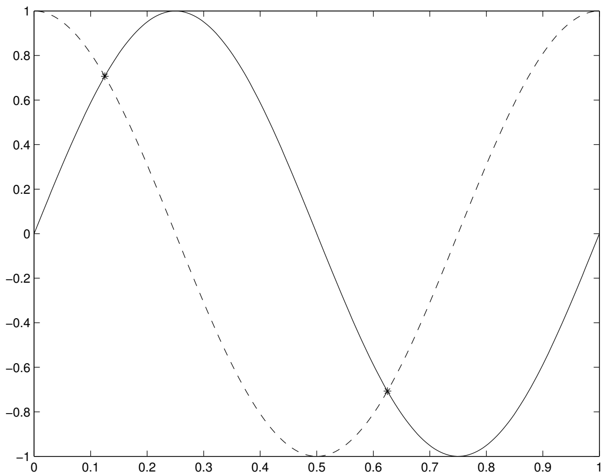


FIGURE 1.6 *Superpositioning*

This illustrates `plot`'s "multigraph" capability. The syntax is as follows:

`plot(`⟨*First Graph Specification*⟩`,...,`⟨*Last Graph Specification*⟩`)`

where each graph specification has the form

⟨*Vector*⟩`,`⟨*Vector*⟩`,`⟨*String (optional)*⟩

If some of the string arguments are missing, then MATLAB chooses them in a way that fosters clarity in the overall plot.

1.2.3 Polygons

Suppose that we have a polygon with n vertices. If `x` and `y` are column vectors that contain the coordinate values, then

```
plot(x,y)
```

does *not* display the polygon because (x_n, y_n) is not connected to (x_1, y_1). To rectify this we merely "tack on" an extra copy of the first point:

```
x = [x;x(1)];
y = [y;y(1)];
plot(x,y)
```

Thus, the three points $(1,2)$, $(4,-2)$, and $(3,7)$ could be represented with the three-vectors `x = [1 4 3]` and `y = [2 -2 7]`. The `x` and `y` updates yield `x = [1 4 3 1]` and `y = [2 -2 7 2]`. Plotting the revised `y` against the revised `x` displays the triangle with the designated vertices.

The preceding "concatenation" of a component to a vector is a special case of a general operation whereby vectors can be glued together. If `r1`, `r2`,...,`rm` are row vectors, then

```
v = [ r1 r2 ...  rm]
```

is also a row vector obtained by placing the component vectors `r1`,...,`rm` side by side. For example,

```
v = [linspace(1,10,10) linspace(20,100,9)];
```

is equivalent to

```
v = [ 1  2  3  4  5  6  7  8  9  10  20  30  40  50  60  70  80  90  100];
```

Similarly, if c1, c2,..., cm are column vectors, then

```
v = [ c1 ; c2 ; ...  ; cm]
```

is also a column vector, obtained by stacking c1,...,cm.

Continuing with our polygon discussion, assume that we have executed the commands

```
t = linspace(0,2*pi,361);
c = cos(t);
s = sin(t);
plot(c,s)
axis off equal
```

The object displayed is a regular 360-gon with "radius" 1. The command axis equal ensures that the x-distance per pixel is the same as the y-distance per pixel. This is important in this application because a regular polygon would not look regular if the two scales were different.

With the preceding sine/cosine vectors computed, it is possible to display various other regular n-gons simply by connecting appropriate subsets of points. For example,

```
x = [c(1) c(121) c(241) c(361)];
y = [s(1) s(121) s(241) s(361)];
plot(x,y)
```

plots the equilateral triangle whose vertices are at the 0^o, 120^o, and 240^o points along the unit circle. This kind of non-unit stride subvector extraction can be elegantly handled in MATLAB using the colon notation. The preceding triplet of commands is equivalent to

```
x = c(1:120:361);
y = s(1:120:361);
plot(x,y)
```

More generally, if sides is a positive integer that is a divisor of 360, then

```
x = c(1:(360/sides):361);
y = s(1:(360/sides):361);
plot(x,y)
```

plots a regular polygon with that number of sides. Here is a script that displays nine regular polygons in nine separate subwindows:

```
% Script File: Polygons
% Plots selected regular polygons.
close all
theta = linspace(0,2*pi,361);
c = cos(theta);
s = sin(theta);
k=0;
for sides = [3 4 5 6 8 10 12 18 24]
   stride = 360/sides;
   k=k+1;
   subplot(3,3,k)
   plot(c(1:stride:361),s(1:stride:361))
   axis equal
end
```

Figure 1.7 shows what is produced when this script is executed.

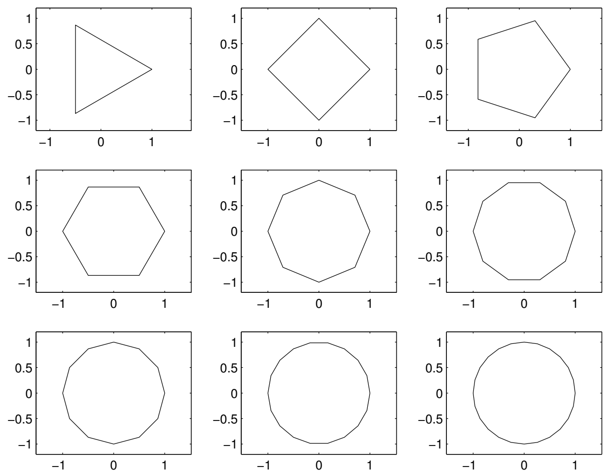


FIGURE 1.7 *Regular polygons*

The key new feature in `Polygons` is `subplot`. The command `subplot(3,3,k)` says "break up the current figure window into a 3-by-3 array of subwindows, and place the next plot in the kth one of these." The subwindows are indexed as follows:

$$\begin{array}{ccc} \boxed{1} & \boxed{2} & \boxed{3} \\ \boxed{4} & \boxed{5} & \boxed{6} \\ \boxed{7} & \boxed{8} & \boxed{9} \end{array}$$

In general, `subplot(m,n,k)` splits the current figure into an m-row by n-column array of subwindows that are indexed left to right, top to bottom.

1.2.4 Some Matrix Computations

Let's consider the problem of plotting the function

$$f(x) = 2\sin(x) + 3\sin(2x) + 7\sin(3x) + 5\sin(4x)$$

across the interval $[-10, 10]$. The scalar-level script

```
n = 200;
x = linspace(-10,10,n)';
y = zeros(n,1);
for k=1:n
   y(k) = 2*sin(x(k)) + 3*sin(2*x(k)) + 7*sin(3*x(k)) + 5*sin(4*x(k));
end
plot(x,y)
title('f(x) = 2sin(x) + 3sin(2x) + 7sin(3x)  +5sin(4x)')
```

does the trick. (See Figure 1.8.) Notice that `x` and `y` are column vectors. The `sin` evaluations can be vectorized giving this superior alternative:

```
n = 200;
x = linspace(-10,10,n)';
y = 2*sin(x) + 3*sin(2*x) + 7*sin(3*x) + 5*sin(4*x);
plot(x,y)
title('f(x) = 2sin(x) + 3sin(2x) + 7sin(3x)  +5sin(4x)')
```

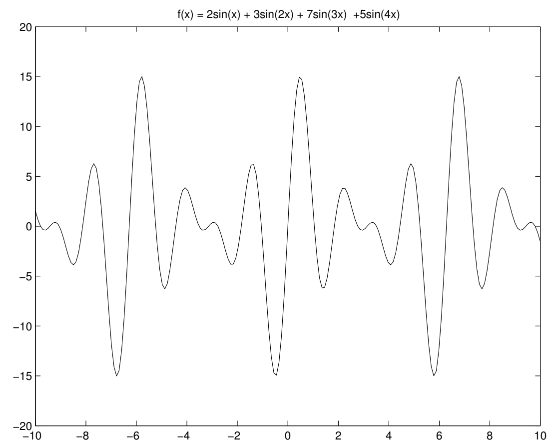


FIGURE 1.8 *A sum of sines*

But any linear combination of vectors is "secretly" a matrix-vector product. That is,

$$2\begin{bmatrix}3\\1\\4\\7\\2\\8\end{bmatrix}+3\begin{bmatrix}5\\0\\3\\8\\4\\2\end{bmatrix}+7\begin{bmatrix}8\\3\\3\\1\\1\\1\end{bmatrix}+5\begin{bmatrix}1\\6\\8\\7\\0\\9\end{bmatrix}=\begin{bmatrix}3&5&8&1\\1&0&3&6\\4&3&3&8\\7&8&1&7\\2&4&1&0\\8&2&1&9\end{bmatrix}\begin{bmatrix}2\\3\\7\\5\end{bmatrix}.$$

MATLAB supports matrix-vector multiplication, and the script

```
A = [3 5 8 1; 1 0 3 6; 4 3 3 8; 7 8 1 7; 2 4 1 0; 8 2 1 9];
y = A*[2;3;7;5];
```

shows how to initialize a small matrix and engage it in a matrix-vector product. Note that the matrix is assembled row by row with semicolons separating the rows. Spaces separate the entries within a row. An ellipsis (...) can be used to spread a long command over more than one line, which is sometimes useful for clarity:

```
A = [3 5 8 1;...
     1 0 3 6;...
     4 3 3 8;...
     7 8 1 7;...
     2 4 1 0;...
     8 2 1 9];
y = A*[2;3;7;5];
```

In the sum-of-sines plotting problem, the vector y can also be constructed as follows:

```
n = 200; m = 4;
x = linspace(-10,10,n)';  A = zeros(n,m);
for j=1:m
   for k=1:n
      A(k,j) = sin(j*x(k));
   end
end
y = A*[2;3;7;5];
plot(x,y)
title('f(x) = 2sin(x) + 3sin(2x) + 7sin(3x)  + 5sin(4x)')
```

This illustrates how a matrix can be initialized at the scalar level. But a matrix is just an aggregation of its columns, and MATLAB permits a column-by-column synthesis, bringing us to the final version of our script:

```
% Script File: SumOfSines
% Plots f(x) = 2sin(x) + 3sin(2x) + 7sin(3x) + 5sin(4x)
% across the interval [-10,10].
close all
x = linspace(-10,10,200)';
A = [sin(x) sin(2*x) sin(3*x) sin(4*x)];
y = A*[2;3;7;5];
plot(x,y)
title('f(x) = 2sin(x) + 3sin(2x) + 7sin(3x)  + 5sin(4x)')
```

An expression of the form

$$\texttt{[}\ \langle \textit{Column 1}\rangle\ \langle \textit{Column 2}\rangle\ \ldots\ \langle \textit{Column m}\rangle\ \texttt{]}$$

is a matrix with m columns. Of course, the participating column vectors must have the same length.

Another way to initialize `A` is to use a single loop whereby each pass sets up a single column:

```
n = 200;
m = 4;
A = zeros(n,m);
for j=1:m
   A(:,j) = sin(j*x);
end
```

The notation `A(:,j)` names the jth column of `A`. Notice that the size of `A` is established with a call to `zeros`. The `size` function can be used to determine the dimensions of any active variable. (Recall that all variables are treated as matrices.) Thus, the script

```
A = [1 2 3;4 5 6];
[r,c] = size(A);
```

assigns 2 (the row dimension) to `r` and 3 (the column dimension) to `c`. Many MATLAB functions return more than one value and `size` is our first exposure to this. Note that the output values are enclosed with square brackets.

Matrices can also be built up by row. In `SumOfSines`, the kth row of `A` is given by `sin(x(k)*(1:4))` so we also initialize `A` as follows:

```
n = 200;
m = 4;
A = zeros(n,m);
for k=1:n
   A(k,:) = sin(x(k)*(1:m));
end
```

The notation `A(k,:)` identifies the kth row of `A`.

As a final example, suppose that we want to plot *both* of the functions

$$\begin{aligned} f(x) &= 2\sin(x) + 3\sin(2x) + 7\sin(3x) + 5\sin(4x) \\ g(x) &= 8\sin(x) + 2\sin(2x) + 6\sin(3x) + 9\sin(4x) \end{aligned}$$

in the same window. Obviously, a double application of the preceding ideas solves the problem:

```
n = 200;
x = linspace(-10,10,n)';
A = [sin(x) sin(2*x) sin(3*x) sin(4*x)];
y1 = A*[2;3;7;5];
y2 = A*[8;2;6;9];
plot(x,y1,x,y2)
```

But a set of matrix-vector products that involve the same matrix is "secretly" a single matrix-matrix product:

$$\left.\begin{array}{c}\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}\begin{bmatrix} 5 \\ 7 \end{bmatrix} = \begin{bmatrix} 19 \\ 43 \end{bmatrix} \\ \\ \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}\begin{bmatrix} 6 \\ 8 \end{bmatrix} = \begin{bmatrix} 22 \\ 50 \end{bmatrix}\end{array}\right\} \equiv \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}\begin{bmatrix} 5 & 6 \\ 7 & 8 \end{bmatrix} = \begin{bmatrix} 19 & 22 \\ 43 & 50 \end{bmatrix}.$$

Since Matlab supports matrix-matrix multiplication, our script transforms to

```
n = 200;
x = linspace(-10,10,n)';
A = [sin(x) sin(2*x) sin(3*x) sin(4*x)];
y = A*[2 8;3 2;7 6;5 9];
plot(x,y(:,1),x,y(:,2))
```

But the `plot` function can accept matrix arguments. The command

```
plot(x,y(:,1),x,y(:,2))
```

is equivalent to

```
plot(x,y)
```

and so we obtain

```
% Script File: SumOfSines2
% Plots the functions
%         f(x) = 2sin(x) + 3sin(2x) + 7sin(3x) + 5sin(4x)
%         g(x) = 8sin(x) + 2sin(2x) + 6sin(3x) + 9sin(4x)
% across the interval [-10,10].
close all
n = 200;
x = linspace(-10,10,n)';
A = [sin(x) sin(2*x) sin(3*x) sin(4*x)];
y = A*[2 8;3 2;7 6;5 9];
plot(x,y)
```

In general, plotting a matrix against a vector is the same thing as plotting each of the matrix columns against the vector. Of course, the row dimension of the matrix must equal the length of the vector.

It is also possible to plot one matrix against another. If `X` and `Y` have the same size, then the corresponding columns will be plotted against each other with the command `plot(X,Y)`.

Finally, we mention the "backslash" operator that can be invoked whenever the solution to a linear system of algebraic equations is required. For example, suppose we want to find scalars $\alpha_1, \ldots, \alpha_4$ so that if

$$f(x) = \alpha_1 \sin(x) + \alpha_2 \sin(2x) + \alpha_3 \sin(3x) + \alpha_4 \sin(4x),$$

then $f(1) = -2$, $f(2) = 0$, $f(3) = 1$, and $f(4) = 5$. These four stipulations imply

$$
\begin{array}{rcrcrcrcr}
\alpha_1 \sin(1) & + & \alpha_2 \sin(2) & + & \alpha_3 \sin(3) & + & \alpha_4 \sin(4) & = & -2 \\
\alpha_1 \sin(2) & + & \alpha_2 \sin(4) & + & \alpha_3 \sin(6) & + & \alpha_4 \sin(8) & = & 0 \\
\alpha_1 \sin(3) & + & \alpha_2 \sin(6) & + & \alpha_3 \sin(9) & + & \alpha_4 \sin(12) & = & 1 \\
\alpha_1 \sin(4) & + & \alpha_2 \sin(8) & + & \alpha_3 \sin(12) & + & \alpha_4 \sin(16) & = & 5
\end{array}
$$

That is,

$$
\begin{bmatrix} \sin(1) & \sin(2) & \sin(3) & \sin(4) \\ \sin(2) & \sin(4) & \sin(6) & \sin(8) \\ \sin(3) & \sin(6) & \sin(9) & \sin(12) \\ \sin(4) & \sin(8) & \sin(12) & \sin(16) \end{bmatrix} \begin{bmatrix} \alpha_1 \\ \alpha_2 \\ \alpha_3 \\ \alpha_4 \end{bmatrix} = \begin{bmatrix} -2 \\ 0 \\ 1 \\ 5 \end{bmatrix}.
$$

Here is how to set up and solve this 4-by-4 linear system:

```
X = [1 2 3 4 ; 2 4 6 8 ; 3 6 9 12 ; 4 8 12 16];
Z = sin(X);
f = [-2; 0; 1; 5]
alpha = Z\f
```

Observe that `sin` applied to a matrix returns the matrix of corresponding sine evaluations. This is typical of many of MATLAB's built-in functions. For linear system solving, the backslash operator requires the matrix of coefficients on the left and the right hand side vector (as a column) on the right. The solution to the preceding example is

$$
\alpha = \begin{bmatrix} -0.2914 \\ -8.8454 \\ -18.8706 \\ -11.8279 \end{bmatrix}.
$$

Problems

P1.2.1 Suppose `z = [10 40 20 80 30 70 60 90]`. Indicate the vectors that are specified by `z(1:2:7)`, `z(7:-2:1)`, and `z([3 1 4 8 1])`.

P1.2.2 Suppose `z = [10 40 20 80 30 70 60 90]`. What does this vector look like after each of these commands?

```
z(1:2:7) = zeros(1,4)
z(7:-2:1) = zeros(1,4)
z([3 4 8 1]) = zeros(1,4)
```

P1.2.3 Given that the commands

```
x = linspace(0,1,200);
y = sqrt(1-x.^2);
```

have been carried out, show how to produce a plot of the circle $x^2 + y^2 = 1$ without any additional square roots or trigonometric evaluations.

P1.2.4 Produce a single plot that displays the graphs of the functions $\sin(kx)$ across $[0, 2\pi]$, $k = 1{:}5$.

P1.2.5 Assume that `m` is an initialized positive integer. Write a MATLAB script that plots in a single window the functions x, x^2, x^3, $\ldots$, x^m across the interval $[0, 1]$.

P1.2.6 Assume that `x` is an initialized MATLAB array and that `m` is a positive integer. Using the `ones` function, the pointwise array multiply operator `.*`, and MATLAB's ability to scale and add arrays, write a fragment that computes an array `y` with the property that the ith component of y has the following value:

$$
y_i = \sum_{k=0}^{m} \frac{x_i^k}{k!}.
$$

P1.2.7 Write a MATLAB fragment to plot the following ellipses in the same window:

$$\begin{array}{lll} \text{Ellipse 1:} & x_1(t) = 3 + 6\cos(t) & y_1(t) = -2 + 9\sin(t) \\ \text{Ellipse 2:} & x_2(t) = 7 + 2\cos(t) & y_2(t) = 8 + 6\sin(t) \end{array}$$

P1.2.8 Consider the following MATLAB script:

```
x = linspace(0,2*pi);
y = sin(x);
plot(x/2,y)
hold on
for k=1:3
   plot((k*pi)+x/2,y)
end
hold off
```

What function is plotted and what is the range of x?

P1.2.9 Assume that `x`, `y`, and `z` are MATLAB arrays initialized as follows:

```
x = linspace(0,2*pi,100);
y = sin(x);
z = exp(-x);
```

Write a MATLAB fragment that plots the function $e^{-x}\sin(x)$ across the interval $[0, 4\pi]$. The fragment should not involve any additional calls to `sin` or `exp`. Hint: exploit the fact that sin has period 2π and that the exponential function satisfies $e^{a+b} = e^a e^b$.

P1.2.10 Modify the script `SumOfSines` so that $f(x) = 2\sin(x) + 3\sin(2x) + 7\sin(3x) + 5\sin(4x)$ is plotted in one window and its derivative in another. Use `subplot` placing one window above the other. Your implementation should not involve any loops and should have appropriate titles on the plots.

1.3 Building Exploratory Environments

A consequence of MATLAB's friendliness and versatility is that it encourages the exploration of mathematical and algorithmic ideas. Many computational scientists like to precede the rigorous analysis of a problem with MATLAB-based experimentation. We use three examples to show this, learning many new features of the system as we go along.

1.3.1 The Up/Down Sequence

Suppose x_1 is a given positive integer and that for $k \geq 1$ we define the sequence $x_1, x_2, \ldots$ as follows:

$$x_{k+1} = \begin{cases} x_k/2 & \text{if } x_k \text{ is even} \\ \\ 3x_k + 1 & \text{if } x_k \text{ is odd} \end{cases}.$$

Thus, if $x_1 = 7$, then the following sequence unfolds:

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

We will call this the *up/down* sequence for obvious reasons. Note that it cycles once the value of 1 is reached. A number of interesting questions are suggested:

- Does the sequence always reach the cycling stage?
- Let n be the smallest index for which $x_n = 1$. How does n behave as a function of the initial value x_1?
- Are there any systematic patterns in the sequence worth noting?

Our goal is to develop a script file that can be used to shed light on these and related issues.

We start with a script that solicits a starting value and then generates the sequence, assembling the values in a vector `x`:

```
x(1) = input('Enter initial positive integer:');
k = 1;
while (x(k) ~= 1)
   if rem(x(k),2) == 0
      x(k+1) = x(k)/2;
   else
      x(k+1) = 3*x(k)+1;
   end
   k = k+1;
end
```

The `input` command is used to set up `x(1)`. It has the form

`input(`⟨*string message*⟩`)`

and prompts for keyboard input. For example,

```
Enter initial positive integer:
```

Whatever number you type, it is assigned to `x(1)`.

After `x(1)` is initialized, the generation of the sequence takes place under the auspices of a `while`-loop. Each pass through the loop requires a test of the current `x(k)` in accordance with the rule for `x(k+1)` given earlier. This is handled by an `if-then-else`.

Let's look at the details. In MATLAB, a test of the form `x(k)==10` renders a one if it is true and a zero if it is false.[1] All the usual comparisons are supported:

Notation	Meaning
`<`	less than
`<=`	less than or equal
`==`	equal
`>=`	greater than or equal
`>`	greater than
`~=`	not equal

A `while`-loop has the form

`while` ⟨*Condition*⟩
 ⟨*Statements*⟩
`end`

An `if-then-else` is structured as follows:

`if` ⟨*Condition*⟩
 ⟨*Statements*⟩
`else`
 ⟨*Statements*⟩
`end`

Both of these control structures operate in the usual way. The condition is numerically valued, and is interpreted as *true* if it is nonzero.

The remainder function `rem` is used to check whether or not `x(k)` is even. Assuming that `a` and `b` are positive integers, a call of the form `rem(a,b)` returns the remainder when `b` is divided into `a`.

Now one of the things we do not know is whether or not the up/down sequence reaches 1. To guard against the production of an unacceptably large x-vector, we can put a limit on how many terms to generate. Setting that limit to 500 and presizing `x` to that length, we obtain

[1] Remember, there is no boolean type in MATLAB.

```
x = zeros(500,1);
x(1) = input('Enter initial positive integer:');
k = 1;
while ((x(k) ~= 1) & (k < 500))
   if rem(x(k),2) == 0
      x(k+1) = x(k)/2;
   else
      x(k+1) = 3*x(k)+1;
   end
   k = k+1;
end
n = k;
x = x(1:n);
```

The index of the first sequence member that equals 1 is assigned to `n` and `x` is "trimmed" to that length with the assignment `x = x(1:n)`. Notice the use of the *and* operator `&` in the `while`-loop condition. The and, or, and not operations are all possible in MATLAB :

Notation	Meaning
`&`	and
`|`	or
`~`	not
`xor`	exclusive or

The usual definitions apply with the understanding that 1 and 0 are used for true and false respectively. Thus `(x(k) == 1) & (k < 500))` has the value of 1 if `x(k)` equals 1 and `k` is strictly less than 500. If either of these conditions is false, then the logical expression equals 0.

Computing `x(1:n)` brings us to the stage where we must decide how to display it and its properties. Of course, we could display the vector simply by leaving off the semicolon in `x = x(1:n);`. Alternatively, we can make use of `fprintf`'s vectorizing capability:

```
fprintf('%10d\n',x)
```

When a vector like `x` is passed to `fprintf` in this way. it just keeps cycling through the format string until every vector component is processed.

Among the numerical properties of `x` that are interesting are the maximum value and the number of integers $\leq x_1$ that are "hit" by the up/down process:

```
[xmax,imax] = max(x);
disp(sprintf('\n x(%1.0f) = %1.0f is the max.',imax,xmax))
density = sum(x<=x(1))/x(1);
disp(sprintf(' The density is %5.3f.',density))
```

When the `max` function is applied to a vector, it returns the maximum value and the index where it occurs. It is also possible to use `max` in an expression. For example,

```
GrowthFactor = max(x)/x(1)
```

assigns to `GrowthFactor` the ratio of the largest value in `x` to `x(1)`. Notice the use of the `1.0f` format. For integers greater than one digit in length, extra space is accorded as necessary. This ensures that there is no gap between the displayed subscript and the right parenthesis, a small aesthetic point.

The assignment to `density` requires two explanations. First, it is legal to compare vectors in MATLAB. The comparison `x<=x(1)` returns a vector of 0's and 1's that is the same size as `x`. If `x(k) <= x(1)` is true, then the kth component of this vector is one. The `sum` function applied to a vector sums its entries. Thus `sum(x<=x(1))` is precisely the number of components in `x` that are less than or equal to `x(1)`.

Graphical display is also in order and can help us appreciate the "flow of events" as the sequence winds its way to unity:

```
close all
figure
plot(x)
title(sprintf('x(1) = %1.0f, n = %1.0f',x(1),n));
figure
plot(sort(x,'descend'))
title('Sequence values sorted.')
I = find(rem(x(1:n-1),2));
if length(I)>1
   figure
   plot((1:n),zeros(1,n),I+1,x(I+1),I+1,x(I+1),'*')
   title('Local Maxima')
end
```

This script involves a number of new features. First, the command `plot(x)` plots the components of `x` against their indices. It is equivalent to `plot((1:n)',x)`.

Second, the `sort` function is used to produce a plot of the sequence with its values ordered from small to large. If `v` is a vector with length `m`, then `u = sort(v)` permutes the values in `v` and assigns them to `u` so that

$$u_1 \le u_2 \le u_3 \le \cdots \le u_m.$$

The command `sort(x,'descend')` produces a "big-to-little" sort.

Third, the expression `rem(x(1:n-1),2) == 1` returns a 0-1 vector that designates which components of `x(1:n-1)` are odd. The function `rem`, like many of MATLAB's built-in functions, accepts vector arguments and merely returns a vector of the function applied to each of the components. The `find` function returns a vector of subscripts that designate which entries in a vector are nonzero. Thus, if

```
x(1:n-1) = [ 17 52 26 13 40 20 10  5  16  8  4  2]'
```

and `r = rem(x(1:n-1),2)` and `I = find(r)`, then

```
r(1:n-1) = [  1  0  0  1  0  0  0  1   0  0  0  0]'
```

and `I = [ 1 4 8]'`. If the vector `I` is nonempty, then a plot of `I+1` is produced showing the pattern of the sequence's "local maxima." (The vector `I+1` contains the indices of values in `x(1:n-1)` that are produced by the "up operation" $3x_k + 1$.)

The last thing to discuss is `figure`. In all prior examples, our plots have appeared in a single window. New plots erase old ones. But with each reference to `figure`, a new window is opened. Figures are indexed from 1 and so `figure(1)` refers to a plot of `x`, `figure(2)` designates the plot of `x` sorted, and if `I` is nonempty, then `figure(3)` contains a plot of its local maxima. The `close all` statement clears all windows and ensures that the figure indexing starts at 1.

The script `UpDown` incorporates all of these features and by repeatedly running it we could bolster our intuition about the up/down sequence. To make this enterprise more convenient, we write a second script file that invokes `UpDown`:

```
% Script File: RunUpDown
% Environment for studying the up/down sequence.
% Stores selected results in file UpDownOutput.
while(input('Another Example? (1=yes, 0=no)'))
   diary UpDownOutput
   UpDown
   diary off
   if (input('Keep Output? (1=yes, 0=no)')~=1)
      delete UpDownOutput
   end
end
```

By using this script we can keep trying new starting values until one of special interest is found. The `while`-loop keeps running as long as you want to test another starting value. Before `UpDown` is run, the

```
diary UpDownOutput
```

command creates a file called `UpDownOutput`. Everything that is now written to the command window during the execution of `UpDown` is now also written to `UpDownOutput`. After `UpDown` is run, we turn off this feature with

```
diary off
```

The script then asks if the output should be kept. If not, then the file `UpDownOutput` is deleted. Note that it is possible to record several possible runs of `UpDown`, but as soon as the `if` condition is true, everything is erased. The advantage of writing output to a file is that it can then be edited to make it look nice. For example,

```
For starting value x(1) = 511, the UpDown sequence is

x(1:62) =

511     1534    767     2302    1151    3454    1727    5182    2591    7774
3887    11662   5831    17494   8747    26242   13121   39364   19682   9841
29524   14762   7381    22144   11072   5536    2768    1384    692     346
173     520     260     130     65      196     98      49      148     74
37      112     56      28      14      7       22      11      34      17
52      26      13      40      20      10      5       16      8       4
2       1
```

The figures from the final `UpDown` run are available for printing as well.

1.3.2 Random Processes

Many simulations performed by computational scientists involve random processes. In order to implement these on a computer, it is necessary to be able to generate sequences of random numbers. In MATLAB this is done with the built-in functions `rand` and `randn`. The command `x = rand(1000,1)` creates a length-1000 column vector of real numbers chosen randomly from the interval $(0, 1)$. The uniform$(0, 1)$ distribution is used, meaning that if $0 < a < b < 1$, then the fraction of values that fall in the range $[a, b]$ will be about $b - a$. The `randn` function should be used if a sequence of normally distributed random numbers is desired. The underlying probability distribution is the normal$(0, 1)$ distribution. A brief, graphically oriented description of these functions should clarify their statistical properties.

Histograms are a common way of presenting statistical data. Here is a script that illustrates `rand` and `randn` using this display technique:

```
% Script File: Histograms
% Histograms of rand(1000,1) and randn(1000,1).
close all
subplot(2,1,1)
x = rand(1000,1);
hist(x,30)
axis([-1 2 0 60])
title('Distribution of Values in rand(1000,1)')
xlabel(sprintf('Mean = %5.3f. Median = %5.3f.',mean(x),median(x)))
subplot(2,1,2)
x = randn(1000,1);
hist(x,linspace(-2.9,2.9,100))
title('Distribution of Values in randn(1000,1)')
xlabel(sprintf('Mean = %5.3f. Standard Deviation = %5.3f',mean(x),std(x)))
```

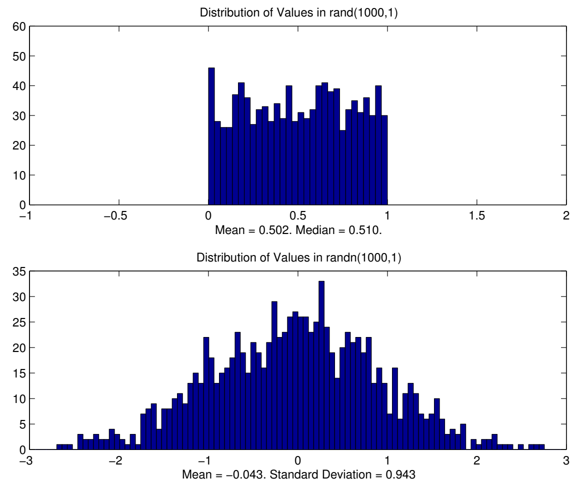


FIGURE 1.9 *The uniform and normal distributions*

(See Figure 1.9.) Notice that `rand` picks values uniformly from $[0, 1]$ while the distribution of values in `randn(1000,1)` follows the familiar "bell shaped curve." The mean, median, and standard deviation functions `mean`, `median`, and `std` are referenced. The histogram function `hist` can be used in several ways and the script shows two of the possibilities. A reference like `hist(x,30)` reports the distribution of the x-values according to where they "belong" with respect to 30 equally spaced bins spread across the interval $[\min(x), \max(x)]$. The bin locations can also be specified by passing `hist` a vector in the second parameter position (e.g., `hist(x,linspace(-2.9,2.9,100))`). This is done for the histogram of the normally distributed data.

Building on `rand` and `randn` through translation and scaling, it is possible to produce random sequences with specified means and variances. For example,

```
x = 10 + 5*rand(n,1);
```

generates a sequence of uniformly distributed numbers from the interval $(10, 15)$. Likewise,

```
x = 10 + 5*randn(n,1);
```

produces a sequence of normally distributed random numbers with mean 10 and standard deviation 5.

It is possible to generate random integers using `rand` (or `randn`) and the `floor` function. The command `z = floor(6*rand(n,1)+1)` computes a random vector of integers selected from $\{1, 2, 3, 4, 5, 6\}$ and assigns them to `z`. This is because `floor` rounds to $-\infty$. The command `z = ceil(6*x)` is equivalent because `ceil` rounds toward $+\infty$. In either case, the vector `z` looks like a recording of n dice throws. Notice that `floor` and `ceil` accept vector arguments and return vectors of the same size. (See also `fix` and `round`.) Here is a script that simulates 1000 rolls of a pair of dice, displaying the outcome in histogram form:

```
% Script File: Dice
% Simulates 1000 rollings of a pair of dice.
close all
First  = 1 + floor(6*rand(1000,1));
Second = 1 + floor(6*rand(1000,1));
Throws = First + Second;
hist(Throws, linspace(2,12,11));
title('Outcome of 1000 Dice Rolls.')
```

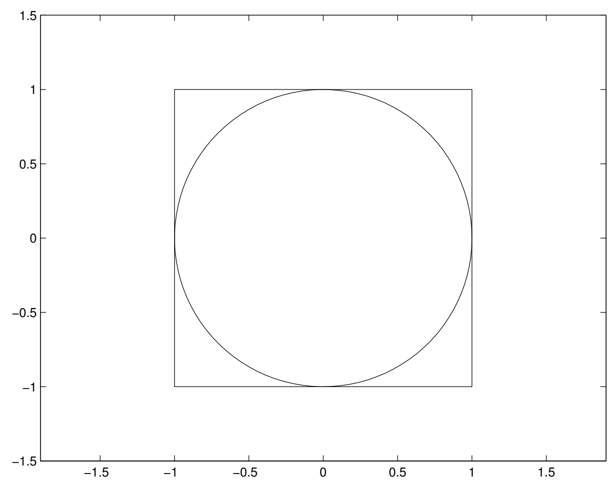


FIGURE 1.10 *A target*

Random simulations can be used to answer "nonrandom" questions. Suppose we throw n darts at the circle-in-square target depicted in Figure 1.10. Assume that the darts land anywhere on the square with equal probability and that the square has side 2 and center $(0, 0)$. After a large number of throws, the fraction of the darts that land inside the circle should be approximately equal to $\pi/4$, the ratio of the circle area to the square's area. Thus,

$$\pi \;\approx\; 4\,\frac{\text{Number of Throws Inside the Circle}}{\text{Total Number of Throws}}.$$

By simulating the throwing of a large number of darts, we can produce an estimate of π. Here is a script file that does just that:

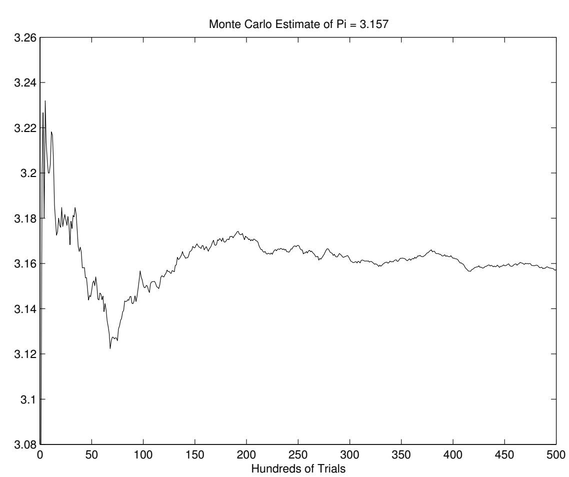


FIGURE 1.11 *A Monte Carlo estimate of* π

```
% Script File: Darts
% Estimates pi using random dart throws.
close all
rand('seed',.123456);
NumberInside = 0;
PiEstimate = zeros(500,1);
for k=1:500
   x = -1+2*rand(100,1);
   y = -1+2*rand(100,1);
   NumberInside = NumberInside + sum(x.^2 + y.^2 <= 1);
   PiEstimate(k) = (NumberInside/(k*100))*4;
end
plot(PiEstimate)
title(sprintf('Monte Carlo Estimate of Pi = %5.3f',PiEstimate(500)));
xlabel('Hundreds of Trials')
```

(See Figure 1.11.) Notice that the estimated values are gradually improving with n, but that the "progress" towards 3.14159... is by no means steady or fast. Simulation in this spirit is called *Monte Carlo*. The command `rand('seed',.123456)` starts the random number sequence with a prescribed *seed*. This enables one to repeat the random simulation with exactly the same sequence of underlying random numbers.

The `any` and `all` functions indicate whether any or all of the components of a vector are nonzero. Thus, if `x` and `y` are vectors of the same length, then `a = any( x.^2 + y.^2 <= 1)` assigns to `a` the value "1" if there is at least one (x_i, y_i) in the unit circle and "0" otherwise. Similarly, `b = all( x.^2 + y.^2 <= 1)` assigns "1" to `b` if all the (x_i, y_i) are in the unit circle and assigns "0" otherwise.

1.3.3 Polygon Smoothing

If `x` and `y` are $n+1$-vectors (of the same type) and $x_1 = x_{n+1}$ and $y_1 = y_{n+1}$, then `plot(x,y,x,y,'*')` displays the polygon obtained by connecting the points $(x_1, y_1), \ldots, (x_{n+1}, y_{n+1})$ in order. If we compute

```
xnew = [(x(1:n)+x(2:n+1))/2;(x(1)+x(2))/2];
ynew = [(y(1:n)+y(2:n+1))/2;(y(1)+y(2))/2];
plot(xnew,ynew)
```

then a new polygon is displayed that is obtained by connecting the side midpoints of the original polygon. We wish to explore what happens when this process is repeated.

The first issue that we have to deal with is how to specify the "starting polygon" such as the one displayed in Figure 1.12. One approach is to use the `ginput` command that supports mouseclick input. It returns the x-y-coordinates of the click with respect to the current axis. Under the control of a `for`-loop an assignment of the form `[x(k),y(k)] = ginput(1)` could be used to places the coordinates of the kth vertex in `x(k)` and `y(k)`, e.g.,

```
n = input('Enter the number of edges:');
figure
axis([0 1 0 1])
axis square
hold on
x = zeros(n,1);
y = zeros(n,1);
for k=1:n
   title(sprintf('Click in %2.0f more points.',n-k+1))
   [x(k) y(k)] = ginput(1);
   plot(x(1:k),y(1:k), x(1:k),y(1:k),'*')
end
```

```
x = [x;x(1)];
y = [y;y(1)];
plot(x,y,x,y,'*')
title('The Original Polygon')
hold off
```

The `for`-loop displays the sides of the polygon as it is "built up." If we did not care about this kind of graphical feedback as we click in the vertices, then the command `[x,y] = ginput(n)` could be used. This just stores the coordinates of the next n mouseclicks in `x` and `y`. Notice how we set up an "empty" figure with a prescribed axis in advance of the data acquisition.

Now that vertices of the starting polygon are available, the connect-the-midpoint process can begin:

```
k=0;
xlabel('Click inside window to smooth, outside window to quit.')
[a,b] = ginput(1);
v = axis;
while (v(1)<=a) & (a<=v(2)) & (v(3)<=b) & (b<=v(4));
   k = k+1;
   x = [(x(1:n)+x(2:n+1))/2;(x(1)+x(2))/2];
   y = [(y(1:n)+y(2:n+1))/2;(y(1)+y(2))/2];
   m = max(abs([x;y])); x = x/m; y = y/m;
   figure
   plot(x,y,x,y,'*')
   axis square
   title(sprintf('Number of Smoothings = %1.0f',k))
   xlabel('Click inside window to smooth, outside window to quit.')
   v = axis;
   [a,b] = ginput(1);
end
```

The command `v = axis` assigns to `v` a 4-vector $[x_{min}, x_{max}, y_{min}, y_{max}]$ that specifies the x and y ranges of the current figure. The `while`-loop that oversees the process terminates as soon as the solicited mouseclick falls outside the plot window. The polygons are scaled so that they are roughly the same size.

Once the execution of the loop is completed, the evolution of the smoothed polygons can be reviewed by using `figure`. For example, the command `figure(2)` displays the polygon after two smoothings. (See Figure 1.13.) This works because a new figure is generated each pass through the `while`-loop so in effect, each plot is saved. The script `Smooth` encapsulates the whole process.

Problems

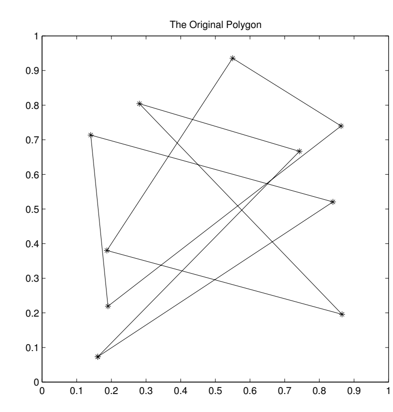


FIGURE 1.12 *The initial polygon*

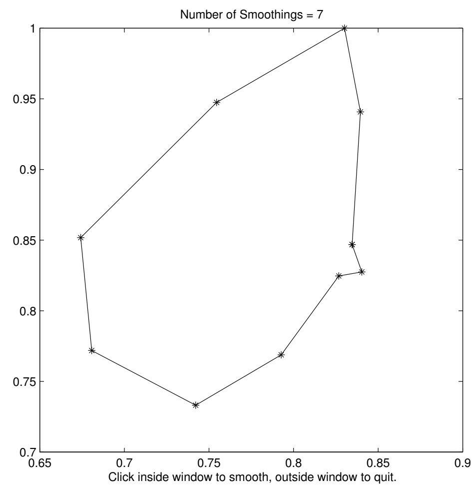


FIGURE 1.13 *A smoothed polygon*

P1.3.1 Suppose $\{x_i\}$ is the up/down sequence with $x_1 = m$. Let $g(m)$ be the index of the first x_i that equals one. Plot the values of g for $m = 1{:}200$.

P1.3.2 Consider the quadratic equation $ax^2 + bx + c = 0$. Let P_1 be the probability that this equation has complex roots, given that the coefficients are random variables with uniform(0,1) distribution. Let $P_1(n)$ be a Monte Carlo estimate of this probability based on n trials. Let P_2 be the probability that this equation has complex roots given that the coefficients are random variables with normal(0,1) distribution. Let $P_2(n)$ be a Monte Carlo estimate of this probability based on n trials. Write a script that prints a nicely formatted table that reports the value of $P_1(n)$ and $P_2(n)$ for $n = 100{:}100{:}800$.

P1.3.3 Write a simulation that estimates the volume of $\{(x_1, x_2, x_3, x_4) : x_1^2+x_2^2+x_3^2+x_4^2 \le 1\}$, the unit sphere in 4-dimensional space.

P1.3.4 Let $S = \{\, (x, y) \mid -1 \le x \le 1,\ -1 \le y \le 1\}$. Let S_0 be the set of points in S that are closer to the point $(.2, .4)$ than to an edge of S. Write a MATLAB script that estimates the area of S_0.

1.4 Error

Errors abound in scientific computation. Rounding errors attend floating point arithmetic, terminal screens are granular, analytic derivatives are approximated with divided differences, a polynomial is used in lieu of the sine function, the data acquired in a lab are correct to only three significant digits, etc. Life in computational science is like this, and we have to build up a facility for dealing with it. In this section we focus on the mathematical errors that arise through discretization and the rounding errors that arise due to finite precision arithmetic.

1.4.1 Absolute and Relative Error

If $\tilde{x}$ approximates a scalar x, then the *absolute error* in $\tilde{x}$ is given by $|\tilde{x} - x|$ while the *relative error* is given by $|\tilde{x} - x|/|x|$. If the relative error is about 10^{-d}, then $\tilde{x}$ has approximately d correct significant digits in that there exists a number τ having the form

$$\tau \;=\; \pm(.\underbrace{00\ldots0}_{d \text{ zeros}}n_{d+1}n_{d+2}\ldots) \times 10^g$$

so that $\tilde{x} = x + \tau$. (Here, g is some integer.)

As an exercise in relative and absolute error, let's examine the quality of the Stirling approximation

$$S_n = \sqrt{2\pi n}\left(\frac{n}{e}\right)^n, \qquad e = \exp(1).$$

to the factorial function $n! = 1 \cdot 2 \cdots n$. Here is a script that produces a table of errors:

```
% Script File: Stirling
% Prints a table showing error in Stirling's formula for n!
clc
disp('                      Stirling         Absolute    Relative')
disp('   n          n!     Approximation        Error       Error')
disp('----------------------------------------------------------------')
e = exp(1);
nfact = 1;
for n = 1:13
   nfact = n*nfact;
   s = sqrt(2*pi*n)*((n/e)^n);
   abserror = abs(nfact - s);
   relerror = abserror/nfact;
   s1 = sprintf('  %2.0f   %10.0f   %13.2f',n,nfact,s);
   s2 = sprintf('   %13.2f    %5.2e',abserror,relerror);
   disp([s1 s2])
end
```

Notice how the strings `s1` and `s2` are concatenated before they are displayed. In general, you should think of a string as a row vector of characters. Concatenation is then just a way of obtaining a new row vector from two smaller ones. This is the logic behind the required square bracket.

The command `clc` clears the command window and moves the cursor to the top. This ensures that the table produced is profiled nicely in the command window. Here it is:

```
                         Stirling         Absolute    Relative
   n          n!     Approximation          Error       Error
----------------------------------------------------------------
   1           1              0.92            0.08    7.79e-02
   2           2              1.92            0.08    4.05e-02
   3           6              5.84            0.16    2.73e-02
   4          24             23.51            0.49    2.06e-02
   5         120            118.02            1.98    1.65e-02
   6         720            710.08            9.92    1.38e-02
   7        5040           4980.40           59.60    1.18e-02
   8       40320          39902.40          417.60    1.04e-02
   9      362880         359536.87         3343.13    9.21e-03
  10     3628800        3598695.62        30104.38    8.30e-03
  11    39916800       39615625.05       301174.95    7.55e-03
  12   479001600      475687486.47      3314113.53    6.92e-03
  13  6227020800     6187239475.19     39781324.81    6.39e-03
```

1.4.2 Taylor Approximation

The partial sums of the exponential satisfy

$$e^x = \sum_{k=0}^{n} \frac{x^k}{k!} \; + \; \frac{e^\eta}{(n+1)!}x^{n+1}$$

for some η in between 0 and x. The mathematics says that if we take enough terms, then the partial sums converge. The script `ExpTaylor` explores this by plotting the partial sum relative error as a function of n.

```
% Script File: ExpTaylor
% Plots, as a function of n, the relative error in the
% Taylor approximation 1 + x + x^2/2! +...+ x^n/n! to exp(x).
close all
nTerms = 50;
for x=[10 5 1 -1 -5 -10]
   figure
   term = 1; s = 1; f = exp(x)*ones(nTerms,1);
   for k=1:nTerms, term = x.*term/k; s = s+ term; err(k) = abs(f(k) - s); end
   relerr = err/exp(x);
   semilogy(1:nTerms,relerr)
   ylabel('Relative Error in Partial Sum.')
   xlabel('Order of Partial Sum.')
   title(sprintf('x = %5.2f',x))
end
```

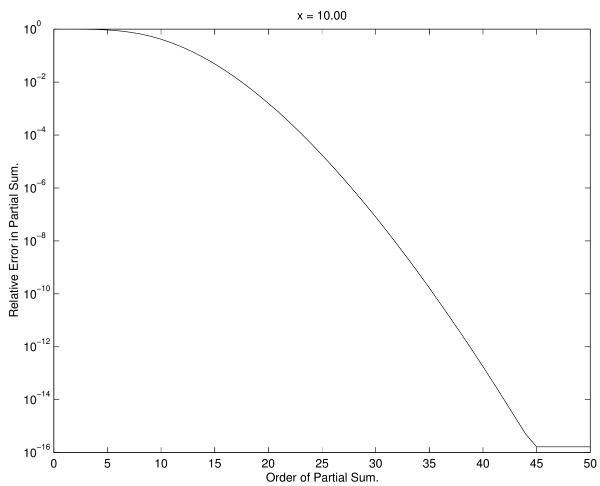


FIGURE 1.14 *Error in Taylor approximations to* $e^x, x = 10$

When plotting numbers that vary tremendously in range, it is useful to use `semilogy`. It works just like `plot`, only the base-10 log of the y-vector is displayed. `ExpTaylor` produces six figure windows, one each for the six x-values. For example, the $x = 10$ plot is in figure 1. By entering the command `figure(1)`, this plot is "brought up" by making the Figure 1 window the active window. It could then (for example) be printed. (See Figures 1.14 and 1.15.)

1.4.3 Rounding Errors

The plots produced by `ExpTaylor` reveal that the mathematical convergence theory does not quite apply. The errors do *not* go to zero as the number of terms in the series increases. In each case, they seem to "bottom out" at some small value. Once that happens, the incorporation of more terms into the partial sum does not make a difference. Moreover, by comparing the plots in Figures 1.14 and 1.15, we observe that where the relative error bottoms out depends on x. The relative error for $x = -10$ is much worse than for $x = 10$.

An explanation of this phenomenon requires an understanding of floating point arithmetic. Like it or not, numerical computation involves working with an inexact computer arithmetic system. This will force us to rethink the connections between mathematics and the development of algorithms. Nothing will be simple ever again.

To dramatize this point, consider the plot of a rather harmless looking function: $p(x) = (x-1)^6$. The script `Zoom` graphs this polynomial over increasingly smaller neighborhoods around $x = 1$, but it uses the formula

$$p(x) = x^6 - 6x^5 + 15x^4 - 20x^3 + 15x^2 - 6x + 1.$$

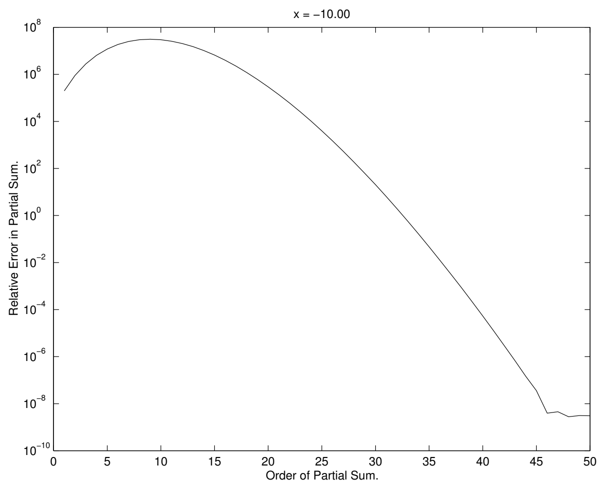


FIGURE 1.15 *Error in Taylor approximations to* $e^x, x = -10$

```
% Script File: Zoom
% Plots (x-1)^6 near x=1 with increasingly refined scale.
% Evaluation via x^6 - 6x^5 + 15x^4 - 20x^3 + 15x^2 - 6x +1
% leads to severe cancelation.

close all
k = 0; n = 100;
for delta = [.1 .01 .008 .007 .005  .003 ]
   x = linspace(1-delta,1+delta,n)';
   y = x.^6 - 6*x.^5 + 15*x.^4 - 20*x.^3 + 15*x.^2  - 6*x + ones(n,1);
   k = k+1; subplot(2,3,k); plot(x,y,x,zeros(1,n))
   axis([1-delta 1+delta -max(abs(y)) max(abs(y))])
end
```

Notice how the x-axis is plotted and how it is forced to appear across the middle of the window. (See Figure 1.16 for a display of the plots.) As we increase the "magnification," a very chaotic behavior unfolds. It seems that $p(x)$ has thousands of zeros!

It turns out that if the plot is based on the formula $(x-1)^6$ instead of its expansion, then the expected graph is displayed and this gets right to the heart of the example. *Algorithms that are equivalent mathematically may behave very differently numerically.* The time has come to look at floating point arithmetic.

1.4.4 The Floating Point Numbers

A nonzero value x in a base-2 floating point number system has the following form:

$$x = \pm 1.b_1 b_2 \ldots b_t \times \beta^e \qquad L \le e \le U$$

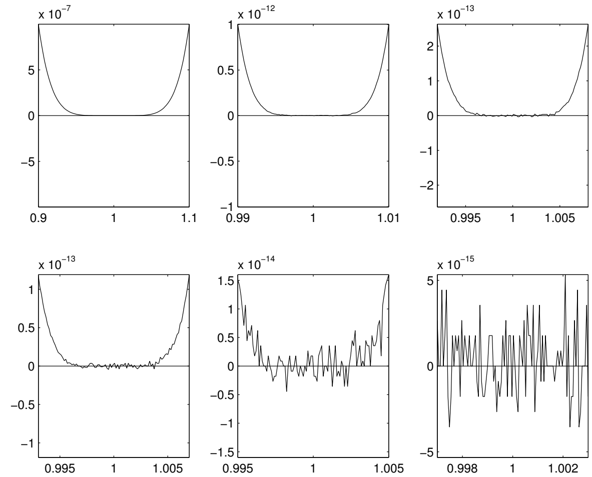


FIGURE 1.16 *Plots of* $(x-1)^6 = x^6 - 6x^5 + 15x^4 - 20x^3 + 15x^2 - 6x + 1$ *near* $x = 1$

The bits $b_1, b_2, \ldots b_t$ make up the *mantissa*. The *exponent* e is restricted to the interval $[L, U]$. Zero is also a floating point number and we assume that in its representation both the mantissa and exponent are set to zero.

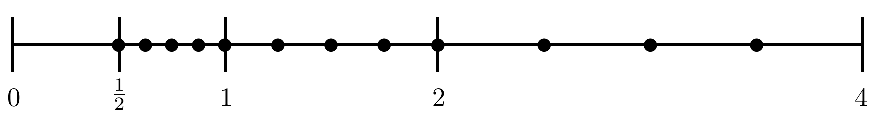
We denote the set of floating point numbers by $\mathbf{F}(t, L, U)$. To emphasize the finiteness of this set, suppose $t = 2$, $L = -1$ and $U = +1$. There are twelve positive floating point numbers:

$$x \;=\; \left\{ \begin{array}{c} (1.00)_2 \\ (1.01)_2 \\ (1.10)_2 \\ (1.11)_2 \end{array} \right\} \times \left\{ \begin{array}{c} 2^{-1} \\ 2^{0} \\ 2^{1} \end{array} \right\}.$$

The base-2 notation is not difficult. Thus, $x = (1.01)_2 \times 2^1$ represents

$$\left(1 \,+\, 0 \cdot \frac{1}{2} \,+\, 1 \cdot \frac{1}{4}\right) \times 2 \;=\; 2.5$$

There is a smallest positive floating point number $(1.00 \times 2^{-1} = .5)$ and a largest floating point number $(1.11 \times 2^1 = 3.75)$. Moreover, the spacing between the floating point numbers is not uniform as can be seen from this display of the positive portion of $\mathbf{F}(2, -1, 1)$:



Extrapolating from this small example we identify three important numbers associated with $\mathbf{F}(t, L, U)$:

`m`	the smallest positive floating point number $= 2^L$.
`M`	the largest positive floating point number $= (2 - 2^{-t})2^U$
`eps`	the distance from 1 to the next largest floating point number $= 2^{-t}$

Note that if x is a floating point number and $2^e < x < 2^{e+1}$, then $x - 2^{e-t}$ is its left "neighbor" and $x + 2^{e-t}$ is its right neighbor.

Now let us talk about the errors associated with the $\mathbf{F}(t, L, U)$ representation. If x is a real number, then let $\mathrm{fl}(x)$ be the nearest floating point number to x. (Assume the existence of a tie-breaking rule.) Think of $\mathrm{fl}(x)$ as the stored version of x. The following theorem bounds the relative error in $\mathrm{fl}(x)$.

Theorem 1 *Suppose we are given a set of floating point numbers with mantissa length t and exponent range $[L, U]$. If $x \in \mathbb{R}$ satisfies $m < |x| < M$, then*

$$\frac{|\mathrm{fl}(x) - x|}{|x|} \leq 2^{-t-1} = \texttt{eps}$$

Proof Without loss of generality, assume that x is positive and that

$$x = (1.b_1 b_2 \ldots b_t b_{t+1} \ldots)_2 \times 2^e.$$

If x is a power of two, then the theorem obviously holds since $\mathrm{fl}(x) = x$ and the relative error is zero. Otherwise we observe that the spacing of the floating point numbers at x is 2^{e-t}. Since $\mathrm{fl}(x)$ is the closest floating number to x, we have

$$|\mathrm{fl}(x) - x| \leq \frac{1}{2} 2^{e-t} = 2^{e-t-1}.$$

From the lower bound $\beta^e < x$ it follows that

$$\frac{|\mathrm{fl}(x) - x|}{|x|} \leq \frac{2^{e-t-1}}{2^e} = 2^{-t-1}. \quad \square$$

Another way of saying the same thing is that

$$\mathrm{fl}(x) = x(1 + \delta)$$

where $|\delta| \leq \texttt{eps}$.

What are the values of t, L and U on a typical computer? For the widely implemented IEEE double precision format, $t = 52$, $L = -1022$ and $U = 1023$. This representation fits into a 64-bit word because we need one bit for the sign and because 11 bits are required to store $e + 1023$. (The last is a clever trick for encoding the sign of the exponent.)

The quantity `eps` is referred to as the machine precision (a.k.a. unit roundoff) and is available in MATLAB through the built-in constant `eps`:

```
>> What_Is_eps = eps

What_Is_eps =

   2.220446049250313e-016
```

Thus, in the IEEE floating point environment, $\texttt{eps} = 2^{-52} \approx 10^{-16}$.

IEEE floating point arithmetic is carefully designed so that when two floating point numbers are combined via $+$, $-$, $\times$, or $/$, then the answer is the nearest floating point number to the exact answer. One way to say this for any of these four "ops" is

$$\mathrm{fl}(x \operatorname{op} y) = (x \operatorname{op} y)(1 + \delta) \qquad |\delta| \leq \texttt{eps}$$

Thus, there is good relative error for an individual floating point operation. As we shall see, it does **not** follow that sequences of floating point operations result in an answer that has $O(\texttt{eps})$ relative error.

Some simple `while`-loop computations can be used to glean information about the underlying floating system. Here is a script that assigns the value of the smallest positive integer so $1 + 1/2^p = 1$ in floating point arithmetic:

```
p = 0; y = 1; z = 1+y;
while z>1
  y = y/2;
  p = p+1;
  z = 1+y;
end
```

With IEEE arithmetic, $p = 53$. Stated another way, $1 + 1/2^{52}$ can be represented exactly but $1 + 1/2^{53}$ cannot.

The finiteness of the exponent range has ramifications too. A floating point operation can result in an answer that is too big to represent. When this happens, it is called *floating point overflow* and a special value called `inf` is produced. Here is a script that assigns to `r` the smallest positive integer so $2^r =$ `inf` in floating point arithmetic:

```
x = 1;
r = 0;
while x~=inf
  x = 2*x;
  r = r+1;
end
```

When IEEE arithmetic is used, $r = 1024$. In other words, 2^{1023} can be represented but 2^{1024} cannot.

At the other end of the scale, if a floating point operation renders a nonzero result that is too small to represent, then an *underflow* results. In light of the fact that the smallest positive floating point number is $\mathrm{m} = 2^{-1022}$, we anticipate that the script

```
x = 1;
q = 0;
while x>0
   x = x/2;
   q = q+1;
end
```

would assign -1023 to `q`. However, the actual value that is assigned to `q` is 1075. This is because the IEEE standard implements what is call *gradual underflow* meaning that the actual smallest floating point number that can be represented is $2^{L-t} = 2^{-1022-52} = 2^{-1074}$.

Sometimes these are just set to zero. Sometimes they result in program termination. Here is a script that assigns to `q` the smallest positive integer so that $1/2^q = 0$ in floating point arithmetic:

Problems

P1.4.1 The binomial coefficient n-choose-k is defined by

$$\left(\begin{array}{c} n \\ k \end{array} \right) = \frac{n!}{k!(n-k)!}.$$

Let $B_{n,k} = S_n/(S_k S_{n-k})$. Write a script analogous to `Stirling` that explores the error in $B_{n,k}$ for the cases $(n,k) = (52,2),(52,3),\ldots,(52,13)$. There are no set rules on output except that it should look nice and clearly present the results.

P1.4.2 The sine function has the power series definition

$$\sin(x) = \sum_{k=0}^{\infty}(-1)^k \frac{x^{2k+1}}{(2k+1)!}.$$

Write a script `SinTaylor` analogous to `ExpTaylor` that explores the relative error in the partial sums.

P1.4.3 Write a script that solicits n and plots both $\sin(x)$ and

$$S_n(x) = \sum_{k=0}^{n}(-1)^k \frac{x^{2k+1}}{(2k+1)!}$$

across the interval $[0, 2\pi]$.

P1.4.4 To affirm your understanding of the floating point representation, what is the largest value of n so that $n!$ can be exactly represented in $\mathbf{F}(52, -1022, 1023)$? Show your work.

P1.4.5 On a base-2 machine, the distance between 7 and the next largest floating point number is 2^{-12}. What is the distance between 70 and the next largest floating point number?

P1.4.6 Assume that x and y are floating point numbers in $\mathbf{F}(t, -10, 10)$. What is the smallest possible value of $y - x$ given that $x < 8 < y$? (Your answer will involve t.)

P1.4.7 What is the largest value of k such that 10^k can be represented exactly in $\mathbf{F}(52, -1022, 1023)$?

P1.4.8 What is the nearest floating point number to 64 on a base-2 computer with 5-bit mantissas? Show work.

P1.4.9 If 127 is the nearest floating point number to 128 on a base-2 computer, then how long is the mantissas? Show work.

1.5 Designing Functions

An ability to write good MATLAB functions is crucial. Two examples are used to clarify the essential ideas: Taylor series and numerical differentiation.

1.5.1 Four Ways to Compute the Exponential of a Vector of Values

Consider once again the Taylor approximation

$$T_n(x) = \sum_{k=0}^{n} \frac{x^k}{k!}$$

to the exponential e^x. It is possible to write functions in MATLAB, and here is one that encapsulates this approximation:

```
  function y = MyExpF(x,n)
% y = MyExpF(x,n)
% x is a scalar, n is a positive integer
% and y = n-th order Taylor approximation to exp(x).
term = 1;
y = 1;
for k = 1:n
   term = x*term/k;
   y = y + term;
end
```

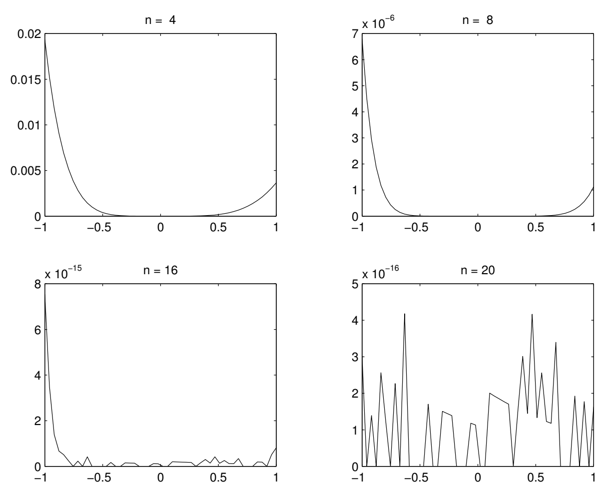


FIGURE 1.17 *Relative error in* $T_n(x)$

The function itself must be placed in a separate `.m` file[2] having the same name as the function, e.g., `MyExpF.m`. Once that is done, it can be referenced like any of the built-in functions. Thus, the script

[2]Subfunctions are an exception. Enter `help function` for details.

```
m = 50;
x = linspace(-1,1,m);
y = zeros(1,m);
exact = exp(x);
k = 0;
for n = [4 8 16 20]
   for i=1:m
      y(i) = MyExpF(x(i),n);
   end
   RelErr = abs(exact - y)./exact;
   k = k+1;
   subplot(2,2,k)
   plot(x,RelErr)
   title(sprintf('n = %2.0f',n))
end
```

plots the relative error in $T_n(x)$ for $n = 4, 8, 16$, and 20 across $[-1, 1]$. (See Figure 1.17.)

When writing a MATLAB function you must adhere to the following rules and guidelines:

- From the example we infer the following general structure for a MATLAB function:

  `function` ⟨*Output Parameter*⟩ `=` ⟨*Name of Function*⟩`(`⟨*Input Parameters*⟩`)`
  `%`
  `%` ⟨*Comments that completely specify the function.*⟩
  `%`
  ⟨*function body*⟩

- Somewhere in the function body the desired value must be assigned to the output variable.

- Comments that completely specify the function should be given immediately after the `function` statement. The specification should detail all input value assumptions (the *pre-conditions*) and what may be assumed about the output value (the *postconditions*).

- The lead block of comments after the `function` statement is displayed when the function is probed using `help` (e.g., `help MyExpF`).

- The input and output parameters are formal parameters. At the time of the call they are replaced by the actual parameters.

- All variables inside the function are local and are not part of the MATLAB workspace.

- If the function file is not in the current directory, then it cannot be referenced unless the appropriate path is established. Type `help path`.

Further experimentation with `MyExpF` shows that if $n = 17$, then full machine precision exponentials are computed for all $x \in [-1, 1]$. With this understanding about the Taylor approximation across $[-1, 1]$, we are ready to develop a "vector version":

```
  function y = MyExp1(x)
% y = MyExp1(x)
% x is a column vector and y is a column vector with the property that
% y(i) is a Taylor approximation to exp(x(i)) for i=1:n.
n = 17; p = length(x);
y = ones(p,1);
for i=1:p
   y(i) = MyExpF(x(i),n);
end
```

This example shows several things: (1) A Matlab function can have vector arguments and can return a vector, (2) the `length` function can be used to determine the size of an input vector, (3) one function can reference another. Here is a script that references `MyExp1`:

```
x = linspace(-1,1,50);
exact = exp(x);
RelErr = abs(exact - MyExp1(x')')./exact;
```

Notice the transpose that is required to ensure that the vector passed to `MyExp1` is a column vector. The other transpose is required to make `MyExp1(x')` a row vector so that it can be combined with `exact`. Here is another implementation that is not sensitive to the shape of `x`:

```
  function y = MyExp2(x)
% y = MyExp2(x)
% x is an n-vector and y is an n-vector with the same shape
% and the property that y(i) is a Taylor approximation to exp(x(i)), i=1:n.

y = ones(size(x));
nTerms = 17;
term = ones(size(x));
for k=1:nTerms
   term = x.*term/k;
   y = y + term;
end
```

The expression `ones(size(x))` creates a vector of ones that is exactly the same shape as `x`. In general, the command `[p,q] = size(A)` returns the number of rows and columns in `A` in `p` and `q`, respectively. If such a 2-vector is passed to `ones`, then the appropriate matrix of ones is established. (The same comment applies to `zeros`.) The new implementation "doesn't care" whether `x` is a row or column vector. The script

```
x = linspace(-1,1,50);
exact = exp(x);
RelErr = abs(exact - MyExp2(x))./exact;
```

produces a vector of relative error exactly the same size as `x`.

Notice the use of pointwise multiplication. In contrast to `MyExp1` which computes the component-level exponentials one at a time, `MyExp2` computes them "at the same time." In general, Matlab runs faster in *vector mode*. Here is a script that quantifies this statement by *benchmarking* these two functions:

```
nRepeat = 100;
disp(' Length(x)     Time(MyExp2)/Time(MyExp1)')
disp('-------------------------------------------')
for L = 1000:100:1500
   xL = linspace(-1,1,L);
   tic
   for k=1:nRepeat, y = MyExp1(xL); end
   T1 = toc;
   tic
   for k=1:nRepeat, y = MyExp2(xL); end
   T2 = toc;
   disp(sprintf('%6.0f  %13.6f  ',L,T2/T1))
end
```

The script makes use of `tic` and `toc`. To time a code fragment, "sandwich" it in between a `tic` and a `toc`. Keep in mind that the clock is discrete and is typically accurate to within a millisecond. Therefore, whatever is timed should take somewhat longer than a millisecond to execute to ensure reliability. To address this issue it is sometimes necessary to time repeated instances of the code fragment as above. Here are some sample results:

```
Length(x)   Time(MyExp2)/Time(MyExp1)
--------------------------------
  1000           0.086525
  1100           0.101003
  1200           0.104044
  1300           0.080007
  1400           0.087395
  1500           0.082073
```

It is important to stress that these are *sample* results. Different timings would result on different computers.

The for-loop implementations in MyExp1 and MyExp2 are flawed in two ways. First, the value of n chosen is machine dependent. A different n would be required on a computer with a different machine precision. Second, the number of terms required for an x value near the origin may be considerably less than 17. To rectify this, we can use a while-loop that keeps adding in terms until the next term is less than or equal to eps times the size of the current partial sum:

```
  function y = MyExpW(x)
% y = MyExpW(x)
% x is a scalar and y is a Taylor approximation to exp(x).
y = 0;
term = 1;
k=0;
while abs(term) > eps*abs(y)
   k = k + 1;
   y = y + term;
   term = x*term/k;
end
```

To produce a vector version, we can proceed as in MyExp1 and simply call MyExpW for each component:

```
  function y = MyExp3(x)
% y = MyExp3(x)
% x is a column n-vector and y is a column n-vector with the property that
% y(i) is a Taylor approximation to exp(x(i)) for i=1:n.
n = length(x);
y = ones(n,1);
for i=1:n
   y(i) = MyExpW(x(i));
end
```

Alternatively, we can follow the MyExp2 idea and vectorize as follows:

```
  function y = MyExp4(x)
% y = MyExp4(x)
% x is an n-vector and y is an n-vector with the same shape and the
% property that y(i) is a Taylor approximation to exp(x(i)) for i=1:n.
y = zeros(size(x));
term = ones(size(x));
k = 0;
while any(abs(term) > eps*abs(y))
   y = y + term;
   k = k+1;
   term = x.*term/k;
end
```

Observe the use of the `any` function. It returns a "1" as long as there is at least one component in `abs(term)` that is larger than `eps` times the corresponding term in `abs(y)`. If `any` returns a zero, then this means that `term` is small relative to `y`. In fact, it is so small that the floating point sum of `y` and `term` is `y`. The `while`-loop terminates as this happens.

1.5.2 Numerical Differentiation

Suppose $f(x)$ is a function whose derivative we wish to approximate at $x = a$. A Taylor series expansion about this point says that

$$f(a+h) = f(a) + f'(a)h + \frac{f''(\eta)}{2}h^2$$

for some $\eta \in [a, a+h]$. Thus,

$$D_h = \frac{f(a+h) - f(a)}{h}$$

provides increasingly good approximations as h gets small since

$$D_h = f'(a) + f''(\eta)\frac{h}{2}.$$

Here is a script that enables us to explore the quality of this approach when $f(x) = \sin(x)$:

```
a = input('Enter a:  ');
h = logspace(-1,-16,16);
Dh = (sin(a+h) - sin(a))./h;
err = abs(Dh - cos(a));
```

Using this to find the derivative of sin at $a = 1$, we see the following:

h	Absolute Error
1.0e-01	0.0429385533327507
1.0e-02	0.0042163248562708
1.0e-03	0.0004208255078129
1.0e-04	0.0000420744495186
1.0e-05	0.0000042073622750
1.0e-06	0.0000004207468094
1.0e-07	0.0000000418276911
1.0e-08	0.0000000029698852
1.0e-09	0.0000000525412660
1.0e-10	0.0000000584810365
1.0e-11	0.0000011687040611
1.0e-12	0.0000432402169239
1.0e-13	0.0007339159003137
1.0e-14	0.0037069761981869
1.0e-15	0.0148092064444385
1.0e-16	0.5403023058681398

The loss of accuracy may be explained as follows. Any error in the computation of the numerator of D_h is magnified by $1/h$. Let us assume that the values returned by `sin` are within `eps` of their true values. Thus, instead of a precise calculus bound

$$|D_h - f'(a)| \leq \frac{h}{2}|f''(\eta)|$$

as predicted earlier, we have a heuristic bound

$$|D_h - f'(a)| \approx \frac{h}{2}|f''(\eta)| + \frac{2\texttt{eps}}{h}.$$

The right-hand side incorporates the "truncation error" due to calculus and the computation error due to roundoff. This quantity is minimized when $h = 2\sqrt{\texttt{eps}/|f''(\eta)|}$.

Let's package these observations and write a function that does numerical differentiation. The key analytical detail is the intelligent choice of h. If we have an upper bound on the second derivative of the form $|f''(x)| \le M_2$, then the truncation error can be bounded as follows:

$$|D_h - f'(a)| \le \frac{M_2}{2}h. \tag{1.1}$$

If the absolute error in a computed function evaluation is bounded by δ, then

$$errD(h) = M_2\frac{h}{2} + \frac{2\delta}{h}$$

is a reasonable model for the total error. This quantity is minimized if

$$h_{opt} = 2\sqrt{\frac{\delta}{M_2}},$$

giving

$$errD(h_{opt}) = 2\sqrt{\delta M_2}.$$

Here is a function that implements this idea:

```
  function [d,err] = Derivative(f,a,delta,M2)
% f is a handle that references a function f(x) whose derivative
% at x = a is sought. delta is the absolute error associated with
% an f-evaluation and M2 is an estimate of the second derivative
% magnitude near a. d is an approximation to f'(a) and err is an estimate
% of its absolute error.
%
% Usage:
%     [d,err] =  Derivative(@f,a)
%     [d,err] =  Derivative(@f,a,delta)
%     [d,err] =  Derivative(@f,a,delta,M2)

if nargin <= 3
   % No derivative bound supplied, so assume the
   % second derivative bound is 1.
   M2 = 1;
end
if nargin == 2
   % No function evaluation error supplied, so
   % set delta to eps.
   delta = eps;
end
% Compute optimum h and divided difference
hopt = 2*sqrt(delta/M2);
d   = (f(a+hopt) - f(a))/hopt;
err = 2*sqrt(delta*M2);
```

There are several new syntactic features associated with this implementation. We identify them through a sequence of examples.

Example 1. Compute the derivative of $f(x) = \exp(x)$ at $x = 5$. Assume that the `exp` function returns values that are correct to machine precision and use the fact that the second derivative of f is bounded by 500:

```
[der_val,err_est] = Derivative(@exp,5,eps,500)
```

To hand over a function to `Derivative`, you pass its *handle*. This is simply the name of the function preceded by the "at" symbol "`@`". In effect `@exp` "points" to the `exp` function. Another aspect of this example is that functions in MATLAB can return more than one item: Just separate the output parameters with commas and enclose with square brackets.

Example 2. Same as Example 1 only (pretend) that we cannot produce an upper bound on the second derivative:

```
[der_val,err_est] = Derivative(@exp,5,eps)
```

The `nargin` command makes it possible to have abbreviated calls. In this case, `Matlab` "knows" that this is a 2-argument call and substitutes a value for the missing input parameter.

Example 3. Same as Example 1 only you don't care about the error estimate:

```
der_val = Derivative(@exp,5,eps,500)
```

In this case

Example 4. Assuming the existence of

```
function y = MyF(x,alfa,beta)
y = alfa*exp(beta*x);
```

estimate the derivative at $x = 10$ assuming that $\alpha = 20$ and $\beta = -2$:

```
alfa = 20;
beta = -2;
der_val = Derivative(@(x) MyF(x,alfa,beta),10);
```

This illustrates the use of the *anonymous* function idea which is very useful when functions depend on parameters.

Problems

P1.5.1 It can be shown that

$$C_h = \frac{f(a+h) - f(a-h)}{2h}$$

satisfies

$$|C_h - f'(a)| \le \frac{M_3}{6}h^2$$

if

$$|f^{(3)}(x)| \le M_3$$

for all x. Model the error in the evaluation of C_h by

$$errC(h) = \frac{M_3 h^2}{6} + 2\frac{\delta}{h}.$$

Generalize `Derivative` so that it has a 5th optional argument `M3` being an estimate of the 3rd derivative. It should compute $f'(a)$ using the better of the two approximations D_h and C_h.

P1.5.2 Consider the ellipse $P(t) = (x(t), y(t))$ with

$$\begin{aligned} x(t) &= a\cos(t) \\ y(t) &= b\sin(t) \end{aligned}$$

and assume that $0 = t_1 < t_2 < \ldots < t_n = \pi/2$. Define the points $Q_1, \ldots, Q_n$ by

$$Q_i = (x(t_i), y(t_i)).$$

Let L_i be the tangent line to the ellipse at Q_i. This line is defined by the parametric equations

$$\begin{aligned} x(t) &= a\cos(t_i) - a\sin(t_i)t \\ y(t) &= b\sin(t_i) + b\cos(t_i)t. \end{aligned}$$

Next, define the points $P_0, \ldots, P_n$ by

$$P_i = \begin{cases} (a,0) & i = 0 \\ \text{intersection of } L_i \text{ and } L_{i+1} & i = 1 \ldots n-1 \\ (0,b) & i = n \end{cases} .$$

For your information, if the lines defined by

$$\begin{aligned} x_1(t) &= \alpha_1 + \beta_1 t \\ y_1(t) &= \gamma_1 + \delta_1 t \\ \\ x_2(t) &= \alpha_2 + \beta_2 t \\ y_2(t) &= \gamma_2 + \delta_2 t \end{aligned}$$

intersect, then the point of their intersection (x_*, y_*) is given by

$$x_* = \frac{\beta_2(\alpha_1\delta_1 - \beta_1\gamma_1) - \beta_1(\alpha_2\delta_2 - \beta_2\gamma_2)}{\delta_1\beta_2 - \beta_1\delta_2} \qquad \text{and} \qquad y_* = \frac{\delta_2(\alpha_1\delta_1 - \beta_1\gamma_1) - \delta_1(\alpha_2\delta_2 - \beta_2\gamma_2)}{\delta_1\beta_2 - \beta_1\delta_2}.$$

Complete the following function:

```
  function [P,Q] = Points(a,b,t)
% a and b are positive, n = length(t)>=2, and 0 = t(1) < t(2) <... < t(n) = pi/2.
% For i=1:n, (Q(i,1),Q(i,2)) is the ith Q-point and (P(i,1),P(i,2)) is the ith P point.
```

Write a script file that calls `Points` with $a = 5$, $b = 2$, and `t = linspace(0,pi/2,4)`. The script should then plot in one window the first quadrant portion of the ellipse, the polygonal line that connects the Q points, and the polygonal line that connects the P points. Use `title` to display PL and QL, the lengths of these two polygonal lines, i.e., `title(sprintf(' QL = %10.6f PL = %10.6f ',QL,PL ))`.

P1.5.3 Write a MATLAB function `Ellipse(P,A,theta)` that plots the "tilted" ellipse defined by

$$\begin{aligned} x(t) &= \cos(\theta)\left[\frac{P-A}{2} + \frac{P+A}{2}\cos(t)\right] - \sin(\theta)\left[\sqrt{A\cdot P}\sin(t)\right] \\ y(t) &= \sin(\theta)\left[\frac{P-A}{2} + \frac{P+A}{2}\cos(t)\right] + \cos(\theta)\left[\sqrt{A\cdot P}\sin(t)\right] \end{aligned}$$

for $0 \le t \le 2\pi$. Your implementation should not have any loops.

P1.5.4 For a scalar z and a nonnegative integer n define

$$f(z,n) = \sum_{k=0}^{n} (-1)^k \frac{z^{2k+1}}{(2k+1)!}.$$

This is an approximation to the function $\sin(z)$. Write a MATLAB function `y = MySin(x,n)` that accepts a vector `x` and a nonnegative integer `n` and returns a vector `y` with the same size and orientation as `x` and with the property that $y_i = f(x_i, n)$ for $i = 1{:}length(x)$. The implementation should not involve any loops. Write a script that graphically reports on the relative error when `MySin` is applied to `x = linspace(.01,pi-.01)` for `n=3:2:9`. Use `semilogy` and present the four plots in a single window using `subplot`. To avoid `log(0)` problems, plot the maximum of the true relative error and `eps`. Label the axes. The title should indicate the value of n and the number of flops required by the call to `MySin`.

P1.5.5 Using `tic` and `toc`, plot the relative error in `pause(k)` for $k = 1{:}10$.

P1.5.6 Complete the following Matlab function

```
   function [cnew,snew] = F(c,s,a,b)
 % a and b are scalars with a<b. c and s are row (n+1)-vectors with the property that
 % c = cos(linspace(a,b,n+1)) and s = sin(linspace(a,b,n+1))
 %
 % cnew and snew are column (2n+1)-vectors with the property that
 % cnew = cos(linspace(a,b,2*n+1)) and snew = sin(linspace(a,b,2*n+1))
```

Your implementation should be vectorized and must make effective use of the trigonometric identities

$$\begin{aligned}\cos(\alpha+\Delta) &= \cos(\alpha)\cos(\Delta) - \sin(\alpha)\sin(\Delta)\\ \sin(\alpha+\Delta) &= \sin(\alpha)\cos(\Delta) + \cos(\alpha)\sin(\Delta)\end{aligned}$$

in order to reduce the number of new cosine and sine evaluations. Hint: Let Δ be the spacing associated with z.

P1.5.7 Complete the following function:

```
function BookCover(a,b,n)
% a and b are real with b<a. n is a positive integer.
% Let r1 = (a+b)/2 and r2 = (a-b)/2. In the same figure draws the ellipse
%
%                  (a*cos(t),b*sin(t))     0<=t<=2*pi,
%
% the "big" circle
%
%                  (r1*cos(t),r1*sin(t))   0<=t<=2*pi,
%
% and n "small" circles. The kth small circle should have radius r2 and center
% (r1*cos(2*pi*k/n),r1*sin(2*pi*k/n). A radius making angle -2*pi*k/n should be drawn
% inside the kth small circle.
```

Use `BookCover` to draw with correct proportions, the ellipse/circle configuration on the cover of the book.

1.6 Structure Arrays and Cell Arrays

As problems get more complicated it is very important to use appropriate data structures. The choice of a good data structure can simplify one's "algorithmic life." To that end we briefly review two ways that more advanced data structures can be used in MATLAB: *structure arrays* and *cell arrays*.

A structure array has fields and values. Thus,

```
A = struct('d',16,'m',23,'s',47);
```

establishes `A` as a structure array with fields "d", "m", and "s". Such a structure might be handy in a geodesy application where latitudes and longitudes are measured in degrees, minutes, and seconds. The field values are accessed with a "dot" notation. The value of `A.d` is 16, the value of `A.m` is 23, and the value of `A.s` is 47. The statement

```
r = pi*(A.d + A.m/60 + A.s/3600)/180;
```

assigns to `r` the radian equivalent of the angle represented by `A`. The triplet

```
NYC_Lat  = struct('d',40,'m',45,'s',27);
NYC_Long = struct('d',75,'m',12,'s',32);
C1 = struct('name','New York','lat',NYC_Lat,'long',NYC_Long);
```

establishes `C1` as a structure array with three fields. The first field is a string and the last two are structure arrays. Note that `C1.long.d` has value 75. One can also have an array of structure arrays:

```
NYC_Lat  = struct('d',16,'m',23,'s',47);
NYC_Long = struct('d',74,'m',2,'s',32);
City(1) = struct('name','New York','lat',NYC_Lat,'long',NYC_Long)
Ith_Lat  = struct('d',42,'m',25,'s',16);
Ith_Long = struct('d',76,'m',29,'s',41);
City(2) = struct('name','Ithaca','lat',Ith_Lat,'long',Ith_Long);
```

In this case, `City(2).lat.d` has value 42. We mention that a structure array can have an array field and functions can have input and output parameters that are structure arrays.

A cell array is basically a matrix in which a given entry can be a matrix, a structure array, or a cell array. If `m` and `n` are positive integers, then

```
C = cell(m,n)
```

establishes C as an m-by-n cell array. Cell entries are referenced with curly brackets. Thus, the cell array C in

```
C = cell(2,2);
C{1,1} = [1 2 ; 3 4];
C{1,2} = [ 5;6];
C{2,1} = [7 8];
C{2,2} = 9;
M = [C{1,1} C{1,2};C{2,1} C{2,2}]
```

is a way of representing the 3-by-3 matrix

$$M = \left[\begin{array}{cc|c} 1 & 2 & 5 \\ 3 & 4 & 6 \\ \hline 7 & 8 & 9 \end{array}\right].$$

1.6.1 Three-digit Arithmetic

Structures and strings are nicely reviewed by developing a three-digit, base-10 floating point arithmetic simulation package. Let's assume that the exponent range is $[-9, 9]$ and that we use a 4-field structure to represent each floating point number as described in the following specification:

```
  function f = Represent(x)
% f = Represent(x)
% Yields a 3-digit floating point representation of f:
%
%    f.mSignBit   mantissa sign bit (0 if x>=0, 1 otherwise)
%    f.m          mantissa (= f.m(1) + f.m(2)/10 + f.m(3)/100)
%    f.eSignBit   the exponent sign bit (0 if exponent nonnegative, 1 otherwise)
%    f.e          the exponent (-9<=f.e<=9)
%
% If x is outside of [-9.99*10^9,9.99*10^9], f.m is set to inf.
% If x is in the range (-1.00*10^-9,1.00*10^-9) f is the representation of zero
% in which both sign bits are 0, e is zero, and m = [0 0 0].
```

Thus, `f = Represent(-237000)` is equivalent to

```
f = struct('mSignBit',1,'m',[2 3 7],'eSignBit',0,'e',6)
```

Complementing `Represent` is the following function, which can take a three-digit representation and compute its value:

```
  function x = Convert(f)
% x = Convert(f)
% f is a is a representation of a 3-digit floating point number.
% x is the value of f.

% Overflow situations
if (f.m == inf) & (f.mSignBit==0)
   x = inf;
   return
end
if (f.m == inf) & (f.mSignBit==1)
   x = -inf;
   return
end
```

```
% Mantissa value
mValue = (100*f.m(1) + 10*f.m(2) + f.m(3))/100;
if f.mSignBit==1
   mValue = -mValue;
end

% Exponent value
eValue = f.e;
if f.eSignBit==1
   eValue = -eValue;
end

x = mValue * 10^eValue;
```

To simulate three-digit floating point arithmetic, we convert the operands to conventional form, do the arithmetic, and then represent the result in 3-digit form. The following function implements this approach:

```
  function z = Float(x,y,op)
% z = Float(x,y,op)
% x and y are representations of a 3-digit floating point number.
% op is one of the strings '+', '-', '*', or '/'.
% z is the 3-digit floating point representation of x op y.

sx = num2str(convert(x));
sy = num2str(convert(y));
z = represent(eval(['(' sx ')' op '(' sy ')' ]));
```

Strings are enclosed in quotes. The conversion of a number to a string is handled by `num2str`. Strings are concatenated by assembling them in square brackets. The `eval` function takes a string for input and returns the value produced when that string is executed.

To "pretty print" the value of a floating point representation, we have

```
  function s = Pretty(f)
% s = Pretty(f)
% f is a representation of a 3-digit floating point number.
% s is a string so that disp(s) "pretty prints" the value of f.
```

As an illustration of how these functions can be used, the script file `Euler` generates the partial sums

$$s_n = 1 + \frac{1}{2} + \cdots + \frac{1}{n}.$$

In exact arithmetic the s_n tend toward ∞, but when we run

```
% Script File: Euler
% Sums the series 1 + 1/2 + 1/3 + .. in 3-digit floating point arithmetic.
% Terminates when the addition of the next term does not change
% the value of the running sum.

oldsum = Represent(0);
one = Represent(1);
sum = one;
k = 1;
while Convert(sum) ~= Convert(oldsum)
   k = k+1;
   kay  = Represent(k);
   term = Float(one,kay,'/');
   oldsum = sum;
   sum  = Float(sum,term,'+');
end
clc
disp(['The sum for ' num2str(k) ' or more terms is ' pretty(sum)])
```

the loop terminates after 200 terms.

1.6.2 Padé Approximants

A very useful class of approximants for the exponential function e^z are the Padé functions defined by

$$R_{pq}(z) \;=\; \left(\sum_{k=0}^{p} \frac{(p+q-k)!p!}{(p+q)!k!(p-k)!} z^k\right) \Bigg/ \left(\sum_{k=0}^{q} \frac{(p+q-k)!q!}{(p+q)!k!(q-k)!} (-z)^k\right).$$

Assuming the availability of

```
  function R = PadeCoeff(p,q)
% R = PadeCoeff(p,q)
% p and q are nonnegative integers and R is a representation of the
% (p,q)-Pade approximation N(x)/D(x) to exp(x):
%
%     R.num  is a row (p+1)-vector whose entries are the coefficients of the
%            p-degree numerator polynomial N(x).
%
%     R.den  is a row (q+1)-vector whose entries are the coefficients of the
%            q-degree denominator polynomial D(x).
%
% Thus,
%                  R.num(1) + R.num(2)x + R.num(3)x^2
%                 ------------------------------------
%                          R.den(1) + R.den(2)x
%
% is the (2,1) Pade approximation.
```

the following function returns a cell array whose entries specify a particular Padé approximation:

```
  function P = PadeArray(m,n)
% P = PadeArray(m,n)
% m and n are nonnegative integers.
% P is an (m+1)-by-(n+1) cell array.
%
% P{i,j} represents the (i-1,j-1) Pade approximation N(x)/D(x) to exp(x).

P = cell(m+1,n+1);
for i=1:m+1
   for j=1:n+1
      P{i,j} = PadeCoeff(i-1,j-1);
   end
end
```

Problems

P1.6.1 Write a function `s = dot3(x,y)` that returns the 3-digit representation of the inner product `x'*y` where `x` and `y` are column vectors of the same length. The inner product should be computed using 3-digit arithmetic. (Make effective use of `represent`, `convert`, and `float`.) The error can be computed via the command `err = x'*y - convert(dot3(x,y))`. Write a script that plots a histogram of the error when `dot3` is applied to 100 random `x'*y` problems of length 5. Use `randn(5,1)` to generate the x and y vectors. Report the results in a histogram with 20 bins.

P1.6.2 Use `PadeArray` to generate representations of the Padé approximants R_{pq} for $0 \le p \le 3$ and $0 \le q \le 3$. Plot the relative error of R_{11}, R_{22} and R_{33} across the interval [-5 5]. Use `semilogy` for the plots.

P1.6.3 The Chebychev polynomials are defined by

$$T_k(x) = \begin{cases} 1 & k = 0 \\ x & k = 1 \\ 2xT_{k-1}(x) - T_{k-2}(x) & k \ge 2 \end{cases}.$$

Write a function `T = ChebyCoeff(n)` that returns an n-by-1 cell array whose ith cell is a length-i array. The elements of the array are the coefficients of T_{i-1}. Thus `T{3} = [-1 0 2]` since $T_2(x) = 2x^2 - 1$.

1.7 More Refined Graphics

Plots can be embellished so that they carry more information and have a more pleasing appearance. In this section we show how to set font, incorporate subscripts and superscripts, and use mathematical and Greek symbols in displayed strings. We also discuss the careful placement of text in a figure window and how to modify what the axes "say". Line thickness and color are also treated.

Because refined graphics is best learned through experimentation, our presentation is basically by example. Formal syntactic definitions are avoided. The reader is encouraged to play with the scripts provided.

1.7.1 Fonts

A font has a name, a size, and a style. Figure 1.18 shows some of the possibilities associated with the Times-Roman font. The script `ShowFonts` displays similar tableaus for the AvantGarde, Bookman, Courier, Helvetica, Helvetica-Narrow, NewCenturySchlbk, Palatino, and Zapfchancery fonts. Here are some sample `text` commands where non-default fonts are used:

```
text(x,y,'Matlab','FontName','Times-Roman','FontSize',12)
text(x,y,'Matlab','FontName','Helvetica','FontSize',12,'FontWeight','bold')
text(x,y,'Matlab','FontName','ZapfChancery','FontSize',12,'FontAngle','oblique')
```

The fonts can also be set when using `title`, `xlabel`, and `ylabel`, e.g.,

```
title('Important Title','FontName','Helvetica','FontSize',18,'FontWeight','bold')
```

1.7.2 Mathematical Typesetting

It is possible to specify subscripts, superscripts, Greek letters, and various mathematical symbols in the strings that are passed to `title`, `xlabel`, `ylabel`, and `text`. For example,

```
title('{\itf}_{1}({\itx}) = sin(2\pi{\itx}){\ite}^{-2{\it\alphax}}')
```

creates a title of the form $\sin(2\pi x)e^{-2\alpha x}$. conventions are followed. "Special characters" are specified with

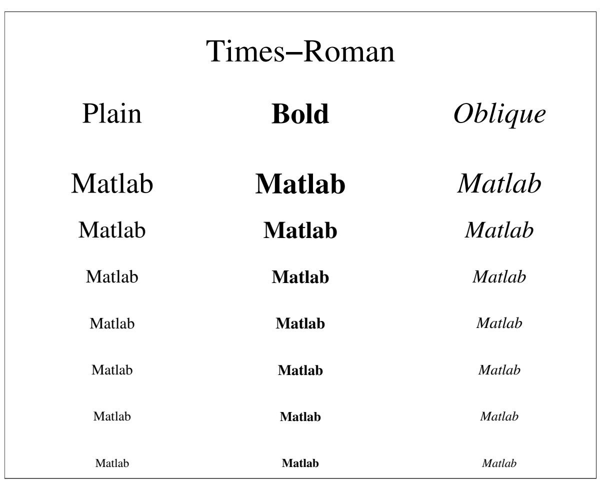


FIGURE 1.18 *Fonts*

a \ prefix and some of the possibilities are given in Figures 1.19 and 1.20. In this setting, curly brackets are used to determine scope. The underscore and caret are used for subscripts and superscripts. It is customary to italicize mathematical expressions, except that numbers and certain function names should remain in plain font. To do this use `\it`.

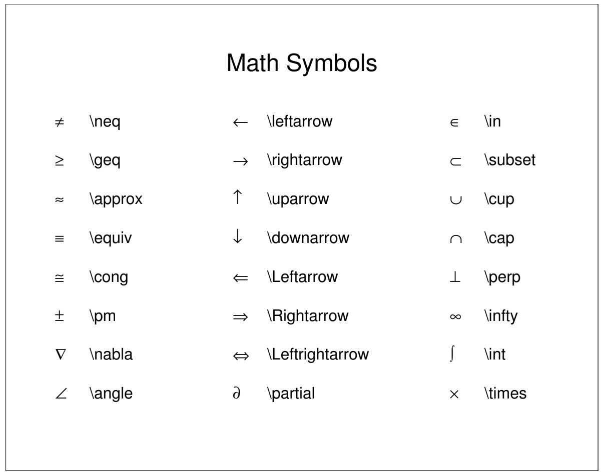


FIGURE 1.19 *Math symbols*

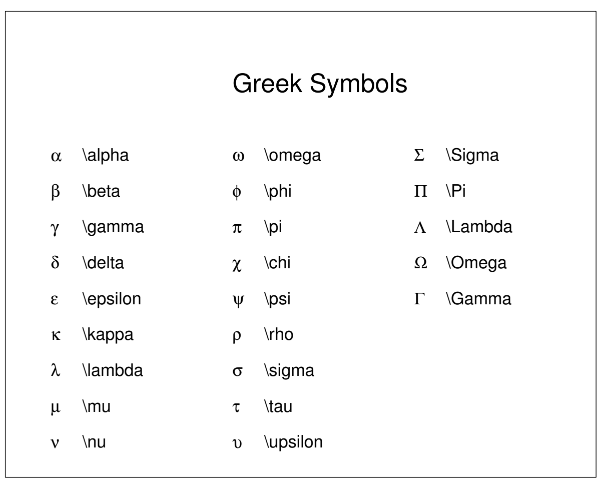


FIGURE 1.20 *Greek symbols*

1.7.3 Text Placement

The accurate placement of labels in a figure window is simplified by using `HorizontalAlignment` and `VerticalAlignment` with suitable modifiers. With its vertices encoded in a pair of length-6 arrays `x` and `y`,

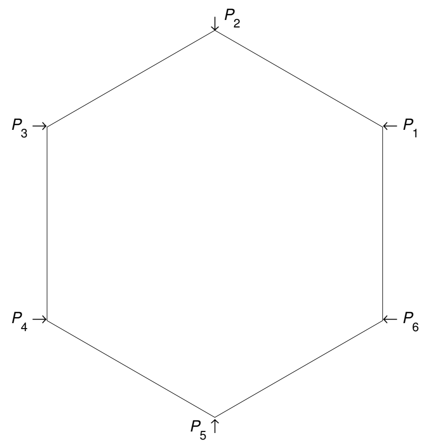


FIGURE 1.21 *Text placements*

the labeled hexagon in Figure 1.21 is produced with the following fragment:

```
HA = 'HorizontalAlignment'; VA = 'VerticalAlignment';
text(x(1),y(1),'\leftarrow {\itP}_{1}',  HA,'left')
text(x(2),y(2),'\downarrow',             HA,'center', VA,'baseline')
text(x(2),y(2),'{  \itP}_{2}',           HA,'left',   VA,'bottom')
text(x(3),y(3),'{\itP}_{3} \rightarrow', HA,'right')
text(x(4),y(4),'{\itP}_{4} \rightarrow', HA,'right')
text(x(5),y(5),'\uparrow',               HA,'center', VA,'top')
text(x(5),y(5),'{\itP}_{5}  ',           HA,'right',  VA,'top')
text(x(6),y(6),'\leftarrow {\itP}_{6}',  HA,'left')
```

1.7.4 Line Width and Axes

It is possible to modify the thickness of the lines that are drawn by `plot`. The fragment

```
h = plot(x,y);
set(h,'LineWidth',3)
```

plots y versus x with the line width attribute set to 3. The effect of various line width settings is shown in Figure 1.22. It is also possible to regulate the font used by `xlabel`, `ylabel`, and `title` and to control the "tick mark" placement along these axes. See Figure 1.23 which is produced by the following script:

```
F = 'Times-Roman'; n = 12; t = linspace(0,2*pi); c = cos(t); s = sin(t);
plot(c,s), axis([-1.3 1.3,-1.3 1.3]), axis equal
title('The Circle ({\itx-a})^{2} + ({\ity-b})^{2} = {\itr}^{2}',...
      'FontName',F,'FontSize',n)
xlabel('x','FontName',F,'FontSize',n)
ylabel('y','FontName',F,'FontSize',n)
set(gca,'XTick',[-.5 0 .5])
set(gca,'YTick',[-.5 0 .5])
grid on
```

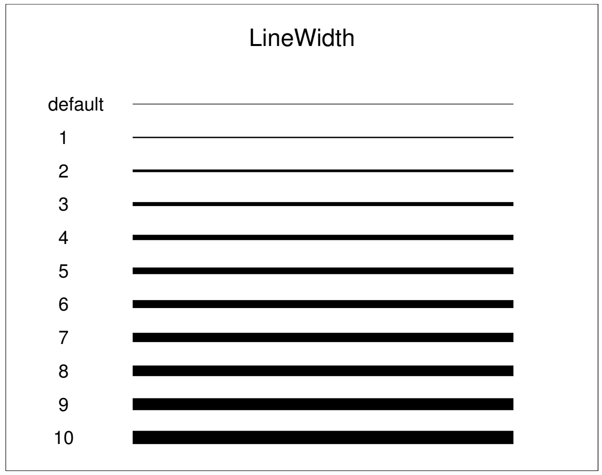


FIGURE 1.22 *Line width*

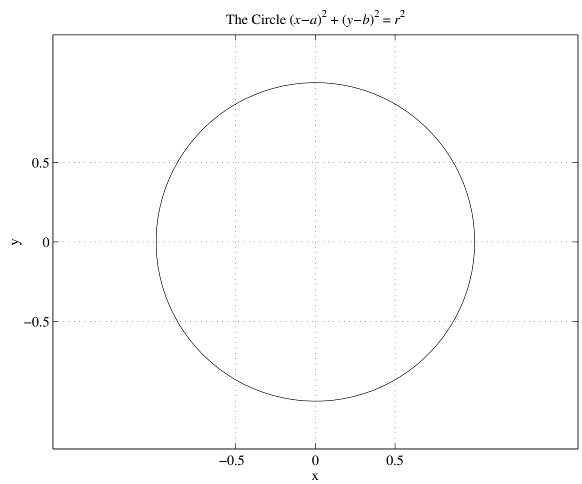


FIGURE 1.23 *Axis design*

We mention that `grid` is a toggle and when it is on, the grid lines associated with the prescribed axis ticks are displayed. All tick marks can be suppressed by using the empty matrix, e.g., `set(gca,'XTick',[])`.

1.7.5 Legends

It is sometimes useful to have a legend in plots that display more than one function. Figure 1.24 is produced by the following script:

```
t = linspace(0,2);
axis([0 2 -1.5 1.5])
y1 = sin(t*pi); y2 = cos(t*pi);
plot(t,y1,t,y2,[0 .5 1 1.5 2],[0 0 0 0 0],'o')
set(gca,'XTick',[]), set(gca,'YTick',[0]), grid on
legend('sin(\pi t)','cos(\pi t)','roots',0)
```

The integer provided to `legend` is used to specify position: 0 = least conflict with data, 1 = upper right-hand corner (default), 2 = upper left-hand corner, 3 = lower left-hand corner, 4 = lower right-hand corner, and -1 = to the right of the plot.

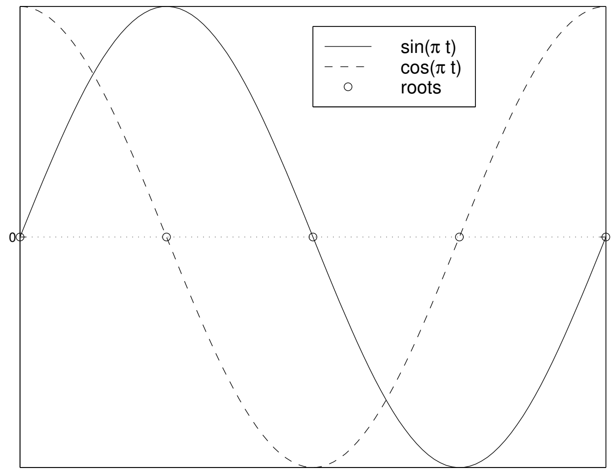


FIGURE 1.24 *Legend placement*

1.7.6 Color

MATLAB comes with 8 predefined colors:

rgb	[0 0 0]	[0 0 1]	[0 1 0]	[0 1 1]	[1 0 0]	[1 0 1]	[1 1 0]	[1 1 1]
color	white	blue	green	cyan	red	magenta	yellow	black
mnemonic	w	b	g	c	r	m	y	k

The "rgb triple" is a 3-vector whose components specify the amount of red, green and blue. The rgb values must be in between 0 and 1. (See Figure 1.25.) To specify that a particular line be drawn with a predefined color, just include its mnemonic in the relevant line type string. Here are some examples:

```
plot(x,y,'g')
plot(x,y,'*g')
plot(x1,y1,'r',x2,y2,'.g',x3,y3,'k.-')
```

The `fill` function can be used to draw filled polygons with a specified color. If `x` and `y` are length-n vectors then

```
fill(x,y,'m')
```

draws a magenta polygon whose vertices are (x_i, y_i), $i = 1{:}n$. "User-defined" colors can also be passed to `fill`,

```
fill(x,y,[.3,.8,.4])
```

It is also possible draw several filled polygons at once:

```
fill(x1,y1,'g',x2,y2,[.3,.8,.4])
```

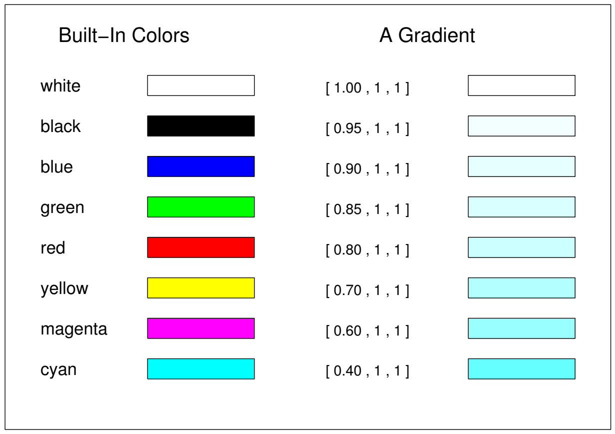


FIGURE 1.25 *Color*

See the script `ShowColor` for more details.

Problems

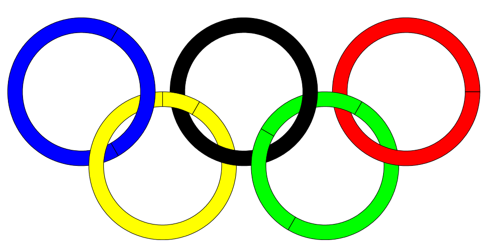
P1.7.1 Complete the following MATLAB function so that it performs as specified:

```
  function arch(a,b,theta1,theta2,r1,r2,ring_color)
%
% Adds an arch with center (a,b), inner radius r1, and outer radius r2 to the current figure.
% The arch is the set of all points of the form (a+r*cos(theta),b+r*sin(theta)) where
% r1 <= r <= r2 and theta1 <= theta <= theta2 where theta1 and theta2 in radians.
% The color of the displayed arch is prescribed by ring_color, a 3-vector encoding the rgb triple.
```

Write a function `OlympicRings(r,n,ring_colors)` with the property that the script

```
close all
ring_colors = [0 0 1 ; 1 1 0 ; 1 1 1 ; 0 1 0 ; 1 0 0];
OlympicRings(1,5,ring_colors)
axis off equal
```

produces the following output (in black and white):



In a call to `OlympicRings`, `r` is the outer radius of each ring and `n` is the number of rings. Index the rings left to right from 0 to $n-1$. The parameter `ring_colors` is an n-by-3 matrix whose $k+1$st row specifies the color of the kth ring. The inner radius of each ring is $.85r$. The center (a_k, b_k) of the kth ring is given by $(1.15rk, 0)$ if k is even and by $(1.15rk, -r)$ if k is odd.

Notice that the rings are interlocking. Thus, to get the right "over-and-under" appearance you cannot simply superimpose the drawing of the 5 rings. You'll have to split up the drawing of each ring into sections and the small little cross lines you see in the above figure are a hint.

M-Files and References

Script Files

`SineTable`	Prints a short table of sine evaluations.
`SinePlot`	Displays a sequence of sin(x) plots.
`ExpPlot`	Plots exp(x) and an approximation to exp(x).
`TangentPlot`	Plots tan(x).
`SineAndCosPlot`	Superimposes plots of sin(x) and cos(x).
`Polygons`	Displays nine regular polygons, one per window.
`SumOfSines`	Displays the sum of four sine functions.
`SumOfSines2`	Displays a pair of sum-of-sine functions.
`UpDown`	Sample core exploratory environment.
`RunUpDown`	Framework for running `UpDown`.
`Histograms`	Displays the distribution of rand and randn.
`Clouds`	Displays 2-dimensional rand and randn.
`Dice`	Histogram of 1000 dice rolls.
`Darts`	Monte Carlo computation of pi.
`Smooth`	Polygon smoothing.
`Stirling`	Relative and absolute error in Stirling formula.
`ExpTaylor`	Plots relative error in Taylor approximation to exp(x).
`Zoom`	Roundoff in the expansion of (x-1)^6.
`FpFacts`	Examines precision, overflow, and underflow.
`TestMyExp`	Examines `MyExp1`, `MyExp2`, `MyExp3`, and `MyExp4`.
`Euler`	Three-digit arithmetic sum of 1 + 1/2 +...+ 1/n.
`ShowPadeArray`	Tests the function `PadeArray`.
`ShowFonts`	Illustrates how to use fonts.
`ShowSymbols`	Shows how to generate math symbols.
`ShowGreek`	Shows how to generate Greek letters.
`ShowText`	Shows how to align with `text`.
`ShowLineWidth`	Shows how vary line width in a plot.
`ShowAxes`	Shows how to set tick marks on axes.
`ShowLegend`	Shows how to add a legend to a plot.
`ShowColor`	Shows how to use built-in colors and user-defined colors.

Function Files

`MyExpF`	For-loop Taylor approximation to exp(x).
`MyExp1`	Vectorized version of MyExpF.
`MyExp2`	Better vectorized version of MyExpF.
`MyExpW`	While-loop Taylor approximation to exp(x).
`MyExp3`	Vectorized version of MyExpW.
`MyExp4`	Better vectorized version of MyExpW.
`Derivative`	Numerical differentiation.
`Represent`	Sets up 3-digit arithmetic representation.
`Convert`	Converts 3-digit representation to float.
`Float`	Simulates 3-digit arithmetic.
`Pretty`	Pretty prints a 3-digit representation.
`PadeArray`	Builds a cell array of Pade coefficients.