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

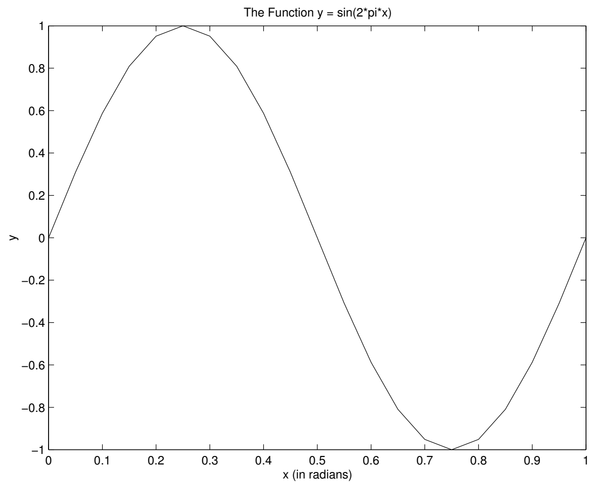


FIGURE 1.1 *A crude plot of* $\sin(2\pi x)$

"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

`disp(`⟨*string*⟩`)`

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

`sprintf(`⟨*String with Format Specifications*⟩,⟨*List-of-Variables*⟩`)`

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')
```

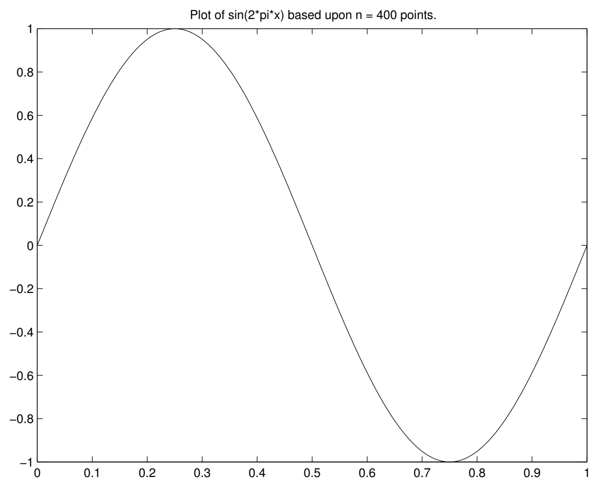


FIGURE 1.4 *A smooth plot of* $\sin(2\pi x)$

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

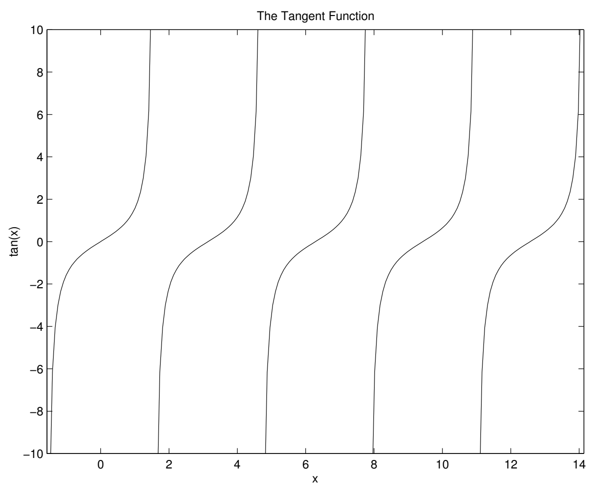


FIGURE 1.5 *A plot of* $\tan(x)$

```
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` $\leq x \leq$ `xmax` and a y-axis range `ymin` $\leq y \leq$ `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)],'*')
```

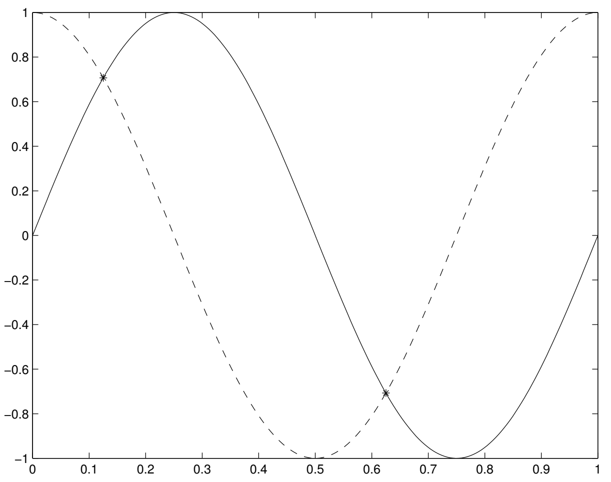


FIGURE 1.6 *Superpositioning*

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.

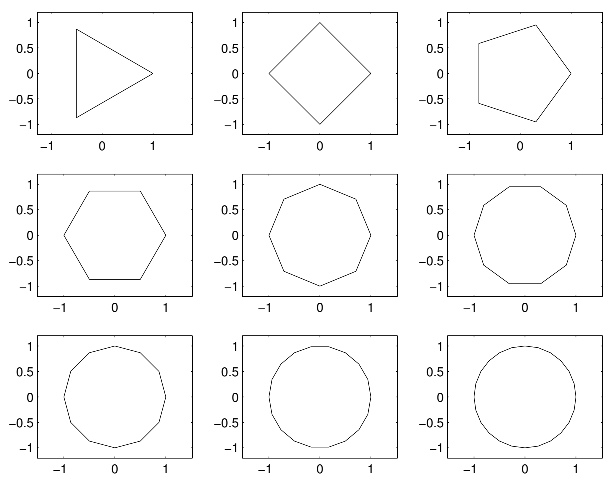


FIGURE 1.7 *Regular polygons*

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)')
```

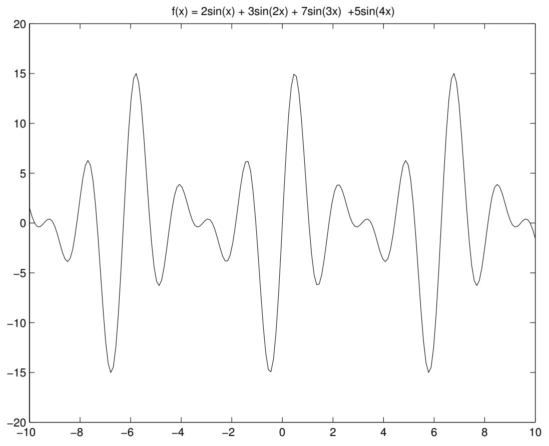


FIGURE 1.8 *A sum of sines*

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

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

That is,

$$\left[\begin{array}{cccc}\sin(1) & \sin(2) & \sin(3) & \sin(4)\\ \sin(2) & \sin(4) & \sin(6) & \sin(8)\\ \sin(3) & \sin(6) & \sin(9) & \sin(12)\\ \sin(4) & \sin(8) & \sin(12) & \sin(16)\end{array}\right]\left[\begin{array}{c}\alpha_1\\ \alpha_2\\ \alpha_3\\ \alpha_4\end{array}\right] = \left[\begin{array}{r}-2\\ 0\\ 1\\ 5\end{array}\right].$$

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

$$\alpha = \left[\begin{array}{r}-0.2914\\ -8.8454\\ -18.8706\\ -11.8279\end{array}\right].$$

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

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

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)))
```

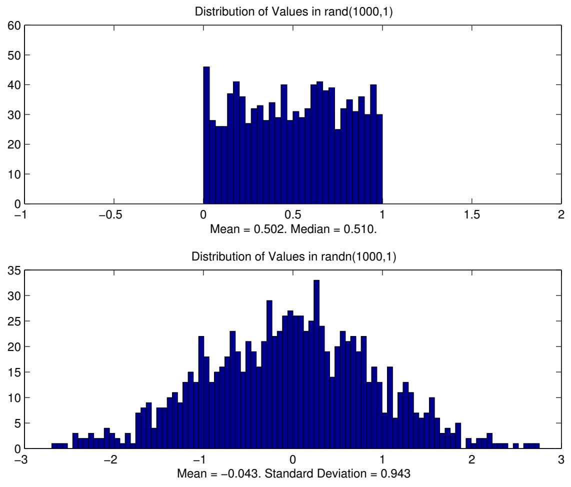


FIGURE 1.9 *The uniform and normal distributions*

(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.')
```

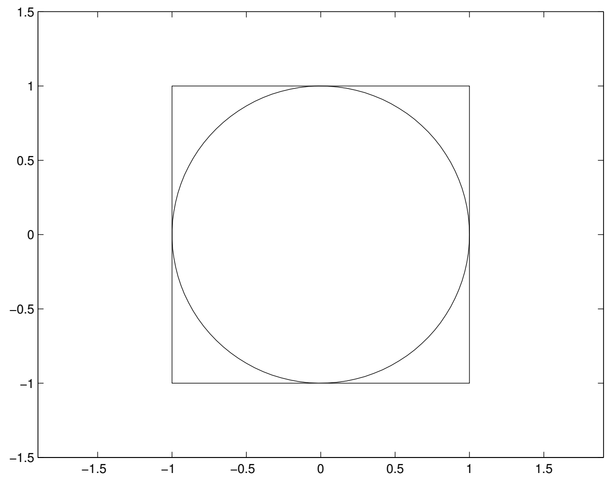


FIGURE 1.10 *A target*

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:

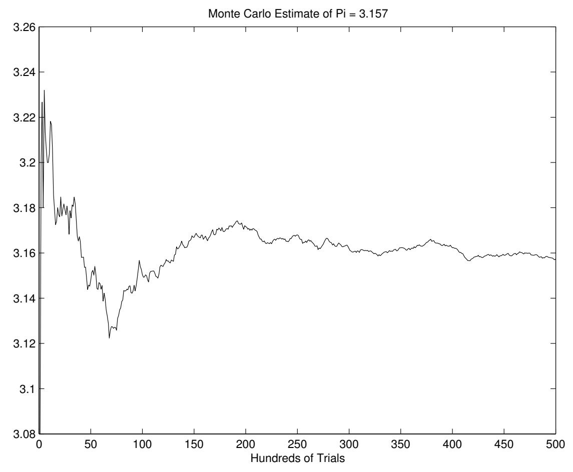


FIGURE 1.11 *A Monte Carlo estimate of π*

```
% 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

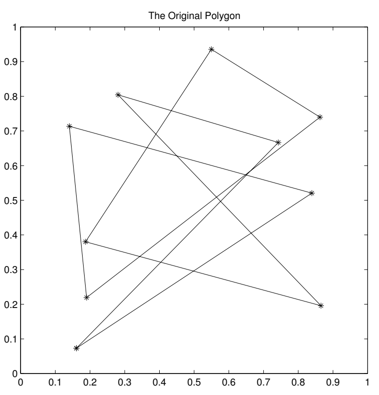


FIGURE 1.12 *The initial polygon*

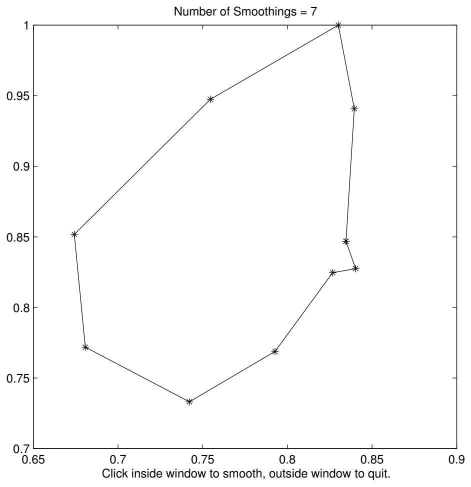


FIGURE 1.13 *A smoothed polygon*

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 $\{(x_1, x_2, x_3, x_4) : x_1^2+x_2^2+x_3^2+x_4^2 \leq 1\}$, the unit sphere in 4-dimensional space.

P1.3.4 Let $S = \{\ (x, y) \mid -1 \leq x \leq 1,\ -1 \leq y \leq 1\}$. 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
```

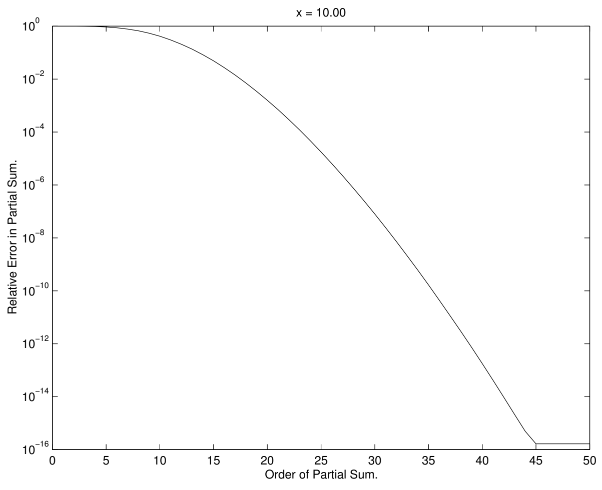


FIGURE 1.14 *Error in Taylor approximations to* $e^x, x = 10$

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

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

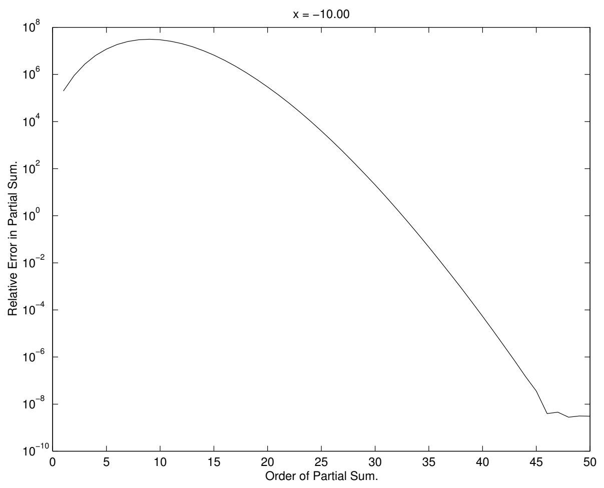


FIGURE 1.15 *Error in Taylor approximations to $e^x, x = -10$*

```
% 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:

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

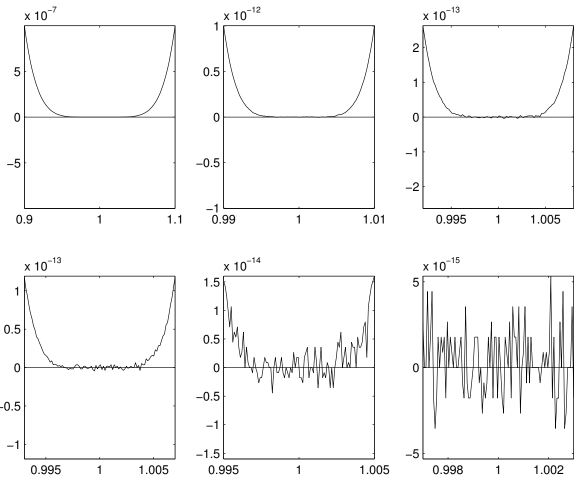


FIGURE 1.16 *Plots of* $(x-1)^6 = x^6 - 6x^5 + 15x^4 - 20x^3 + 15x^2 - 6x + 1$ *near* $x = 1$

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.

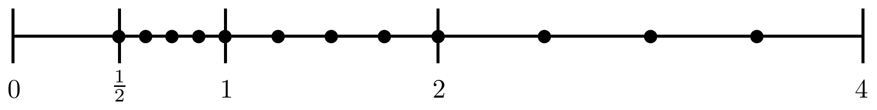
We denote the set of floating point numbers by $\mathbf{F}(t, L, U)$. To emphasize the finiteness of this set, suppose $t = 2$, $L = -1$ and $U = +1$. There are twelve positive floating point numbers:

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

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* $x \in \mathrm{I\!R}$ *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

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

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

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

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 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

$$\begin{pmatrix} n \\ k \end{pmatrix} = \frac{n!}{k!(n-k)!}.$$

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
```

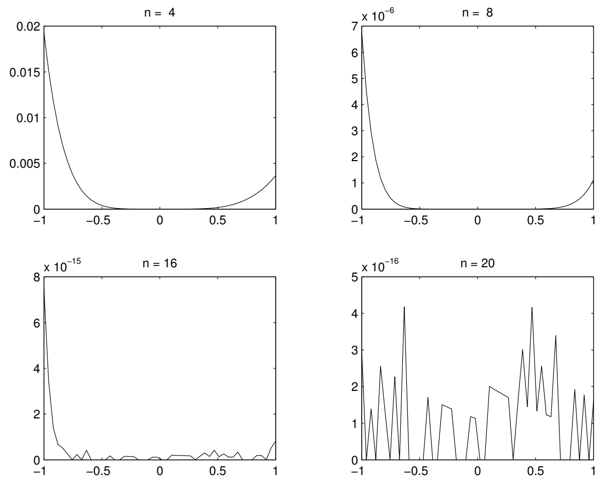


FIGURE 1.17 *Relative error in $T_n(x)$*

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

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

if

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

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

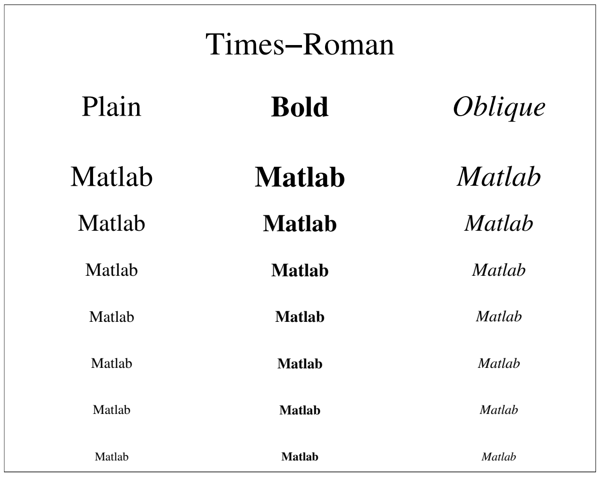
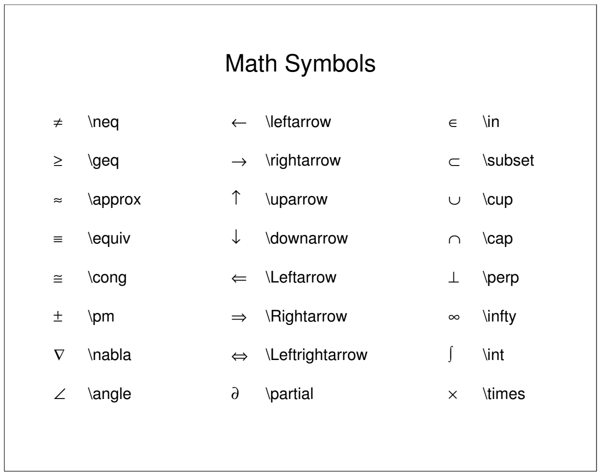


FIGURE 1.18 *Fonts*

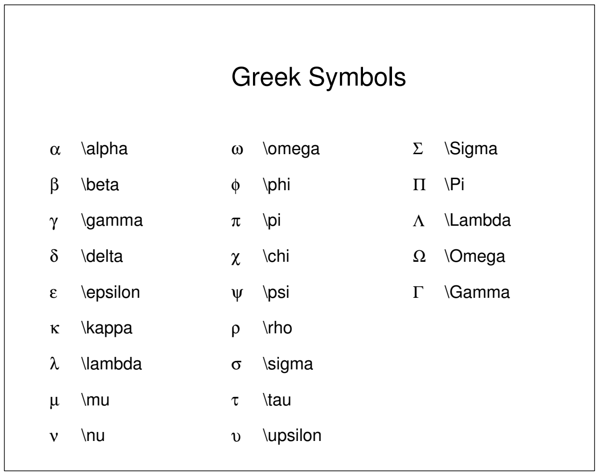
a \ prefix and some of the possibilities are given in Figures 1.19 and 1.20. In this setting, curly brackets are used to determine scope. The underscore and caret are used for subscripts and superscripts. It is customary to italicize mathematical expressions, except that numbers and certain function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



Math Symbols

≠	\neq	←	\leftarrow	∈	\in
≥	\geq	→	\rightarrow	⊂	\subset
≈	\approx	↑	\uparrow	∪	\cup
≡	\equiv	↓	\downarrow	∩	\cap
≅	\cong	⇐	\Leftarrow	⊥	\perp
±	\pm	⇒	\Rightarrow	∞	\infty
∇	\nabla	⇔	\Leftrightarrow	∫	\int
∠	\angle	∂	\partial	×	\times

FIGURE 1.19 *Math symbols*



Greek Symbols

α	\alpha	ω	\omega	Σ	\Sigma
β	\beta	ϕ	\phi	Π	\Pi
γ	\gamma	π	\pi	Λ	\Lambda
δ	\delta	χ	\chi	Ω	\Omega
ε	\epsilon	ψ	\psi	Γ	\Gamma
κ	\kappa	ρ	\rho		
λ	\lambda	σ	\sigma		
μ	\mu	τ	\tau		
ν	\nu	υ	\upsilon		

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`,

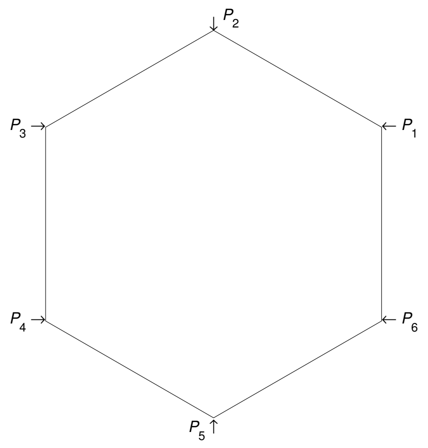


FIGURE 1.21 *Text placements*

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
```

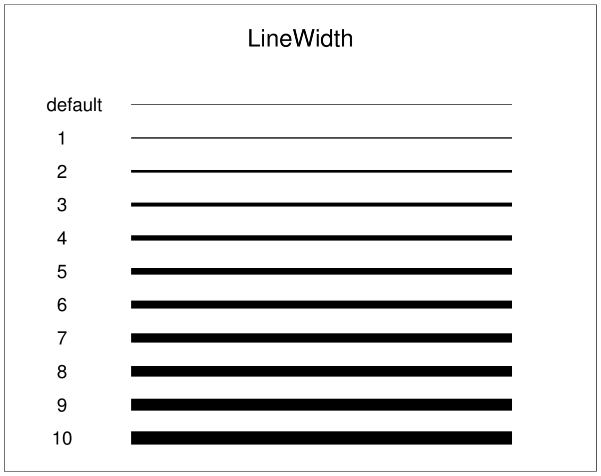


FIGURE 1.22 *Line width*

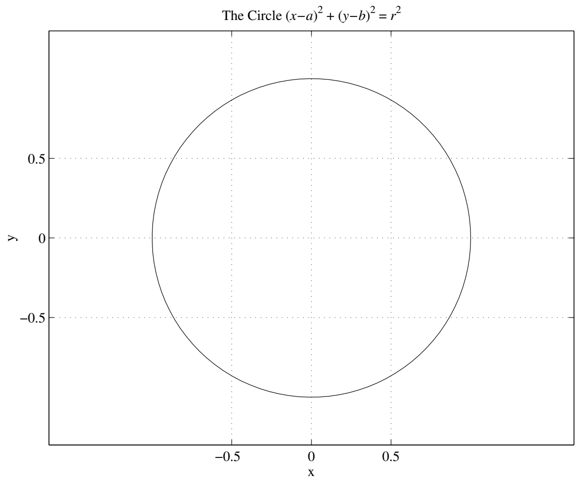


FIGURE 1.23 *Axis design*

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.

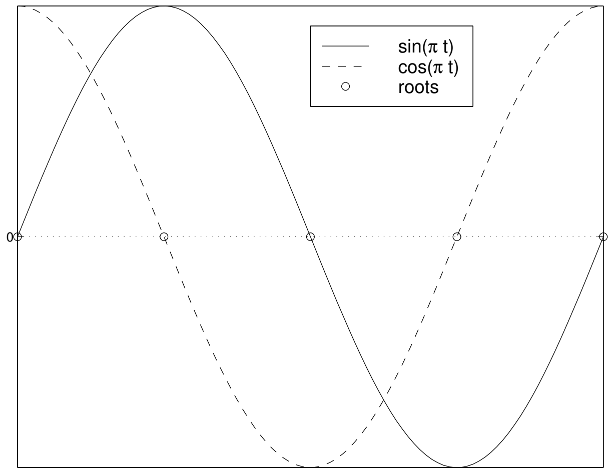


FIGURE 1.24 *Legend placement*

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])
```

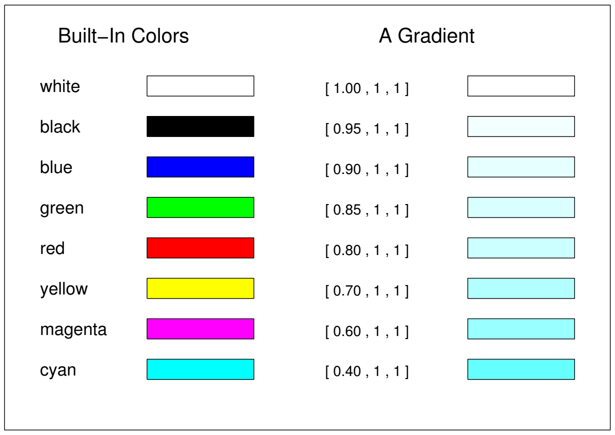


FIGURE 1.25 *Color*

See the script `ShowColor` for more details.

Problems

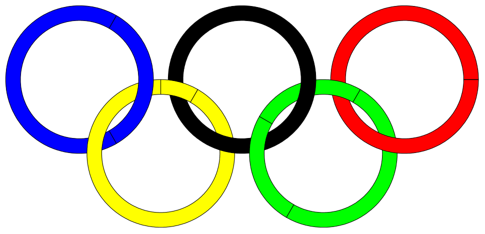
P1.7.1 Complete the following MATLAB function so that it performs as specified:

```
  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.

Chapter 2

Polynomial Interpolation

In the problem of *data approximation*, we are given some points $(x_1, y_1), \ldots, (x_n, y_n)$ and are asked to find a function $\phi(x)$ that "captures the trend" of the data. If the trend is one of decay, then we may seek a ϕ of the form $a_1 e^{-\lambda_1 x} + a_2 e^{-\lambda_2 x}$. If the trend of the data is oscillatory, then a trigonometric approximant might be appropriate. Other settings may require a low-degree polynomial. Regardless of the type of function used, there are many different metrics for success, e.g., least squares.

A special form of the approximation problem ensues if we insist that ϕ actually "goes through" the data, as shown in Figure 2.1. This means that $\phi(x_i) = y_i$, $i = 1{:}n$ and we say that ϕ *interpolates* the data. The polynomial interpolation problem is particularly important:

Given $x_1, \ldots, x_n$ (distinct) and $y_1, \ldots, y_n$, find a polynomial $p_{n-1}(x)$ of degree $n-1$ (or less) such that $p_{n-1}(x_i) = y_i$ for $i = 1{:}n$.

Thus, $p_2(x) = 1 + 4x - 2x^2$ interpolates the points $(-2, -15)$, $(3, -5)$, and $(1, 3)$.

Each (x_i, y_i) pair can be regarded as a snapshot of some function $f(x)$: $y_i = f(x_i)$. The function f may be explicitly available, as when we want to interpolate $\sin(x)$ at $x = 0, \pi/2$, and π with a quadratic. On other occasions, f is implicitly defined, as when we want to interpolate the solution to a differential equation at a discrete number of points.

The discussion of polynomial interpolation revolves around how it can be represented, computed, and evaluated:

- How do we *represent* the interpolant $p_{n-1}(x)$? Instead of expressing the interpolant in terms of the "usual" basis polynomials 1, x, and x^2, we could use the alternative basis 1, $(x+2)$, and $(x+2)(x-3)$. Thus,
$$p_2(x) = -15 + 2(x+2) - 2(x+2)(x-3)$$
is another way to express the quadratic interpolant of the data $(-2, -15)$, $(3, -5)$, and $(1, 3)$. Different bases have different computational virtues.

- Once we have settled on a representation for the polynomial interpolant, how do we determine the associated coefficients? It turns out that this aspect of the problem involves the solution of a linear system of equations with a highly structured coefficient matrix.

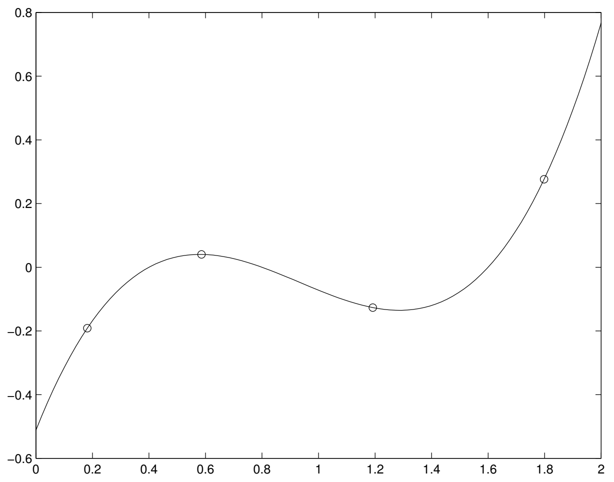


FIGURE 2.1 *The interpolation of four data points with a cubic polynomial*

- After we have computed the coefficients, how can the interpolant be evaluated with efficiency? For example, if the interpolant is to be plotted then we are led to the problem of evaluating a polynomial on a vector of values.

In MATLAB these issues can be handled by `polyfit` and `polyval`. The script

```
x = [-2 3 1];
y = [-15 -5 3];
a = polyfit(x,y,2)
xvals = linspace(-3,2,100);
pvals = polyval(a,xvals);
plot(xvals,pvals)
```

plots the polynomial interpolant of the data $(-2,-15)$, $(3,-5)$, and $(1,3)$. The interpolant is given by $p(x) = 1 + 4x - 2x^2$ and the call to `polyfit` computes a representation of this polynomial. In particular, `a` is assigned the vector `[-2 4 1]`.

In general, if `x` and `y` are n-vectors, then `a = polyfit(x,y,n-1)` assigns a length-n vector to `a` with the property that the polynomial

$$p(x) = a_n + a_{n-1}x + a_{n-2}x^2 + \cdots + a_1x^{n-1}$$

interpolates the data $(x_1, y_1), \ldots, (x_n, y_n)$.

The function `polyval` is used to evaluate polynomials in the MATLAB representation. In the above script `polyval(a,xvals)` is a vector of interpolant evaluations.

In this chapter we start with what we call the "Vandermonde" approach to the polynomial interpolation problem. The Newton representation is considered in §2.2 and accuracy issues in §2.3. Divided differences, inverse interpolation, interpolation in the plane, and trigonmetric interpolation are briefly discussed in §2.4.

2.1 The Vandermonde Approach

In the Vandermonde approach, the interpolant is expressed as a linear combination of 1, x, x^2, etc. Although monomials are not the best choice for a basis, our familiarity with this way of "doing business" with polynomials makes them a good choice to initiate the discussion.

2.1.1 A Four-point Interpolation Problem

Let us find a cubic polynomial

$$p_3(x) = a_1 + a_2x + a_3x^2 + a_4x^3$$

that interpolates the four data points $(-2, 10)$, $(-1, 4)$, $(1, 6)$, and $(2, 3)$. Note that this is the "reverse" of MATLAB 's convention for representing polynomials. [1] Each point of interpolation leads to a linear equation that relates the four unknowns a_1, a_2, a_3, and a_4:

$$\begin{array}{lclcrcrcrcrcr} p_3(-2) = 10 & \Rightarrow & a_1 & - & 2a_2 & + & 4a_3 & - & 8a_4 & = & 10 \\ p_3(-1) = 4 & \Rightarrow & a_1 & - & a_2 & + & a_3 & - & a_4 & = & 4 \\ p_3(1) \;\;= 6 & \Rightarrow & a_1 & + & a_2 & + & a_3 & + & a_4 & = & 6 \\ p_3(2) \;\;= 3 & \Rightarrow & a_1 & + & 2a_2 & + & 4a_3 & + & 8a_4 & = & 3 \end{array}$$

Expressing these four equations in matrix/vector terms gives

$$\begin{bmatrix} 1 & -2 & 4 & -8 \\ 1 & -1 & 1 & -1 \\ 1 & 1 & 1 & 1 \\ 1 & 2 & 4 & 8 \end{bmatrix} \begin{bmatrix} a_1 \\ a_2 \\ a_3 \\ a_4 \end{bmatrix} = \begin{bmatrix} 10 \\ 4 \\ 6 \\ 3 \end{bmatrix}.$$

The solution $a = [4.5000\ 1.9167\ 0.5000\ \ -0.9167]^T$ to this 4-by-4 system can be found as follows:

```
y = [10; 4; 6; 3];
V = [1 -2 4 -8; 1 -1 1 -1; 1 1 1 1; 1 2 4 8];
a = V\y;
```

2.1.2 The General n Case

From this example, it looks like the polynomial interpolation problem reduces to a linear equation problem. For general n, the goal is to determine $a_1, \ldots, a_n$ so that if

$$p_{n-1}(x) = a_1 + a_2x + a_3x^2 + \cdots + a_nx^{n-1},$$

then

$$p_{n-1}(x_i) \;=\; a_1 + a_2x_i + a_3x_i^2 + \cdots + a_nx_i^{n-1} \;=\; y_i$$

for $i = 1{:}n$. By writing these equations in matrix-vector form, we obtain

$$\begin{bmatrix} 1 & x_1 & x_1^2 & \cdots & x_1^{n-1} \\ 1 & x_2 & x_2^2 & \cdots & x_2^{n-1} \\ 1 & x_3 & x_3^2 & \cdots & x_3^{n-1} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & x_n & x_n^2 & \cdots & x_n^{n-1} \end{bmatrix} \begin{bmatrix} a_1 \\ a_2 \\ a_3 \\ \vdots \\ a_n \end{bmatrix} = \begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ \vdots \\ y_n \end{bmatrix}.$$

Designate the matrix of coefficients by V. The solvability of the interpolation problem hinges on the nonsingularity of V. Suppose there is a vector c such that $Vc = 0$. It follows that the polynomial

$$q(x) = c_1 + c_2x + \cdots + c_nx^{n-1}$$

is zero at $x = x_1, \ldots, x = x_n$. This says that we have a degree $n - 1$ polynomial with n roots. The only way that this can happen is if q is the zero polynomial (i.e., $c = 0$). Thus V is nonsingular because the only vector that it zeros is the zero vector.

[1] MATLAB would represent the sought-after cubic as $p_3 = a_4 + a_3x + a_2x^2 + a_1x^3$. Our chosen style is closer to what one would find in a typical math book: $p_3(x) = a_0 + a_1x + a_2x^2 + a_3x^3$.

2.1.3 Setting Up and Solving the System

Let us examine the construction of the Vandermonde matrix V. Our first method is based on the observation that the ith row of V involves powers of x_i and that the powers increase from 0 to $n-1$ as the row is traversed from left to right. A conventional double-loop approach gives

```
n = length(x); V = zeros(n,n);
for i=1:n
   % Set up row i.
   for j=1:n
      V(i,j) = x(i)^(j-1);
   end
end
```

Algorithms that operate on a two-dimensional array in row-by-row fashion are *row oriented.*

The inner-loop in the preceding script can be *vectorized* because MATLAB supports pointwise exponentiation. For example, `u = [1 2 3 4] .^[3 5 2 3]` assigns to `u` the row vector [1 32 9 64]. The i-th row of V requires exponentiating the scalar x_i to each of the values in the row vector $0{:}n-1 = (0,\ 1,\ \ldots,\ n-1)$. Thus, `row = (x(i)*ones(1,n)).^(0:n-1)` assigns the vector $(1, x_i, x_i^2, \ldots, x_i^{n-1})$ to `row`, precisely the values that make up the ith row of V. The ith row of a matrix V may be referenced by `V(i,:)`, and so we obtain

```
n = length(x); V = zeros(n,n);
for i=1:n
   % Set up the i-th row of V.
   V(i,:)  = (x(i)*ones(1,n)).^(0:n-1);
end
```

By reversing the order of the loops in the original set-up script, we obtain a *column oriented* algorithm:

```
n = length(x); V = zeros(n,n);
for j=1:n
   % Set up column j.
   for i=1:n
      V(i,j) = x(i)^(j-1);
   end
end
```

If $j > 1$, then $V(i,j)$ is the product of $x(i)$ and $V(i,j-1)$, the matrix entry to its left. This suggests that the required exponentiations can be obtained through repeated multiplication:

```
n = length(x);
V = ones(n,n);
for j=2:n
   % Set up column j.
   for i=1:n
      V(i,j) = x(i)*V(i,j-1)
   end
end
```

The generation of the jth column involves *pointwise vector multiplication*:

$$\begin{bmatrix} x_1 \\ \vdots \\ x_n \end{bmatrix} .* \begin{bmatrix} v_{1,j-1} \\ \vdots \\ v_{n,j-1} \end{bmatrix} = \begin{bmatrix} v_{1,j} \\ \vdots \\ v_{n,j} \end{bmatrix}.$$

This may be implemented by `V(:,j) = x .* V(:,j-1)`. Basing our final implementation on this, we obtain

```
  function a = InterpV(x,y)
% a = InterpV(x,y)
% This computes the Vandermonde polynomial interpolant where
% x is a column n-vector with distinct components and y is a
% column n-vector.
%
% a is a column n-vector with the property that if
%
%        p(x) = a(1) + a(2)x + ... a(n)x^(n-1)
% then
%        p(x(i)) = y(i), i=1:n

n = length(x);
V = ones(n,n);
for j=2:n
   % Set up column j.
   V(:,j) = x.*V(:,j-1);
end
a = V\y;
```

Column-oriented, matrix-vector implementations will generally be favored in this text. One reason for doing this is simply to harmonize with the traditions of linear algebra, which is usually taught with a column-oriented perspective.

2.1.4 Nested Multiplication

We now consider the evaluation of $p_{n-1}(x) = a_1 + \cdots + a_n x^{n-1}$ at $x = z$, assuming that `z` and `a(1:n)` are available. The routine approach

```
n = length(a);
zpower = 1;
pVal = a(1);
for i=2:n
   zpower = z*zpower;
   pVal = pVal + a(i)*zpower;
end
```

assigns the value of $p_{n-1}(z)$ to `pVal`.

A more efficient algorithm is based on a nested organization of the polynomial, which we illustrate for the case $n = 4$:

$$p_3(x) = a_1 + a_2x + a_3x^2 + a_4x^3 = ((a_4x + a_3)x + a_2)x + a_1.$$

Note that the fragment

```
pVal = a(4);
pVal = z*pVal + a(3);
pVal = z*pVal + a(2);
pVal = z*pVal + a(1);
```

assigns the value of $p_3(z)$ to `pVal`. For general n, this nested multiplication idea takes on the following form:

```
n = length(a);
pVal = a(n);
for i=n-1:-1:1
   pVal = z*pVal + a(i);
end
```

This is widely known as *Horner's rule.*

Before we encapsulate the Horner idea in a MATLAB function, let us examine the case when the interpolant is to be evaluated at many different points. To be precise, suppose `z(1:m)` is initialized and that for $i = 1{:}m$, we want to assign the value of $p_{n-1}(z(i))$ to `pVal(i)`. One obvious approach is merely to repeat the preceding Horner iteration at each point. Instead, we develop a vectorized implementation that can be obtained if we think about the "simultaneous" evaluation of the interpolants at each z_i. Suppose $m = 5$ and $n = 4$ (i.e, the case when a cubic interpolant is to be evaluated at five different points). The first step in the five applications of the Horner idea may be summarized as follows:

$$\begin{bmatrix} \texttt{pVal(1)} \\ \texttt{pVal(2)} \\ \texttt{pVal(3)} \\ \texttt{pVal(4)} \\ \texttt{pVal(5)} \end{bmatrix} = \begin{bmatrix} \texttt{a(4)} \\ \texttt{a(4)} \\ \texttt{a(4)} \\ \texttt{a(4)} \\ \texttt{a(4)} \end{bmatrix}.$$

In vector terms `pVal = a(n)*ones(m,1)`. The next step requires a multiply-add of the following form:

$$\begin{bmatrix} \texttt{pVal(1)} \\ \texttt{pVal(2)} \\ \texttt{pVal(3)} \\ \texttt{pVal(4)} \\ \texttt{pVal(5)} \end{bmatrix} = \begin{bmatrix} \texttt{z(1)*pVal(1)} \\ \texttt{z(2)*pVal(2)} \\ \texttt{z(3)*pVal(3)} \\ \texttt{z(4)*pVal(4)} \\ \texttt{z(5)*pVal(5)} \end{bmatrix} + \begin{bmatrix} \texttt{a(3)} \\ \texttt{a(3)} \\ \texttt{a(3)} \\ \texttt{a(3)} \\ \texttt{a(3)} \end{bmatrix}.$$

That is,

```
pVal = z.*pVal + a(3)
```

The pattern is clear for the cubic case:

```
pVal = a(4)*ones(m,1);
pVal = z .* pVal + a(3);
pVal = z .* pVal + a(2);
pVal = z .* pVal + a(1);
```

From this we generalize to the following:

```
  function pVal = HornerV(a,z)
% pVal = HornerV(a,z)
% evaluates the Vandermonde interpolant on z where
% a is an n-vector and z is an m-vector.
%
% pVal is a vector the same size as z with the property that if
%
%         p(x) = a(1) + .. +a(n)x^(n-1)
% then
%         pVal(i) = p(z(i))  ,  i=1:m.

n = length(a);
m = length(z);
pVal = a(n)*ones(size(z));
for k=n-1:-1:1
   pVal = z.*pVal + a(k);
end
```

Each update of `pval` requires $2m$ flops so approximately $2mn$ flops are required in total.

As an application, here is a script that displays cubic interpolants of $\sin(x)$ on $[0, 2\pi]$. The abscissas are chosen randomly.

```
% Script File: ShowV
% Plots 4 random cubic interpolants of sin(x) on [0,2pi].
% Uses the Vandermonde method.

close all
x0 = linspace(0,2*pi,100)';
y0 = sin(x0);
for eg=1:4
   x = 2*pi*sort(rand(4,1));
   y = sin(x);
   a = InterpV(x,y);
   pVal = HornerV(a,x0);
   subplot(2,2,eg)
   plot(x0,y0,x0,pVal,'--',x,y,'*')
   axis([0 2*pi -2 2])
end
```

Figure 2.2 displays a sample output.

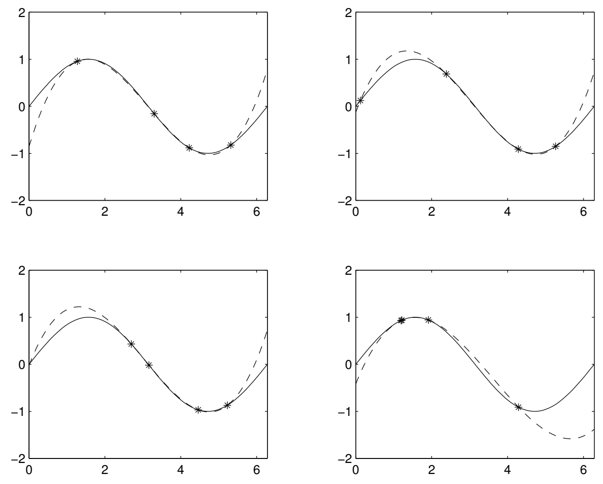


FIGURE 2.2 *Random cubic interpolants of* $\sin(x)$ *on* $[0,2\pi]$

Problems

P2.1.1 Instead of expressing the polynomial interpolant in terms of the basis functions $1, x, \ldots, x^{n-1}$, we can work with the alternative representation

$$p_{n-1}(x) = \sum_{k=1}^{n} a_k \left(\frac{x-u}{v}\right)^{k-1}.$$

Here u and v are scalars that serve to shift and scale the x-range. Generalize `InterpV` so that it can be called with either two, three, or four arguments. A call of the form `a = InterpV(x,y)` should assume that $u = 0$ and $v = 1$. A call of the form `a = InterpV(x,y,u)` should assume that $v = 1$ and that `u` houses the shift factor. A call of the form `a = InterpV(x,y,u,v)` should assume that `u` and `v` house the shift and scale factors, respectively.

P2.1.2 A polynomial of the form

$$p(x) = a_1 + a_2x^2 + \cdots + a_mx^{2m-2}$$

is said to be *even*, while a polynomial of the form

$$p(x) = a_1x + a_3x^3 + \cdots + a_mx^{2m-1}$$

is said to be *odd*. Generalize `HornerV(a,z)` so that it has an optional third argument `type` that indicates whether or not the underlying polynomial is even or odd. In particular, a call of the form `HornerV(a,z,'even')` should assume that a_k is the coefficient of x^{2k-2}. A call of the form `HornerV(a,z,'odd')` should assume that a_k is the coefficient of x^{2k-1}.

P2.1.3 Assume that `z` and `a(1:n)` are initialized and define

$$p(x) = a_1 + a_2x + \cdots + a_nx^{n-1}.$$

Write a script that evaluates (1) $p(z)/p(-z)$, (2) $p(z)+p(-z)$, (3) $p'(z)$, (4) $\int_0^1 p(x)dx$, and (e) $\int_{-z}^{z} p(x)dx$. Make effective use of `HornerV`.

P2.1.4 (a) Assume that `L` (scalar), `R` (scalar), and `c(1:4)` are given. Write a script that computes `a(1:4)` so that if $p(x) = a_1 + a_2x + a_3x^2 + a_4x^3$, then $p(L) = c_1$, $p'(L) = c_2$, $p''(L) = c_3$, and $p(R) = c_4$. Use \ to solve any linear system that arises in your method. (b) Write a function `a = TwoPtInterp(L,cL,R,cR)` that returns the coefficients of a polynomial $p(x) = a_1 + a_2x + \cdots + a_nx^n$ that satisfies $p^{(k-1)}(L) =$ `cL(k)` for $k = 1$:`length(cL)` and $p^{(k-1)}(R) =$ `cR(k)` for $k = 1$:`length(cR)`. The degree of p should be one less than the total number of end conditions. (The problem of determining a cubic polynomial whose value and slope are prescribed at two points is discussed in detail in §3.2.1. It is referred to as the *cubic Hermite interpolation problem.*)

P2.1.5 Write a function `PlotDerPoly(x,y)` that plots the derivative of the polynomial interpolant of the data (x_i, y_i), $i = 1{:}n$. Assume that $x_1 < \cdots < x_n$ and the plot should be across the interval $[x_1, x_n]$. Use `polyfit` and `polyval`.

2.2 The Newton Representation

We now look at a form of the polynomial interpolant that is generally more useful than the Vandermonde representation.

2.2.1 A Four-Point Example

To motivate the idea, consider once again the problem of interpolating the four points (x_1, y_1), (x_2, y_2), (x_3, y_3), and (x_4, y_4) with a cubic polynomial $p_3(x)$. However, instead of expressing the interpolant in terms of the "canonical" basis 1, x, x^2, and x^3, we use the basis 1, $(x-x_1)$, $(x-x_1)(x-x_2)$, and $(x-x_1)(x-x_2)(x-x_3)$. This means that we are looking for coefficients c_1, c_2, c_3, and c_4 so that if

$$p_3(x) = c_1 + c_2(x-x_1) + c_3(x-x_1)(x-x_2) + c_4(x-x_1)(x-x_2)(x-x_3), \tag{2.1}$$

then $y_i = p_3(x_i) = y_i$ for $i = 1{:}4$. In expanded form, these four equations state that

$$\begin{aligned}
y_1 &= c_1 \\
y_2 &= c_1 + c_2(x_2 - x_1) \\
y_3 &= c_1 + c_2(x_3 - x_1) + c_3(x_3 - x_1)(x_3 - x_2) \\
y_4 &= c_1 + c_2(x_4 - x_1) + c_3(x_4 - x_1)(x_4 - x_2) + c_4(x_4 - x_1)(x_4 - x_2)(x_4 - x_3).
\end{aligned}$$

By rearranging these equations, we obtain the following four-step solution process:

$$\begin{aligned}
c_1 &= y_1 \\
c_2 &= \frac{y_2 - c_1}{x_2 - x_1} \\
c_3 &= \frac{y_3 - (c_1 + c_2(x_3 - x_1))}{(x_3 - x_1)(x_3 - x_2)} \\
c_4 &= \frac{y_4 - (c_1 + c_2(x_4 - x_1) + c_3(x_4 - x_1)(x_4 - x_2))}{(x_4 - x_1)(x_4 - x_2)(x_4 - x_3)}.
\end{aligned}$$

This sequential solution process is made possible by the clever choice of the basis polynomials and the result is the *Newton representation* of the interpolating polynomial.

To set the stage for the general-n algorithm, we redo the $n = 4$ case using matrix-vector notation to discover a number of simplifications. The starting point is the system of equations that we obtained previously which can be expressed in the following form:

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 1 & (x_2-x_1) & 0 & 0 \\ 1 & (x_3-x_1) & (x_3-x_1)(x_3-x_2) & 0 \\ 1 & (x_4-x_1) & (x_4-x_1)(x_4-x_2) & (x_4-x_1)(x_4-x_2)(x_4-x_3) \end{bmatrix} \begin{bmatrix} c_1 \\ c_2 \\ c_3 \\ c_4 \end{bmatrix} = \begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ y_4 \end{bmatrix}.$$

From this we see immediately that $c_1 = y_1$. We can eliminate c_1 from equations 2, 3, and 4 by subtracting equation 1 from equations 2, 3, and 4:

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & (x_2-x_1) & 0 & 0 \\ 0 & (x_3-x_1) & (x_3-x_1)(x_3-x_2) & 0 \\ 0 & (x_4-x_1) & (x_4-x_1)(x_4-x_2) & (x_4-x_1)(x_4-x_2)(x_4-x_3) \end{bmatrix} \begin{bmatrix} c_1 \\ c_2 \\ c_3 \\ c_4 \end{bmatrix} = \begin{bmatrix} y_1 \\ y_2-y_1 \\ y_3-y_1 \\ y_4-y_1 \end{bmatrix}.$$

If we divide equations 2, 3, and 4 by $(x_2 - x_1)$, $(x_3 - x_1)$, and $(x_4 - x_1)$, respectively, then the system transforms to

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 1 & (x_3-x_2) & 0 \\ 0 & 1 & (x_4-x_2) & (x_4-x_2)(x_4-x_3) \end{bmatrix} \begin{bmatrix} c_1 \\ c_2 \\ c_3 \\ c_4 \end{bmatrix} = \begin{bmatrix} y_1 \\ y_{21} \\ y_{31} \\ y_{41} \end{bmatrix},$$

where y_{21}, y_{31}, and y_{41} are defined by

$$y_{21} = \frac{y_2-y_1}{x_2-x_1} \qquad y_{31} = \frac{y_3-y_1}{x_3-x_1} \qquad y_{41} = \frac{y_4-y_1}{x_4-x_1}.$$

Notice that

$$\begin{bmatrix} y_{21} \\ y_{31} \\ y_{41} \end{bmatrix} = \left(\begin{bmatrix} y_2 \\ y_3 \\ y_4 \end{bmatrix} - \begin{bmatrix} y_1 \\ y_1 \\ y_1 \end{bmatrix} \right) ./ \left(\begin{bmatrix} x_2 \\ x_3 \\ x_4 \end{bmatrix} - \begin{bmatrix} x_1 \\ x_1 \\ x_1 \end{bmatrix} \right) = (y(2{:}4) - y(1))./(x(2{:}4) - x(1)).$$

The key point is that we have reduced the size of problem by one. The remaining unknowns satisfy a 3-by-3 system:

$$\begin{bmatrix} 1 & 0 & 0 \\ 1 & (x_3-x_2) & 0 \\ 1 & (x_4-x_2) & (x_4-x_2)(x_4-x_3) \end{bmatrix} \begin{bmatrix} c_2 \\ c_3 \\ c_4 \end{bmatrix} = \begin{bmatrix} y_{21} \\ y_{31} \\ y_{41} \end{bmatrix}.$$

This is exactly the system obtained were we to seek the coefficients of the quadratic

$$q(x) = c_2 + c_3(x - x_2) + c_4(x - x_2)(x - x_3)$$

that interpolates the data (x_2, y_{21}), (x_3, y_{31}), and (x_4, y_{41}).

2.2.2 The General n Case

For general n, we see that if $c_1 = y_1$ and

$$q(x) = c_2 + c_3(x - x_2) + \cdots + c_n(x - x_2)\cdots(x - x_{n-1})$$

interpolates the data

$$\left(x_i, \frac{y_i - y_1}{x_i - x_1}\right) \qquad i = 2{:}n,$$

then

$$p(x) = c_1 + (x - x_1)q(x)$$

interpolates $(x_1, y_1), \ldots, (x_n, y_n)$. This is easy to verify. Indeed, for $j = 1{:}n$

$$p(x_j) = c_1 + (x_j - x_1)q(x_j) = y_1 + (x_j - x_1)\frac{y_j - y_1}{x_j - x_1} = y_j.$$

This sets the stage for a recursive formulation of the whole process:

```
  function c = InterpNRecur(x,y)
% c = InterpNRecur(x,y)
% The Newton polynomial interpolant.
% x is a column n-vector with distinct components and y is
% a column n-vector. c is  a column n-vector with the property that if
%
%      p(x) = c(1) + c(2)(x-x(1))+...+ c(n)(x-x(1))...(x-x(n-1))
% then
%      p(x(i)) = y(i), i=1:n.

n = length(x); c = zeros(n,1); c(1) = y(1);
if n > 1
   c(2:n) = InterpNRecur(x(2:n),(y(2:n)-y(1))./(x(2:n)-x(1)));
end
```

If $n = 1$, then the constant interpolant $p(x) \equiv y_1$ is returned (i.e., $c_1 = y_1$.) Otherwise, the final c-vector is a "stacking" of y_1 and the solution to the reduced problem. The recursive call obtains the coefficients of the interpolant $q(x)$ mentioned earlier.

To develop a nonrecursive implementation, we return to our four-point example and the equation

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 1 & (x_3 - x_2) & 0 \\ 0 & 1 & (x_4 - x_2) & (x_4 - x_2)(x_4 - x_3) \end{bmatrix} \begin{bmatrix} c_1 \\ c_2 \\ c_3 \\ c_4 \end{bmatrix} = \begin{bmatrix} y_1 \\ y_{21} \\ y_{31} \\ y_{41} \end{bmatrix}.$$

From this we see that $c_2 = y_{21}$. Now subtract equation 2 from equation 3 and divide by $(x_3 - x_2)$. Next, subtract equation 2 from equation 4 and divide by $(x_4 - x_2)$. With these operations we obtain

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & (x_4 - x_3) \end{bmatrix} \begin{bmatrix} c_1 \\ c_2 \\ c_3 \\ c_4 \end{bmatrix} = \begin{bmatrix} y_1 \\ y_{21} \\ y_{321} \\ y_{421} \end{bmatrix},$$

where

$$y_{321} = \frac{y_{31} - y_{21}}{x_3 - x_2} \qquad y_{421} = \frac{y_{41} - y_{21}}{x_4 - x_2}.$$

At this point we see that $c_3 = y_{321}$. Finally, by subtracting the third equation from the fourth equation and dividing by $(x_4 - x_3)$, we obtain

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} c_1 \\ c_2 \\ c_3 \\ c_4 \end{bmatrix} = \begin{bmatrix} y_1 \\ y_{21} \\ y_{321} \\ y_{4321} \end{bmatrix},$$

where

$$y_{4321} = \frac{y_{421} - y_{321}}{x_4 - x_3}.$$

Clearly, $c_4 = y_{4321}$. The pattern for the general n case should be apparent:

```
for k=1:n-1
    $c_k = y_k$
    for $j = k+1{:}n$
        Subtract equation $k$ from equation $j$ and divide the result by $(x_j - x_k)$.
    end
end
$c_n = y_n$
```

However, when updating the equations *we need only keep track of the changes in the y-vector.* For example,

$$\begin{aligned} y(k+1{:}n) &\leftarrow \left(\begin{bmatrix} y_{k+1} \\ \vdots \\ y_n \end{bmatrix} - \begin{bmatrix} y_k \\ \vdots \\ y_k \end{bmatrix} \right) ./ \left(\begin{bmatrix} x_{k+1} \\ \vdots \\ x_n \end{bmatrix} - \begin{bmatrix} x_k \\ \vdots \\ x_k \end{bmatrix} \right) \\ &= (y(k+1{:}n) - y(k)) \;./\; (x(k+1) - x(k)). \end{aligned}$$

This leads to

```
  function c = InterpN(x,y)
% c = InterpN(x,y)
% The Newton polynomial interpolant.
% x is a column n-vector with distinct components and y is
% a column n-vector. c is a column n-vector with the property that if
%
%     p(x) = c(1) + c(2)(x-x(1))+...+ c(n)(x-x(1))...(x-x(n-1))
% then
%     p(x(i)) = y(i), i=1:n.
n = length(x);
for k = 1:n-1
   y(k+1:n) = (y(k+1:n)-y(k)) ./ (x(k+1:n) - x(k));
end
c = y;
```

2.2.3 Nested Multiplication

As with the Vandermonde representation, the Newton representation permits an efficient nested multiplication scheme. For example, to evaluate $p_3(x)$ at $x = z$, we have the nesting

$$p_3(x) = ((c_4(x - x_3) + c_3)(x - x_2) + c_2)(x - x_1) + c_1.$$

The fragment

```
pVal = c(4);
pVal = (z-x(3))*pVal + c(3);
pVal = (z-x(2))*pVal + c(2);
pVal = (z-x(1))*pVal + c(1);
```

assigns the value of $p_3(z)$ to `pVal`. If `z` is a vector, then this becomes

```
pVal = c(4)*ones(size(z));
pVal = (z-x(3)).*pVal + c(3);
pVal = (z-x(2)).*pVal + c(2);
pVal = (z-x(1)).*pVal + c(1);
```

In general, we have

```
    function pVal = HornerN(c,x,z)
  % pVal = HornerN(c,x,z)
  % Evaluates the Newton interpolant on z where c and x are n-vectors, z is an
  % m-vector, and pVal is a vector the same size as z with the property that if
  %
  %         p(x) = c(1) +  c(2)(x-x(1))+ ... + c(n)(x-x(1))...(x-x(n-1))
  % then
  %         pVal(i) = p(z(i)) , i=1:m.

  n = length(c);
  pVal = c(n)*ones(size(z));
  for k=n-1:-1:1
     pVal = (z-x(k)).*pVal + c(k);
  end
```

The script `ShowN` illustrates `HornerN` and `InterpN`.

Problems

P2.2.1 Write a MATLAB function `a = N2V(c,x)`, where `c` is a column n-vector, `x` is a column $(n-1)$-vector and `a` is a column n-vector, so that if

$$p(x) = c_1 + c_2(x-x_1) + \cdots + c_n(x-x_1)(x-x_2)\cdots(x-x_{n-1}),$$

then

$$p(x) = a_1 + a_2 x + \cdots + a_n x^{n-1}.$$

In other words, `N2V` converts from the Newton representation to the Vandermonde representation.

P2.2.2 Suppose we are given the data (x_i, y_i), $i = 1{:}n$. Assume that the x_i are distinct and that $n \geq 2$. Let $p_L(x)$ and $p_R(x)$ be degree $n-2$ polynomials that satisfy

$$\begin{aligned} p_L(x_i) &= y_i \qquad i = 1{:}n-1 \\[2ex] p_R(x_i) &= y_i \qquad i = 2{:}n. \end{aligned}$$

Note that if

$$p(x) = \frac{(x-x_n)p_L(x) - (x-x_1)p_R(x)}{x_1 - x_n},$$

then $p(x_i) = y_i$, $i = 1{:}n$. In other words, $p(x)$ is the unique degree $n-1$ interpolant of (x_i, y_i), $i = 1{:}n$. Using this result, complete the following function:

```
    function pVal = RecurEval(x,y,z);
  %
  % x is column n-vector with distinct entries, y is a column n-vector, and z is
  % a column m-vector.
  %
  % pVal is a column m-vector with the property that pVal(i) = p(z(i))
  % where p(x) is the degree n-1 polynomial interpolant of (x(i),y(i)), i=1:n.
```

The implementation should be recursive and vectorized. No loops are necessary! Use `RecurEval` to produce an interpolant of $\sin(2\pi x)$ at $x = 0{:}.25{:}1$.

P2.2.3 Write a MATLAB script that solicits the name of a built-in function (as a string), the left and right limits of an interval $[L, R]$, and a positive integer n and then displays both the function and the $n-1$ degree interpolant of it at `linspace(L,R,n)`.

P2.2.4 Assume that `n`, `z(1:n)`, `L`, `R`, and `a(1:6)` are available. Write an efficient MATLAB script that assigns to `q(i)` the value of the polynomial

$$q(x) = a_1 + a_2(x-L) + a_3(x-L)^2 + a_4(x-L)^3 + a_5(x-L)^3(x-R) + a_6(x-L)^3(x-R)^2$$

at $x = z_i$, $i = 1{:}n$. It doesn't matter whether `q(1:n)` is a row vector or a column vector.

P2.2.5 Write a MATLAB script that plots a closed curve

$$(p_x(t), p_y(t)) \qquad 0 \leq t \leq 1$$

that passes through the points (0,0), (0,3), (4,0). The functions p_x and p_y should be cubic polynomials. Make effective use of `InterpN` and `HornerN`. The plot should be based on one hundred evaluations of p_x and p_y.

2.3 Properties

With two approaches to the polynomial interpolation problem, we have an occasion to assess their relative merits. Speed and accuracy are the main concerns.

2.3.1 Efficiency

One way to talk about the efficiency of a numerical method such as `InterpV` or `InterpN` is to relate the number of required flops to the "length" of the input. For `InterpV`, the amount of required arithmetic grows as the cube of n, the number of interpolation points. We say that `InterpV` is an $O(n^3)$ method meaning that work goes up by a factor of 8 if n is doubled. (An n-by-n linear equation solve requires about $2n^3/3$ flops.) On the other hand, `InterpN` is an $O(n^2)$ method. If we double n then work increases by an approximate factor of 4.

Generally speaking quadratic methods (like `InterpN`) are to be preferred to cubic methods (like `InterpV`) especially for large values of n. However, the "big-oh" predictions are typically not realized in practice for small values of n. Moreover, counting flops does not take into account overheads associated with function calls and memory access. Benchmarking these two methods using `tic` and `toc` would reveal that they are not terribly different for modest n.

So far we have just discussed execution efficiency. Memory efficiency is also important. `InterpV` requires an n-by-n array, while `InterpN` needs just a few n-vectors. In this case we say that `InterpV` is quadratic in memory while `InterpN` is linear in memory.

2.3.2 Accuracy

We know that the polynomial interpolant exists and is unique, but how well does it approximate? The answer to the question depends on the derivatives of the function that is being interpolated.

Theorem 2 *Suppose $p_{n-1}(x)$ interpolates the function $f(x)$ at the distinct points $x_1, \ldots, x_n$. If f is n times continuously differentiable on an interval I containing the x_i, then for any $x \in I$*

$$f(x) = p_{n-1}(x) \;+\; \frac{f^{(n)}(\eta)}{n!}(x-x_1)\cdots(x-x_n)$$

where $a \le \eta \le b$.

Proof For clarity and with not a tremendous loss of generality, we prove the theorem for the $n = 4$ case. Consider the function

$$F(t) = f(t) - p_3(t) - cL(t),$$

where

$$c = \frac{f(x) - p_3(x)}{(x-x_1)(x-x_2)(x-x_3)(x-x_4)}$$

and $L(t) = (t-x_1)(t-x_2)(t-x_3)(t-x_4)$. Note that $F(x) = 0$ and $F(x_i) = 0$ for $i = 1{:}4$. Thus, F has at least five zeros in I. In between these zeros F' has a zero and so F' has at least four zeros in I. Continuing in this way, we conclude that

$$F^{(4)}(t) = f^{(4)}(t) - p_3^{(4)}(t) - cL^{(4)}(t)$$

has at least one zero in I which we designate by η_x. Since p_3 has degree ≤ 3, $p_3^{(4)}(t) \equiv 0$. Since L is a monic polynomial with degree 4, $L_3^{(4)}(t) = 4!$. Thus,

$$0 = F^{(4)}(\eta_x) = f^{(4)}(\eta_x) - p_3^{(4)}(\eta_x) - cL^{(4)}(\eta_x) = f^{(4)}(\eta_x) - c \cdot 4!. \;\square$$

This result shows that the quality of $p_{n-1}(x)$ depends on the size of the nth derivative. If we have a bound on this derivative, then we can compute a bound on the error. To illustrate this point in a practical way, suppose $|f^{(n)}(x)| \le M_n$ for all $x \in [a, b]$. It follows that for any $z \in [a, b]$ we have

$$|f(z) - p_{n-1}(z)| \le \frac{M_n}{n!} \max_{a \le x \le b} |(x-x_1)(x-x_2)\cdots(x-x_n)|.$$

If we base the interpolant on the equally spaced points

$$x_i = a + \left(\frac{b-a}{n-1}\right)(i-1), \qquad i = 1{:}n$$

then, by a simple change of variable,

$$|f(z) - p_{n-1}(z)| \le M_n \left(\frac{b-a}{n-1}\right)^n \max_{0\le s\le n-1} \left|\frac{s(s-1)\cdots(s-n+1)}{n!}\right|.$$

It can be shown that the max is no bigger than $1/(4n)$, from which we conclude that

$$|f(z) - p_{n-1}(z)| \le \frac{M_n}{4n}\left(\frac{b-a}{n-1}\right)^n. \tag{2.2}$$

Thus, if a function has ill-behaved higher derivatives, then the quality of the polynomial interpolants may actually decrease as the degree increases.

A classic example of this is the problem of interpolating the function $f(x) = 1/(1+25x^2)$ across the interval $[-1,1]$. See Figure 2.3. While the interpolant "captures" the trend of the function in the middle part of the interval, it blows up near the endpoints. The script **RungeEg** explores the phenomenon in greater detail.

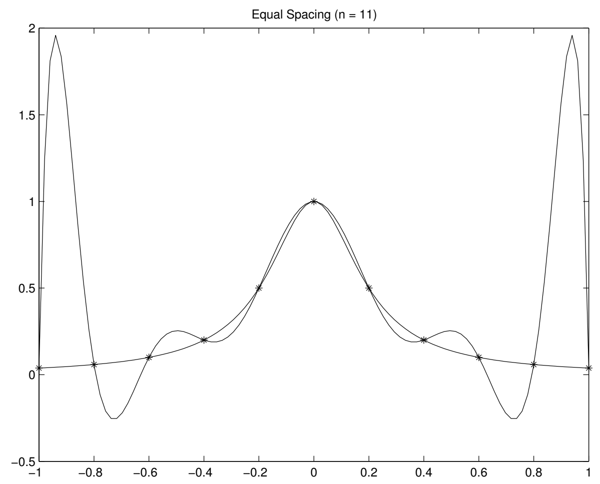


FIGURE 2.3 *The Runge phenomenon*

Problems

P2.3.1 Write a MATLAB script that compares `HornerN` and `HornerV` from the flop point of view.

P2.3.2 Write a MATLAB script that repeatedly solicits an integer n and produces a reasonable plot of the function $e(s) = |s(s-1)\cdots(s-n+1)/n!|$ on the interval $[0, n-1]$. Verify experimentally that this function is never bigger than 1, a fact that we used to establish (2.2).

P2.3.3 Write a MATLAB function `nBest(L,R,a,delta)` that returns an integer n such that if $p_{n-1}(x)$ interpolates e^{ax} at $L+(i-1)(R-L)/(n-1)$, $i = 1{:}n$, then $|p_{n-1}(z) - e^{az}| \le \delta$ for all $z \in [L, R]$. Try to make the value of n as small as you can.

2.4 Special Topics

As a follow-up to the preceding developments, we briefly discuss properties and algorithms associated with divided differences, inverse interpolation, and two-dimensional linear interpolation. We also introduce the important idea of trigonometric interpolation.

2.4.1 Divided Differences

Returning to the $n = 4$ example used in the previous section, we can express c_1, c_2, c_3, and c_4 in terms of the x_i and f:

$$\begin{aligned}
c_1 &= f(x_1) \\
c_2 &= \frac{f(x_2) - f(x_1)}{x_2 - x_1} \\
c_3 &= \frac{\dfrac{f(x_3) - f(x_1)}{x_3 - x_1} - \dfrac{f(x_2) - f(x_1)}{x_2 - x_1}}{x_3 - x_2} \\
c_4 &= \frac{\dfrac{\dfrac{f(x_4) - f(x_1)}{x_4 - x_1} - \dfrac{f(x_2) - f(x_1)}{x_2 - x_1}}{x_4 - x_2} - \dfrac{\dfrac{f(x_3) - f(x_1)}{x_3 - x_1} - \dfrac{f(x_2) - f(x_1)}{x_2 - x_1}}{x_3 - x_2}}{x_4 - x_3}.
\end{aligned}$$

The coefficients are called *divided differences.* To stress the dependence of c_k on f and $x_1, \ldots, x_k$, we write

$$c_k = f[x_1, \ldots, x_k]$$

and refer to this quantity as the $k-1$*st order divided difference.* Thus,

$$p_{n-1}(x) = \sum_{k=1}^{n} f[x_1, \ldots, x_k] \left(\prod_{j=1}^{k-1} (x - x_j) \right)$$

is the n-point polynomial interpolant of f at $x_1, \ldots, x_n$.

We now establish another recursive property that relates the divided differences of f on designated subsets of $\{x_1, \ldots, x_n\}$. Suppose $p_L(x)$ and $p_R(x)$ are the interpolants of f on $\{x_1, \ldots, x_{k-1}\}$ and $\{x_2, \ldots, x_k\}$, respectively. It is easy to confirm that if

$$p(x) = \frac{(x - x_k)p_L(x) - (x - x_1)p_R(x)}{x_1 - x_k}, \tag{2.3}$$

then $p(x_i) = f(x_i)$, $i = 1{:}k$. Thus $p(x)$ is the interpolant of f on $\{x_1, \ldots, x_k\}$ and so

$$p(x) = f[x_1] + f[x_1, x_2](x - x_1) + \cdots + f[x_1, \ldots, x_n](x - x_1) \cdots (x - x_{k-1}). \tag{2.4}$$

Note that since

$$p_L(x) = f[x_1] + f[x_1, x_2](x - x_1) + \cdots + f[x_1, \ldots, x_{k-1}](x - x_1) \cdots (x - x_{k-2}),$$

the coefficient of x^{k-2} is given by $f[x_1, \ldots, x_{k-1}]$. Likewise, since

$$p_R(x) = f[x_2] + f[x_2, x_3](x - x_2) + \cdots + f[x_2, \ldots, x_k](x - x_2) \cdots (x - x_{k-1}),$$

the coefficient of x^{k-2} is given by $f[x_2, \ldots, x_k]$. Comparing the coefficients of x^{k-1} in (2.3) and (2.4), we conclude that

$$f[x_1, \ldots, x_k] = \frac{f[x_2, \ldots, x_k] - f[x_1, \ldots, x_{k-1}]}{x_k - x_1}. \tag{2.5}$$

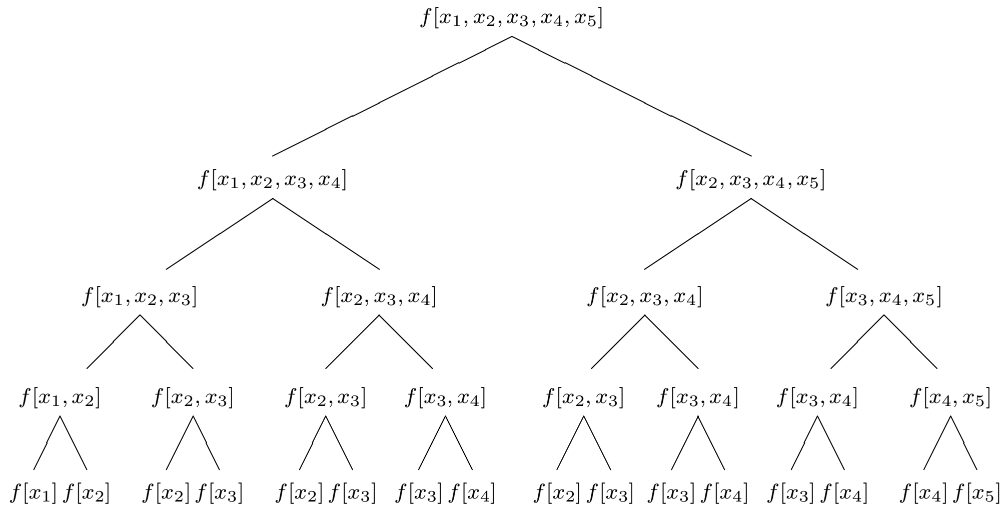


FIGURE 2.4 *Divided differences*

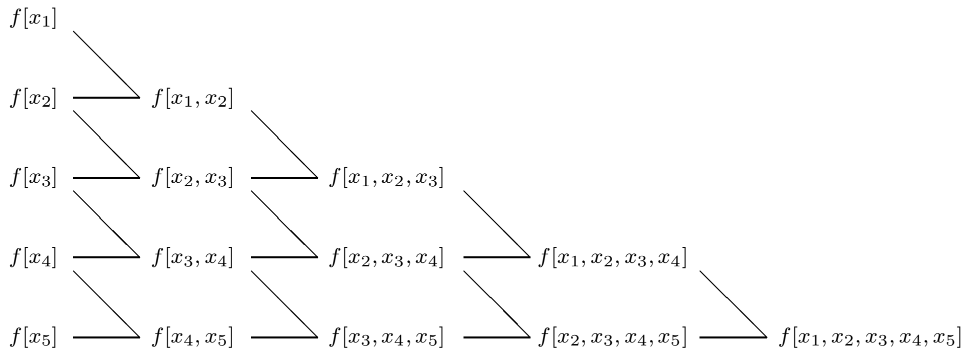


FIGURE 2.5 *Efficient computation of divided differences*

The development of higher-order divided differences from lower order divided differences is illustrated in Figure 2.4. Observe that the sought-after divided differences are along the left edge of the tree. Pruning the excess, we see that the required divided differences can be built up as shown in Figure 2.5. This enables us to rewrite `InterpN` as follows:

```
  function c = InterpN2(x,y)
% c = InterpN2(x,y)
% The Newton polynomial interpolant.
% x is a column n-vector with distinct components and y is
% a column n-vector. c is a column n-vector with the property that if
%      p(x) = c(1) + c(2)(x-x(1))+...+ c(n)(x-x(1))...(x-x(n-1))
% then
%      p(x(i)) = y(i), i=1:n.
n = length(x);
for k = 1:n-1
   y(k+1:n) = (y(k+1:n)-y(k:n-1)) ./ (x(k+1:n) - x(1:n-k));
end
c = y;
```

A number of simplifications result if the x_i are equally spaced. Assume that

$$x_i = x_1 + (i-1)h,$$

where $h > 0$ is the spacing. From (2.5) we see that

$$f[x_1,\ldots,x_k] = \frac{f[x_2,\ldots,x_k] - f[x_1,\ldots,x_{k-1}]}{h(k-1)}.$$

This makes divided difference a scaling of the *differences* $\Delta f[x_1,\ldots,x_k]$, which we define by

$$\Delta f[x_1,\ldots,x_k] \;=\; \begin{cases} f(x_1) & \text{if } k=1 \\ \\ \Delta f[x_2,\ldots,x_k] - \Delta f[x_1,\ldots,x_{k-1}] & \text{if } k>1 \end{cases}.$$

For example,

0th Order	1st Order	2nd Order	3rd Order	4th Order
f_1				
f_2	$f_2 - f_1$			
f_3	$f_3 - f_2$	$f_3 - 2f_2 + f_1$		
f_4	$f_4 - f_3$	$f_4 - 2f_3 + f_2$	$f_4 - 3f_3 + 3f_2 - f_1$	
f_5	$f_5 - f_4$	$f_5 - 2f_4 + f_3$	$f_5 - 3f_4 + 3f_3 - f_2$	$f_5 - 4f_4 + 6f_3 - 4f_2 + f_1$

It is not hard to show that

$$f[x_1,\ldots,x_k] = \frac{\Delta f[x_1,\ldots,x_k]}{h^{k-1}(k-1)!}.$$

The built-in function `diff` can be used to compute differences. In particular, if `y` is an n-vector, then

```
d = diff(y)
```

and

```
d = y(2:n) - y(1:n-1)
```

are equivalent. A second argument can be used to compute higher-order differences. For example,

```
d = diff(y,2)
```

computes the second-order differences:

```
d = y(3:n) - 2*y(2:n-1) + y(1:n-2)
```

Problems

P2.4.1 Compare the computed c_i produced by `InterpN` and `InterpN2`.

P2.4.2 Complete the following MATLAB function:

```
function [c,x,y] = InterpNEqual(fname,L,R,n)
```

Make effective use of the `diff` function.

2.4.2 Inverse Interpolation

Suppose the function $f(x)$ has an inverse on $[a, b]$. This means that there is a function g so that $g(f(x)) = x$ for all $x \in [a, b]$. Thus $g(x) = \sqrt{x}$ is the inverse of $f(x) = x^2$ on $[0, 1]$. If

$$a = x_1 < x_2 < \cdots < x_n = b$$

and $y_i = f(x_i)$, then the polynomial that interpolates the data (y_i, x_i), $i = 1{:}n$ is an interpolate of f's inverse. Thus the script

```
x = linspace(0,1,6)';
y = x.*x;
a  = InterpV(y,x);
yvals = linspace(y(1),y(6));
xvals = HornerV(a,yvals);
plot(yvals,xvals);
```

plots a quintic interpolant of the square root function. This is called *inverse* interpolation, and it has an important application in zero finding. Suppose $f(x)$ is continuous and either monotone increasing or decreasing on $[a, b]$. If $f(a)f(b) < 0$, then f has a zero in $[a, b]$. If $q(y)$ is an inverse interpolant, then $q(0)$ can be thought of as an approximation to this root.

Problems

P2.4.3 Suppose we have three data points (x_1, y_1), (x_2, y_2), and (x_3, y_3) with the property that $x_1 < x_2 < x_3$ and that y_1 and y_3 are opposite in sign. Write a function `root = InverseQ(x,y)` that returns the value of the inverse quadratic interpolant at 0.

2.4.3 Interpolation in Two Dimensions

Suppose $(\tilde{x}, \tilde{y})$ is inside the rectangle

$$R = \{(x, y) : a \leq x \leq b, \qquad c \leq y \leq d\}.$$

Suppose $f(x, y)$ is defined on R and that we have its values on the four corners

$$\begin{aligned} f_{ac} &= f(a, c) \\ f_{bc} &= f(b, c) \\ f_{ad} &= f(a, d) \\ f_{bd} &= f(b, d). \end{aligned}$$

Our goal is to use linear interpolation to obtain an estimate of $f(\tilde{x}, \tilde{y})$. Suppose $\lambda \in [0, 1]$ with the property that $\tilde{x} = (1 - \lambda)a + \lambda b$. It follows that

$$\begin{aligned} f_{xc} &= (1 - \lambda)f_{ac} + \lambda f_{bc} \\ f_{xd} &= (1 - \lambda)f_{ad} + \lambda f_{bd} \end{aligned}$$

are linearly interpolated estimates of $f(\tilde{x}, c)$ and $f(\tilde{x}, d)$, respectively. Consequently, if $\mu \in [0, 1]$ with $\tilde{y} = (1 - \mu)c + \mu d$, then a second interpolation between f_1 and f_2 gives an estimate of $f(\tilde{x}, \tilde{y})$:

$$z = (1 - \mu)f_{xc} + \mu f_{xd} \approx f(\tilde{x}, \tilde{y}).$$

Putting it all together, we see that

$$\begin{aligned} z &= (1 - \mu)((1 - \lambda)f_{ac} + \lambda f_{bc}) + \mu((1 - \lambda)f_{ad} + \lambda f_{bd}) \\ &\approx f((1 - \lambda)a + \lambda b, (1 - \mu)c + \mu d). \end{aligned}$$

Figure 2.6 depicts the interpolation points. To interpolate the values in a matrix of $f(x, y)$ evaluations it is necessary to "locate" the point at which the interpolation is required. The four relevant values from the array must then be combined as described above:

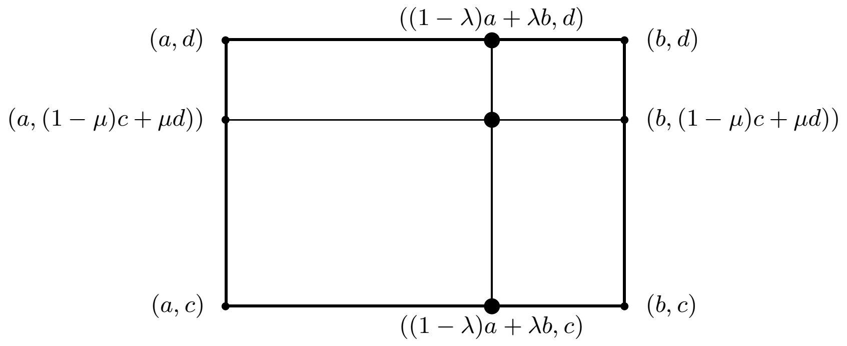


FIGURE 2.6 *Linear interpolation in two dimensions*

```
  function z = LinInterp2D(xc,yc,a,b,c,d,fA)
% z = LinInterp2D(xc,yc,a,b,n,c,d,m,fA)
% Linear interpolation on a grid of f(x,y) evaluations.
% xc, yc, a, b, c, and d  are scalars that satisfy a<=xc<=b and c<=yc<=d.
% fA is an n-by-m matrix with the property that
%
%     A(i,j) = f(a+(i-1)(b-a)/(n-1),c+(j-1)(d-c)/(m-1)) , i=1:n, j=1:m
%
% z is a linearly interpolated value of f(xc,yc).

[n,m] = size(fA);
% xc  = a+(i-1+dx)*hx    0<=dx<=1
hx = (b-a)/(n-1); i  = max([1 ceil((xc-a)/hx)]); dx = (xc - (a+(i-1)*hx))/hx;
% yc  = c+(j-1+dy)*hy    0<=dy<=1
hy = (d-c)/(m-1); j  = max([1 ceil((yc-c)/hy)]); dy = (yc - (c+(j-1)*hy))/hy;
z = (1-dy)*((1-dx)*fA(i,j)+dx*fA(i+1,j)) + dy*((1-dx)*fA(i,j+1)+dx*fA(i+1,j+1));
```

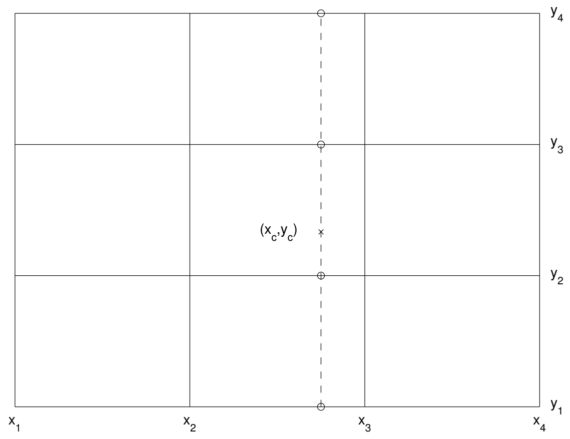
The following can be used for the table-generation across a uniform grid:

```
  function fA = SetUp(f,a,b,n,c,d,m)
% Sets up a matrix of f(x,y) evaluations.
% f is a handle to a function of the form f(x,y).
% a, b, c, and d  are scalars that satisfy a<=b and c<=d.
% n and m are integers >=2.
% fA is an n-by-m matrix with the property that
%
%     A(i,j) = f(a+(i-1)(b-a)/(n-1),c+(j-1)(d-c)/(m-1)) , i=1:n, j=1:m

x = linspace(a,b,n);
y = linspace(c,d,m);
fA = zeros(n,m);
for i=1:n
   for j=1:m
      fA(i,j) = f(x(i),y(j));
   end
end
```

Problems

P2.4.4 Analogous to `LinInterp2D`, write a function `CubicInterp2D(xc,yc,a,b,n,c,d,m,fA)` that does cubic interpolation from a matrix of $f(x,y)$ evaluations. Start by figuring out "where" (x_c, y_c) is in the grid with respect to `x = linspace(a,b,n)` and `y = linspace(c,d,m)`. Suppose this is the situation: as in `LinearInterp2D`.



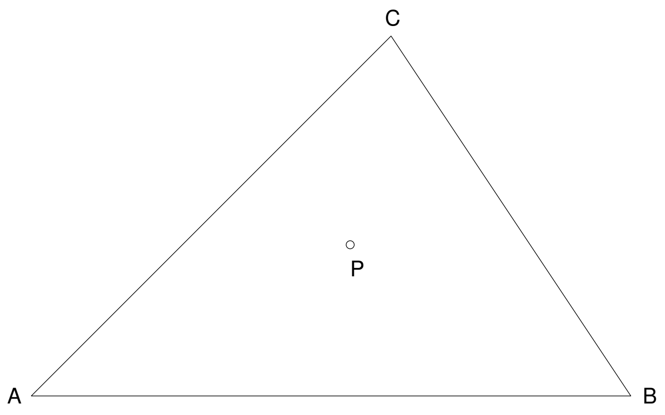
For $i = 1{:}4$, construct a cubic $p_i(x)$ that interpolates f at $(x_1, y_i), (x_2, y_i), (x_3, y_i)$ and (x_4, y_i). Then construct a cubic $q(y)$ that interpolates $(x_c, p_i(x_c))$, $i = 1{:}4$ and return the value of $q(y_c)$. The above picture/plan assumes that (x_c, y_c) is not in an "edge tile" so you'll have to work out something reasonable to do if that is the case.

P2.4.5 (a) Use `SetUp` to produce a matrix of function evaluations for

$$f(x,y) = \frac{1}{.2(x-3)^2 \; + \; .3*(y-1)^2 + .2}.$$

Set $(a, b, n, c, d, m) = (0, 5, 300, 0, 3, 150)$. (b) Produce a plot that shows what f "looks like" along the line segment $\{(x, y)\,|\, x = 5 - 5t,\ y = 3t,\ 0 \le t \le 1\}$. Do this by interpolating f at a sufficiently large number of points along the line segment.

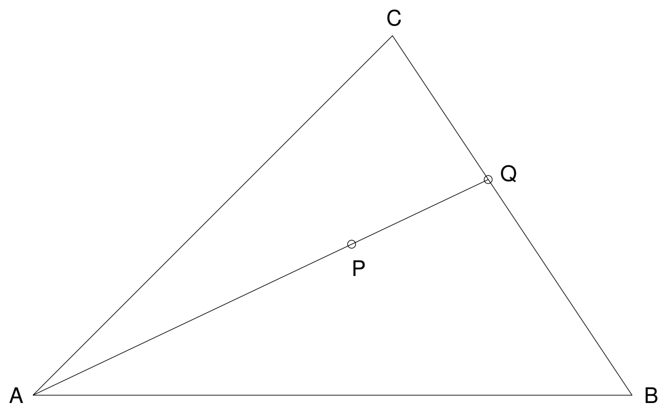
P2.4.6 This problem is about two-dimensional linear interpolation inside a triangle. Suppose that we know the value of a function $f(u, v)$ at the vertices of triangle ABC and that we wish to estimate its value at a point P inside the following triangle:



Consider the following method for doing this:

- Compute the intersection Q of line AP and line BC.
- Use linear interpolation to estimate f at Q from its value at B and C.
- Use linear interpolation to estimate f at P from its value at A and its estimate at Q.

Complete the following function so that it implements this method. Vectorization is not important.



```
  function fp = InterpTri(x,y,fvals,p)
% Suppose f(u,v) is a function of two variables defined everywhere in the plane.
% Assume that x, y, and fvals are column 3-vectors and p is a column 2-vector.
% Assume that (p(1),p(2)) is inside the triangle defined by (x(1),y(1)), (x(2),y(2)),
% and (x(3),y(3)) and that fvals(i) = f(x(i),y(i)) for i=1:3. fp is an estimate of
% f(p(1),p(2)) obtained by linear interpolation.
```

2.4.4 Trigonometric Interpolation

Suppose $f(t)$ is a periodic function with period T, $n = 2m$, and that we want to interpolate the data $(t_0, f_0), \ldots, (t_n, f_n)$ where $f_k = f(t_k)$ and

$$t_k = k\frac{T}{n}, \qquad k = 0{:}n.$$

Because the data is periodic it makes more sense to interpolate with a periodic function rather than with a polynomial. So let us pursue an interpolant that is a linear combination of cosines and sines rather than an interpolant that is a linear combination of 1, x, x^2, etc.

Assuming that j is an integer, the functions $\cos(2\pi jt/T)$ and $\sin(2\pi jt/T)$ have the same period as f prompting us to seek real scalars $a_0, \ldots, a_m$ and $b_0, \ldots, b_m$ so that if

$$F(t) = \sum_{j=0}^{m} \left[a_j \cos\left(\frac{2\pi j}{T}t\right) \;+\; b_j \sin\left(\frac{2\pi j}{T}t\right)\right],$$

then $F(t_k) = f_k$ for $k = 0{:}n$. This is a linear system that consists of $n+1$ equations in $2(m+1) = n+2$ unknowns. However, we note that b_0 and b_m are not involved in any equation since $\sin(2\pi jt/T) = 0$ if $t = t_0 = 0$ or $t = t_n = T$. Moreover, the $k = 0$ equation and the $k = n$ equation are identical because of periodicity. Thus, we really want to determine $a_0, \ldots, a_m$ and $b_1, \ldots, b_{m-1}$ so that if

$$F(t) = a_0 + \sum_{j=1}^{m-1} \left[a_j \cos\left(\frac{2\pi j}{T}t\right) \;+\; b_j \sin\left(\frac{2\pi j}{T}t\right)\right] + a_m \cos\left(\frac{2\pi m}{T}t\right),$$

then $F(t_k) = f(t_k) = f_k$ for $k = 0{:}n-1$. This is an n-by-n linear system in n unknowns:

$$f_k = a_0 + \sum_{j=1}^{m-1} \left(a_j \cos(kj\pi/m) \;+\; b_j \sin(kj\pi/m)\right) + (-1)^k a_m \qquad k = 0{:}n-1.$$

Here is what the system looks like for the case $n = 6$ with angles specified in degrees:

$$\begin{bmatrix} f_0 \\ f_1 \\ f_2 \\ f_3 \\ f_4 \\ f_5 \end{bmatrix} = \begin{bmatrix} \cos(0) & \cos(0) & \cos(0) & \cos(0) & \sin(0) & \sin(0) \\ \cos(0) & \cos(60) & \cos(120) & \cos(180) & \sin(60) & \sin(120) \\ \cos(0) & \cos(120) & \cos(240) & \cos(360) & \sin(120) & \sin(240) \\ \cos(0) & \cos(180) & \cos(360) & \cos(540) & \sin(180) & \sin(360) \\ \cos(0) & \cos(240) & \cos(480) & \cos(720) & \sin(240) & \sin(480) \\ \cos(0) & \cos(300) & \cos(600) & \cos(900) & \sin(300) & \sin(600) \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \\ b_1 \\ b_2 \end{bmatrix}$$

$$= \begin{bmatrix} 1 & 1 & 1 & 1 & 0 & 0 \\ 1 & 1/2 & -1/2 & -1 & \sqrt{3}/2 & \sqrt{3}/2 \\ 1 & -1/2 & -1/2 & 1 & \sqrt{3}/2 & -\sqrt{3}/2 \\ 1 & -1 & 1 & -1 & 0 & 0 \\ 1 & -1/2 & -1/2 & 1 & -\sqrt{3}/2 & \sqrt{3}/2 \\ 1 & 1/2 & -1/2 & -1 & -\sqrt{3}/2 & -\sqrt{3}/2 \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \\ b_1 \\ b_2 \end{bmatrix}.$$

For general even n, `CSInterp` sets up the defining linear system and solves for a and b:

```
  function F = CSInterp(f)
% F = CSInterp(f)
% f is a column n vector and n = 2m.
% F.a is a column m+1 vector and F.b is a column m-1 vector so that if
% tau = (pi/m)*(0:n-1)', then
%         f = F.a(1)*cos(0*tau) +...+ F.a(m+1)*cos(m*tau) +
%             F.b(1)*sin(tau)   +...+ F.b(m-1)*sin((m-1)*tau)

  n = length(f); m = n/2;
  tau = (pi/m)*(0:n-1)';
  P = [];
  for j=0:m,   P = [P cos(j*tau)]; end
  for j=1:m-1, P = [P sin(j*tau)]; end
  y = P\f;
  F = struct('a',y(1:m+1),'b',y(m+2:n));
```

Note that the a and b vectors are returned in a structure. The matrix of coefficients can be shown to be nonsingular so the interpolation process that we have presented is well-defined. However, it involves $O(n^3)$ flops because of the linear system solve. In P2.4.7 we show how to reduce this to $O(n^2)$. The evaluation of the trigonmetric interpolant can be handled by

```
  function Fvals = CSeval(F,T,tvals)
% F.a is a length m+1 column vector, F.b is a length m-1 column vector,
% T is a positive scalar, and tvals is a column vector.
% If
%   F(t)  = F.a(1) + F.a(2)*cos((2*pi/T)*t) +...+ F.a(m+1)*cos((2*m*pi/T)*t) +
%                    F.b(1)*sin((2*pi/T)*t) +...+ F.b(m-1)*sin((2*m*pi/T)*t)
%
% then Fvals = F(tvals).
  Fvals = zeros(length(tvals),1);
  tau = (2*pi/T)*tvals;
  for j=0:length(F.a)-1, Fvals = Fvals + F.a(j+1)*cos(j*tau); end
  for j=1:length(F.b),   Fvals = Fvals + F.b(j)*sin(j*tau); end
```

We close this section by applying `CSinterp` and `CSeval` to a data fitting problem that confronted Gauss in connection with the asteroid Pallas. The problem is to interpolate the following ascension-declination data

α	0	30	60	90	120	150	180	210	240	270	300	330
d	408	89	-66	10	338	807	1238	1511	1583	1462	1183	804

with a function of the form

$$d(\alpha) = a_0 \;+\; \sum_{j=1}^{5}\left[a_j\cos(2\pi j\alpha/360) + b_j\sin(2\pi j\alpha/360)\right] \;+\; a_6\cos(12\pi\alpha/360).$$

Here is a script that does this and plots the results shown in Figure 2.7:

```
% Script File: Pallas
% Plots the trigonometric interpolant of the Gauss Pallas data.
A = linspace(0,360,13)';
D = [ 408 89 -66 10 338 807 1238 1511 1583 1462 1183 804 408]';
Avals = linspace(0,360,200)';
F = CSInterp(D(1:12));
Fvals = CSeval(F,360,Avals);

plot(Avals,Fvals,A,D,'o')
axis([-10 370 -200 1700])
set(gca,'xTick',linspace(0,360,13))
xlabel('Ascension (Degrees)')
ylabel('Declination (minutes)')
```

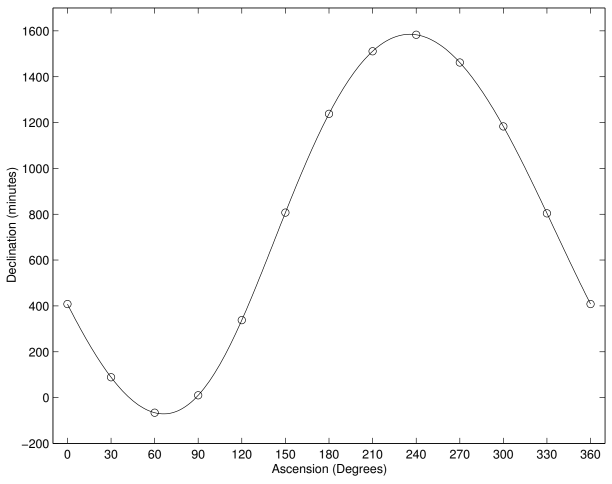


FIGURE 2.7 *Fitting the Pallas data*

Problems

P2.4.7 Observe that the matrix of coefficients P in `CSinterp` has the property that P^TP is diagonal. Use this fact to reduce the flop count in that function from $O(n^3)$ to $O(n^2)$. (With the *fast Fourier transform* it is possible to actually the flop count to an amazing $O(n\log n)$.)

M-Files and References

Script Files

`ShowV`	Illustrates `InterpV` and `HornerV`
`ShowN`	Illustrates `InterpN` and `HornerN`
`ShowRungePhenom`	Examines accuracy of interpolating polynomial.
`TableLookUp2D`	Illustrates `SetUp` and `LinInterp2D.`
`Pallas`	Fits periodic data with `CSInterp` and `CSEval`.

Function Files

`InterpV`	Construction of Vandermonde interpolating polynomial.
`HornerV`	Evaluates the Vandermonde interpolating polynomial.
`InterpNRecur`	Recursive construction of the Newton interpolating polynomial.
`InterpN`	Nonrecursive construction of the Newton interpolating polynomial.
`InterpN2`	Another nonrecursive construction of the Newton interpolating polynomial.
`HornerN`	Evaluates the Newton interpolating polynomial.
`SetUp`	Sets up matrix of f(x,y) evaluation.
`LinInterp2D`	2-Dimensional Linear Interpolation.
`Humps2D`	A sample function of two variables.
`CSInterp`	Fits periodic data with sines and cosines.
`CSEval`	Evaluates sums of sines and cosines.
`ShowMatPolyTools`	Illustrates `polyfit` and `polyval`.

References

W.L. Briggs and V.E. Henson (1995). *The DFT: An Owner's Manual for the Discrete Fourier Transform*, SIAM Publications, Philadelphia, PA.

S.D. Conte and C. de Boor (1980). *Elementary Numerical Analysis: An Algorithmic Approach, Third Edition*, McGraw-Hill, New York.

P. Davis (1963). *Interpolation and Approximation*, Blaisdell, New York.

Chapter 3

Piecewise Polynomial Interpolation

An important lesson from Chapter 2 is that high-degree polynomial interpolants at equally-spaced points should be avoided. This can pose a problem if we are to produce an accurate interpolant across a wide interval $[\alpha, \beta]$. One way around this difficulty is to partition $[\alpha, \beta]$,

$$\alpha = x_1 < x_2 < \cdots < x_n = \beta$$

and then interpolate the given function on each subinterval $[x_i, x_{i+1}]$ with a polynomial of low degree. This is the *piecewise polynomial* interpolation idea. The x_i are called *breakpoints*.

We begin with piecewise linear interpolation working with both fixed and adaptively determined breakpoints. The latter requires a classical divide-and-conquer approach that we shall use again in later chapters.

Piecewise linear functions do not have a continuous first derivative, and this creates problems in certain applications. Piecewise cubic Hermite interpolants address this issue. In this setting, the value of the interpolant and its derivative is specified at each breakpoint. The local cubics join in a way that forces first derivative continuity.

Second derivative continuity can be achieved by carefully choosing the first derivative values at the breakpoints. This leads to the topic of splines, a very important idea in the area of approximation and interpolation. It turns out that cubic splines produce the smoothest solution to the interpolation problem.

3.1 Piecewise Linear Interpolation

Assume that $x(1{:}n)$ and $y(1{:}n)$ are given where $\alpha = x_1 < \cdots < x_n = \beta$ and $y_i = f(x_i)$, $i = 1{:}n$. If you connect the dots $(x_1, y_1), \ldots, (x_n, y_n)$ with straight lines, as in Figure 3.1, then the graph of a *piecewise linear* function is displayed. We already have considerable experience with such functions, for this is what `plot(x,y)` displays.

3.1.1 Set-Up

The piecewise linear interpolant is built upon the local linear interpolants

$$L_i(z) = a_i + b_i(z - x_i),$$

where for $i = 1{:}n-1$ the coefficients are defined by

$$a_i = y_i \quad \text{and} \quad b_i = \frac{y_{i+1} - y_i}{x_{i+1} - x_i}.$$

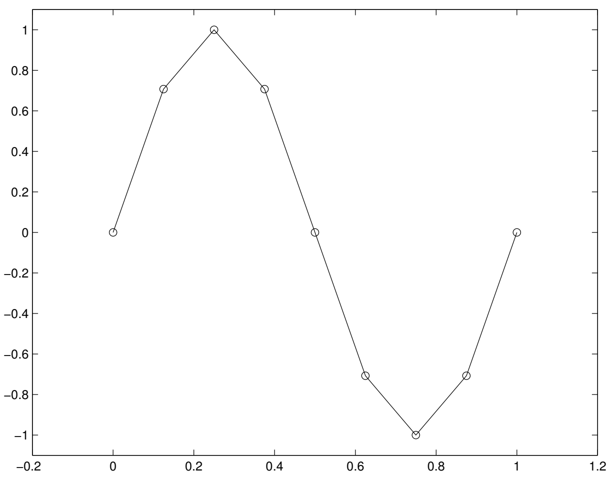


FIGURE 3.1 *A piecewise linear function*

Note that $L_i(z)$ is just the linear interpolant of f at the points $x = x_i$ and $x = x_{i+1}$. We then define

$$L(z) \;=\; \begin{cases} L_1(z) & \text{if} \quad x_1 \le z < x_2 \\ L_2(z) & \text{if} \quad x_2 \le z < x_3 \\ \vdots & \qquad\qquad \vdots \\ L_{n-1}(z) & \text{if} \quad x_{n-1} \le z \le x_n \end{cases} .$$

The act of setting up L is the act of solving each of the local linear interpolation problems. The $n-1$ divided differences $b_1, \ldots, b_{n-1}$ can obviously be computed by a loop,

```
for i=1:n-1
   b(i) = (y(i+1)-y(i))/(x(i+1)-x(i));
end
```

or by using pointwise division,

```
b = (y(2:n)-y(1:n-1)) ./ (x(2:n)-x(1:n-1))
```

or by using the built-in function `diff`:

```
b = diff(y) ./ diff(x)
```

Packaging these operations we obtain

```
  function [a,b] = pwL(x,y)
% Generates the piecewise linear interpolant of the data specified by the
% column n-vectors x and y. It is assumed that x(1) < x(2) < ... < x(n).
%
% a and b are column (n-1)-vectors with the property that for i=1:n-1, the
% line L(z) = a(i) + b(i)z passes though the points (x(i),y(i)) and (x(i+1),y(i+1)).

n = length(x);
a = y(1:n-1);
b = diff(y) ./ diff(x);
```

Thus,

```
z = linspace(0,1,9);
[a,b] = pwL(z,sin(2*pi*z));
```

sets up a piecewise linear interpolant of $\sin(2\pi z)$ on a uniform, nine-point partition of $[0, 1]$.

3.1.2 Evaluation

To evaluate L at a point $z \in [\alpha, \beta]$, it is necessary to determine the subinterval that contains z. In our problem `x` has the property that $x_1 < \cdots < x_n$ and so `sum(x<=z)` is the number of x_i that are to the left of z or equal to z. It follows that

```
if z == x(n);
   i = n-1;
else
   i = sum(x<=z);
end
```

determines the index i so that $x_i \le z \le x_{i+1}$. Notice the special handling of the case when z equals x_n. (Why?) A total of n comparisons are made because every component in `x` is compared to `z`.

A better approach is to exploit the monotonicity of the x_i and to use binary search. Here is the main idea. Suppose we have indices *Left* and *Right* so that $x_{Left} \le z \le x_{Right}$. If $mid = \text{floor}((Left + Right)/2)$, then by checking z's relation to x_{mid} we can halve the search space by redefining $Left$ or $Right$ accordingly:

```
mid = floor((Left+Right)/2);
if z < x(mid)
   Right = mid;
else
   Left = mid;
end
```

Repeated application of this process eventually identifies the subinterval that houses z:

```
if z == x(n)
   i = n-1;
else
   Left = 1; Right = n;
   while Right > Left+1
      % z is in [x(Left),x(Right)].
      mid = floor((Left+Right)/2);
      if z < x(mid)
         Right = mid;
      else
         Left = mid;
      end
   end
   i = Left;
end
```

Upon completion, `i` contains the index of the subinterval that contains z. If $n = 10$ and $z \in [x_6, x_7]$, then here is the succession of `Left` and `Right` values produced by the binary search method:

Left	Right	mid
1	10	5
5	10	7
5	7	6
6	7	-

Roughly $\log_2(n)$ comparisons are required to locate the appropriate subinterval. If n is large, then this is much more efficient than the `sum(x<z)` method, which requires n comparisons.

For "random" z, we can do no better than binary search. However, if L is to be evaluated at an ordered succession of points, then we can improve the subinterval location process. For example, suppose we want to plot L on $[\alpha, \beta]$. This requires the assembly of the values $L(z_1), \ldots, L(z_m)$ in a vector where m is a typically large integer and $\alpha \le z_1 \le \cdots \le z_m \le \beta$. Rather than locate each z_i via binary search, it is more efficient to exploit the systematic "migration" of the evaluation point as it moves left to right across the subintervals. Chances are that if i is the subinterval index associated with the current z-value, then i will be the correct index for the next z-value. This "guess" at the correct subinterval can be checked before we launch the binary search process.

```
  function i = locate(x,z,g)
% Locates z in a partition x.
% x is column n-vector with x(1) < x(2) <...<x(n) and
% z is a scalar with x(1) <= z <= x(n).
% g (1<=g<=n-1) is an optional input parameter
% i is an integer such that x(i) <= z <= x(i+1). Before the general
% search for i begins, the value i=g is tried.
if nargin==3
   % Try the initial guess.
   if (x(g)<=z) & (z<=x(g+1))
      i = g;
      return
   end
end
n = length(x);
if z==x(n)
   i = n-1;
else
   % Binary Search
   Left = 1; Right = n;
   while Right > Left+1
      % x(Left) <= z <= x(Right)
      mid = floor((Left+Right)/2);
      if z < x(mid)
         Right = mid;
      else
         Left = mid;
      end
   end
   i = Left;
end
```

This function makes use of the `return` command. This terminates the execution of the function. It is possible to restructure `locate` to avoid the `return`, but the resulting logic would be cumbersome. As an application of `locate`, here is a function that produces a vector of L-values:

```
  function LVals = pwLeval(a,b,x,zVals)
% Evaluates the piecewise linear polynomial defined by the column (n-1)-vectors
% a and b and the column n-vector x. It is assumed that x(1) < ... < x(n).
% zVals  is a column m-vector with each component in [x(1),x(n)].
% LVals is a column m-vector with the property that LVals(j) = L(zVals(j))
% for j=1:m where L(z)= a(i) + b(i)(z-x(i)) for x(i)<=z<=x(i+1).

m = length(zVals);  LVals = zeros(m,1);  g = 1;
for j=1:m
   i = locate(x,zVals(j),g);
   LVals(j) = a(i) + b(i)*(zVals(j)-x(i));
   g = i;
end
```

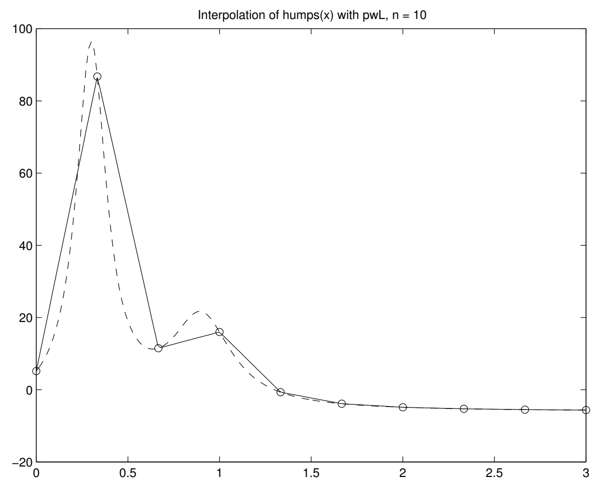


FIGURE 3.2 *Piecewise linear approximation*

The following script illustrates the use of this function, producing a sequence of piecewise linear approximations to the built-in function

$$\text{humps}(x) = \frac{1}{(x-.3)^2+.01} + \frac{1}{(x-.9)^2+.04} - 6.$$

```
% Script File: ShowPWL1
% Convergence of the piecewise linear interpolant to
% humps(x) on [0,3]
close all
z = linspace(0,3,200)';
fvals = humps(z);
for n = [5 10 25 50]
   figure
   x = linspace(0,3,n)';
   y = humps(x);
   [a,b] = pwL(x,y);
   Lvals = pwLEval(a,b,x,z);
   plot(z,Lvals,z,fvals,'--',x,y,'o');
   title(sprintf('Interpolation of humps(x) with pwL, n = %2.0f',n))
end
```

(See Figure 3.2 for the 10-point case.) Observe that more interpolation points are required in regions where `humps` is particularly nonlinear.

3.1.3 A Priori Determination of the Breakpoints

Let us consider how many breakpoints we need to obtain a satisfactory piecewise linear interpolant. If $z \in [x_i, x_{i+1}]$, then from Theorem 2,

$$f(z) = L(z) + \frac{f^{(2)}(\eta)}{2}(z-x_i)(z-x_{i+1}),$$

where $\eta \in [x_i, x_{i+1}]$. If the second derivative of f on $[\alpha, \beta]$ is bounded by M_2 and if $\bar{h}$ is the length of the longest subinterval in the partition, then it is not hard to show that

$$|f(z) - L(z)| \;\leq\; \frac{M_2\bar{h}^2}{8}$$

for all $z \in [\alpha, \beta]$.

A typical situation where this error bound can be put to good use is in the design of the underlying partition upon which L is based. Assume that $L(x)$ is based on the uniform partition

$$\alpha = x_1 < x_2 < \cdots < x_n = \beta,$$

where

$$x_i = \alpha + \frac{i-1}{n-1}(\beta - \alpha).$$

To ensure that the error between L and f is less than or equal to a given positive tolerance δ, we insist that

$$|f(z) - L(z)| \;\le\; \frac{M_2\bar{h}^2}{8} \;=\; \frac{M_2}{8}\left(\frac{\beta-\alpha}{n-1}\right)^2 \;\le\; \delta.$$

From this we conclude that n must satisfy

$$n \;\ge\; 1 + (\beta - \alpha)\sqrt{M_2/8\delta}.$$

For the sake of efficiency, it makes sense to let n be the smallest integer that satisfies this inequality:

```
  function [x,y] = pwLstatic(f,M2,alpha,beta,delta)
% Generates interpolation points for a piecewise linear approximation of
% prescribed accuracy.
%
% f is a handle that references a function f(x).
% Assume that f can take vector arguments.
% M2 is an upper bound for|f"(x)| on [alpha,beta].
% alpha and beta are scalars with alpha<beta.
% delta is a positive scalar.
%
% x and y  column n-vectors with the property that y(i) = f(x(i)), i=1:n.
% The piecewise linear interpolant L(x) of this data satisfies
% |L(z) - f(z)| <= delta for x(1) <= z <= x(n).

n = max(2,ceil(1+(beta-alpha)*sqrt(M2/(8*delta))));
x = linspace(alpha,beta,n)';
y = f(x);
```

The partition produced by `pwLstatic` *does not* take into account the sampled values of f. As a result, the uniform partition produced may be much too refined in regions where f'' is much smaller than the bound M_2.

3.1.4 Adaptive Piecewise Linear Interpolation

Suppose f is very nonlinear over just a small portion of $[\alpha, \beta]$ and very smooth elsewhere. (See Figure 3.2.) This means that if we use `pwLstatic` to generate the partition, then we are compelled to use a large `M2`. Lots of subintervals and (perhaps costly) f-evaluations will be required. Over regions where f is smooth, the partition will be overly refined.

To address this problem, we develop a recursive partitioning algorithm that "discovers" where f is "extra nonlinear" and that clusters the breakpoints accordingly. A definition simplifies the discussion. We say that the subinterval $[xL, xR]$ is *acceptable* if

$$\left| f\left(\frac{xL + xR}{2}\right) \;-\; \frac{f(xL) + f(xR)}{2} \right| \;\le\; \delta$$

or if

$$xR - xL \le h_{min},$$

where $\delta > 0$ and $h_{min} > 0$ are (typically small) refinement parameters. The first condition measures the discrepancy between the line that connects $(xL, f(xL))$ and $(xR, f(xR))$ and the function $f(x)$ at the interval midpoint $m = (xL+xR)/2$. The second condition says that sufficiently short subintervals are also acceptable where "sufficiently short" means less than h_{min} in length.

One more definition is required before we can describe the complete partitioning process. A partition $x_1 < \cdots < x_n$ is *acceptable* if each subinterval is acceptable. Note that if

$$xL = x_1^{(L)} < \cdots < x_n^{(L)} = m$$

is an acceptable partition of $[xL, m]$ and if

$$m = x_1^{(R)} < \cdots < x_n^{(R)} = xR$$

is an acceptable partition of $[m, xR]$, then

$$xL = x_1^{(L)} < \cdots < x_n^{(L)} < x_2^{(R)} < \cdots < x_n^{(R)} = xR$$

is an acceptable partition of $[xL, xR]$. This sets the stage for a recursive determination of an acceptable partition:

```
  function [x,y] = pwLadapt(f,xL,fL,xR,fR,delta,hmin)
% Adaptively determines interpolation points for a piecewise linear
% approximation of a specified function.
%
% f is a handle that references a function of the form y = f(u).
% xL and xR are real scalars and fL = f(xL) and fR = f(xR).
% delta and hmin are positive real scalars that determine accuracy.
%
% x and y are column n-vectors with the property that
%              xL = x(1) < ... < x(n) = xR
% and y(i) = f(x(i)), i=1:n. Each subinterval [x(i),x(i+1)] is
% either <= hmin in length or has the property that at its midpoint m,
% |f(m) - L(m)| <= delta where L(x) is the line that connects (x(i),y(i))
% and (x(i+1),y(i+1)).

if (xR-xL) <= hmin
   % Subinterval is acceptable
   x = [xL;xR];
   y = [fL;fR];
else
   mid  = (xL+xR)/2;
   fmid = f(mid);
   if (abs(((fL+fR)/2) - fmid) <= delta )
      % Subinterval accepted.
      x = [ xL;xR];
      y = [fL;fR];
   else
      % Produce left and right partitions, then synthesize.
      [xLeft,yLeft]   = pwLAdapt(f,xL,fL,mid,fmid,delta,hmin);
      [xRight,yRight] = pwLAdapt(f,mid,fmid,xR,fR,delta,hmin);
      x = [ xLeft;xRight(2:length(xRight))];
      y = [ yLeft;yRight(2:length(yRight))];
   end
end
```

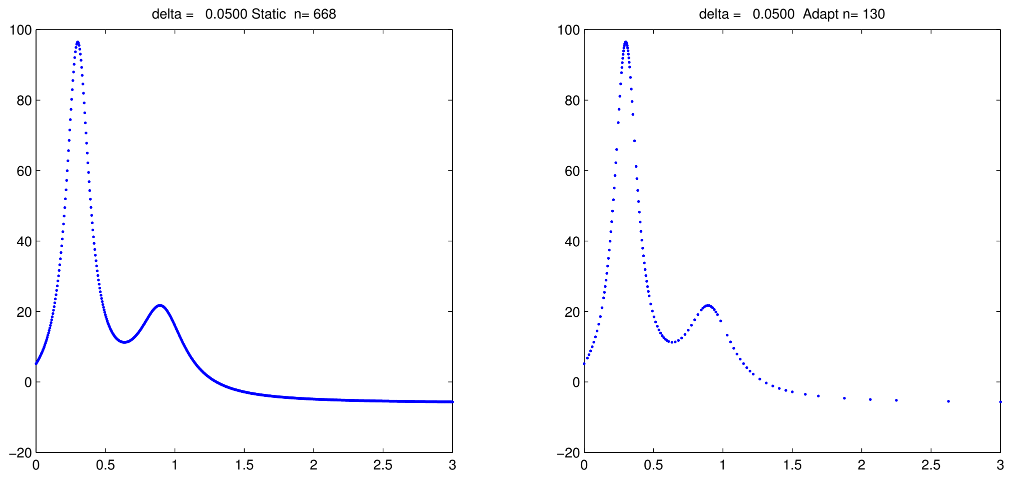


Figure 3.3 *Static versus adaptive approximation*

The idea behind the function is to check and see if the input interval is acceptable. If it is not, then acceptable partitions are obtained for the left and right half intervals. These are then "glued" together to obtain the final, acceptable partition.

The distinction between static and adaptive piecewise linear interpolation is revealed by running the following script:

```
% Script File: ShowpwL2
% Compares pwLstatic and pwLsdapt on [0,3] using the function
%
%   humps(x) = 1/((x-.3)^2 + .01)  +  1/((x-.9)^2+.04)
%
close all
% Second derivative estimate based on divided differences
z = linspace(0,1,101);
humpvals = humps(z);
M2 = max(abs(diff(humpvals,2)/(.01)^2));
for delta = [1 .5 .1 .05 .01]
   figure
   [x,y] = pwLstatic(@humps,M2,0,3,delta);
   subplot(1,2,1)
   plot(x,y,'.');
   title(sprintf('delta = %8.4f Static  n= %2.0f',delta,length(x)))
   [x,y] = pwLadapt(@humps,0,humps(0),3,humps(3),delta,.001);
   subplot(1,2,2)
   plot(x,y,'.');
   title(sprintf('delta = %8.4f  Adapt n= %2.0f',delta,length(x)))
   set(gcf,'position',[200 200 1200 500])
end
```

(See Figure 3.3.) The `humps` function is very nonlinear in the vicinity of $x = .3$. A second derivative bound is approximated with differences and used in `pwLstatic`. In the example approximately four times as many function evaluations are required when the static approach is taken.

Problems

P3.1.1 Generalize `locate` so that it tries $i = g + 1$ and $i = g - 1$ before resorting to binary search. (Take care to guard against subscript out-of-range.) Implement `pwLeval` with this modified subinterval locator and document the speed-up.

P3.1.2 Write a function `i = LocateUniform(alpha,beta,n,z)` that assumes $[\alpha, \beta]$ is partitioned into $n-1$ subintervals of equal length and returns the index of the interval that houses z.

P3.1.3 What happens if `pwLadapt` is applied to $\sin(x)$ with $[\alpha, \beta] = [0, 2\pi]$?

P3.1.4 Describe what would happen if `pwLadapt` is called with `delta = 0`.

P3.1.5 Describe why the number of recursive calls in `pwLadapt` is bounded if $|f''(x)|$ is bounded on $[\alpha, \beta]$.

P3.1.6 Modify `pwLadapt` so that a subinterval is accepted if $|f(p) - \lambda(p)|$ *and* $|f(q) - \lambda(q)|$ are less than or equal to `delta`, where $p = (2xL + xR)/3$, $q = (xL + 2xR)/3$, and $\lambda(x)$ is the line that connects `(xL,fL)` and `(xR,fR)`. Avoid redundant function evaluations.

P3.1.7 If `pwLadapt` is applied to the function $f(x) = \sqrt{x}$ on the interval [0,1], then a partition $x(1{:}n)$ is produced that satisfies

$$x_2 - x_1 \le x_3 - x_2 \le \cdots \le x_n - x_{n-1}.$$

Why?

P3.1.8 Generalize `pwLadapt(f,xL,fL,xR,fR,delta,hmin)` to

```
    function [x,y,eTaken] = pwLadapt(f,xL,fL,xR,fR,delta,hmin,eMax)
```

so that no more than `eMax` function evaluations are taken. The value of `eTaken` should be the actual number of function evaluations spent. Let $n = \mathit{length}(x)$. In a "successful" call, `x(n)` should equal `xR`, meaning that a satisfactory piecewise linear approximation was found extending across the entire interval $[xL, xR]$. If this is not the case, then the evaluation limit was encountered before xR was reached and `x(n)` will be less than `xR`. In this situation vectors returned define a satisfactory piecewise linear approximation across $[x(1), x(n)]$.

P3.1.9 Notice that in `pwLadapt` the vector `y` does not include all the computed function evaluations. So that these evaluations are not lost, generalize `pwLadapt` to

```
  [x,y,xUnused,yUnused] = pwLadaptGen(f,xL,fL,xR,fR,delta,hmin,...)
```

where **(a)** the `x` and `y` vectors are identical to what `pwLadapt` computes and **(b)** `xUnused` and `yUnused` are column vectors that contain the x-values and function values that were computed, but not included in `x` and `y`. Thus, the `xUnused` and `yUnused` vectors should have the property that `yUnused(i) = feval(fname,xUnused(i))`, `i = 1:length(xUnused)`. You are allowed to extend the calling sequence if convenient. In that case, indicate the values that should be passed through these new parameters at the top-level call. `xUnused` and `yUnused` should be assigned the empty vector [ ] if `xR-xL<hmin`. The order of the values in `xUnused` is not important.

P3.1.10 Vectorize `locate` and `pwLeval`.

3.2 Piecewise Cubic Hermite Interpolation

Now let's graduate to piecewise cubic functions. With the increase in degree we can obtain a smoother fit to a given set of n points. The idea is to interpolate both f and its derivative with a cubic on each of the subintervals.

3.2.1 Cubic Hermite Interpolation

So far we have only considered the interpolation of function values at distinct points. In the *Hermite* interpolation problem, both the function and its derivative are interpolated. To illustrate the idea, we consider the interpolation of the function $f(z) = \cos(z)$ at the points $x_1 = 0$, $x_2 = \delta$, $x_3 = 3\pi/2 - \delta$, and $x_4 = 3\pi/2$ by a cubic $p_3(z)$. For small δ we notice that $p_3(z)$ seems to interpolate both f and f' at $z = 0$ and $z = 3\pi/2$. The interpolation shown in Figure 3.4 on the next page was obtained by running the following script:

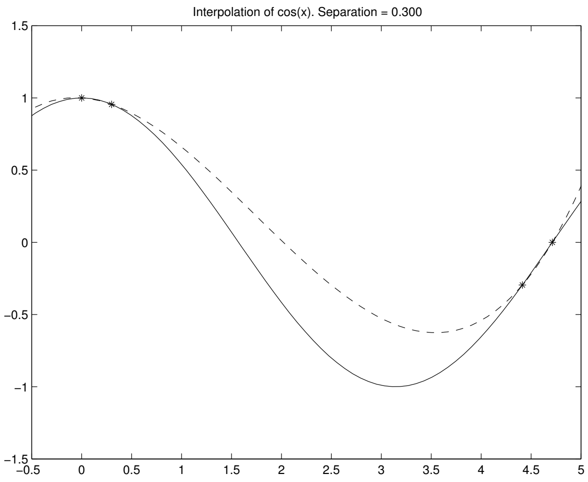


FIGURE 3.4 *A "nearly" Hermite interpolation*

```
% Script File: ShowHermite
% Plots a succession of cubic interpolants to cos(x).
% x(2) converges to x(1) = 0 and x(3) converges to x(4) = 3pi/2.
close all
z = linspace(-pi/2,2*pi,100);
CosValues = cos(z);
for d = [1 .5  .3  .1  .05 .001]
   figure
   xvals = [0;d;(3*pi/2)-d;3*pi/2];
   yvals = cos(xvals);
   c = InterpN(xvals,yvals);
   CubicValues = HornerN(c,xvals,z);
   plot(z,CosValues,z,CubicValues,'--',xvals,yvals,'*')
   axis([-.5 5 -1.5 1.5])
   title(sprintf('Interpolation of cos(x). Separation = %5.3f',d))
end
```

As the points coalesce, the cubic converges to a cubic interpolant of the cosine and its derivative at the points 0 and $3\pi/2$. This is called the *Hermite cubic interpolant.*

In the general cubic Hermite interpolation problem, we are given function values y_L and y_R and derivative values s_L and s_R and seek coefficients a, b, c, and d so that if

$$q(z) = a + b(z - x_L) + c(z - x_L)^2 + d(z - x_L)^2(z - x_R),$$

then

$$\begin{aligned} q(x_L) &= y_L & \qquad q(x_R) &= y_R \\ q'(x_L) &= s_L & \qquad q'(x_R) &= s_R. \end{aligned}$$

Each of these equations "says" something about the unknown coefficients. Noting that

$$q'(z) = b + 2c(z - x_L) + d(2(z - x_L)(z - x_R) + (z - x_L)^2),$$

we see that

$$\begin{aligned} a &= y_L & \qquad a + b\Delta x + c(\Delta x)^2 &= y_R \\ b &= s_L & \qquad b + 2c\Delta x + d(\Delta x)^2 &= s_R, \end{aligned}$$

where $\Delta x = x_R - x_L$. Expressing this in matrix-vector we obtain

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 1 & \Delta x & (\Delta x)^2 & 0 \\ 0 & 1 & 2\Delta x & (\Delta x)^2 \end{bmatrix} \begin{bmatrix} a \\ b \\ c \\ d \end{bmatrix} = \begin{bmatrix} y_L \\ s_L \\ y_R \\ s_R \end{bmatrix}.$$

The solution to this triangular system is straightforward:

$$\begin{aligned} a &= y_L \\ b &= s_L \\ c &= \frac{y'_L - s_L}{\Delta x} \\ d &= \frac{s_R + s_L - 2y'_L}{(\Delta x)^2} \end{aligned}$$

where

$$y'_L = \frac{y_R - y_L}{\Delta x} = \frac{y_R - y_L}{x_R - x_L}.$$

Thus, we obtain

```
  function [a,b,c,d] = HCubic(xL,yL,sL,xR,yR,sR)
% Cubic Hermite interpolation
% (xL,yL,sL) and (xR,yR,sR) are x-y-slope triplets with xL and xR distinct.
% a,b,c,d are real numbers with the property that if
%           p(z) = a + b(z-xL) + c(z-xL)^2 + d(z-xL)^2(z-xR)
% then p(xL)=yL, p'(xL)=sL, p(xR)=yR, p'(xR)=sR.
a = yL; b = sL; delx = xR - xL;
yp = (yR - yL)/delx;
c  = (yp - sL)/delx;
d  = (sL - 2*yp + sR)/(delx*delx);
```

An error expression for the cubic Hermite interpolant can be derived from Theorem 3.

Theorem 3 *Suppose $f(z)$ and its first four derivatives are continuous on $[x_L, x_R]$ and that the constant M_4 satisfies*

$$|f^{(4)}(z)| \le M_4$$

for all $z \in [L, R]$. If q is the cubic Hermite interpolant of f at x_L and x_R, then

$$|f(z) - q(z)| \le \frac{M_4}{384} h^4,$$

where $h = x_R - x_L$.

Proof If $q_\delta(z)$ is the cubic interpolant of f at x_L, $x_L + \delta$, $x_R - \delta$, and x_R, then from Theorem 2 we have

$$|f(z) - q_\delta(z)| \le \frac{M_4}{24} |(z - x_L)(z - x_L - \delta)(z - x_R + \delta)(z - x_R)|$$

for all $z \in [x_L, x_R]$. We assume without proof[1] that

$$\lim_{\delta \to 0} q_\delta(z) = q(z)$$

and so

$$|f(z) - q(z)| \le \frac{M_4}{24} |(z - x_L)(z - x_L)(z - x_R)(z - x_R)|.$$

[1] But check out `ShowHermite`

The maximum value of the quartic polynomial on the right occurs at the midpoint $z = x_L + h/2$ and so for all z in the interval $[x_L, x_R]$ we have

$$|f(z) - q(z)| \leq \frac{M_4}{24}\left(\frac{h}{2}\right)^4 = \frac{M_4}{384}h^4. \ \Box$$

Theorem 3 says that if the interval length is divided by 10, then the error bound is reduced by a factor of 10^4.

3.2.2 Representation and Set-Up

We now show how to glue a sequence of Hermite cubic interpolants together so that the resulting piecewise cubic polynomial $C(z)$ interpolates the data $(x_1, y_1), \ldots, (x_n, y_n)$, with the prescribed slopes $s_1, \ldots, s_n$. To that end we assume $x_1 < x_2 < \cdots < x_n$ and define the ith *local cubic* by

$$q_i(z) = a_i + b_i(z - x_i) + c_i(z - x_i)^2 + d_i(z - x_i)^2(z - x_{i+1}).$$

Define the piecewise cubic polynomial by

$$C(z) \ = \left\{ \begin{array}{lcc} q_1(z) & \text{if} & x_1 \leq z < x_2 \\ q_2(z) & \text{if} & x_2 \leq z < x_3 \\ \vdots & & \vdots \\ q_{n-1}(z) & \text{if} & x_{n-1} \leq z \leq x_n \end{array} \right. .$$

Our goal is to determine $a(1{:}n-1)$, $b(1{:}n-1)$, $c(1{:}n-1)$, and $d(1{:}n-1)$ so that

$$\begin{array}{lcl} C(x_i) & = & y_i \\ C'(x_i) & = & s_i \end{array}, \qquad i = 1{:}n$$

This will be the case if we simply solve the following $n-1$ cubic Hermite problems:

$$\begin{array}{lcl} q_i(x_i) & = & y_i \\ q_i'(x_i) & = & s_i \\ q_i(x_{i+1}) & = & y_{i+1} \\ q_i'(x_{i+1}) & = & s_{i+1} \end{array}$$

The results of §3.2.1 apply:

$$a_i = y_i, \qquad b_i = s_i, \qquad c_i = \frac{y_i' - s_i}{\Delta x_i}, \qquad d_i = \frac{s_{i+1} + s_i - 2y_i'}{(\Delta x_i)^2},$$

where $\Delta x_i = x_{i+1} - x_i$ and

$$y_i' \ = \ \frac{y_{i+1} - y_i}{\Delta x_i} \ = \ \frac{y_{i+1} - y_i}{x_{i+1} - x_i}.$$

We could use `HCubic` to resolve the coefficients:

```
for i=1:n-1
   [a(i), b(i), c(i), d(i)] = HCubic(x(i),y(i),s(i),x(i+1),y(i+1),s(i+1))
end
```

But a better solution is to vectorize the computation, and this gives

```
  function [a,b,c,d] = pwC(x,y,s)
% Piecewise cubic Hermite interpolation.
%
% x,y,s  column n-vectors with x(1) < ... < x(n)
%
% a,b,c,d  column (n-1)-vectors that define a continuous, piecewise
% cubic polynomial q(z) with the property that for i = 1:n,
%
%            q(x(i)) = y(i) and q'(x(i)) = s(i).
%
% On the interval [x(i),x(i+1)],
%
%      q(z) = a(i) + b(i)(z-x(i)) + c(i)(z-x(i))^2  + d(i)(z-x(i))^2(z-x(i+1)).
n  = length(x);
a  = y(1:n-1);
b  = s(1:n-1);
Dx = diff(x);
Dy = diff(y);
yp = Dy ./ Dx;
c  = (yp - s(1:n-1)) ./ Dx;
d  = (s(2:n) + s(1:n-1) - 2*yp) ./ (Dx.* Dx);
```

If M_4 bounds $|f^{(4)}(x)|$ on the interval $[x_1, x_n]$, then Theorem 3 implies that

$$|f(z) - C(z)| \le \frac{M_4}{384}\bar{h}^4$$

for all $z \in [x_1, x_n]$, where $\bar{h}$ is the length of the longest subinterval (i.e., $\max_i |x_{i+1} - x_i|$.)

3.2.3 Evaluation

The evaluation of $C(z)$ has two parts. As with any piecewise polynomial that must be evaluated, the position of z in the partition must be ascertained. Once that is accomplished, the relevant local cubic must be evaluated. Here is a function that can be used to evaluate C at a vector of z values:

```
  function Cvals = pwCeval(a,b,c,d,x,zVals)
% Evaluates the pwC defined by the column (n-1)-vectors a,b,c, and
% d and the column n-vector x. It is assumed that x(1) < ... < x(n).
% zVals  is a column m-vector with each component in [x(1),x(n)].
%
% CVals is a column m-vector with the property that CVals(j) = C(zVals(j))
% for j=1:m where on the interval [x(i),x(i+1)]
%
%   C(z)= a(i) + b(i)(z-x(i)) + c(i)(z-x(i))^2 + d(i)(z-x(i))^2(z-x(i+1))
m = length(zVals);
Cvals = zeros(m,1);
g=1;
for j=1:m
   i = Locate(x,zVals(j),g);
   Cvals(j) = d(i)*(zVals(j)-x(i+1)) + c(i);
   Cvals(j) = Cvals(j)*(zVals(j)-x(i)) + b(i);
   Cvals(j) = Cvals(j)*(zVals(j)-x(i)) + a(i);
   g = i;
end
```

Analogous to `pwLeval`, we use `Locate` to determine the subinterval that houses the jth evaluation point z_j. The cubic version of `HornerN` is then used to evaluate the appropriate local cubic. The following script file illustrates the use of `pwC` and `pwCeval`:

```
% Script File: ShowpwCH
% Convergence of the piecewise cubic hermite interpolant to
% exp(-2x)sin(10*pi*x) on [0,1].)
close all
z = linspace(0,1,200)';
fvals = exp(-2*z).*sin(10*pi*z);
for n = [4 8 16 24]
   x = linspace(0,1,n)';
   y = exp(-2*x).*sin(10*pi*x);
   s = 10*pi*exp(-2*x).*cos(10*pi*x)-2*y;
   [a,b,c,d] = pwC(x,y,s);
   Cvals = pwCeval(a,b,c,d,x,z);
   figure
   plot(z,fvals,z,Cvals,'--',x,y,'*');
   title(sprintf('Interpolation of exp(-2x)sin(10pi*x) with pwCH, n = %2.0f',n))
end
legend('e^{-2z}sin(10\pi z)','The pwC interpolant')
```

Sample output is displayed in Figure 3.5.

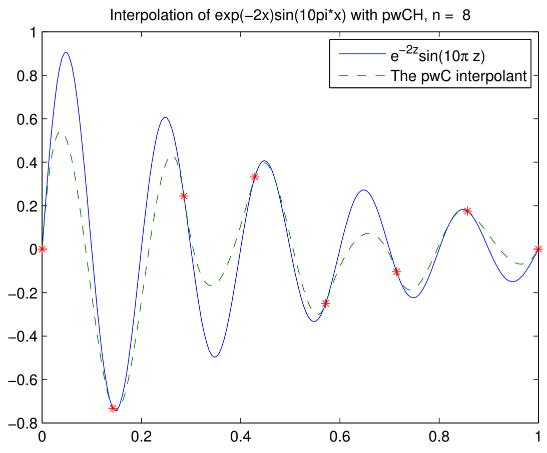


FIGURE 3.5 *Piecewise cubic Hermite interpolant of* $e^{-2x}\sin(10\pi x)$, $n = 8$

Problems

P3.2.1 Write a function `[a,b,c,d] = pwCstatic(f,fp,M4,alpha,beta,delta)` analogous to `pwLstatic`. It should produce a piecewise cubic Hermite approximation with uniform spacing. It should use the error result of Theorem 3 and the 4th derivative bound `M4` to determine the partition. The parameters `f` and `fp` should be handles that reference the function and its derivative respectively.

P3.2.2 Write a recursive function

```
function [x,y,s] = pwCAdapt(f,fp,L,fL,DfL,R,fR,DfR,delta,hmin)
```

analogous to `pwLadapt`. Use the same interval acceptance tests as in `pwLadapt`. The parameters `f` and `fp` should be handles that reference the function and its derivative respectively. Use both `pwLAdapt` and `pwCAdapt` to produce approximations to $f(x) = \sqrt{x}$ on the interval $[.001, 9]$. Fix $h_{min} = .001$. Print a table that shows the number of partition points computed by `pwLadapt` and `pwCadapt` for `delta` = .1, .01, .001, .0001, and .00001.

P3.2.3 Complete the following function:

```
  function [R,fR] = stretch(L,fL,tol);
% L,fL are scalars that satisfy fL = exp(-L) and tol is a positive real.
% R,fR are scalars that satisfy fR = exp(-R) with the property that if q(z) is the cubic
% hermite interpolant of exp(-z) at z=L and z=R, then |q(z) - exp(-z)| <= tol on [L,R].
```

Make effective use of the error bound in Theorem 3 when choosing R. Hint: How big can you make R and still guarantee the required accuracy? Making effective use of `stretch` complete the following:

```
  function [x,y] = pwCexp(a,b,tol)
% a,b are scalars that satisfy a < b and tol is a positive real.
% x,y are column n-vectors where a = x(1) < x(2) < ... < x(n) = b
% and y(i) = exp(-x(i)), i=1:n. The partition is chosen so that if C(z)
% is the piecewise cubic hermite interpolant of exp(-z) on this partition,
% then |C(z) - exp(-z)| <= tol for all z in [a,b]
```

P3.2.4 We want to interpolate a function f on $[a, b]$ with error less than *tol*. When is it cheaper to set up a piecewise linear interpolant $L(z)$ with a uniform partition than a piecewise cubic hermite interpolant $C(z)$ with a uniform partition? Your answer should make use of the following facts and assumptions:

- If ℓ is the linear interpolant of f on an interval $[\alpha, \beta]$, then on that interval the error is no bigger than $M_2(\beta - \alpha)^2/8$, where M_2 is an upper bound for $|f^{(2)}(z)|$. Assume that M_2 is known.
- If p is the cubic hermite interpolant of f on an interval $[\alpha, \beta]$, then on that interval the error is no bigger than $M_4(\beta - \alpha)^4/384$, where M_4 is an upper bound for $|f^{(4)}(z)|$. Assume that M_4 is known.
- A vectorized MATLAB implementation of the function f is available and it requires σn seconds to execute when applied to an n-vector. Assume that σ is known.
- A vectorized MATLAB implementation of the function f' is available and it requires τn seconds to execute when applied to an n-vector. Assume that τ is known.

P3.2.5 Consider the quartic polynomial $q(t)$ having the form

$$q(t) = a_1 + a_2 t + a_3 t^2 + a_4 t^2(t - 1) + a_5 t^2(t - 1)^2.$$

Given scalars v_0, s_0, v_1, s_1, v_τ, and τ, our goal is to determine the a_i so that

$$q(0) = v_0 \qquad q'(0) = s_0 \qquad q(1) = v_1 \qquad q'(1) = s_1 \qquad q(\tau) = v_\tau$$

We refer to this fourth degree Hermite interpolation problem as the "H4 problem" and to q as an "H4 interpolant." Note that its value is prescribed at three points and that at two of those points we also specify its slope. Complete the following function:

```
  function A = H4(v0,s0,v1,s1,vtau,tau)
%
% Assume that the six inputs are length-n column vectors.
% A is an n-by-5 matrix with the property that if qi(t) is the quartic polynomial
%
%          qi(t) = A(i,1) + A(i,2)t + A(i,3)t^2 + A(i,4)t^2(t-1) + A(i,5)t^2(t-1)^2
%
% then qi(0) = v0(i), qi'(0) = s0(i), qi(1) = v1(i), qi'(1) = s1(i), and qi(tau(i)) = vtau(i)
% for i=1:n.
```

Your implementation should not involve any loops. Also develop a vectorized implementation for evaluation:

```
  function Y = H4Eval(A,tval)
% Assume that A is an n-by-5 matrix and that tval is a length-m row vector.
% For i=1:n, let qi(t) be the quartic polynomial
%
%    qi(t) = A(i,1) + A(i,2)t + A(i,3)t^2 + A(i,4)t^2(1-t) + A(i,5)t^2(1-t)^2.
%
% Y is an n-by-m matrix with the property that Y(i,j) = qi(tval(j)) for
% i=1:n and j=1:m.
```

To test your implementations, write a script that plots in a single window the functions $f(t)$, $q_1(t)$, and $q_2(t)$ where $f(t) = e^{-t}\sin(5t)$ and q_1 and q_2 are H4 interpolants that satisfy

$$q_1(0) = f(0) \quad q_1'(0) = f'(0) \quad q_1(1) = f(1) \quad q_1'(1) = f'(1) \quad q_1(.5) = f(.5)$$

$$q_2(1) = f(1) \quad q_2'(1) = f'(1) \quad q_2(2) = f(2) \quad q_2'(2) = f'(2) \quad q_2(1.5) = f(1.5).$$

There should be just a single call to `H4` and `H4Eval`. In the same plot window, plot q_1 across [0,1] and q_2 across [1,2]. Print the coefficients of the two interpolants.

3.3 Cubic Splines

In the piecewise cubic Hermite interpolation problem, we are given n triplets

$$(x_1, y_1, s_1), \ldots, (x_n, y_n, s_n)$$

and determine a function $C(x)$ that is piecewise cubic on the partition $x_1 < \cdots < x_n$ with the property that $C(x_i) = y_i$ and $C'(x_i) = s_i$ for $i = 1{:}n$. This interpolation strategy is subject to a number of criticisms:

- The function $C(z)$ does not have a continuous second derivative: Its display may be too crude in graphical applications, because the human eye can detect discontinuities in the second derivative.
- In other applications where C and its derivatives are part of a larger mathematical landscape, there may be difficulties if $C''(x)$ is discontinuous. For example, trouble arises if C is a distance function.
- In experimental settings where the y_i are "instrument readings," we may not have the first derivative information required by the cubic Hermite process. Indeed, the underlying function f may not be known explicitly.

These reservations prompt us to pose the *cubic spline interpolation problem*:

> Given $(x_1, y_1), \ldots, (x_n, y_n)$ with $\alpha = x_1 < \cdots < x_n = \beta$, find a piecewise cubic interpolant $S(z)$ with the property that S, S', and S'' are continuous.

The function $S(z)$ that solves this problem is a *cubic spline interpolant.* This can be accomplished by *choosing* the appropriate slope values $s_1, \ldots, s_n$.

3.3.1 Continuity at the Interior Breakpoints

Assume that $S(z)$ is the cubic Hermite interpolant of the data (x_i, y_i, s_i) for $i = 1{:}n$. We ask the following question: Is it possible to choose $s_1, \ldots, s_n$ so that the second derivative of S is continuous? Let us look at what happens to S'' at each of the "interior" breakpoints $x_2, \ldots, x_{n-1}$. To the left of x_{i+1}, $S(z)$ is defined by the local cubic

$$q_i(z) = y_i + s_i(z - x_i) + \frac{y_i' - s_i}{\Delta x_i}(z - x_i)^2 + \frac{s_i + s_{i+1} - 2y_i'}{(\Delta x_i)^2}(z - x_i)^2(z - x_{i+1}),$$

where $y_i' = (y_{i+1} - y_i)/(x_{i+1} - x_i)$ and $\Delta x_i = x_{i+1} - x_i$. The second derivative of this local cubic is given by

$$q_i''(z) \;=\; 2\frac{y_i' - s_i}{\Delta x_i} \;+\; \frac{s_i + s_{i+1} - 2y_i'}{(\Delta x_i)^2}\left[4(z - x_i) + 2(z - x_{i+1})\right]. \qquad (3.1)$$

Likewise, to the right of x_{i+1} the piecewise cubic $C(z)$ is defined by

$$q_{i+1}(z) = y_{i+1} + s_{i+1}(z - x_{i+1}) + \frac{y_{i+1}' - s_{i+1}}{\Delta x_{i+1}}(z - x_{i+1})^2 + \frac{s_{i+1} + s_{i+2} - 2y_{i+1}'}{(\Delta x_{i+1})^2}(z - x_{i+1})^2(z - x_{i+2}).$$

The second derivative of this local cubic is given by

$$q_{i+1}''(z) \;=\; 2\frac{y_{i+1}' - s_{i+1}}{\Delta x_{i+1}} \;+\; \frac{s_{i+1} + s_{i+2} - 2y_{i+1}'}{(\Delta x_{i+1})^2}\left[4(z - x_{i+1}) + 2(z - x_{i+2})\right]. \qquad (3.2)$$

To force second derivative continuity at x_{i+1}, we insist that

$$q_i''(x_{i+1}) = \frac{2}{\Delta x_i}(2s_{i+1} + s_i - 3y_i')$$

and

$$q''_{i+1}(x_{i+1}) = \frac{2}{\Delta x_{i+1}}(3y'_{i+1} - 2s_{i+1} - s_{i+2})$$

be equal. That is,

$$\Delta x_{i+1}s_i \;+\; 2\,(\Delta x_i + \Delta x_{i+1})\,s_{i+1} \;+\; \Delta x_i s_{i+2} \;=\; 3\left(\Delta x_{i+1}y'_i + \Delta x_i y'_{i+1}\right) \tag{3.3}$$

for $i = 1{:}n-2$. If we choose $s_1, \ldots, s_n$ to satisfy these equations, then $S''(z)$ is continuous.

Before we plunge into the resolution of these equations for general n, we acquire some intuition by examining the $n = 7$ case. The equations designated by (3.3) are as follows:

$$\begin{array}{lcrcl}
i=1 & \Rightarrow & \Delta x_2 s_1 + 2(\Delta x_1 + \Delta x_2)s_2 + \Delta x_1 s_3 & = & 3(\Delta x_2 y'_1 + \Delta x_1 y'_2) \\
i=2 & \Rightarrow & \Delta x_3 s_2 + 2(\Delta x_2 + \Delta x_3)s_3 + \Delta x_2 s_4 & = & 3(\Delta x_3 y'_2 + \Delta x_2 y'_3) \\
i=3 & \Rightarrow & \Delta x_4 s_3 + 2(\Delta x_3 + \Delta x_4)s_4 + \Delta x_3 s_5 & = & 3(\Delta x_4 y'_3 + \Delta x_3 y'_4) \\
i=4 & \Rightarrow & \Delta x_5 s_4 + 2(\Delta x_4 + \Delta x_5)s_5 + \Delta x_4 s_6 & = & 3(\Delta x_5 y'_4 + \Delta x_4 y'_5) \\
i=5 & \Rightarrow & \Delta x_6 s_5 + 2(\Delta x_5 + \Delta x_6)s_6 + \Delta x_5 s_7 & = & 3(\Delta x_6 y'_5 + \Delta x_5 y'_6).
\end{array}$$

Notice that we have five constraints and seven parameters and therefore two "degrees of freedom." If we move two of the parameters (s_1 and s_7) to the right hand side and assemble the results in matrix-vector form, then we obtain a 5-by-5 linear system

$$Ts(2{:}6) \;=\; T\begin{bmatrix} s_2 \\ s_3 \\ s_4 \\ s_5 \\ s_6 \end{bmatrix} \;=\; \begin{bmatrix} 3(\Delta x_2 y'_1 + \Delta x_1 y'_2) - \Delta x_2 s_1 \\ 3(\Delta x_3 y'_2 + \Delta x_2 y'_3) \\ 3(\Delta x_4 y'_3 + \Delta x_3 y'_4) \\ 3(\Delta x_5 y'_4 + \Delta x_4 y'_5) \\ 3(\Delta x_6 y'_5 + \Delta x_5 y'_6) - \Delta x_5 s_7 \end{bmatrix} \;=\; r,$$

where

$$T \;=\; \begin{bmatrix} 2(\Delta x_1 + \Delta x_2) & \Delta x_1 & 0 & 0 & 0 \\ \Delta x_3 & 2(\Delta x_2 + \Delta x_3) & \Delta x_2 & 0 & 0 \\ 0 & \Delta x_4 & 2(\Delta x_3 + \Delta x_4) & \Delta x_3 & 0 \\ 0 & 0 & \Delta x_5 & 2(\Delta x_4 + \Delta x_5) & \Delta x_4 \\ 0 & 0 & 0 & \Delta x_6 & 2(\Delta x_5 + \Delta x_6) \end{bmatrix}.$$

Matrices like this that are zero everywhere except on the diagonal, subdiagonal, and superdiagonal are said to be *tridiagonal.*

Different choices for the end slopes s_1 and s_n yield different cubic spline interpolants. Having defined the end slopes, the interior slopes $s(2{:}n-1)$ ar determined by solving an $(n-2)$-by-$(n-2)$ linear system. In each case that we consider here, the matrix of coefficients looks like

$$T = \begin{bmatrix} t_{11} & t_{12} & 0 & \cdots & 0 \\ \Delta x_3 & 2(\Delta x_2 + \Delta x_3) & \Delta x_2 & & \vdots \\ \vdots & \ddots & \ddots & \ddots & \vdots \\ 0 & & \Delta x_{n-2} & 2(\Delta x_{n-3} + \Delta x_{n-2}) & \Delta x_{n-3} \\ 0 & \cdots & 0 & t_{n-2,n-3} & t_{n-2,n-2} \end{bmatrix},$$

while the right-hand side r has the form

$$r = \begin{bmatrix} r_1 \\ 3(\Delta x_3 y_2' + \Delta x_2 y_3') \\ \vdots \\ 3(\Delta x_{n-2} y_{n-3}' + \Delta x_{n-3} y_{n-2}') \\ r_{n-2} \end{bmatrix}.$$

As we show in the next subsection, the values of t_{11}, t_{12}, and r_1 depend on how s_1 is chosen. The values of $t_{n-2,n-3}$, $t_{n-2,n-2}$, and r_{n-2} depend on how s_n is defined. Moreover, the T matrices that emerge can be shown to be nonsingular.

The following fragment summarizes what we have established so far about the linear system $Ts(2{:}n-1) = r$:

```
n=length(x);
Dx = diff(x);
yp = diff(y) ./ Dx;
T = zeros(n-2,n-2);
r = zeros(n-2,1);
for i=2:n-3
   T(i,i) = 2(Dx(i)+Dx(i+1));
   T(i,i-1) = Dx(i+1);
   T(i,i+1) = Dx(i);
   r(i) = 3(Dx(i+1)*yp(i) + Dx(i)*yp(i+1));
end
```

This sets up all but the first and last rows of T and all but the first and last components of r. How T and r are completed depends on the end conditions that are imposed on the spline.

3.3.2 The Complete Spline

The *complete spline* is obtained by setting $s_1 = \mu_L$ and $s_n = \mu_R$, where μ_L and μ_R are given real values. With these constraints, setting $i = 1$ and $i = n-2$ in (3.3) gives

$$\begin{aligned} \Delta x_2 \mu_L + 2(\Delta x_1 + \Delta x_2)s_2 + \Delta x_1 s_3 &= 3(\Delta x_2 y_1' + \Delta x_1 y_2') \\ \Delta x_{n-1} s_{n-2} + 2(\Delta x_{n-2} + \Delta x_{n-1})s_{n-1} + \Delta x_{n-2}\mu_R &= 3(\Delta x_{n-1} y_{n-2}' + \Delta x_{n-2} y_{n-1}'), \end{aligned}$$

and so the first and last equations are given by

$$\begin{aligned} 2(\Delta x_1 + \Delta x_2)s_2 + \Delta x_1 s_3 &= 3(\Delta x_2 y_1' + \Delta x_1 y_2') - \Delta x_2 \mu_L \\ \Delta x_{n-1} s_{n-2} + 2(\Delta x_{n-2} + \Delta x_{n-1})s_{n-1} &= 3(\Delta x_{n-1} y_{n-2}' + \Delta x_{n-2} y_{n-1}') - \Delta x_{n-2}\mu_R. \end{aligned}$$

Thus, the setting up of T and r and the resolution of s are completed with the fragment

```
T(1,1) = 2*(Dx(1) + Dx(2));
T(1,2) = Dx(1);
r(1) = 3*(Dx(2)*yp(1) + Dx(1)*yp(2)) - Dx(2)*muL;
T(n-2,n-2) = 2*(Dx(n-2) + Dx(n-1));
T(n-2,n-3) = Dx(n-1);
r(n-2) = 3*(Dx(n-1)*yp(n-2) + Dx(n-2)*yp(n-1)) - Dx(n-2)*muR;
s = [ muL; T \ r(1:n-2) ; muR];
```

assuming that `muL` and `muR` house μ_L and μ_R, respectively.

3.3.3 The Natural Spline

Instead of prescribing the slope of the spline at the endpoints, we can prescribe the value of its second derivative. In particular, if we insist that $\mu_L = q_1''(x_1)$, then from (3.1) it follows that

$$\mu_L = 2\frac{y_1' - s_1}{\Delta x_1} - 2\frac{s_1 + s_2 - 2y_1'}{\Delta x_1},$$

from which we conclude that

$$s_1 = \frac{1}{2}\left(3y_1' - s_2 - \frac{\mu_L}{2}\Delta x_1\right).$$

Substituting this result into the $i = 1$ case of (3.3) and rearranging, we obtain

$$(2\Delta x_1 + 1.5\Delta x_2)s_2 + \Delta x_1 s_3 \;=\; 1.5\Delta x_2 y_1' + 3\Delta x_1 y_2' + \frac{\mu_L}{4}\Delta x_1 \Delta x_2.$$

Likewise, by setting $\mu_R = q_{n-1}''(x_n)$, then (3.2) implies

$$\mu_R = 2\frac{y_{n-1}' - s_{n-1}}{\Delta x_{n-1}} + 4\frac{s_{n-1} + s_n - 2y_{n-1}'}{\Delta x_{n-1}},$$

from which we conclude that

$$s_n = \frac{1}{2}\left(3y_{n-1}' - s_{n-1} + \frac{\mu_R}{2}\Delta x_{n-1}\right).$$

Substituting this result into the $i = n-2$ case of (3.3) and rearranging we obtain

$$\Delta x_{n-1}s_{n-2} + (1.5\Delta x_{n-2} + 2\Delta x_{n-1})s_{n-1} \;=\; 3\Delta x_{n-1}y_{n-2}' + 1.5\Delta x_{n-2}y_{n-1}' - \frac{\mu_R}{4}\Delta x_{n-2}\Delta x_{n-1}.$$

Thus, the setting up of T and r and the resolution of s are completed with the fragment

```
T(1,1) = 2*Dx(1) + 1.5*Dx(2);
T(1,2) = Dx(1);
r(1) = 1.5*Dx(2)*yp(1) + 3*Dx(1)*yp(2)) + Dx(1)*Dx(2)*muL/4;
T(n-2,n-2) = 1.5*Dx(n-2) + 2*Dx(n-1);
T(n-2,n-3) = Dx(n-1);
r(n-2) = 3*Dx(n-1)*yp(n-2) + 1.5*Dx(n-2)*yp(n-1) -Dx(n-2)*Dx(n-1)*muR;
stilde = T \ r;
s1 = (3*yp(1) - stilde(1) - muL*Dx(1)/2)/2;
sn = (3*yp(n-1) - stilde(n-2) + muR*Dx(n-1)/2)/2;
s = [s1; stilde; sn];
```

If $\mu_L = \mu_R = 0$, then the resulting spline is called *the natural spline.*

3.3.4 The Not-a-Knot Spline

This method for prescribing the end conditions is appropriate if no endpoint derivative information is available. It produces the *not-a-knot* spline. The idea is to ensure third derivative continuity at both x_2 and x_{n-1}. Note from (3.1) that

$$q_i'''(x) = 6\frac{s_i + s_{i+1} - 2y_i'}{(\Delta x_i)^2},$$

and so $q_1'''(x_2) = q_2'''(x_2)$ says that

$$\frac{s_1 + s_2 - 2y_1'}{(\Delta x_1)^2} \;=\; \frac{s_2 + s_3 - 2y_2'}{(\Delta x_2)^2}.$$

It follows that this will be the case if we set

$$s_1 = -s_2 + 2y_1' + \left(\frac{\Delta x_1}{\Delta x_2}\right)^2, (s_2 + s_3 - 2y_2').$$

As a result of making the third derivative continuous at x_2, the cubics $q_1(x)$ and $q_2(x)$ are identical.

Likewise, $q_{n-2}'''(x_{n-1}) = q_{n-1}'''(x_{n-1})$ says that

$$\frac{s_{n-2} + s_{n-1} - 2y_{n-2}'}{(\Delta x_{n-2})^2} = \frac{s_{n-1} + s_n - 2y_{n-1}'}{(\Delta x_{n-1})^2}.$$

It follows that this will be the case if we set

$$s_n = -s_{n-1} + 2y_{n-1}' + \left(\frac{\Delta x_{n-1}}{\Delta x_{n-2}}\right)^2 (s_{n-2} + s_{n-1} - 2y_{n-2}').$$

Thus, the first and last equations for the not-a-knot spline are set up as follows:

```
q = Dx(1)*Dx(1)/Dx(2);
T(1,1)= 2*Dx(1) +Dx(2) + q;
T(1,2) = Dx(1) + q;
r(1) = Dx(2)*yp(1) + Dx(1)*yp(2)+2*yp(2)*(q+Dx(1));
q= Dx(n-1)*Dx(n-1)/Dx(n-2);
T(n-2,n-2) = 2*Dx(n-1) + Dx(n-2)+q;
T(n-2,n-3) = Dx(n-1)+q;
r(n-2) = Dx(n-1)*yp(n-2) + Dx(n-2)*yp(n-1) +2*yp(n-2)*(Dx(n-1)+q);
stilde = T\ r;
s1 = -stilde(1)+2*yp(1);
s1 = s1 + ((Dx(1)/Dx(2))^2)*(stilde(1)+stilde(2)-2*yp(2));
sn = -stilde(n-2) +2*yp(n-1);
sn=sn+((Dx(n-1)/Dx(n-2))^2)*(stilde(n-3)+stilde(n-2)-2*yp(n-2));
s=[s1;stilde;sn];
```

3.3.5 The Cubic Spline Interpolant

The function `CubicSpline` can be used to construct the cubic spline interpolant with any of the three aforementioned types of end conditions. Here is its specification:

```
  function [a,b,c,d] = CubicSpline(x,y,derivative,muL,muR)
% [a,b,c,d] = CubicSpline(x,y,derivative,muL,muR)
% Cubic spline interpolation with prescribed end conditions.
%
% x,y are column n-vectors. It is assumed that n >= 4 and x(1) < ... x(n).
% derivative is an integer (1 or 2) that specifies the order of the endpoint derivatives.
% muL and muR are the endpoint values of this derivative.
%
% a,b,c, and d are column (n-1)-vectors that define the spline S(z). On [x(i),x(i+1)],
%
%          S(z) =  a(i) + b(i)(z-x(i)) + c(i)(z-x(i))^2 + d(i)(z-x(i))^2(z-x(i+1).
%
% Usage:
%   [a,b,c,d] = CubicSpline(x,y,1,muL,muR)   S'(x(1))  = muL, S'(x(n))  = muR
%   [a,b,c,d] = CubicSpline(x,y,2,muL,muR)   S''(x(1)) = muL, S''(x(n)) = muR
%   [a,b,c,d] = CubicSpline(x,y)             S'''(z) continuous at x(2) and x(n-1)
%
```

Notice that a two-argument call is all that is required to produce the not-a-knot spline. The script `ShowSpline` examines various `CubicSpline` interpolants to the sine function.

Error bounds for the cubic spline interpolant are complicated to derive. The bounds are *not* good if the end conditions are improperly chosen. Figure 3.6 shows what can happen if the complete spline is used with end conditions that are at variance with the behavior of the function being interpolated. However, if the

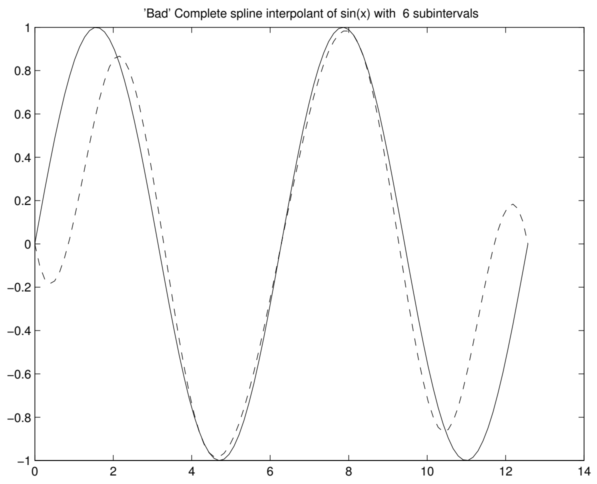


FIGURE 3.6 *Bad end conditions*

end values are properly chosen or if the not-a-knot approach is used, then the error bound has the form $M_4\bar{h}^4$ where $\bar{h}$ is the maximum subinterval length and M_4 bounds the 4th derivative of the function being interpolated. The script `ShowSplineErr` confirms this for the case of an "easy" $f(x)$. It produces the plots shown in Figure 3.7. Notice that the error is reduced by a factor of 10^4 if the subinterval length is reduced

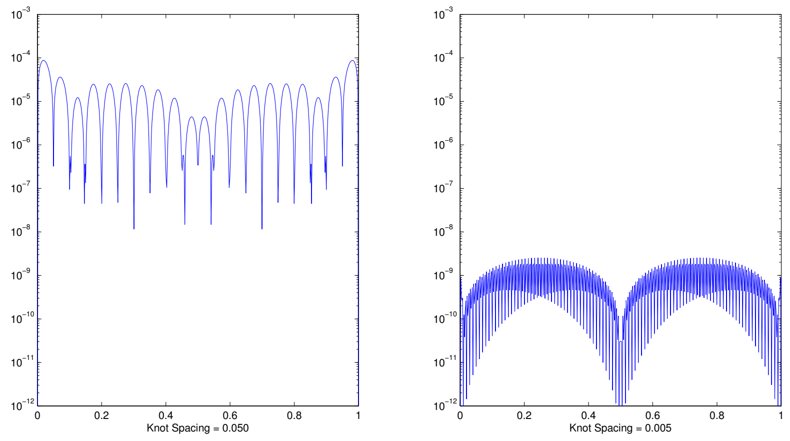


FIGURE 3.7 *Not-a-knot spline error*

by a factor of ten.

3.3.6 MATLAB Spline Tools

The MATLAB function `spline` can be used to compute not-a-knot spline interpolants. It can be called with either two or three arguments. The script

```
    z = linspace(-5,5);
    x = linspace(-5,5,9);
    y = atan(x);
    Svals = spline(x,y,z);
    plot(z,Svals);
```

illustrates a three-argument call. It plots the $n = 9$ not-a-knot spline interpolant of the function $f(x) = \arctan(x)$ across the interval $[-5, 5]$. The first two arguments in the call to `spline` specify the interpolation points that define the spline S. The spline is then evaluated at `z` with the values returned in `Svals`. Thus $S($`x(i)`$) =$ `y(i)` for `i=1:length(x)` and $S($`z(i)`$) =$ `Svals(i)` for `i=1:length(z)`.

A 2-argument call to `spline` returns what is called the *pp-representation* of the spline. This type of reference is required whenever one has to manipulate the local cubics that make up the spline. The pp-representation of a spline is different from the four-vector representation that we have been using for piecewise cubics. For one thing, it is more general because it can accommodate piecewise polynomials of arbitrary degree.

To gain a facility with MATLAB's piecewise polynomial tools, let's consider the problem of constructing the pp-representation of the derivative of a cubic spline S. In particular, let's plot S' where S is a nine-point, equally spaced, not-a-knot spline interpolant of the arctangent function across the interval $[-5, 5]$. We start by constructing the pp-representation of S:

```
    x = linspace(-5,5,9);
    y = atan(x);
    S = spline(x,y)
```

A two-argument call to `spline` such as this produces the pp-representation of the spline. The `ppval` function can be used to evaluate a piecewise polynomial in this representation:

```
    z = linspace(-5,5);
    Svals = ppval(S,z);
    plot(z,Svals)
```

The call to `ppval` returns the value of the spline at `z`. The vector `Svals` contains the values of the spline on `z`. These values are then plotted.

The derivative of the spline is a piecewise *quadratic* polynomial, and by using the functions `unmkpp` and `mkpp` we can produce its pp-representation. A call to `unmkpp` unveils the four major components of the *pp*-representation:

```
    [x,rho,L,k] = unmkpp(S)
```

The x-values are returned in `x`. The coefficients of the local polynomials are assembled in an `L`-by-`k` matrix `rho`. `L` is the number of local polynomials and `k-1` is their degree. So in our case, `x = linspace(-5,5,9)`, `L = 8`, and `k=4`. The coefficients of the i-th local cubic are stored in ith row of the `rho` matrix. In particular, the spline is defined by

$$S(z) = \rho_{i,4} + \rho_{i,3}(z - x_i) + \rho_{i,2}(z - x_i)^2 + \rho_{i,1}(z - x_i)^3$$

on the interval $[x_i, x_{i+1}]$. Thus, `rho(i,j)` is the ith local polynomial coefficient of $(x - x_i)^{k-j+1}$.

The function `mkpp` takes the breakpoints and the array of coefficients and produces the pp-representation of the piecewise polynomial so defined. Thus, to set up the *pp*-representation of the spline's derivative, we execute

```
    drho = [3*rho(:,1) 2*rho(:,2) rho(:,3)];
    dS = mkpp(x,drho);
```

The set-up of the three-column matrix `drho` follows from the observation that

$$S'(x) = \rho_{i,3} + 2\rho_{i,2}(x - x_i) + 3\rho_{i,1}(x - x_i)^2$$

on the interval $[x_i, x_{i+1}]$. Putting it all together, we obtain

```
% Script File: ShowSplineTools
% Illustrates the Matlab functions spline, ppval, mkpp, unmkpp
close all
% Set Up Data:
n = 9;
x = linspace(-5,5,n);
y = atan(x);
%   Compute the spline interpolant and its derivative:
S  = spline(x,y);
[x,rho,L,k] = unmkpp(S);
drho = [3*rho(:,1) 2*rho(:,2) rho(:,3)];
dS = mkpp(x,drho);
%   Evaluate S and dS:
z = linspace(-5,5);
Svals = ppval(S,z);
dSvals = ppval(dS,z);

%   Plot:
atanvals = atan(z);
figure
plot(z,atanvals,z,Svals,x,y,'*');
title(sprintf('n = %2.0f Spline Interpolant of atan(x)',n))
datanvals = ones(size(z))./(1 + z.*z);
figure
plot(z,datanvals,z,dSvals)
title(sprintf('Derivative of n = %2.0f Spline Interpolant of atan(x)',n))
```

Problems

P3.3.1 What can you say about the $n = 4$ not-a-knot spline interpolant of $f(x) = x^3$?

P3.3.2 Suppose $S(z)$ is the not-a-knot spline interpolant of (x_1, y_1), (x_2, y_2), (x_3, y_3), and (x_4, y_4) where it is assumed that the x_i are distinct. Suppose $p(x)$ is the cubic interpolant at same four points. Explain why $S(z) = p(z)$ for all z.

P3.3.3 Let $S(z)$ be the natural spline interpolant of z^3 at $z = -3$, $z = -1$, $z = 1$, $z = 3$. What is $S(0)$?

P3.3.4 Given $\sigma > 0$, (x_i, y_i, s_i), and $(x_{i+1}, y_{i+1}, s_{i+1})$, show how to determine a_i, b_i, c_i, and d_i so that

$$g_i(x) = a_i + b_i(x - x_i) + c_i e^{\sigma(x-x_i)} + d_i e^{-\sigma(x-x_i)}$$

satisfies $g_i(x_i) = y_i$, $g_i'(x_i) = s_i$, $g_i(x_{i+1}) = y_{i+1}$, and $g_i'(x_{i+1}) = s_{i+1}$.

P3.3.5 Another approach that can be used to make up for a lack of endpoint derivative information is to glean that information from a four-point cubic interpolant. For example, if $q_L(x)$ is the cubic interpolant of (x_1, y_1), (x_2, y_2), (x_3, y_3), and (x_4, y_4), then either of the endpoint conditions

$$\begin{aligned} q_1'(x_1) &= q_L'(x_1) \\ q_1''(x_1) &= q_L''(x_1) \end{aligned}$$

is reasonable, where $q_1(x)$ is the leftmost local cubic. Likewise, if $q_R(x)$ is the cubic interpolant of (x_{n-3}, y_{n-3}), (x_{n-2}, y_{n-2}), (x_{n-1}, y_{n-1}), and (x_n, y_n), then either of the right endpoint conditions

$$\begin{aligned} q_{n-1}'(x_n) &= q_R'(x_n) \\ q_{n-1}''(x_n) &= q_R''(x_n) \end{aligned}$$

is reasonable, where $q_{n-1}(x)$ is the rightmost local cubic.

Modify `CubicSpline` so that it invokes this strategy whenever the function call involves just three arguments, (i.e., `[a,b,c,d] = CubicSpline(x,y,derivative`.) The value of `derivative` should determine which derivative is to be matched at the endpoints.

(Its value should be 1 or 2.) Augment the script file `ShowSpline` so that it graphically depicts the splines that are produced by this method.

P3.3.6 Explain how MATLAB's spline tools can be used to compute

$$\int_{\alpha}^{\beta} [S''(x)]^2 dx,$$

where $S(x)$ is a cubic spline.

P3.3.7 Suppose $S(x)$ is a cubic spline interpolant of the data $(x_1, y_1), \ldots, (x_n, y_n)$ obtained using `spline`. Write a MATLAB function `d3 = MaxJump(S)` that returns the maximum jump in the third derivative of the spline `S` assumed to be in the *pp*-representation. Vectorize as much as possible. Use the `max` function.

P3.3.8 Write a MATLAB function `S = Convert(a,b,c,d,x)` that takes our piecewise cubic interpolant representation and converts it into pp form.

P3.3.9 Complete the following function:

```
  function [a,b,c,d] = SmallSpline(z,y)
% z is a scalar and y is 3-vector.
% a,b,c,d are column 2-vectors with the property that if
%
%     S(x) = a(1) + b(1)(x - z) + c(1)(x - z)^2 + d(1)(x - z)^3  on [z-1,z]
% and
%     S(x) = a(2) + b(2)(x - z) + c(2)(x - z)^2 + d(2)(x - z)^3  on [z,z+1]
% then
%                         (a) S(z-1) = y(1), S(z) = y(2), S(z+1) = y(3),
%                         (b) S''(z-1) = S''(z+1) = 0
%                         (c) S, S', and S'' are continuous on [z-1,z+1]
%
```

P3.3.10 In computerized typography the problem arises of finding an interpolant to points that lie on a path in the plane (e.g., a printed capital S). Such a shape cannot be represented as a function of x because it is not single valued. One approach is to number the points $(x_1, y_1), \ldots, (x_n, y_n)$ as we traverse the curve. Let d_i be the straight-line distance between (x_i, y_i) and (x_{i+1}, y_{i+1}), $i = 1{:}n-1$. Set $t_i = d_1 + \cdots + d_{i-1}$, $i = 1{:}n$. Suppose $S_x(t)$ is a spline interpolant of $(t_1, x_1), \ldots, (t_n, x_n)$ and that $S_y(t)$ is a spline interpolant of $(t_1, y_1), \ldots, (t_n, y_n)$.

It follows that the curve $\Lambda = \{(S_x(t), S_y(t)) : t_1 \le t \le t_n\}$ is smooth and passes through the n points. Write a MATLAB function `[xi,yi] = SplineInPlane(x,y,m)` that returns in `xi(1:m)` and `yi(1:m)` the x-y coordinates of m points on the curve Λ. Use the MATLAB `Spline` function to determine the splines $S_x(t)$ and $S_y(t)$.

To test `SplineInPlane` write a script that solicits an arbitrary number of points from the plot window using `ginput`. It should echo your mouseclicks by placing an asterisk at each point. After all the points are acquired it should compute the splines S_x and S_y defined above and then plot the curve Λ. Use `hold on` so that the asterisks are also displayed.

Submit listings and sample output showing a personally designed letter "S". The number of input points used is up to you.

P3.3.11 Let $S(x)$ be the not-a-knot cubic spline interpolant of (0,0), (1,1), (2,8), (3,27). Explain why $S(3/2) = (3/2)^3$.

P3.3.12 Suppose `x` and `y` are column n-vectors with $x_1 < x_2 < \cdots < x_n$. If `z` is a column m-vector, then `sval = spline(x,y,z)` is a column m-vector with the property that `sval(i)` $= S(z_i)$, where S the not-a-knot spline interpolant of $(x_1, y_1), \ldots, (x_n, y_n)$.

Let

$S_1(x)$	be the not-a-knot spline interpolant of sin(x) at	$x_i = (i-1)/10$, $i = 1{:}21$
$S_2(x)$	be the not-a-knot spline interpolant of exp(x) at	$x_i = (i-1)/10$, $i = 1{:}21$
$S_3(x)$	be the not-a-knot spline interpolant of sin$(x) \cdot$ exp(x) at	$x_i = (i-1)/10$, $i = 1{:}21$
$S_4(x)$	be the not-a-knot spline interpolant of $2\sin(x) + 3\exp(x)$ at	$x_i = (i-1)/10$, $i = 1{:}21$

Write a vectorized MATLAB script that plots in a single window these four splines across the interval [0,2]. The plots should be based on one-hundred, equally-spaced evaluations. Avoid unnecessary function calls. You do not have to exploit any trigonometric or exponential identities.

P3.3.13 Produce a plot that shows that it is a bad idea to interpolate with the natural spline if the second derivative of the underlying function is not zero at the endpoints.

P3.3.14 Suppose $f(t)$ and its first two derivatives are defined everywhere. If f has period T (positive), then $f(t+T) = f(t)$ for all t. Consider the problem of interpolating such a function on an interval $[\tau, \tau + T]$ with a spline S having breakpoints

$$\tau = t_1 < \cdots < t_n = \tau + T.$$

It makes sense to require

$$\begin{aligned} S'(\tau) &= S'(\tau+T) \\ S''(\tau) &= S''(\tau+T), \end{aligned}$$

since $f'(\tau) = f'(\tau+T)$ and $f''(\tau) = f''(\tau+T)$. Moreover, we can then extend S periodically off the "base" interval $[\tau, \tau+T]$ and obtain a piecewise cubic interpolant that is continuous through the second derivative. **(a)** Modify `CubicSpline` so that a 3-argument call of the form

```
        [a,b,c,d] = CubicSpline(x,y,0)
```

produces the periodic spline interpolant. In other words,

$$\begin{aligned} S(x_i) &= y_i \\ S'(x_1) &= S'(x_n) \\ S''(x_1) &= S''(x_n) \end{aligned}$$

where $i = 1{:}n$ and n is the length of `x`. Test your adaptation with the function

$$f(x) = \sin(2\pi x) - .3 \cdot \cos(4\pi x) + .6 \cdot \sin(6\pi x) + .2 \cdot \cos(8\pi x)$$

by generating its periodic spline interpolant on `linspace(0,1,15)`. Print a table of the coefficients $a(1{:}14)$, $b(1{:}14)$, $c(1{:}14)$, and $d(1{:}14)$ and plot both f and the spline across $[0,1]$. **(b)** A not-a-knot spline interpolant of f across $[\tau, \tau+T]$ will in general not be periodic. However, we can make it "almost" periodic by choosing $t_2 = t_1 + \delta$ and $t_{n-1} = t_n - \delta$ for small δ. Write a function

```
    function s = Periodic(f,t1,T,n,del)
  % f is a handle that references an available function f(t) that has period T and is defined
  % everywhere. t1 is a real scalar, n is an integer >= 4, and del a positive scalar that
  % satisfies del < T/2.
  %
  % s is the pp-form of the not-a-knot spline that interpolates f at t(1),...,t(n) where
  %
  %                      t1                                  if k=1
  %                      t1 + del                            if k=2
  %            t(k)  =   t1 + del + (k-2)*(T-2*del)/(n-3)    if k=3:n-2
  %                      t1 + T-del                          if k=n-1
  %                      t1 + T                              if k=n
  %
  %     A four-parameter reference of the form s = Periodic(f,t1,T,n) should
  %     return the not-a-knot spline interpolant of f at linspace(t1,t1+T,n).
```

M-Files and References

Script Files

`ShowpwL1`	Illustrates `pwL` and `pwLeval`.
`ShowpwL2`	Compares pwLstatic and `pwLadapt`.
`ShowHermite`	Illustrates the Hermite interpolation idea.
`ShowpwC`	Illustrates `pwC` and `pwCeval`.
`ShowSpline`	Illustrates `CubicSpline`.
`ShowSplineErr`	Explores the not-a-knot spline interpolant error.
`ShowMatSplineTools`	Illustrates MATLAB spline tools.

Function Files

`Locate`	Determines the subinterval in a mesh that houses a given x-value
`pwL`	Sets up a piecewise linear interpolant.
`pwLeval`	Evaluates a piecewise linear function.
`pwLstatic`	A priori determination of a mesh for a pwL approximation.
`pwLadapt`	Dynamic determination of a mesh for a pwL approximation.
`HCubic`	Constructs the cubic Hermite interpolant.
`pwC`	Sets up a piecewise cubic Hermite interpolant.
`pwCeval`	Evaluates a piecewise cubic function.
`CubicSpline`	Constructs complete, natural, or not-a-knot spline.

References

R. Bartels, J. Beatty, and B. Barsky (1987). *An Introduction to Splines for Use in Computer Graphics and Geometric Modeling*, Morgan Kaufmann, Los Altos, CA.

C. de Boor (1978). *A Practical Guide to Splines*, Springer, Berlin.

Chapter 4

Numerical Integration

An m-point *quadrature rule* Q for the definite integral

$$I(f,a,b) = \int_a^b f(x)dx \tag{4.1}$$

is an approximation of the form

$$I_Q(f,a,b) \;=\; (b-a)\sum_{k=1}^{m} w_k f(x_k). \tag{4.2}$$

The x_k are the *abscissas* and the w_k are the *weights*. The abscissas and weights define the rule and are chosen so that $I_Q(f,a,b) \approx I(f,a,b)$. *Efficiency essentially depends upon the number of function evaluations.* This is because the time needed to evaluate f at the x_i is typically *much* greater than the time needed to form the required linear combination of function values. Thus, a six-point quadrature rule is twice as expensive as a three-point rule.

We start by presenting the the *Newton-Cotes* family of quadrature rules. These rules are derived by integrating a polynomial interpolant of the integrand $f(x)$. Composite rules based on a partition of $[a,b]$ into subintervals are then discussed in §4.2. In a composite rule, a simple rule is applied to each subintegral and the result summed. The adaptive determination of the partition with error control is presented in §4.3. The partition is determined recursively using heuristic estimates of the integrand's behavior. In §4.4 we discuss the "super accuracte" Gauss quadrature idea and also how to approach the quadrature problem using splines when the integrand is only known through a discrete set of sample points.

4.1 The Newton-Cotes Rules

One way to derive a quadrature rule Q is to integrate a polynomial approximation $p(x)$ of the integrand $f(x)$. The philosophy is that $p(x) \approx f(x)$ implies

$$\int_a^b f(x)dx \approx \int_a^b p(x)dx.$$

(See Figure 4.1.) The *Newton-Cotes* quadrature rules are obtained by integrating uniformly spaced polynomial interpolants of the integrand. The m-point Newton-Cotes rule ($m \geq 2$) is defined by

$$Q_{\text{NC}(m)} = \int_a^b p_{m-1}(x)dx, \tag{4.3}$$

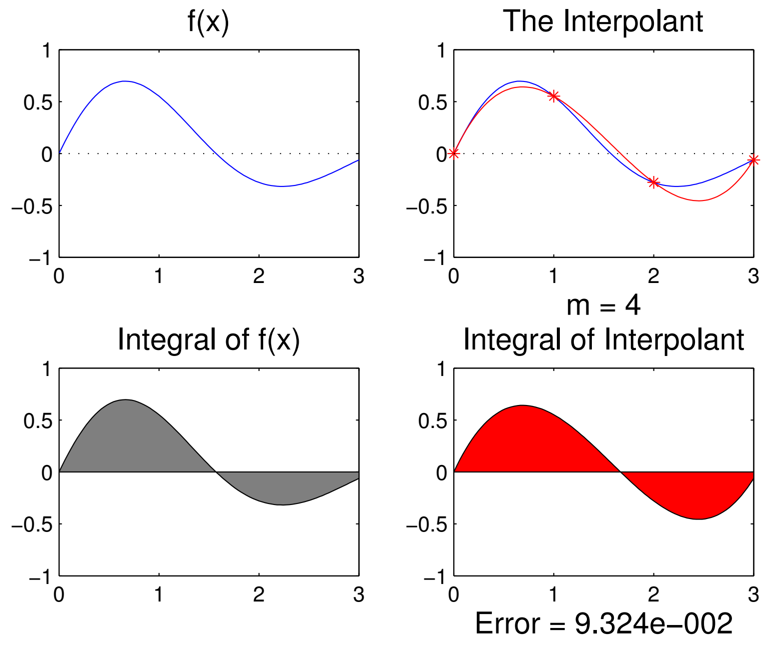


FIGURE 4.1 *The Newton-Cotes idea*

where $p_{m-1}(x)$ interpolates $f(x)$ at

$$x_i = a + \frac{i-1}{m-1}(b-a), \qquad i = 1{:}m.$$

If $m = 2$, then we obtain the *trapezoidal rule*:

$$\begin{aligned} Q_{\text{NC}(2)} &= \int_a^b \left(f(a) + \frac{f(b)-f(a)}{b-a}(x-a) \right) dx \\ &= (b-a)\left(\frac{1}{2}f(a) + \frac{1}{2}f(b) \right). \end{aligned}$$

If $m = 3$ and $c = (a+b)/2$, then we obtain the *Simpson rule*:

$$\begin{aligned} Q_{\text{NC}(3)} &= \int_a^b \left(f(a) + \frac{f(c)-f(a)}{c-a}(x-a) + \frac{\frac{f(b)-f(c)}{b-c} - \frac{f(c)-f(a)}{c-a}}{b-a}(x-a)(x-c) \right) dx \\ &= \frac{b-a}{6}\left(f(a) + 4f\left(\frac{a+b}{2}\right) + f(b) \right). \end{aligned}$$

From these low-degree examples, it appears that a linear combination of f-evaluations is obtained upon expansion of the right-hand side in (4.3).

4.1.1 Derivation

For general m, we proceed by substituting the Newton representation

$$p_{m-1}(x) = \sum_{k=1}^{m}\left(c_k \prod_{i=1}^{k-1}(x - x_i)\right)$$

into (4.3):

$$Q_{\mathrm{NC}(m)} = \int_a^b p_{m-1}(x)dx \;=\; \sum_{k=1}^{m} c_k \int_a^b \left(\prod_{i=1}^{k-1}(x - x_i)\right)dx.$$

If we set $x = a + sh$, where $h = (b-a)/(m-1)$, then this transforms to

$$Q_{\mathrm{NC}(m)} \;=\; \int_a^b p_{m-1}(x)dx \;=\; h\int_0^{m-1} p_{m-1}(a+sh)ds \;=\; \sum_{k=1}^{m} c_k h^k S_{mk},$$

where

$$S_{mk} \;=\; \int_0^{m-1}\left(\prod_{i=1}^{k-1}(s - i + 1)\right)ds.$$

The c_k are divided differences. Because of the equal spacing, they have a particularly simple form in terms of the f_i, as was shown in §2.4.1. For example,

$$\begin{aligned}
c_1 &= f_1\\
c_2 &= (f_2 - f_1)/h\\
c_3 &= (f_3 - 2f_2 + f_1)/(2h^2)\\
c_4 &= (f_4 - 3f_3 + 3f_2 - f_1)/(3!h^3).
\end{aligned}$$

Recipes for the S_{mk} can also be derived. Here are a few examples:

$$\begin{aligned}
S_{m1} &= \textstyle\int_0^{m-1} 1\cdot ds &&= (m-1)\\
S_{m2} &= \textstyle\int_0^{m-1} sds &&= (m-1)^2/2\\
S_{m3} &= \textstyle\int_0^{m-1} s(s-1)ds &&= (m-1)^2(m-5/2)/3\\
S_{m4} &= \textstyle\int_0^{m-1} s(s-1)(s-2)ds &&= (m-1)^2(m-3)^2/4
\end{aligned}$$

Using these tabulations we can readily derive the weights for any particular m-point rule. For example, if $m = 4$, then

$$S_{41} = 3 \quad S_{42} = 9/2 \quad S_{43} = 9/2 \quad S_{44} = 9/4.$$

Thus,

$$\begin{aligned}
Q_{\mathrm{NC}(4)} &= S_{41}c_1h + S_{42}c_2h^2 + S_{43}c_3h^3 + S_{44}c_4h^4\\
&= 3f_1h + \frac{9}{2}\frac{f_2 - f_1}{h}h^2 + \frac{9}{2}\frac{f_3 - 2f_2 + f_1}{2h^2}h^3 + \frac{9}{4}\frac{f_4 - 3f_3 + 3f_2 - f_1}{6h^3}h^4\\
&= \frac{3h}{8}(f_1 + 3f_2 + 3f_3 + f_4)\\
&= (b-a)(f_1 + 3f_2 + 3f_3 + f_4)/8
\end{aligned}$$

showing that [1 3 3 1]/8 is the weight vector for $Q_{\mathrm{NC}(4)}$.

4.1.2 Implementation

For convenience in subsequent computations, we "package" the Newton-Cotes weight vectors in the following function:

```
  function w = NCWeights(m)
% w = NCWeights(m)
%
% w is a column m-vector consisting of the weights for the m-point Newton-Cotes rule.
% m is an integer that satisfies 2 <= m <= 11.

if m==2
   w=[1 1]'/2;
elseif m==3
   w=[1 4 1]'/6;
elseif m==4
   w=[1 3 3 1]'/8;
elseif m==5
   w=[7 32 12 32 7]'/90;
    :
end
```

Notice that the weight vectors are symmetric about their middle in that $w(1{:}m) = w(m{:}-1{:}1)$.

Turning now to the evaluation of $Q_{\text{NC}(m)}$ itself, we see from

$$Q_{\text{NC}(m)} = (b-a)\sum_{i=1}^{m} w_i f_i \;=\; (b-a)\begin{bmatrix} w_1 & \cdots & w_m \end{bmatrix}\begin{bmatrix} f(x_1) \\ \vdots \\ f(x_m) \end{bmatrix}$$

that it is a scaled inner product of the weight vector w and the vector of function values. Therefore, we obtain

```
  function numI = QNC(f,a,b,m)
% m-point Newton-Cotes quadrature across the interval [a b].
% f is a handle that points to a function of the form f(x) where x is a
% scalar. f must be defined on [a,b] and it must return a column vector if
% x is a column vector.
% m is an integer that satisfies 2 <= m <= 11.
% numI is the m-point Newton-Cotes approximation of the integral of f from
% a to b.
w = NCweights(m);
x = linspace(a,b,m)';
fvals = f(x);
numI = (b-a)*(w'*fvals);
```

We mention that $Q_{\text{NC}(2)}$ and $Q_{\text{NC}(3)}$ are referred to as the *trapezoidal rule* and *Simpson's rule* respectively.

Let us see how well `QNC` does when it is applied to the problems

$$I_1 \;=\; \int_0^1 e^{-x}dx \;=\; 1 - e^{-1}$$

and

$$I_2 \;=\; \int_0^1 e^{-20x}dx \;=\; (1 - e^{-20})/20$$

Setting `Q1 = QNC(@(x) exp(-x),0,1,m)` and `Q2 = QNC(@(x) exp(-20*x),0,1,m)` we find

```
 m        |Q1 - I1|             |Q2 - I2|
----------------------------------------------
 2    0.0518191617571635    0.4500000011336345
 3    0.0002131211751050    0.1166969337330916
 4    0.0000950324202655    0.0754778453850014
 5    0.0000003161797660    0.0301796546189490
 6    0.0000001782491539    0.0208012561376684
 7    0.0000000003894651    0.0080385105198381
 8    0.0000000002389524    0.0056365811921616
 9    0.0000000000003593    0.0019118765020265
10    0.0000000000002303    0.0013508599157407
11    0.0000000000000003    0.0003884845483225
```

We need a theory that explains why the results for I_2 are so inferior!

4.1.3 Newton-Cotes Error

How good are the Newton-Cotes rules? Since they are based on the integration of a polynomial interpolant, the answer clearly depends on the quality of the interpolant. Here is a result for Simpson's rule:

Theorem 4 *If $f(x)$ and its first four derivatives are continuous on $[a,b]$, then*

$$\left|\int_a^b f(x)dx - Q_{\mathrm{NC}(3)}\right| \le \frac{(b-a)^5}{2880}M_4,$$

where M_4 is an upper bound on $|f^{(4)}(x)|$ on $[a,b]$.

Proof Suppose

$$p(x) = c_1 + c_2(x-a) + c_3(x-a)(x-b) + c_4(x-a)(x-b)(x-c)$$

is the Newton form of the cubic interpolant to $f(x)$ at the points a, b, c, and d. If c is the midpoint of the interval $[a,b]$, then

$$\int_a^b (c_1 + c_2(x-a) + c_3(x-a)(x-b))\,dx = Q_{\mathrm{NC}(3)},$$

because the first three terms in the expression for $p(x)$ specify the quadratic interpolant of $(a, f(a))$, $(c, f(c))$, and $(b, f(b))$, on which the three-point Newton-Cotes rule is based. By symmetry we have

$$\int_a^b (x-a)(x-b)(x-c)dx = 0$$

and so

$$\int_a^b p(x)dx = Q_{\mathrm{NC}(3)}.$$

The error in $p(x)$ is given by Theorem 2,

$$f(x) - p(x) = \frac{f^{(4)}(\eta_x)}{24}(x-a)(x-b)(x-c)(x-d)$$

and thus,

$$\int_a^b f(x)dx - Q_{\mathrm{NC}(3)} = \int_a^b \left(\frac{f^{(4)}(\eta_x)}{24}(x-a)(x-b)(x-c)(x-d)\right)dx.$$

Taking absolute values, we obtain

$$\left|\int_a^b f(x)dx - Q_{\mathrm{NC}(3)}\right| \le \frac{M_4}{24}\int_a^b |(x-a)(x-b)(x-c)(x-d)|\,dx.$$

If we set $d = c$, then $(x-a)(x-b)(x-c)(x-d)$ is always negative and it is easy to verify that

$$\int_a^b |(x-a)(x-b)(x-c)(x-d)|\, dx = \frac{(b-a)^5}{120}$$

and so

$$\left| \int_a^b f(x)dx - Q_{\text{NC}(3)} \right| \le \frac{M_4}{24}\frac{(b-a)^5}{120} = \frac{M_4}{2880}(b-a)^5. \square$$

Note that if $f(x)$ is a cubic polynomial, then $f^{(4)} = 0$ and so Simpson's rule is exact. This is somewhat surprising because the rule is based on the integration of a *quadratic* interpolant.

In general, it can be shown that

$$\int_a^b f(x)dx \;=\; Q_{\text{NC}(m)} \;+\; c_m f^{(d+1)}(\eta)\left(\frac{b-a}{m-1}\right)^{d+2}, \tag{4.4}$$

where c_m is a small constant, η is in the interval $[a,b]$, and

$$d \;=\; \begin{cases} m-1 & \text{if } m \text{ is even} \\ m & \text{if } m \text{ is odd} \end{cases}.$$

Notice that if m is odd, as in Simpson's rule, then an extra degree of accuracy results. See P4.1.3 for details.

From (4.4), we see that knowledge of $f^{(d+1)}$ is required in order to say something about the error in $Q_{\text{NC}(m)}$. For example, if $|f^{(d+1)}(x)| \le M_{d+1}$ on $[a,b]$, then

$$\left| Q_{\text{NC}(m)} - \int_a^b f(x)dx \right| \;\le\; |c_m| M_{d+1}\left(\frac{b-a}{m-1}\right)^{d+2}. \tag{4.5}$$

The following function can be used to return this upper bound given the interval $[a,b]$, m, and the appropriate derivative bound:

```
  function error = QNCError(a,b,m,M)
% The error bound for the m-point Newton-Cotes rule when applied to
% the integral from a to b of a function f(x). It is assumed that
% a<=b and 2<=m<=11. M is an upper bound for the (d+1)-st derivative of the
% function f(x) on [a,b] where d = m if m is odd, and m-1 if m is even.
if     m==2,   d=1;  c = -1/12;
elseif m==3,   d=3;  c = -1/90;
elseif m==4,   d=3;  c = -3/80;
elseif m==5,   d=5;  c = -8/945;
elseif m==6,   d=5;  c = -275/12096;
elseif m==7,   d=7;  c = -9/1400;
elseif m==8,   d=7;  c = -8183/518400;
elseif m==9,   d=9;  c = -2368/467775;
elseif m==10,  d=9;  c = -173/14620;
else           d=11; c = -1346350/326918592;
end
error = abs( c*M*((b-a)/(m-1))^(d+2));
```

From this we see that if you are contemplating an even m rule, then the $(m-1)$-point rule is probably just as good and requires one less function evaluation. The following table summarizes the error when the m-point Newton-Cotes rule is applied to

$$I \;=\; \int_0^{\pi/2} \sin(x)dx.$$

m	`QNC(@sin,0,pi/2,m)`	Actual Error	Error Bound
2	0.7853981633974483	2.146e-01	3.230e-01
3	1.0022798774922104	2.280e-03	3.321e-03
4	1.0010049233142790	1.005e-03	1.476e-03
5	0.9999915654729927	8.435e-06	1.219e-05
6	0.9999952613861667	4.739e-06	6.867e-06
7	1.0000000258372355	2.584e-08	3.714e-08
8	1.0000000158229039	1.582e-08	2.277e-08
9	0.9999999999408976	5.910e-11	8.466e-11
10	0.9999999999621675	3.783e-11	5.417e-11
11	1.0000000000001021	1.021e-13	1.460e-13

Problems

P4.1.1 Let $C(x)$ be the cubic Hermite interpolant of $f(x)$ at $x = a$ and b. Show that

$$\int_a^b C(x)dx = \frac{h}{2}(f(a) + f(b)) + \frac{h^2}{12}(f'(a) - f'(b)).$$

This is sometimes called the *corrected trapezoidal rule*. Write a function `CorrTrap(f,fp,a,b)` that computes this value. Here, `f` and `fp` are handles that reference the integrand and its derivative respectively. The error in this rule has the form $ch^4 f^{(4)}(\eta)$. Determine c (approximately) through experimentation.

P4.1.2 This problem is about the computation of the closed Newton-Cotes weights by solving an appropriate linear system. Observe that the m-point rule should compute the integral

$$\int_0^1 x^{i-1}dx = \frac{1}{i}$$

exactly for $i = 1{:}m$. For this calculation, the abscissas are given by $x_j = (j-1)/(m-1)$, $i = 1{:}m$. Thus the weights $w_1, \ldots, w_m$ satisfy

$$w_1 x_1^{i-1} + w_2 x_2^{i-1} + \cdots + w_m x_m^{i-1} = \frac{1}{i}$$

for $i = 1{:}m$. This defines a linear system whose solution is the weight vector for the m-point rule. Write a function `MyNCweights(m)` that computes the weights by setting up the preceding linear system and solving for w using the backslash operation. Compare the output of `NCweights` and `MyNCweights` for $m = 2{:}11$.

P4.1.3 (a) Suppose m is odd and that $c = (a + b)/2$. Show that $Q_{\mathrm{NC}(m)}$ is exact if applied to

$$I = \int_a^b (x - c)^k dx$$

when k is odd. (b) If $p(x)$ has degree m, then it can be written in the form $p(x) = q(x) + \alpha(x - c)^m$ where q has degree $m - 1$ and α is a scalar. Use this fact with $c = (a + b)/2$ to show that if m is odd, then $Q_{\mathrm{NC}(m)}$ is exact when applied to

$$I = \int_a^b p(x)dx.$$

P4.1.4 Augment `ShowQNCError` so that it also prints a table of errors and error bounds for the integral

$$I = \int_0^1 \frac{dx}{1 + 10x}.$$

Explain clearly the derivative bounds that are used.

4.2 Composite Rules

We will not be happy with the error bound (4.5) unless $b - a$ is sufficiently small. Fortunately, there is an easy way to organize the computation of an integral so that small-interval quadratures prevail.

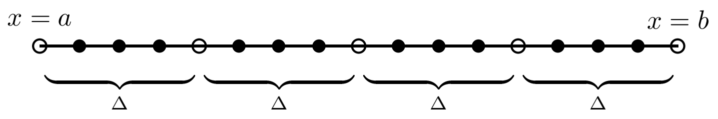


FIGURE 4.2 *Function evaluations in* $Q^{(4)}_{\mathrm{NC}(5)}$

4.2.1 Derivation

If we have a partition

$$a = z_1 < z_2 < \cdots < z_{n+1} = b,$$

then

$$\int_a^b f(x)dx \;=\; \sum_{i=1}^{n} \int_{z_i}^{z_{i+1}} f(x)dx.$$

If we apply $Q_{\mathrm{NC}(m)}$ to each of the subintegrals, then a *composite quadrature rule* based on $Q_{\mathrm{NC}(m)}$ results. For example, if $\Delta_i = z_{i+1} - z_i$ and $z_{i+1/2} = (z_i + z_{i+1})/2$, $i = 1{:}n$, then

$$Q = \sum_{i=1}^{n} \frac{\Delta_i}{6}\left(f(z_i) + 4f(z_{i+1/2}) + f(z_{i+1})\right) \tag{4.6}$$

is a composite Simpson rule. In general, if `z` houses a partition of $[a, b]$ and `f` is a handle that references a function, then

```
numI=0
for i=1:length(z)-1
   numI = numI + QNC(@f,z(i),z(i+1),m);
end
```

assigns to `numI` the composite m-point Newton-Cotes estimate of the integral based on the partition housed in `z`.

In §4.4 we will show how to automate the choice of a good partition. In the remainder of this section, we focus on composite rules that are based on uniform partitions. In these rules, $n \geq 1$,

$$z_i = a + (i-1)\Delta, \qquad \Delta = \frac{b-a}{n}$$

for $i = 1{:}n+1$, and the composite rule evaluation has the form

```
numI = 0;
Delta=(b-a)/n;
for i=1:n
   numI = numI + QNC(@f,a+(i-1)*Delta,a+i*Delta,m);
end
```

We designate the estimate produced by this quadrature rule by $Q^{(n)}_{\mathrm{NC}(m)}$. The computation is a little inefficient because it involves $n-1$ extra function evaluations and a `for`-loop. The rightmost f-evaluation in the ith call to `QNC` is the same as the leftmost f-evaluation in the $i+1$st call. Figure 4.2 depicts the situation in the four-subinterval, five-point rule case.

To avoid redundant f-evaluation and a `for`-loop with repeated function calls, it is better not to apply `QNC` to each of the n subintegrals. Instead, we precompute *all* the required function evaluations and store them in a single column vector `fval(1:n(m-1)+1)`. The linear combination that defines the composite rule is then calculated. In the preceding $Q^{(4)}_{\mathrm{NC}(5)}$ example, the 17 required function evaluations are assembled in `fval(1:17)`. If w is the weight vector for $Q_{\mathrm{NC}(5)}$, then

$$Q^{(4)}_{\mathrm{NC}(5)} = \Delta\left(w^T fval(1{:}5) + w^T fval(5{:}9) + w^T fval(9{:}13) + w^T fval(13{:}17)\right).$$

From this we conclude that $Q^{(n)}_{\text{NC}(m)}$ is a summation of n inner products, each of which involves the weight vector w of the underlying rule and a portion of the $fval$-vector. The following function is organized around this principle:

```
  function numI = CompQNC(f,a,b,m,n)
% Composite Newton-Cotes rule for the integral of f from a to b.
% f is a handle that points to a function of the form f(x) where x is a
% scalar. f must be defined on [a,b] and it must return a column vector if x is a
% column vector.
% m is an integer that satisfies 2 <= m <= 11.
% n is a positive integer.
% numI is the composite m-point Newton-Cotes approximation of the integral of f
% from a to b with n equal length subintervals.

w = NCweights(m);
x = linspace(a,b,n*(m-1)+1)';
f = f(x);
numI = 0; first = 1; last = m;
for i=1:n
   %Add in the inner product for the i-th subintegral.
   numI = numI + w'*f(first:last);
   first = last;
   last = last+m-1;
end
numI = Delta*numI;
```

4.2.2 Error

Let us examine the error. Suppose Q_i is the m-point Newton-Cotes estimate of the ith subintegral. If this rule is exact for polynomials of degree d, then using (4.4) we obtain

$$\int_a^b f(x)dx \;=\; \sum_{i=1}^n \int_{z_i}^{z_{i+1}} f(x)dx = \sum_{i=1}^n \left(Q_i + c_m f^{(d+1)}(\eta_i) \left(\frac{z_{i+1}-z_i}{m-1} \right)^{d+2} \right).$$

By definition

$$Q^{(n)}_{\text{NC}(m)} = \sum_{i=1}^n Q_i$$

and

$$z_{i+1} - z_i = \Delta = \frac{b-a}{n}.$$

Moreover, it can be shown that

$$\frac{1}{n}\sum_{i=1}^n f^{(d+1)}(\eta_i) = f^{(d+1)}(\eta)$$

for some $\eta \in [a,b]$ and so

$$\int_a^b f(x)dx \;=\; Q^{(n)}_{\text{NC}(m)} + c_m \left(\frac{b-a}{n(m-1)} \right)^{d+2} n f^{(d+1)}(\eta). \tag{4.7}$$

If $|f^{(d+1)}(x)| \le M_{d+1}$ for all $x \in [a,b]$, then

$$\left| Q^{(n)}_{\text{NC}(m)} - \int_a^b f(x)dx \right| \;\le\; \left[|c_m| M_{d+1} \left(\frac{b-a}{m-1} \right)^{d+2} \right] \frac{1}{n^{d+1}}. \tag{4.8}$$

Comparing with (4.5), we see that the error in the composite rule is the error in the corresponding "simple" rule divided by n^{d+1}. Thus, with m fixed it is possible to exercise error control by choosing n sufficiently large. For example, suppose that we want to approximate the integral with a uniformly spaced composite Simpson rule so that the error is less than a prescribed tolerance *tol.* If we know that the fourth derivative of f is bounded by M_4, then we choose n so that

$$\frac{1}{90} M_4 \left(\frac{b-a}{2}\right)^5 \frac{1}{n^4} \le tol.$$

To keep the number of function evaluations as small as possible, n should be the smallest positive integer that satisfies

$$n \ge (b-a)\sqrt[4]{\frac{M_4(b-a)}{2880 \cdot tol}}.$$

The script file `ShowCompQNC` displays the error properties of the composite Newton-Cotes rules for three different integrands. (See Figure 4.3.)

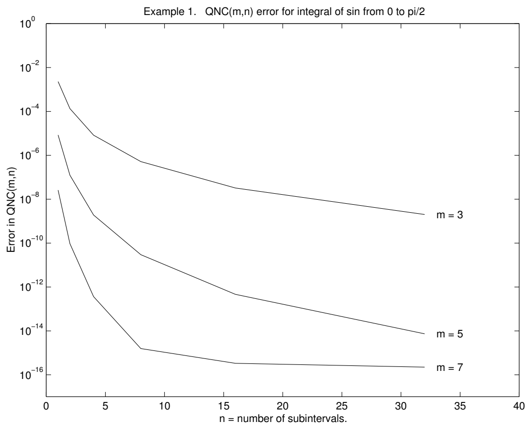


FIGURE 4.3 *Error in composite Newton-Cotes rules*

Problems

P4.2.1 Write a function `error = CompQNCerror(a,b,m,DerBound,n)` that returns an upper bound for the error in the uniformly spaced composite m-point Newton-Cotes quadrature rule applied to the integral of $f(x)$ from a to b. Use `errNC`.

P4.2.2 Rewrite `CompQNC` so that only one call to the integrand function is required.

P4.2.3 Write a function: `n = nBest(a,b,m,DerBound,tol)` that returns an integer n such that the error bound for $Q^{(n)}_{\mathrm{NC}(m)}$ is less than *tol.*

P4.2.4 Let $C(x)$ be the piecewise cubic Hermite interpolant of of $f(x)$ on $[a,b]$. Develop a uniformly spaced composite rule based on this interpolant.

P4.2.5 What can you say about the *approximate* value of T_2/T_1, where T_1 is the time required to compute a certain integral using a composite m-point Newton-Cotes rule with n subintervals, and T_2 is the time required to compute the same integral using a composite $2m$-point Newton-Cotes rule with $10n$ subintervals.

4.3 Adaptive Quadrature

Uniformly spaced composite rules that are exact for degree d polynomials are efficient if $f^{(d+1)}$ is uniformly behaved across $[a,b]$. However, if the magnitude of this derivative varies widely across the interval of integration, then the error control process discussed in §4.2 may result in an unnecessary number of function evaluations. This is because n is determined by an interval-wide derivative bound M_{d+1}. In regions where $f^{(d+1)}$ is small compared to this value, the subintervals are (possibly) much shorter than necessary. *Adaptive quadrature* methods address this problem by "discovering" where the integrand is ill behaved and shortening the subintervals accordingly.

4.3.1 An Adaptive Newton-Cotes Procedure

To obtain a good partition of $[a, b]$, we need to be able to estimate error. That way the partition can be refined if the error is not small enough. One idea is to use two different quadrature rules. The difference between the two predicted values of the integral could be taken as a measure of their inaccuracy:

```
function numI = AdaptQNC(f,a,b,...)
   Compute the integral from a to b in two ways.  Call the values $A_1$ and $A_2$
      and assume that $A_2$ is better.
   Estimate the error in $A_2$ based on $|A_1 - A_2|$.
   If the error is sufficiently small, then
      numI = $A_2$;
   else
      mid = (a+b)/2;
      numI = AdaptQNC(f,a,mid,...)  + AdaptQNC(f,mid,b,...);
   end
```

This divide-and-conquer framework is similar to the one we developed for adaptive piecewise linear approximation.

The filling in of the details begins with the development of a method for estimating the error. Fix m and set $A_1 = Q^{(1)}_{\mathrm{NC}(m)}$ and $A_2 = Q^{(2)}_{\mathrm{NC}(m)}$. Thus A_1 is the "simple" m-point rule estimate and A_2 is the two-interval, m-point rule estimate. If these rules are exact for degree d polynomials, then it can be shown that

$$I = A_1 + \left[c_m f^{(d+1)}(\eta_1)\left(\frac{b-a}{m-1}\right)^{d+2}\right] \tag{4.9}$$

$$I = A_2 + \left[c_m f^{(d+1)}(\eta_2)\left(\frac{b-a}{m-1}\right)^{d+2}\right]\frac{1}{2^{d+1}} \tag{4.10}$$

where η_1 and η_2 are in the interval $[a, b]$. We now make the assumption $f^{(d+1)}(\eta_1) = f^{(d+1)}(\eta_2)$. This is reasonable if $f^{(d+1)}$ does not vary much on $[a, b]$. (The shorter the interval, the more likely this is to be the case.) Thus

$$I = A_1 + C$$

and

$$I = A_2 + C/2^{d+1},$$

where

$$C = \left[c_m f^{(d+1)}(\eta_1)\left(\frac{b-a}{m-1}\right)^{d+2}\right].$$

By subtracting these two equations for I from each other and solving for C, we get

$$C = \frac{A_2 - A_1}{1 - \frac{1}{2^{d+1}}}$$

and so

$$|I - A_2| \approx \frac{|A_1 - A_2|}{2^{d+1} - 1}.$$

Thus, the discrepancy between the two estimates divided by $2^{d+1} - 1$ provides a reasonable estimate of the error in A_2. If our goal is to produce an estimate of I that has absolute error *tol* or less, then the recursion may be organized as follows:

```
  function numI = AdaptQNC(f,a,b,m,tol)
% f is a handle that points to a function of the form f(x) where x is a
% scalar. f must be defined on [a,b] and it must return a column vector if x is a
% column vector.
% a,b are real scalars, m is an integer that satisfies 2 <= m <=11, and
% tol is a positive real.
% numI is a composite m-point Newton-Cotes approximation of the
% integral of f(x) from a to b, where the subinterval partition is
% determined adaptively.

% Estimates based on composite rule with 1 and 2 subintervals...
A1 = CompQNC(f,a,b,m,1);
A2 = CompQNC(f,a,b,m,2);
% The error estimate...
d = 2*floor((m-1)/2)+1;
error = (A2-A1)/(2^(d+1)-1);
% Accept of reject A2?
if abs(error) <= tol
   % A2 is acceptable
   numI = A2+error;
else
   % Subdivide the problem...
   mid = (a+b)/2;
   numI = AdaptQNC(f,a,mid,m,tol/2) + AdaptQNC(f,mid,b,m,tol/2);
end
```

If the heuristic estimate of the error is greater than *tol*, then two recursive calls are initiated to obtain estimates

$$Q_L \approx \int_a^{mid} f(x)dx = I_L$$

and

$$Q_R \approx \int_{mid}^{b} f(x)dx = I_R$$

that satisfy

$$|I_L - Q_L| \le tol/2$$

and

$$|I_R - Q_R| \le tol/2.$$

Setting $Q = Q_L + Q_R$, we see that

$$|I - Q| = |(I_L - Q_L) + (I_R - Q_R)| \le |I_L - Q_L| + |I_R - Q_R| \le (tol/2) + (tol/2) = tol.$$

Insight into the economies that are realized by the adaptive framework can be obtained by applying `AdaptQNC` to the integral of the built-in function

$$\texttt{humps}(x) \;=\; \frac{1}{0.01 + (x - 0.3)^2} \;+\; \frac{1}{0.04 + (x - 0.9)^2} \;-\; 6$$

from 0 to 1. The tables in FIGURE 4.4 and FIGURE 4.5 report on the number of required function evaluations associated with the call `AdaptQNC(@humps,0,1,m,tol)` for various choices of `m` and `tol`. These values would be much higher if we used `CompQNC(f,a,b,m,n)` to attain the same level of accuracy. This is because higher derivatives of `humps` are modest in size except near $x = .3$ and $x = .9$. To handle these "rough spots" we would need a large number of subintervals, i.e., a large value for `n` in the call to `CompQNC`.

	m = 3	m = 5	m = 7	m = 9
tol = .01	26	14	6	2
tol = .001	54	22	6	2
tol = .0001	94	30	14	10
tol = .00001	174	46	26	14

FIGURE 4.4 *Number of Scalar f-evaluations required by* `QNC(@humps,0,1,m,tol)`

	m = 3	m = 5	m = 7	m = 9
tol = .01	104	98	60	26
tol = .001	216	154	60	26
tol = .0001	376	210	140	130
tol = .00001	696	322	260	182

FIGURE 4.5 *Number of Vector f-evaluations required by* `QNC(@humps,0,1,m,tol)`

Problems

P4.3.1 The one-panel midpoint rule Q_1 for the integral

$$I = \int_a^b f(x)dx$$

is defined by

$$Q_1 \;=\; (b-a)f\left(\frac{a+b}{2}\right).$$

The two-panel midpoint rule Q_2 for I is given by

$$Q_2 \;=\; \frac{b-a}{2}\left(f\left(\frac{3a+b}{4}\right)+f\left(\frac{a+3b}{4}\right)\right).$$

Using the heuristic $|I-Q_2| \le |Q_2-Q_1|$, write an efficient MATLAB adaptive quadrature routine of the form `Adapt(f,a,b,tol,...)` that returns an estimate of I that is accurate to within the tolerance given by *tol*. You may extend the parameter list, and you may use `nargin` as required. You may ignore the possibility of infinite recursion.

P4.3.2 A number of efficiency improvements can be made to `AdaptQNC`. A casual glance at `AdaptQNC` reveals two sources of redundant function evaluations: First, each function evaluation required in the assignment to `A1` is also required in the assignment to `A2`. Second, the recursive calls could (but do not) make use of previous function evaluations. In addressing these deficiencies, you are to follow these ground rules:

- A call of the form `AdaptQNC1(@f,a,b,m,tol)` must produce the same value as a call of the form `AdaptQNC(@f,a,b,m,tol)`.
- No global variables are allowed.

To "transmit" appropriate function values in the recursive calls, you will want to design `AdaptQNC1` so that it has an "optional" sixth argument `fValues`. By making this argument optional, the same five-parameter calls at the top level are permitted.

P4.3.3 An implementation `y = MyF(x)` of the function $f(x)$ has the property that it returns $f(x_i)$ in y_i for $i = 1{:}n$ where n is the length of x. Assume that the cost of a `MyF` evaluation is constant and independent of the length of the input vector `x`. We want to compute

$$I = \int_a^b f(x)dx$$

with specified accuracy. Explain why it might be more efficient to use a composite trapezoidal rule with uniform length subintervals than an adaptive trapezoidal rule *if* we have information about the second derivative of f.

P4.3.4 Assume that `MyF` is a given implementation of the function $f(x)$ and that f has positive period T. Write an efficient MATLAB script for computing the integral

$$I = \int_a^b f(x)dx$$

with absolute error $\le 10^{-6}$. Assume that a and b are given and make effective use of the Matlab quadrature function `quad`. The absolute error is no bigger than `tol`.

P4.3.5 Let Q_n be the equal spacing composite trapezoidal rule:

$$Q_n = h\left(\frac{1}{2}f(x_1) + f(x_2) + \cdots + f(x_{n-1}) + \frac{1}{2}f(x_n)\right) \qquad h = \frac{b-a}{n-1},$$

where $x = linspace(a, b, n)$ and we assume that $n \geq 2$. Assume that there is a constant C (independent of n), such that

$$I = \int_a^b f(x)fx = Q_n + Ch^2.$$

(a) Give an expression for $|I - Q_{2n}|$ in terms of $|Q_{2n} - Q_n|$. (b) Write an *efficient* function `Q = TrapRecur(f,a,b,tol` that returns in `Q` the value of Q_{2^k+1}, where k is the smallest positive integer so that $|I - Q_{2^k+1}|$ is smaller than the given positive tolerance `tol`.

P4.3.6 Assume that the function $f(x)$ is available and define

$$\phi(z) = \int_{-z}^{z} f(x)dx.$$

Using `quad`, show how to compute an array `phiVals(1:100)` with the property that $\phi(k)$ is assigned to `phiVals(k)` for `k=1:100`.

P4.3.7 Give a solution procedure for computing

$$I = \int_a^b \left(\int_a^x f(x,y)dy\right) dx,$$

where `f(x,y)` is a given. All integrals in your method must be computed using `quad`. Clearly define the functions that are required by your method. Note: The built-in MATLAB function `dblquad` can be used to evaluate double integrals of the form

$$I = \int_a^b \int_c^d f(x,y)dxdy,$$

but this does not help in this problem.

4.4 Gauss Quadrature and Spline Quadrature

We discuss two other approaches to the quadrature problem. Gauss quadrature rules are of great interest because they optimize accuracy for a given number of f-evaluations. They also have merit in certain problems where the integrand has singularities. In situations where the function evaluations are experimentally determined, spline quadrature has a certain appeal.

4.4.1 Gauss Quadrature

In the Newton-Cotes framework, the integrand is sampled at regular intervals across $[a, b]$. In the *Gauss quadrature* framework, the abscissas are positioned in such a way that the rule is correct for polynomials of maximal degree.

A simple example clarifies the main idea. Let us try to determine weights w_1 and w_2 and abscissas x_1 and x_2 so that

$$w_1 f(x_1) + w_2 f(x_2) = \int_{-1}^{1} f(x)dx$$

for polynomials of degree 3 or less. This is plausible since there are four parameters to choose (w_1, w_2, x_1, x_2) and four constraints obtained by forcing the rule to be exact for the functions 1, x, x^2, and x^3:

$$\begin{aligned} w_1 + w_2 &= 2 \\ w_1x_1 + w_2x_2 &= 0 \\ w_1x_1^2 + w_2x_2^2 &= 2/3 \\ w_1x_1^3 + w_2x_2^3 &= 0 \end{aligned}$$

By multiplying the second equation by x_1^2 and subtracting it from the fourth equation we get $w_2x_2(x_1^2 - x_2^2) = 0$, and so $x_2 = -x_1$. It follows from the second equation that $w_1 = w_2$ and thus, from the first equation, $w_1 = w_2 = 1$. From the third equation, $x_1^2 = 1/3$ and so $x_1 = -1/\sqrt{3}$ and $x_2 = 1/\sqrt{3}$. Thus, for any $f(x)$ we have

$$\int_{-1}^{1} f(x)dx \approx f(-1/\sqrt{3}) + f(1/\sqrt{3}).$$

This is the two-point *Gauss-Legendre* rule.

The m-point Gauss-Legendre rule has the form

$$Q_{\mathrm{GL}(m)} = w_1 f(x_1) + \cdots + w_m f(x_m),$$

where the w_i and x_i are chosen to make the rule exact for polynomials of degree $2m-1$. One way to define these $2m$ parameters is by the $2m$ *nonlinear* equations

$$w_1 x_1^k + w_2 x_2^k + \cdots + w_m x_m^k = \frac{1-(-1)^{k+1}}{k+1}, \qquad k = 0{:}2m-1.$$

The kth equation is the requirement that the rule

$$w_1 f(x_1) + \cdots + w_m f(x_m) \; = \; \int_{-1}^{1} f(x)dx$$

be exact for $f(x) = x^k$. It turns out that this system has a unique solution, which we encapsulate in the following function for the cases $m = 2{:}6$:

```
  function [w,x] = GLweights(m)
% [w,x] = GLWeights(m)
% w is a column m-vector consisting of the weights for the m-point Gauss-Legendre rule.
% x is a column m-vector consisting of the abscissae.
% m is an integer that satisfies 2 <= m <= 6.
w = ones(m,1);
x = ones(m,1);
if m==2
   w(1) =  1.000000000000000; w(2) =  w(1);
   x(1) = -0.577350269189626; x(2) = -x(1);
elseif m==3
    :
end
```

The Gauss-Legendre rules

$$Q_{\mathrm{GL}(m)} = w_1 f(x_1) + \cdots + w_m f(x_m) \; \approx \; \int_{-1}^{1} f(x)dx$$

are not restrictive even though they pertain to integrals from -1 to 1. By a change of variable, we have

$$\int_a^b f(x)dx = \frac{b-a}{2}\int_{-1}^{1} g(x)dx,$$

where

$$g(x) = f\left(\frac{a+b}{2} + \frac{b-a}{2}x\right),$$

and so

$$\frac{b-a}{2}\left(w_1 f\left(\frac{a+b}{2} + \frac{b-a}{2}x_1\right) \; + \cdots + \; w_m f\left(\frac{a+b}{2} + \frac{b-a}{2}x_m\right)\right) \; \approx \; \int_a^b f(x)dx.$$

This gives

```
  function numI = QGL(f,a,b,m)
% f is a handle that references a function of the form f(x) that
% is defined on [a,b]. f should return a column vector if x is a column vector.
% a,b are real scalars.
% m is an integer that satisfies 2 <= m <= 6.
% numI is the m-point Gauss-Legendre approximation of the
% integral of f(x) from a to b.
[w,x] = GLWeights(m);
fvals = f((b-a)/2)*x + ((a+b)/2)*ones(m,1));
numI = ((b-a)/2)*w'*fvals;
```

It can be shown that

$$\left| \int_a^b f(x)dx - Q_{\text{GL}(m)} \right| \;\leq\; \frac{(b-a)^{2m+1}(m!)^4}{(2m+1)[(2m)!]^3} M_{2m},$$

where M_{2m} is a constant that bounds $|f^{2m}(x)|$ on $[a,b]$. The script file `GLvsNC` compares the $Q_{\text{NC}(m)}$ and $Q_{\text{GL}(m)}$ rules when they are applied to the integral of $\sin(x)$ from 0 to $\pi/2$:

```
  m          NC(m)                  GL(m)
--------------------------------------------------
  2   0.7853981633974483    0.9984726134041148
  3   1.0022798774922104    1.0000081215555008
  4   1.0010049233142790    0.9999999771971151
  5   0.9999915654729927    1.0000000000395670
  6   0.9999952613861668    0.9999999999999533
```

Notice that for this particularly easy problem, $Q_{\text{GL}(m)}$ has approximately the accuracy of $Q_{\text{NC}(2m)}$.

It is possible to formulate an adaptive quadrature procedure that is based on a Gauss-Legendre rule. However, the "weird" location of the abscissae creates a problem. The f-evaluations that are required when we apply an m-point rule across $[a,b]$ are not shared by the m-point rules applied to the half-interval problems. The *Gauss-Kronrod* framework circumvents this problem. The basic idea is to work with a pair of rules that share f-evaluations. The (15,7) Gauss-Kronrod procedure, works with a 15-point rule

$$\int_{-1}^{1} f(x)dx \;\approx\; Q_{\text{GK}(15)} \;=\; \sum_{k=1}^{15} \omega_k^{(15)} f(x_k^{(15)})$$

and a 7-point rule,

$$\int_{-1}^{1} f(x)dx \;\approx\; Q_{\text{GK}(7)} \;=\; \sum_{k=1}^{7} \omega_k^{(7)} f(x_k^{(7)}).$$

The key connection between $x^{(15)}$ and $x^{(7)}$ is this:

$$x^{(7)} \;=\; x^{(15)}(2{:}2{:}15).$$

See FIGURE 4.x. Moreover, there is a heuristic argument that says

$$\left| \int_{-1}^{1} f(x)dx - Q_{\text{GK}(15)} \right| \;\approx\; 200|Q_{\text{GK}(15)} \;-\; Q_{\text{GK}(7)}|^{1.5}. \qquad (4.11)$$

The demo function `ShowGK` affirms this result.

One can formulate an adaptive procedure based on these two rules that use these facts. We compute $Q_{\text{GK}(15)}$ and get $Q_{\text{GK}(7)}$ "for free" because of the shared f-evaluations. If the discrepancy between the two rules is too large, then we subdivide the problem and repeat the process on each half-interval. The MATLAB procedure `quadgk` is based on this idea.

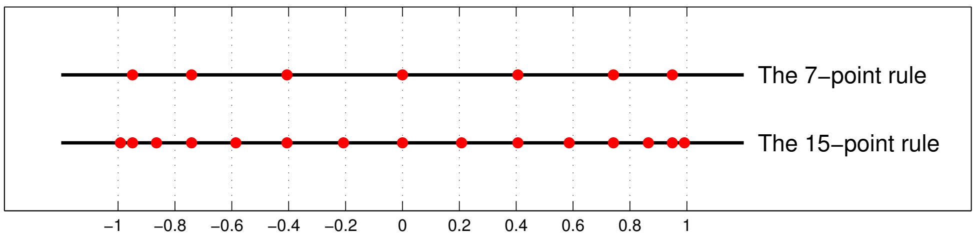


FIGURE 4.6 *Abscissa location for the (15,7) Gauss-Kronrod Pair*

Problems

P4.4.1 If $Q_{\mathrm{GL}(m)}$ is the m-point Gauss-Legendre estimate for

$$I = \int_a^b f(x)dx,$$

then it can be shown that

$$|I - Q_{\mathrm{GL}(m)}| \;\le\; \frac{(b-a)^{2m+1}(m!)^4}{(2m+1)[(2m)!]^3}M_{2m} \;\equiv\; E_m,$$

where the constant M_{2m} satisfies $|f^{(2m)}(x)| \le M_{2m}$ for all $x \in [a,b]$. The following questions apply to the case when $f(x) = e^{cx}$, where $c > 0$. Assume that $a < b$. (a) Give a good choice for M_{2m}. (b) Give an expression for E_{m+1}/E_m. (c) Write a MATLAB script that determines the smallest positive integer m so that E_m is less than `tol`.

P4.4.2 Write a function `numI = CompQGL(f,a,b,m,n)` that approximates the integral of a function from `a` to `b` by applying the m-point Gauss-Legendre rule on n equal-length subintervals of $[a,b]$.

P4.4.3 Develop an addaptive quadrature procedure `numI = AdapkGK(f,a,b,tol)` that is based on the (15,7) Gauss-Kronrod pair and the error heuristic (4.11).

4.4.2 Spline Quadrature

Suppose $S(x)$ is a cubic spline interpolant of (x_i, y_i), $i = 1{:}n$ and that we wish to compute

$$I = \int_{x_1}^{x_n} S(x)dx.$$

If the ith local cubic is represented by

$$q_i(x) = \rho_{i4} + \rho_{i,3}(x-x_i) + \rho_{i,2}(x-x_i)^2 + \rho_{i,1}(x-x_i)^3,$$

then

$$\int_{x_i}^{x_{i+1}} q_i(x)dx = \rho_{i,4}h_i + \frac{\rho_{i,3}}{2}h_i^2 + \frac{\rho_{i,2}}{3}h_i^3 + \frac{\rho_{i,1}}{4}h_i^4,$$

where $h_i = x_{i+1} - x_i$. By summing these quantities from $i = 1{:}n-1$, we obtain the sought-after spline integral:

```
  function numI = SplineQ(x,y)
% Integrates the spline interpolant of the data specified by the
% column n-vectors x and y. It is a assumed that x(1) < ... < x(n)
% and that the spline is produced by the Matlab function spline.
% The integral is from x(1) to x(n).
```

```
S = spline(x,y);
[x,rho,L,k] = unmkpp(S);
sum = 0;
for i=1:L
   % Add in the integral from x(i) to x(i+1).
   h = x(i+1)-x(i);
   subI = h*(((rho(i,1)*h/4 + rho(i,2)/3)*h + rho(i,3)/2)*h + rho(i,4));
   sum = sum + subI;
end
numI = sum;
```

The script file `ShowSplineQ` uses this function to produce the following estimates for the integral of sine from 0 to $\pi/2$:

```
   m     Spline Quadrature
----------------------------
   5    1.0001345849741938
  50    0.9999999990552404
 500    0.9999999999998678
```

Here, the spline interpolates the sine function at `x = linspace(0,pi/2,m)`.

Problems

P4.4.4 Modify `SplineQ` so that a four-argument call `SplineQ(x,y,a,b)` returns the integral of the spline interpolant from a to b. Assume that $x_1 \le a \le b \le x_n$.

P4.4.5 Let $a(t)$ denote the acceleration of an object at time t. If v_0 is the object's velocity at $t = 0$, then the velocity at time t is prescribed by

$$v(t) = v_0 + \int_0^t a(\tau)d\tau.$$

Likewise, if x_0 is the position at $t = 0$, then the position at time t is given by

$$x(t) = x_0 + \int_0^t v(\tau)d\tau.$$

Now suppose that we have snapshots $a(t_i)$ of the acceleration at times t_i, $i = 1{:}m$, $t_1 = 0$. Assume that we know the initial position x_0 and velocity v_0. Our goal is to estimate position from this data. Spline quadrature will be used to approximate the preceding integrals. Let $S_a(t)$ be the not-a-knot spline interpolant of the acceleration data $(t_i, a(t_i))$, $i = 1{:}m$, and define

$$\tilde{v}(t) = v_0 + \int_0^t S_a(\tau)d\tau.$$

Let $S_v(t)$ be the not-a-knot spline interpolant of the data $(t_i, \tilde{v}(t_i))$, $i = 1{:}m$, and define

$$\tilde{x}(t) = x_0 + \int_0^t S_v(\tau)d\tau.$$

The spline interpolant $S_x(t)$ of the data $(t_i, \tilde{x}(t_i))$ is then an approximation of the true position. Write a function

```
  function Sx = PosVel(a,t,x0,v0)}
%
% t is an m-vector of equally spaced time values with t(1) = 0, m>=2.
% a is an m-vector of accelerations, a(i) = acceleration at time t(i).
% x0 and v0 are the  position and velocity  at t=0
%
% Sx the pp-representation of a spline that approximates position.
```

Try it out on the data `t = linspace(0,50,500)`, with $a(t) = 10e^{-t/25}\sin(t)$. However, before you turn the `a` vector over to `PosVel`, contaminate it with noise: `a = a + .01*randn(size(a))`. Produce a plot of the exact and estimated positions across [0,50] and a separate plot of $x(t) - S_x(t)$ across $[0, 50]$. Also print the value of $S_x(t)$ at $t = 50$. Repeat with $m = 50$ instead of 500. Use the MATLAB `spline` function.

P4.4.6 Assume that we have a vectorized implementation `f.m` of a positive-valued function $f(x)$ and that `x` is a given column n-vector with $x_1 < ... < x_n$. (a) Write a Matlab fragment that sets up a column n-vector `q` with the property that

$$\left| q_i - \int_{x_1}^{x_i} f(x)dx \right| \le \texttt{tol}$$

for $i = 1{:}n$. Assume that `tol` is a given positive tolerance. Make effective use of `quad`. (By setting the relative error tolerance to zero, `quad` will return an approximation of the integral that satisfies the absolute error tolerance.) (b) Assume that the array `q` has been successfully computed in (a). Making effective use of `spline`, `ppval`, and the idea of inverse interpolation, show how to estimate x_* so that

$$\int_{x_1}^{x_*} f(x) = \frac{1}{2}\int_{x_1}^{x_n} f(x)dx.$$

P4.4.7 Let $(x_1, y_1), \ldots, (x_n, y_n)$ be given points in the plane. Let d_i be the straight-line distance between (x_i, y_i) and (x_{i+1}, y_{i+1}), $i = 1{:}n-1$. Set $t_i = d_1 + \cdots + d_{i-1}$, $i = 1{:}n$. Suppose $S_x(t)$ is a spline interpolant of $(t_1, x_1), \ldots, (t_n, x_n)$ and that $S_y(t)$ is a spline interpolant of $(t_1, y_1), \ldots, (t_n, y_n)$. It follows that the curve $\Lambda = \{(S_x(t), S_y(t)) : t_1 \le t \le t_n\}$ is smooth and passes through the n points. Write a Matlab function `[Sx,Sy,L] = Arc(x,y)` that returns the two splines interpolants (in pp-form) and the length of Λ, i.e.,

$$L = \int_{t_1}^{t_n} \sqrt{[S_x'(t)]^2 + [S_y'(t)]^2}dt.$$

Use `quad` for the integral with the default tolerance. You will have to set up an integrand function that accesses the piecewise quadratic functions $S_x'(t)$ and $S_y'(t)$. Write a script that displays the curve Λ where the input points are prescribed by

```
x = [ 3  2  1  2  4  5  4  3  2  4  5  5  3];
y = [ 7  6  5  4  3  2  1  1  2  4  5  6  7];
```

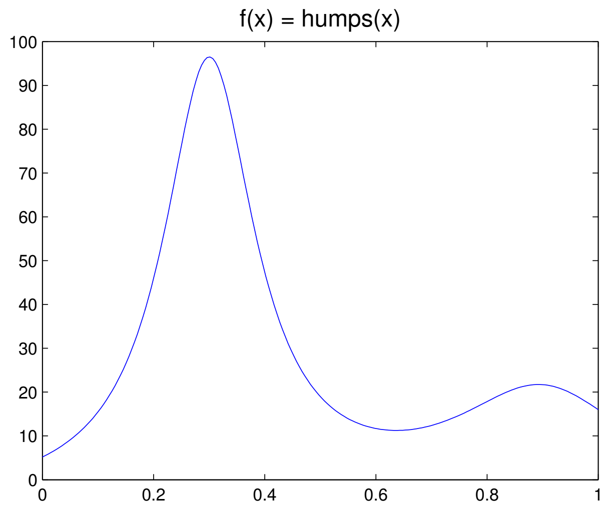
Print the curve length in the title of the plot.

4.5 Matlab's Quadrature Tools

Consider the function $f(x) = \texttt{humps}(x)$ where `humps` is the built-in Matlab function

$$\texttt{humps}(x) = \frac{1}{(x-.3)^2+.01} + \frac{1}{(x-.9)^2+.04} - 6.$$

This function's higher derivatives are large near $x = .3$ and $x = .9$:



The function `quad` can be used to approximate the integral of this function from 0 to 1:

```
>>   Q = quad(@humps,0,1)
     Q = 29.858326128427638
```

The number of f-evaluations can be obtained by supplying a second output parameter:

```
>>   [Q,fevals] = quad(@humps,0,1)
     Q = 29.858326128427638
     fevals = 145
```

Unless it is told otherwise, `quad` aims to compute the required integral with absolute error bounded by .000001. The error tolerance can be modified:

```
>>  [Q,fevals] = quad(@humps,0,1,10^-12)
    Q = 29.858325395498067
    fevals = 2321
```

The function `quad` implements an adaptive version of the *composite Simpson rule.* See §4.4. If high accuracy is required, then it is sometimes more economical to use the MATLAB quadrature function `quadl`:

```
>>  [Q,fevals] = quadl(@humps,0,1,10^-12)
    Q = 29.858325395498671
    fevals = 1608
```

The function `ShowQUADs(f,a,b)` approximates $I(f,a,b)$ and can be used to experiment with these two quadrature procedures for various choices of error tolerance. `ShowQUADs(@sin,0,pi)` tells us that `quadl` is to be preferred for very smooth integrands like $f(x) = \sin(x)$:

```
                       quad                          quadl
   tol      Approximation   f-evals      Approximation   f-evals
------------------------------------------------------------------
1.0e-003    1.999993496535     13        1.999999977471     18
1.0e-006    1.999999996398     33        1.999999977471     18
1.0e-009    1.999999999999    129        2.000000000000     48
1.0e-012    2.000000000000    497        2.000000000000     48
```

On the other hand, `ShowQuads(@(x) sin(1./x),.01,1)` reveals that for nasty integrands like $\sin(1/x)$ it is better to use a low-order rule like `quad`, especially for modest tolerances:

```
                      quad                          quadl
   tol      Approximation   f-evals      Approximation   f-evals
------------------------------------------------------------------
1.0e-003    0.463673444706     25        0.504011796906    138
1.0e-006    0.504041285733    237        0.503981893171    558
1.0e-009    0.503981892714    981        0.503981893175   1338
1.0e-012    0.503981893175   3985        0.503981893175   3648
```

The function `quadgk` offers greater control over error (absolute or relative) and can report back an estimate of the error if required. A call of the form

```
[Q,est] = quadgk(@f,a,b,'AbsTol',tol1,'RelTol',tol2)
```

attempts to return a value in `Q` that satisfies

$$|I(f,a,b) - \mathtt{Q}| \;\leq\; \max\{\mathtt{AbsTol},\mathtt{RelTol}\}.$$

If relative error is critical, then set `tol1`=0. If absolute error is the concern, set `tol2` = 0. In either case, the estimate returned in `est` is an estimate of the absolute error. If `quadgk` spots a problem with its error control, then it may suggest an increase in the value of `MaxIntervalCount` which permits the procedure to get a more accurate answer by evaluating f and more points. In this case you can try again with a response of t he form

```
        [Q,est] = quadgk(@f,a,b,'AbsTol',tol1,'RelTol',tol2,'MaxIntervalCount',BiggerValue)
```

Here are some results when `quadgk` is used to compute

$$I = \int_0^1 100\sin\left(\frac{1}{x}\right)dx$$

with `BiggerValue = 100000`:

```
  Result Via       Error         AbsTol  RelTol
    quadgk        Estimate
------------------------------------------------
  50.40654795    0.00061442      0.0010  0.0000
  50.40465490    0.03100950      0.0000  0.0010
  50.40670252    0.00007224      0.0001  0.0000
  50.40658290    0.00430233      0.0000  0.0001
```

In some cases, `quadgk` can handle endpoint singularities. For example,

```
   Q = quadgk(@(x) 1./sqrt(x),0,1)
   Q = 1.999999999999763
```

The procedure can also accommodate infinite endpoints as in

$$I \;=\; \frac{1}{\sqrt{2\pi}}\int_{-\infty}^{+\infty} e^{-(x-\mu)^2/(2\sigma^2)}\,dx.$$

Thus,

```
  mu = 1;
  sigma = 3;
  Q = quadgk(@(x) exp(-((x-mu).^2/(2*sigma^2)))/(sigma*sqrt(2*pi)),-inf,inf)
  Q = 1.000000000146726
```

thereby affirming that the area under the normal distribution $N(\mu,\sigma)$ equals one.

Problems

P4.5.1 Consider the function

$$I(\alpha) \;=\; (2+\sin(10\alpha))\int_0^2 x^\alpha \sin\left(\frac{\alpha}{2-x}\right)dx$$

Write a script that confirms the fact that

$$\max_{0\le\alpha\le5} \;I(\alpha) \;=\; I(.7859336743...)$$

Make effective use of MATLAB's quadrature software.

P4.5.2 It turns out that

$$\lim_{\epsilon\to0}\int_\epsilon^1 \frac{1}{x}\cdot\cos\left(\frac{\ln(x)}{x}\right)dx \;=\; .3233674316...$$

Write the most efficient script you can that confirms this result. Make effective use of MATLAB's quadrature software.

Script Files

`ShowNCError`	Illustrates `NCerror`.
`ShowCompQNC`	Illustrates `CompQNC` on three examples.
`ShowAdapts`	Illustrates `AdaptQNC`.
`GLvsNC`	Compares Gauss-Legendre and Newton-Cotes rules.
`ShowSplineQ`	Illustrates `SplineQ`.
`ShowGK`	Illustrates the (15,7) Gauss-Kronrod rule.

Function Files

`ShowQuads`	Illustrates `quad`, `quadl`, and `quadgk`.
`ShowNCIdea`	Displays the idea behind the Newton-Cotes rules.
`NCWeights`	Constructs the Newton-Cotes weight vector.
`QNC`	The simple Newton-Cotes rule.
`NCError`	Error in the simple Newton-Cotes rule.
`CompQNC`	Equally-spaced, composite Newton-Cotes rule.
`AdaptQNC`	Adaptive Newton-Cotes quadrature.
`SpecHumps`	The humps function with function call counters.
`GLWeights`	Constructs the Gauss-Legendre weight vector.
`QGL`	The simple Gauss-Legendre rule.
`SplineQ`	Spline quadrature.

References

P. Davis and P. Rabinowitz (1984). *Methods of Numerical Integration, 2nd Ed.*, Academic Press, New York.

G.H. Golub and J.M. Ortega (1993). *Scientific Computing: An Introduction with Parallel Computing,* Academic Press, Boston.

A. Stroud (1972). *Approximate Calculation of Multiple Integrals*, Prentice Hall, Englewood Cliffs, NJ.

Chapter 9

The Initial Value Problem

The goal in the *initial value problem* (IVP) is to find a function $y(t)$ given its value at some initial time t_0 and a recipe $f(t, y)$ for its slope:

$$y'(t) = f(t, y(t)), \qquad y(t_0) = y_0.$$

In applications we may want to plot an approximation to $y(t)$ over a designated interval of interest $[t_0, t_{max}]$ in an effort to discover qualitative properties of the solution. Or we may require a highly accurate estimate of $y(t)$ at some single, prescribed value $t = T$.

The methods we develop produce a sequence of solution snapshots $(t_1, y_1), (t_2, y_2), \ldots$ that are regarded as approximations to $(t_1, y(t_1), (t_2, y(t_2))$, etc. All we have at our disposal is the "slope function" $f(t, y)$, best thought of as a Matlab function `f(t,y)`, that can be called whenever we need information about where $y(t)$ is "headed." IVP solvers differ in how they use the slope function.

In §9.1 we use the Euler methods to introduce the basic ideas associated with approximate IVP solving: discretization, local error, global error, stability, etc. In practice the IVP usually involves a vector of unknown functions, and the treatment of such problems is also covered in §9.1. In this setting the given slope function $f(t, y)$ is a vector of scalar slope functions, and its evaluation tells us how each component in the unknown $y(t)$ vector is changing with t.

The Runge-Kutta and Adams methods are then presented in §9.2 and §9.3 together with the built-in Matlab IVP solvers `ode23` and `ode45`. We also discuss stepsize control, a topic of great practical importance and another occasion to show off the role of calculus-based heuristics in scientific computing.

Quality software for the IVP is very complex. Years of research and development stand behind codes like `ode23` and `ode45`. The implementations that we develop in this chapter are designed to build intuition and, if anything, are just the first step in the long journey from textbook formula to production software.

9.1 Basic Concepts

A "family" of functions generally satisfies a differential equation of the form $y'(t) = f(t, y)$. The initial condition $y(t_0) = y_0$ singles out one of these family members for the solution to the IVP. For example, functions of the form $y(t) = ce^{-5t}$ satisfy $y'(t) = -5y(t)$. If we stipulate that $y(0) = 1$, then $y(t) = e^{-5t}$ is the unique solution to the IVP. (See Figure 9.1.) Our goal is to produce a sequence of points (t_i, y_i) that reasonably track the solution curve as time evolves. The Euler methods that we develop in this section organize this tracking process around a linear model.

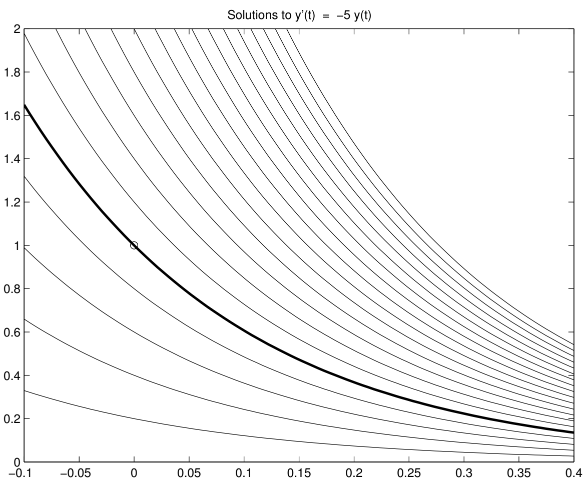


FIGURE 9.1 *Solution curves*

9.1.1 Derivation of the Euler Method

From the initial condition, we know that (t_0, y_0) is on the solution curve. At this point the slope of the solution is computable via the function f:

$$f_0 = f(t_0, y_0).$$

To estimate $y(t)$ at some future time $t_1 = t_0 + h_0$ we consider the following Taylor expansion:

$$y(t_0 + h_0) \approx y(t_0) + h_0 y'(t_0) = y_0 + h_0 f(t_0, y_0).$$

This suggests that we use

$$y_1 = y_0 + h_0 f(t_0, y_0)$$

as our approximation to the solution at time t_1. The parameter $h_0 > 0$ is the *step*, and it can be said that with the production of y_1 we have "integrated the IVP forward" to $t = t_1$.

With $y_1 \approx y(t_1)$ in hand, we try to push our knowledge of the solution one step further into the future. Let h_1 be the next step. A Taylor expansion about $t = t_1$ says that

$$y(t_1 + h_1) \approx y(t_1) + h_1 y'(t_1) = y(t_1) + h_1 f(t_1, y(t_1)).$$

Note that in this case the right-hand side is not computable because we do not know the exact solution at $t = t_1$. However, if we are willing to use the approximations

$$y_1 \approx y(t_1)$$

and

$$f_1 = f(t_1, y_1) \approx f(t_1, y(t_1)),$$

then at time $t_2 = t_1 + h_1$ we have

$$y(t_2) \approx y_2 = y_1 + h_1 f_1.$$

The pattern is now clear. At each step we evaluate f at the current approximate solution point (t_n, y_n) and then use that slope information to get y_{n+1}. The key equation is

$$y_{n+1} = y_n + h_n f(t_n, y_n),$$

and its repeated application defines the *Euler method*:

```
$n = 0$
Repeat:
    $f_n = f(t_n, y_n)$
    Determine the step $h_n > 0$ and set $t_{n+1} = t_n + h_n$.
    $y_{n+1} = y_n + h_n f_n$ .
    $n = n + 1$
```

The script file `ShowEuler` solicits the time steps interactively and applies the Euler method to the problem $y' = -5y,\ y(0) = 1$. (See Figure 9.2.) The determination of the step size is crucial.

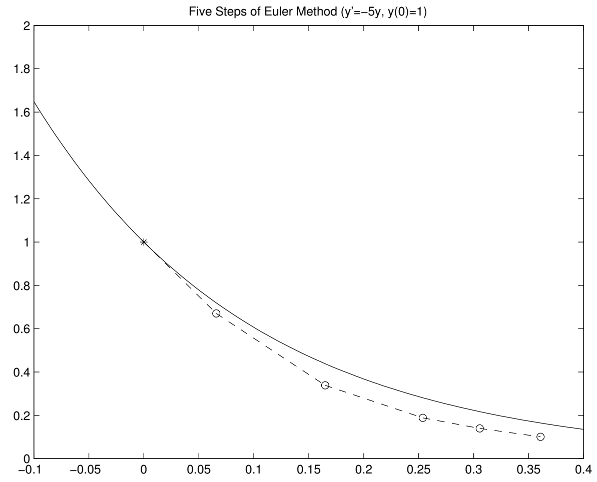


FIGURE 9.2 *Five steps of Euler's method*

Our intuition says that we can control error by choosing h_n appropriately. Accuracy should increase with shorter steps. On the other hand, shorter steps mean more f-evaluations as we integrate across the interval of interest. As in the quadrature problem and the nonlinear equation-solving problem, the number of f-evaluations usually determines execution time, and the efficiency analysis of any IVP method must include a tabulation of this statistic. The basic game to be played is to get the required snapshots of $y(t)$ with sufficient accuracy, evaluating $f(t, y)$ as infrequently as possible. To see what we are up against, we need to understand how the errors in the local model compound as we integrate across the time interval of interest.

9.1.2 Local Error, Global Error, and Stability

Assume in the Euler method that y_{n-1} is exact and let $h = h_{n-1}$. By subtracting $y_n = y_{n-1} + hf_{n-1}$ from the Taylor expansion

$$y(t_n) = y_{n-1} + hy'(t_{n-1}) + \frac{h^2}{2}y^{(2)}(\eta), \qquad \eta \in [t_{n-1}, t_n],$$

we find that

$$y(t_n) - y_n = \frac{h^2}{2}y^{(2)}(\eta).$$

This is called the *local truncation error* (LTE) In general, the LTE for an IVP method is the error that results when a single step is performed with exact "input data." It is a key attribute of any IVP solver, and the *order* of the method is used to designate its form. A method has order k if its LTE goes to zero like h^{k+1}. Thus, the Euler method has order 1. The error in an individual Euler step depends on the square of the step and the behavior of the second derivative. Higher-order methods are pursued in the next two sections.

A good way to visualize the LTE is to recognize that at each step, (t_n, y_n) sits on some solution curve $y_n(t)$ that satisfies the differential equation $y'(t) = f(t, y(t))$. With each step we jump to a new solution curve, and the size of the jump is the LTE. (See Figure 9.3.)

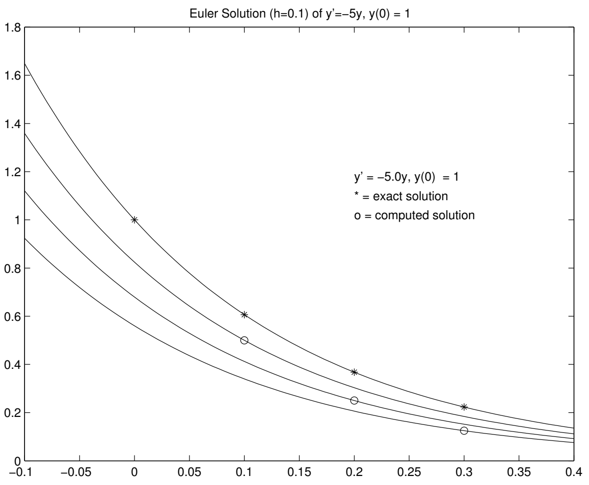


FIGURE 9.3 *Jumping trajectories*

Distinct from the local truncation error is the *global error.* The global error g_n is the actual difference between the $t = t_n$ solution y_n produced by the IVP solver and the *true* IVP solution $y(t_n)$:

$$g_n = y(t_n) - y_n.$$

As we have mentioned, the local truncation error in getting y_n is defined by

$$LTE_n = y_{n-1}(t_n) - y_n,$$

where $y_{n-1}(t)$ satisfies the IVP

$$y'(t) = f(t, y(t)), \qquad y(t_{n-1}) = y_{n-1}.$$

LTE is tractable analytically and, as we shall see, it can be estimated in practice. However, in applications it is the global error that is usually of interest. It turns out that it is possible to control global error by controlling the individual LTEs if the underlying IVP is *stable.* We discuss this after we prove the following result.

Theorem 9 *consec Assume that, for $n = 0{:}N$, a function $y_n(t)$ exists that solves the IVP*

$$y'(t) = f(t, y(t)), \qquad y(t_n) = y_n,$$

where $(t_0, y_0), \ldots, (t_N, y_N)$ are given and $t_0 < t_1 < \cdots < t_N$. Define the global error by

$$g_n \;=\; y_0(t_n) - y_n$$

and the local truncation error by

$$LTE_n \;=\; y_{n-1}(t_n) - y_n.$$

If

$$f_y \;=\; \frac{\partial f(t, y)}{\partial y} \leq 0$$

for all $t \in [t_0, t_N]$ and none of the trajectories

$$\{(t, y_n(t)) : t_0 \leq t \leq t_N\}, \qquad n = 0{:}N$$

intersect, then for $n = 1{:}N$

$$|g_n| \;\leq\; \sum_{k=1}^{n} |LTE_k|.$$

Proof If $y_0(t_n) > y_{n-1}(t_n)$, then because f_y is negative we have

$$\int_{t_{n-1}}^{t_n} (f(t, y_0(t)) - f(t, y_{n-1}(t))\, dt < 0.$$

It follows that

$$\begin{aligned} 0 \;<\; y_0(t_n) - y_{n-1}(t_n) \;&=\; (y_0(t_{n-1}) - y_{n-1}(t_{n-1})) + \int_{t_{n-1}}^{t_n} (f(t, y_0(t)) - f(t, y_{n-1}(t))\, dt \\ &<\; (y_0(t_{n-1}) - y_{n-1}(t_{n-1}))\,, \end{aligned}$$

and so

$$|y_0(t_n) - y_{n-1}(t_n)| \;\leq\; |y_0(t_{n-1}) - y_{n-1}(t_{n-1})|. \tag{9.1}$$

Likewise, if $y_0(t_n) < y_{n-1}(t_n)$, then

$$\int_{t_{n-1}}^{t_n} (f(t, y_{n-1}(t)) - f(t, y_0(t))\, dt < 0,$$

and so

$$\begin{aligned} 0 \;<\; y_{n-1}(t_n) - y_0(t_n) \;&=\; (y_{n-1}(t_{n-1}) - y_0(t_{n-1})) + \int_{t_{n-1}}^{t_n} (f(t, y_{n-1}(t)) - f(t, y_0(t))\, dt \\ &<\; y_{n-1}(t_{n-1}) - y_0(t_{n-1}). \end{aligned}$$

Thus, in either case (9.1) holds and so

$$\begin{aligned} |g_n| \;&=\; |y_0(t_n) - y_n| \\ &\leq\; |y_0(t_n) - y_{n-1}(t_n)| \;+\; |y_{n-1}(t_n) - y_n| \\ &<\; |y_0(t_{n-1}) - y_{n-1}(t_{n-1})| \;+\; |y_{n-1}(t_n) - y_n| \\ &=\; |g_{n-1}| \;+\; |LTE_n|. \end{aligned}$$

The theorem follows by induction since $g_1 = LTE_1$. $\Box$

The theorem essentially says that if $\partial f/\partial y$ is negative across the interval of interest, then global error at $t = t_n$ is less than the sum of the local errors made by the IVP solver in reaching t_n. The sign of this partial derivative is tied up with the stability of the IVP. Roughly speaking, if small changes in the initial value induce correspondingly small changes in the IVP solution, then we say that the IVP is *stable*. The concept is much more involved than the condition/stability issues that we talked about in connection with the $Ax = b$ problem. The mathematics is deep and interesting but beyond what we can do here.

So instead we look at the model problem $y'(t) = ay(t), y(0) = c$ and deduce some of the key ideas. In this example, $\partial f/\partial y \;=\; a$ and so Theorem 9 applies if $a < 0$. We know that the solution $y(t) = ce^{at}$ decays if and only if a is negative. If $\tilde{y}(t)$ solves the same differential equation with initial value $y(0) = \tilde{c}$, then

$$|\tilde{y}(t) - y(t)| \;=\; |\tilde{c} - c|e^{at},$$

showing how "earlier error" is damped out as t increases.

To illustrate how global error might be controlled in practice, consider the problem of computing $y(t_{max})$ to within a tolerance *tol*, where $y(t)$ solves a stable IVP $y'(t) = f(t, y(t))$, $y(t_0) = y_0$. Assume that a fixed-step Euler method is to be used and that we have a bound M_2 for $|y^{(2)}(t)|$ on the interval $[t_0, t_{max}]$. If $h = (t_{max} - t_0)/N$ is the step size, then from what we know about the local truncation error of the method,

$$|LTE_n| \leq M_2 \frac{h^2}{2}.$$

Assuming that Theorem 9 applies,

$$|y(t_{max}) - y_N| \;\leq\; \sum_{n=1}^{N} |LTE_n| \;=\; M_2 N \frac{h^2}{2} \;=\; \frac{t_{max} - t_0}{2} M_2 h.$$

Thus, to make this upper bound less than a prescribed $tol > 0$, we merely set N to be the smallest integer that satisfies

$$\frac{(t_{max} - t_0)^2}{2N} M_2 \leq tol.$$

Here is an implementation of the overall process:

```
  function [tvals,yvals] = FixedEuler(f,y0,t0,tmax,M2,tol)
% Fixed step Euler method.
%
% f is a handle that references a function of the form f(t,y).
% M2 a bound on the second derivative of the solution to
%                y' = f(t,y),   y(t0) = y0
% on the interval [t0,tmax].

% Determine positive n so that if tvals = linspace(t0,tmax,n), then
% y(i) is within tol of the true solution y(tvals(i)) for i=1:n.
n = ceil(((tmax-t0)^2*M2)/(2*tol))+1;
h = (tmax-t0)/(n-1);
yvals = zeros(n,1);
tvals = linspace(t0,tmax,n)';
yvals(1) = y0;
for k=1:n-1
   fval = f(tvals(k),yvals(k));
   yvals(k+1) = yvals(k)+h*fval;
end
```

Figure 9.4 shows the error when this solution framework is applied to the model problem $y' = -y$ across the interval $[0, 5]$. The trouble with this approach to global error control is that (1) we rarely have good bound information about $|y^{(2)}|$ and (2) it would be better to determine h adaptively so that longer step sizes can be taken in regions where the solution is smooth. This matter is pursued in §9.3.5.

Rounding errors are also an issue in IVP solving, especially when lots of very short steps are taken. In Figure 9.5 we plot the errors sustained when we solve $y' = -y,\ y(0) = 1$ across $[0, 1]$ with Euler's method in a three-digit floating point environment. The results for steps h =1/140, 1/160, and 1/180 are reported. Note that the error gets worse as h gets smaller because the step sizes are in the neighborhood of unit roundoff. However, for the kind of problems that we are looking at, it is the discretization errors that dominate the discussion of accuracy.

Another issue that colors the performance of an IVP solver is the stability of *the method* itself. This is quite distinct from the notion of problem stability discussed earlier. It is possible for a method with a particular h to be unstable when it is applied to a stable IVP. For example, if we apply the Euler method to $y'(t) = -10y(t)$, then the iteration takes the form

$$y_{n+1} = (1 - 10h)y_n.$$

To ensure that the errors are not magnified as the iteration progresses, we must insist that

$$|1 - 10h| < 1$$

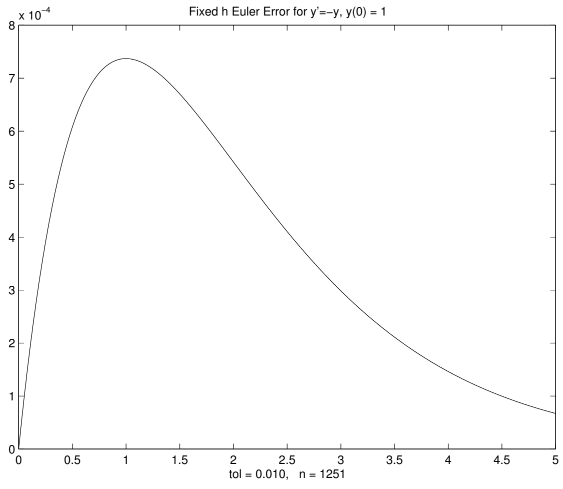


FIGURE 9.4 *Error in fixed-step Euler*

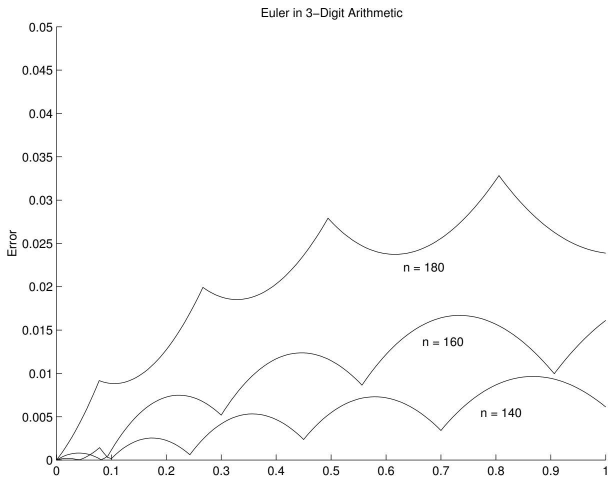


FIGURE 9.5 *Roundoff error in fixed-step Euler*

(i.e., $h < 1/5$). For all h that satisfy this criterion, the method is *stable*. If $h > 1/5$, then any error δ in the initial condition will result in a $(1 - 10h)^n\delta$ contamination of the nth iterate. With this kind of error magnification, we say that the method is *unstable*. Different methods have different h restrictions in order to guarantee stability, and sometimes these restrictions force us to choose h much smaller than we would like.

9.1.3 The Backward Euler Method

To clarify this point about method stability, we examine the *backward Euler method.* The (forward) Euler method is derived from a Taylor expansion of the solution $y(t)$ about $t = t_n$. If instead we work with the approximation

$$y(t_{n+1} + h) \approx y(t_{n+1}) + y'(t_{n+1})h = y(t_{n+1}) + f(t_{n+1}, y(t_{n+1}))h$$

and set $h = -h_n = (t_n - t_{n+1})$, then we get

$$y(t_n) \approx y(t_{n+1}) - h_n f(t_{n+1}, y(t_{n+1})).$$

Substituting y_n for $y(t_n)$ and y_{n+1} for $y(t_{n+1})$, we are led to

$$y_{n+1} = y_n + h_n f(t_{n+1}, y_{n+1})$$

and, with repetition, the *backward Euler framework*:

```
$n = 0$
Repeat:
    Determine the step $h_n > 0$.
    $t_{n+1} = t_n + h_n$.
    Let $y_{n+1}$ solve $F(z) = z - h_n f(t_{n+1}, z) - y_n = 0$.
    $n = n + 1$
```

Like the Euler method, the backward Euler method is first order. However, the two techniques differ in a very important aspect. Backward Euler is an *implicit method* because it defines y_{n+1} implicitly. For a simple problem like $y' = ay$ this poses no difficulty:

$$y_{n+1} = y_n + h_n a y_{n+1} = \frac{1}{1 - h_n a} y_n.$$

Observe that if $a < 0$, then the method is stable for *all* choices of positive step size. This should be contrasted with the situation in the Euler setting, where $|1 + ah| < 1$ is required for stability.

Euler's is an example of an *explicit method*, because y_{n+1} is defined explicity in terms of quantities already computed. [e.g., y_n, $f(t_n, y_n)$]. Implicit methods tend to have better stability properties than their explicit counterparts. But there is an implementation penalty to be paid, because y_{n+1} is defined as a zero of a nonlinear function. In backward Euler, y_{n+1} is a zero of $F(z) = z - h_n f(t_{n+1}, z)$. Fortunately, this does not necessarily require the application of the Chapter 8 root finders. A simpler, more effective approach is presented in §9.3.

9.1.4 Systems

We complete the discussion of IVP solving basics with comments about systems of differential equations. In this case the unknown function $y(t)$ is a vector of unknown functions:

$$y(t) = \begin{bmatrix} z_1(t) \\ \vdots \\ z_d(t) \end{bmatrix}.$$

(We name the component functions with a z instead of a y to avoid confusion with earlier notation.) In this case, we are given an initial value for each component function and a recipe for its slope. This recipe generally involves the value of all the component functions:

$$\begin{bmatrix} z_1'(t) \\ \vdots \\ z_d'(t) \end{bmatrix} = \begin{bmatrix} f_1(t, z_1(t), \ldots, z_d(t)) \\ \vdots \\ f_m(t, z_1(t), \ldots, z_d(t)) \end{bmatrix} \qquad \begin{array}{c} z_1(t_0) = z_{10} \\ \vdots \\ z_d(t_0) = z_{d0} \end{array}.$$

In vector language, $y'(t) = f(t, y(t)), y(t_0) = y_0$, where the y's are now column d-vectors. Here is a $d = 2$ example:

$$\begin{array}{rcl} u'(t) & = & 2u(t) - .01u(t)v(t) \\ v'(t) & = & -v(t) + .01u(t)v(t) \end{array}, \qquad u(0) = u_0,\ v(0) = v_0.$$

It describes the density of rabbit and fox populations in a classical predator-prey model. The rate of change of the rabbit density $u(t)$ and the fox density $v(t)$ depend on the current rabbit/fox densities.

Let's see how the derivation of Euler's method proceeds for a systems problem like this. We start with a pair of time-honored Taylor expansions:

$$\begin{array}{rcccl} u(t_{n+1}) & \approx & u(t_n) + u'(t_n)h_n & = & u(t_n) + h_n(2u(t_n) - .01u(t_n)v(t_n)) \\ v(t_{n+1}) & \approx & v(t_n) + v'(t_n)h_n & = & v(t_n) + h_n(-v(t_n) + .01u(t_n)v(t_n)) \end{array}$$

Here (as usual), $t_{n+1} = t_n + h_n$. With the definitions

$$y_n = \left[\begin{array}{c} u_n \\ v_n \end{array}\right] \approx \left[\begin{array}{c} u(t_n) \\ v(t_n) \end{array}\right] = y(t_n)$$

and

$$f_n = f(t_n, y_n) = \left[\begin{array}{c} 2u_n - .01u_nv_n \\ -v_n + .01u_nv_n \end{array}\right] \approx \left[\begin{array}{c} 2u(t_n) - .01u(t_n)v(t_n) \\ -v(t_n) + .01u(t_n)v(t_n) \end{array}\right] = f(t_n, y(t_n)),$$

we obtain he following vector implementation of the Euler method:

$$\left[\begin{array}{c} u_{n+1} \\ v_{n+1} \end{array}\right] = \left[\begin{array}{c} u_n \\ v_n \end{array}\right] + h_n \left[\begin{array}{c} 2u_n - .01u_nv_n \\ -v_n + .01u_nv_n \end{array}\right].$$

In full vector notation, this can be written as

$$y_{n+1} = y_n + h_nf_n,$$

which is exactly the same formula that we developed in the scalar case.

As we go through the next two sections presenting more sophisticated IVP solvers, we shall do so for scalar ($d = 1$) problems, being mindful that all method-defining equations apply at the system level with no modification.

Systems can arise in practice from the conversion of *higher-order* IVPs. In a kth order IVP, we seek a function $y(t)$ that satisfies

$$y^{(k)}(t) = f(t, y(t), y^{(1)}(t), \ldots, y^{(k-1)}(t)) \qquad \text{where} \qquad \left\{\begin{array}{lcl} y(t_0) & = & y_0 \\ y^{(1)}(t_0) & = & y_0^{(1)} \\ & \vdots & \\ y^{(k-1)}(t_0) & = & y_0^{(k-1)} \end{array}\right.$$

and $y_0, y_0^{(1)}, \ldots, y_0^{(k-1)}$ are given initial values. Higher order IVPs can be solved through conversion to a system of first-order IVPs. For example, to solve

$$v''(t) = 2v(t) + v'(t)\sin(t), \qquad v(0) = \alpha,\ v'(0) = \beta,$$

we define $z_1(t) = v(t)$ and $z_2(t) = v'(t)$. The problem then transforms to

$$\begin{array}{l} z_1'(t) = z_2(t) \\ z_2'(t) = 2z_1(t) + z_2(t)\sin(t) \end{array}, \qquad z_1(0) = \alpha,\ z_2(0) = \beta.$$

Problems

P9.1.1 Produce a plot of the solution to

$$y'(t) = -ty + \frac{1}{y^2}, \qquad y(1) = 1$$

across the interval $[1, 2]$. Use the Euler method.

P9.1.2 Compute an approximation to $y(1)$ where

$$\begin{array}{rcl} x''(t) & = & (3 - \sin(t))x'(t) + x(t)/(1 + [y(t)]^2), \\ y'(t) & = & -\cos(t)y(t) - x'(t)/(1 + t^2), \end{array}$$

$x(0) = 3$, $x'(0) = -1$, and $y(0) = 4$. Use the forward Euler method with fixed step determined so that three significant digits of accuracy are obtained. Hint: Define $z(t) = x'(t)$ and rewrite the recipe for x'' as a function of x, y, and z. This yields a $d = 3$ system.

P9.1.3 Plot the solutions to

$$y'(t) = \left[\begin{array}{cc} -1 & 4 \\ -4 & -1 \end{array}\right] y(t), \qquad y(0) = \left[\begin{array}{c} 2 \\ -1 \end{array}\right]$$

across the interval $[0, 3]$.

P9.1.4 Consider the initial value problem

$$Ay'(t) = By(t), \qquad y(0) = y_0,$$

where A and B are given n-by-n matrices with A nonsingular. For fixed step size h, explain how the backwards Euler method can be used to compute approximate solutions at $t = kh$, $k = 1{:}100$.

9.2 The Runge-Kutta Methods

In an Euler step, we “extrapolate into the future” with only a single sampling of the slope function $f(t,y)$. The method has order 1 because its LTE goes to zero as h^2. Just as we moved beyond the trapezoidal rule in Chapter 4, so we must now move beyond the Euler framework with more involved models of the slope function. In the Runge-Kutta framework, we sample f at several judiciously chosen spots and use the information to obtain y_{n+1} from y_n with the highest possible order of accuracy.

9.2.1 Derivation

The Euler methods evaluate f once per step and have order 1. Let's sample f twice per step and see if we can obtain a second-order method. We arrange it so that the second evaluation depends on the first:

$$\begin{aligned} k_1 &= hf(t_n, y_n) \\ k_2 &= hf(t_n + \alpha h, y_n + \beta k_1) \\ y_{n+1} &= y_n + ak_1 + bk_2\,. \end{aligned}$$

Our goal is to choose the parameters α, β, a, and b so that the LTE is $O(h^3)$. From the Taylor series we have

$$y(t_{n+1}) \;=\; y(t_n) + y^{(1)}(t_n)h + y^{(2)}(t_n)\frac{h^2}{2} + O(h^3).$$

Since

$$\begin{aligned} y^{(1)}(t_n) &= f \\ y^{(2)}(t_n) &= f_t + f_y f \end{aligned}$$

where

$$\begin{aligned} f &= f(t_n, y_n) \\ f_t &= \frac{\partial f(t_n, y_n)}{\partial t} \\ f_y &= \frac{\partial f(t_n, y_n)}{\partial y}, \end{aligned}$$

it follows that

$$y(t_{n+1}) \;=\; y(t_n) + fh \;+\; (f_t + f_y f)\,\frac{h^2}{2} \;+\; O(h^3). \tag{9.2}$$

On the other hand,

$$k_2 = hf(t_n + \alpha h, y_n + \beta k_1) \;=\; h\left(f + \alpha h f_t + \beta k_1 f_y + O(h^2)\right)$$

and so

$$y_{n+1} = y_n + ak_1 + bk_2 = y_n + (a+b)\,fh + b\,(\alpha f_t + \beta f f_y)\,h^2 + O(h^3). \tag{9.3}$$

For the LTE to be $O(h^3)$, the equation

$$y(t_{n+1}) - y_{n+1} = O(h^3)$$

must hold. To accomplish this, we compare terms in (9.2) and (9.3) and require

$$\begin{aligned} a + b &= 1 \\ 2b\alpha &= 1 \\ 2b\beta &= 1\,. \end{aligned}$$

There are an infinite number of solutions to this system, the canonical one being $a = b = 1/2$ and $\alpha = \beta = 1$. With this choice the LTE is $O(h^3)$, and we obtain a second-order Runge-Kutta method:

$$\begin{aligned} k_1 &= hf(t_n, y_n) \\ k_2 &= hf(t_n + h, y_n + k_1) \\ y_{n+1} &= y_n + (k_1 + k_2)/2\,. \end{aligned}$$

The actual expression for the LTE is given by

$$\mathrm{LTE}(RK2) = \frac{h^3}{12}(f_{tt} + 2ff_{ty} + f^2 f_{yy} - 2f_t f_y - 2ff_y^2),$$

where the partials on the right are evaluated at some point in $[t_n, t_n + h]$. Notice that two f-evaluations are required per step.

The most famous Runge-Kutta method is the classical fourth order method:

$$\begin{aligned} k_1 &= hf(t_n, y_n) \\ k_2 &= hf(t_n + \tfrac{h}{2}, y_n + \tfrac{1}{2}k_1) \\ k_3 &= hf(t_n + \tfrac{h}{2}, y_n + \tfrac{1}{2}k_2) \\ k_4 &= hf(t_n + h, y_n + k_3) \\ y_{n+1} &= y_n + \tfrac{1}{6}(k_1 + 2k_2 + 2k_3 + k_4)\,. \end{aligned}$$

This can be derived using the same Taylor expansion technique illustrated previously. It requires four f-evaluations per step.

The function `RKStep` can be used to carry out a Runge-Kutta step of prescribed order. Here is its specification along with an abbreviated portion of the implementation:

```
  function [tnew,ynew,fnew] = RKstep(f,tc,yc,fc,h,k)
% f is a handle that references a function of the form f(t,y)
% where t is a scalar and y is a column d-vector.
% yc is an approximate solution to y'(t) = f(t,y(t)) at t=tc.
% fc = f(tc,yc).
% h is the time step.
% k is the order of the Runge-Kutta method used, 1<=k<=5.
% tnew=tc+h, ynew is an approximate solution at t=tnew, and
% fnew = f(tnew,ynew).

   if k==1
      k1 = h*fc;
      ynew = yc + k1;
   elseif k==2
      k1 = h*fc;
      k2 = h*f(tc+(h),yc+(k1));
      ynew  = yc + (k1 + k2)/2;
   elseif k==3
      k1 = h*fc;
      k2 = h*f(tc+(h/2),yc+(k1/2));
      k3 = h*f(tc+h,yc-k1+2*k2);
      ynew  = yc + (k1 + 4*k2 + k3)/6;
   elseif k==4
       :
   end
   tnew = tc+h;
   fnew = f(tnew,ynew);
```

As can be imagined, symbolic algebra tools are useful in the derivation of such an involved sampling and combination of f-values.

Problems

P9.2.1 The RKF45 method produces both a fourth order estimate and a fifth order estimate using six function evaluations:

$$\begin{aligned}
k_1 &= hf(t_n, y_n) \\
k_2 &= hf(t_n + \tfrac{h}{4}, y_n + \tfrac{1}{4}k_1) \\
k_3 &= hf(t_n + \tfrac{3h}{8}, y_n + \tfrac{3}{32}k_1 + \tfrac{9}{32}k_2) \\
k_4 &= hf(t_n + \tfrac{12h}{13}, y_n + \tfrac{1932}{2197}k_1 - \tfrac{7200}{2197}k_2 + \tfrac{7296}{2197}k_3) \\
k_5 &= hf(t_n + h, y_n + \tfrac{439}{216}k_1 - 8k_2 + \tfrac{3680}{513}k_3 - \tfrac{845}{4104}k_4) \\
k_6 &= hf(t_n + \tfrac{h}{2}, y_n - \tfrac{8}{27}k_1 + 2k_2 - \tfrac{3544}{2565}k_3 + \tfrac{1859}{4104}k_4 - \tfrac{11}{40}k_5) \\
y_{n+1} &= y_n + \tfrac{25}{216}k_1 + \tfrac{1408}{2565}k_3 + \tfrac{2197}{4104}k_4 - \tfrac{1}{5}k_5 \\
z_{n+1} &= y_n + \tfrac{16}{135}k_1 + \tfrac{6656}{12825}k_3 + \tfrac{28561}{56430}k_4 - \tfrac{9}{50}k_5 + \tfrac{2}{55}k_6 \,.
\end{aligned}$$

Write a script that discovers which of y_{n+1} and z_{n+1} is the fourth order estimate and which is the fifth order estimate.

9.2.2 Implementation

Runge-Kutta steps can obviously be repeated, and if we keep the step size fixed, then we obtain the following implementation:

```
  function [tvals,yvals] = FixedRK(f,t0,y0,h,k,n)
% [tvals,yvals] = FixedRK(fname,t0,y0,h,k,n)
% Produces approximate solution to the initial value problem
%
%      y'(t) = f(t,y(t))     y(t0) = y0
%
% using a strategy that is based upon a k-th order Runge-Kutta method. Stepsize
% is fixed. f is a handle that references the function f, t0 is the initial time,
% y0 is the initial condition vector, h is the stepsize, k is the order of
% method (1<=k<=5), and n is the number of steps to be taken,

% tvals(j) = t0 + (j-1)h, j=1:n+1
% yvals(j,:) = approximate solution at t = tvals(j), j=1:n+1

tc = t0; tvals = tc;
yc = y0; yvals = yc';
fc = f(tc,yc);
for j=1:n
   [tc,yc,fc] = RKstep(f,tc,yc,fc,h,k);
   yvals = [yvals; yc'];
   tvals = [tvals tc];
end
```

The function file ShowRK can be used to illustrate the performance of the Runge-Kutta methods on the IVP $y' = -y$, $y(0) = 1$. The results are reported in Figure 9.6. All the derivatives of f are "nice," which means that if we increase the order and keep the step size fixed, then the errors should diminish by a factor of h. Thus for $n = 500$, $h = 1/100$ and we find that the error in the kth order method is about 100^{-k}.

Do not conclude from the example that higher-order methods are necessarily more accurate. If the higher derivatives of the solution are badly behaved, then it may well be the case that a lower-order method gives more accurate results. One must also be mindful of the number of f-evaluations that are required to purchase a given level of accuracy. The situation is analogous to what we found in the quadrature unit. Of course, the best situation is for the IVP software to handle the selection of method and step.

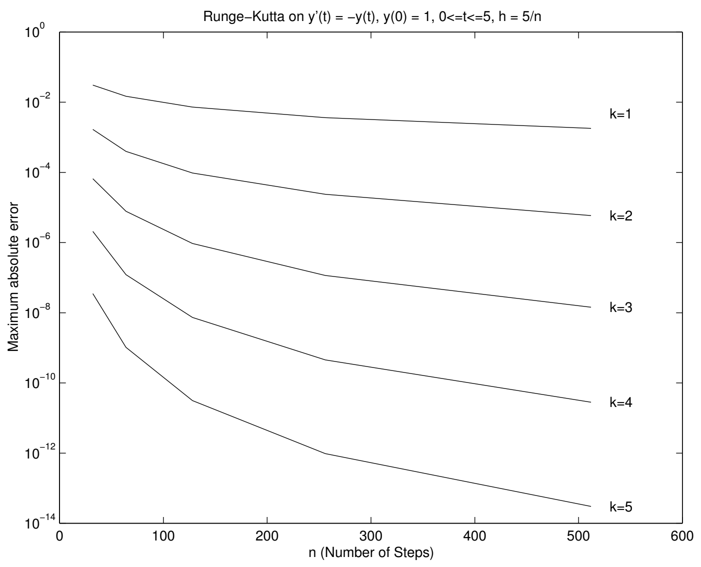


FIGURE 9.6 *Runge-Kutta error*

Problems

P9.2.2 For $k = 1{:}5$, how many f-evaluations does the kth-order Runge-Kutta method require to solve $y'(t) = -y(t),\ y(0) = 1$ with error $\leq 10^{-6}$ across $[0, 1]$?

9.2.3 The MATLAB IVP Solving Tools

MATLAB supplies a number of techniques for solving initial value problems. We start with `ode23`, which is based on a pair of second- and third-order Runge-Kutta methods. With two methods for predicting y_{n+1}, it uses the discrepancy of the predictions to determine heuristically whether the current step size is "safe" with respect to the given tolerances.

Both codes can be used to solve systems, and to illustrate how they are typically used, we apply them to the following initial value problem:

$$\begin{aligned}
\ddot{x}(t) &= -\frac{x(t)}{(x(t)^2 + y(t)^2)^{3/2}} & x(0) = .4 \quad \dot{x}(0) = 0 \\
\ddot{y}(t) &= -\frac{y(t)}{(x(t)^2 + y(t)^2)^{3/2}} & y(0) = 0 \quad \dot{y}(0) = 2\,.
\end{aligned}$$

These are Newton's equations of motion for the two-body problem. As t ranges from 0 to 2π, $(x(t), y(t))$ defines an ellipse.

Both `ode23` and `ode45` require that we put this problem in the standard $y' = f(t, y)$ form. To that end, we define $u_1(t) = x(t)$, $u_2(t) = \dot{x}(t)$, $u_3(t) = y(t)$, $u_4(t) = \dot{y}(t)$. The given IVP problem transforms to

$$\begin{aligned}
\dot{u}_1(t) &= u_2(t) & u_1(0) = .4 \\
\dot{u}_2(t) &= -u_1(t)/(u_1(t)^2 + u_3(t)^2)^{3/2} & u_2(0) = 0 \\
\dot{u}_3(t) &= u_4(t) & u_3(0) = 0 \\
\dot{u}_4(t) &= -u_3(t)/(u_1(t)^2 + u_3(t)^2)^{3/2} & u_4(0) = 2\,.
\end{aligned}$$

We then write the following function, which returns the derivative of the u vector:

```
  function up = Kepler(t,u)
% up = Kepler(t,u)
% t (time) is a scalar and u is a 4-vector whose components satisfy
%
%           u(1) = x(t)      u(2) = (d/dt)x(t)
%           u(3) = y(t)      u(4) = (d/dt)y(t)
%
% where (x(t),y(t)) are the equations of motion in the 2-body problem.
%
% up is a 4-vector that is the derivative of u at time t.

r3 = (u(1)^2 + u(3)^2)^1.5;
up = [  u(2)     ;...
      -u(1)/r3  ;...
       u(4)     ;...
      -u(3)/r3] ;
```

With this function available, we can call ode23 and plot various results:

```
tInitial = 0;
tFinal   =  2*pi;
uInitial = [ .4; 0 ; 0 ; 2];
tSpan = [tInitial tFinal];
[t, u] = ode23(@Kepler, tSpan, uInitial);
```

ode23 requires that we pass the name of the "slope function", the span of integration, and the initial condition vector. The slope function must be of the form f(t,y) where t is a scalar and y is a vector. It must return a column vector. In this call the tSpan vector simply specifies the initial and final times. The output produced is a column vector of times t and a matrix u of solution snapshots. If n = length(t) then (a) t(0) = tInitial, t(n) = tFinal, and u(k,:) is an approximation to the solution at time t(k). The time step lengths and (therefore their number) is determined by the default error tolerance: Reltol $= 10^{-3}$ and AbsTol $= 10^{-6}$. Basically, ode23 integrates from tInitial to tFinal "as quick as possible" subject to these two tolerances. We can display the orbit via

```
plot(u(:,1),u(:,3))
```

i.e., by plotting the computed y-values against the computed x-values. (See Figure 9.7.) From the output we display in Figure 9.8 the step lengths with

```
plot(t(2:length(t)),diff(t))
```

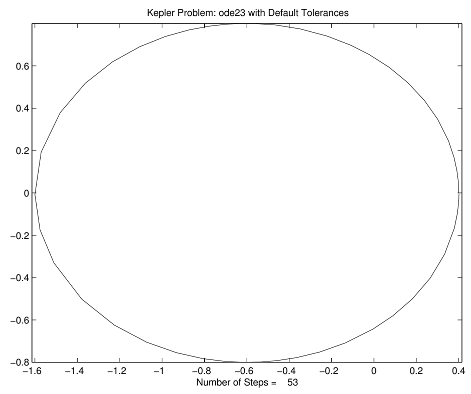


FIGURE 9.7 *Generation of the Orbit via* ode23

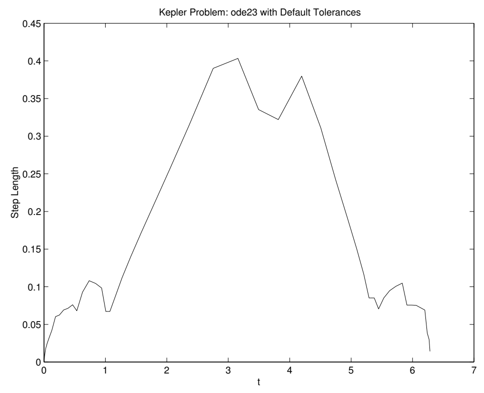


FIGURE 9.8 *Step length with* `ode23` *default tolerances*

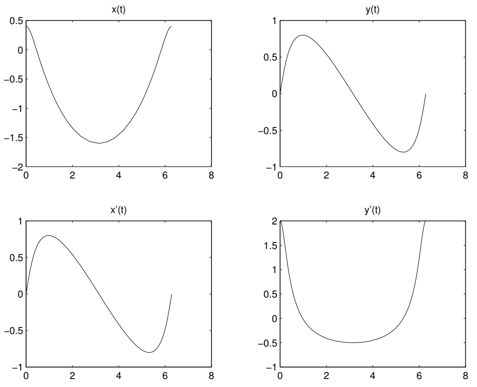


FIGURE 9.9 *Component solutions*

and in Figure 9.9 the component solutions via

```
subplot(2,2,1), plot(t,u(:,1)), title('x(t)')
subplot(2,2,2), plot(t,u(:,2)), title('y(t)')
subplot(2,2,3), plot(t,u(:,3)), title('x''(t)')
subplot(2,2,4), plot(t,u(:,4)), title('y''(t)')
```

The function `ode23` can also be asked to return the solution at specified times. Here is a script that generates 20 solution snapshots and does a spline fit of the output

```
tSpan = linspace(tInitial,tFinal,20);
[t, u] = ode23('Kepler', tSpan, uInitial);
xvals = spline(t,u(:,1),linspace(0,2*pi));
yvals = spline(t,u(:,3),linspace(0,2*pi));
plot(xvals,yvals,u(:,1),u(:,3),'o')
```

(See Figure 9.10.)

Using the function `odeset` it is possible to specify various parameters that are used by `ode23`. For example,

```
tSpan = [tInitial tFinal];
options = odeset('AbsTol',.00000001,'RelTol',.000001,'stats','on');
disp(sprintf('\n Stats for ode23 Call:\n'))
[t, u] = ode23('Kepler', tSpan, uInitial,options);
```

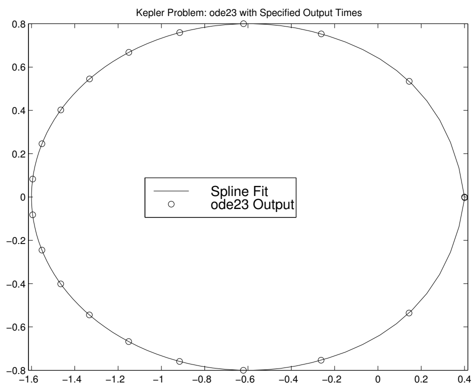


FIGURE 9.10 *Specified output times and spline fit*

overrides the default tolerance for relative error and absolute error and activates the 'stats' option. As expected, the time steps are now shorter as shown in Figure 9.11.

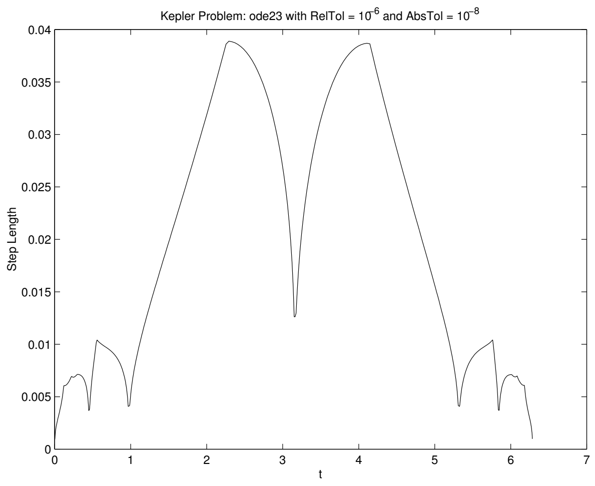


FIGURE 9.11 `ode23` *Timesteps with more stringent tolerances*

The "cost" statistics associated with the call are displayed in the command window:

```
517 successful steps
0 failed attempts
1552 function evaluations
0 partial derivatives
0 LU decompositions
0 solutions of linear systems
```

Sometimes a higher order method can achieve the same accuracy with fewer function evaluations. To illustrate this we apply `ode45` to the same problem:

```
tSpan = [tInitial tFinal];
options = odeset('AbsTol',.00000001,'RelTol',.000001,'stats','on');
disp(sprintf('\n Stats for ode45 Call:\n'))
[t, u] = ode45('Kepler', tSpan, uInitial,options);
```

`ode45` is just like `ode23` except that it uses a mix of 4th and 5th order Runge-Kutta methods. For this problem `ode45` can take decidedly longer time steps in the "high curvature" regions of the orbit. (Compare Figure 9.11 and Figure 9.12.) The output statistics reveal that just 337 function evaluations are required.

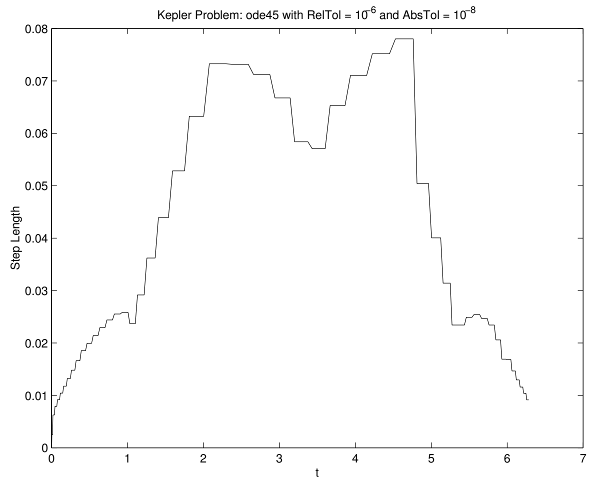


FIGURE 9.12 *Stepsize with ode45*

Problems

P9.2.3 This is about `ode23` vs `ode45`. Suppose it takes T_1 seconds to execute `[t,y] = ode23(@MyF,[0,4],y0)` and T_2 seconds to execute `[t,y] = ode45(@MyF,[0,8],y0)` What factors determine T_2/T_1?

P9.2.4 Our goal is to produce a plot of an approximate solution to the *boundary value problem*

$$y''(t) = f(t, y(t), y'(t)), \qquad y(a) = \alpha,\ y(b) = \beta.$$

Assume the availability of a MATLAB function `f(t,y,yp)`. **(a)** Write a function `g(mu)` that returns an estimate of $y(b)$ where $y(t)$ solves

$$y''(t) = f(t, y(t), y'(t)), \qquad y(a) = \alpha,\ y'(a) = \mu.$$

Use `ode23` and define the function that must be passed to it. **(b)** How could μ_* be computed so that $g(\mu_*) = \beta$? **(c)** Finally, how could a plot of the boundary value problem solution across $[a, b]$ be obtained?

P9.2.5 Consider the following initial value problem

$$A\dot{y} = By + u(t), \qquad y(0) = y_0,$$

where $A \in \mathbb{R}^{n\times n}$ is nonsingular, $B \in \mathbb{R}^{n\times n}$, and $u(t) \in \mathbb{R}^n$. Making effective use of `ode45` with default tolerances, write a MATLAB fragment that assigns to `yFinal` an estimate of $y(t_{final})$. Write out completely the function that your script passes to `ode45`. Assume that `A`, `B`, `y0`, and `tfinal` are given and that `u.m` implements $u(t)$.

P9.2.6 Consider the following IVP:

$$\begin{aligned}\ddot{x}(t) &= 2\dot{y}(t) + x(t) - \frac{\mu_*(x(t)+\mu)}{r_1^3} - \frac{\mu_(x(t)-\mu_*)}{r_2^3}, &\qquad x(0) = 1.2 \quad \dot{x}(0) = 0,\\ \ddot{y}(t) &= -2\dot{x}(t) + y(t) - \frac{\mu_* y(t)}{r_1^3} - \frac{\mu y(t)}{r_2^3}, &\qquad y(0) = 0 \qquad \dot{y}(0) = -1.0493575,\end{aligned}$$

where $\mu = 1/82.45$, $\mu_* = 1 - \mu$, and

$$r_1 = \sqrt{((x(t)+\mu)^2 + y(t)^2}$$
$$r_2 = \sqrt{((x(t)-\mu_*)^2 + y(t)^2}$$

It describes the orbit of a spacecraft that starts behind the Moon (located at $(1-\mu, 0)$), swings by the Earth (located at $(-\mu, 0)$), does a large loop, and returns to the vicinity of the Earth before returning to its initial position behind the Moon at time $T_0 = 6.19216933$. Here, $\mu = 1/82.45$.

(a) Apply `ode45` with $t_{initial} = 0$, $t_{final} = T_0$, and $tol = 10^{-6}$. Plot the orbit twice, once with the default "pen" and once with `'.'` so that you can see how the time step varies. **(b)** Using the output from the `ode45` call in part (a), plot the distance of the spacecraft to Earth as a function of time across $[0, T_0]$. Use `spline` to fit the distance "snapshots." To within a mile, how close does the spacecraft get to the Earth's surface? Assume that the Earth is a sphere of radius 4000 miles and that the Earth-Moon separation is 238,000 miles. Use `fmin` with an appropriate spline for the objective function. Note that the IVP is scaled so that one unit of distance is 238,000 miles. **(c)** Repeat Part **(a)** with `ode23`. **(d)** Apply `ode45` with $t_{initial} = 0$, $t_{final} = 2T_0$, and $tol = 10^{-6}$, but change $\dot{y}(0)$ to and $\dot{y}(0) = -.8$. Plot the orbit. For a little more insight into what happens, repeat with $t_{final} = 8 * T_0$. **(e)** To the nearest minute, compute how long the spacecraft is hidden to an observer on earth as it swings behind the Moon during its orbit. Assume that the observer is at $(-\mu, 0)$ and that the Moon has diameter 2160 miles. Make intelligent use of `fzero`. **(f)** Find t_* in the interval $[0, T_0/2]$ so that at time t_*, the spacecraft is equidistant from the Moon and the Earth.

9.3 The Adams Methods

From the fundamental theorem of calculus, we have

$$y(t_{n+1}) \;=\; y(t_n) + \int_{t_n}^{t_{n+1}} y'(t)dt,$$

and so

$$y(t_{n+1}) \;=\; y(t_n) + \int_{t_n}^{t_{n+1}} f(t, y(t))dt.$$

The Adams methods are based on the idea of replacing the integrand with a polynomial that interpolates $f(t, y)$ at selected solution points (t_j, y_j). The kth order Adams-Bashforth method is explicit and uses the current point (t_n, y_n) and $k-1$ "historical" points. The kth order Adams-Moulton method is implicit and uses the future point (t_{n+1}, y_{n+1}), the current point, and $k-2$ historical points. The implementation and properties of these two IVP solution frameworks are presented in this section.

9.3.1 Derivation of the Adams-Bashforth Methods

In the kth order *Adams-Bashforth* (AB) method, we set

$$y_{n+1} = y_n + \int_{t_n}^{t_{n+1}} p_{k-1}(t)dt, \tag{9.4}$$

where $p_{k-1}(t)$ interpolates $f(t, y)$ at (t_{n-j}, y_{n-j}), $j = 0{:}k-1$. We are concerned with the first five members of this family:

Order	Interpolant	AB Interpolation Points
1st	constant	(t_n, f_n)
2nd	linear	$(t_n, f_n), (t_{n-1}, f_{n-1})$
3rd	quadratic	$(t_n, f_n), (t_{n-1}, f_{n-1}), (t_{n-2}, f_{n-2})$
4th	cubic	$(t_n, f_n), (t_{n-1}, f_{n-1}), (t_{n-2}, f_{n-2}), (t_{n-3}, f_{n-3})$
5th	quartic	$(t_n, f_n), (t_{n-1}, f_{n-1}), (t_{n-2}, f_{n-2}), (t_{n-3}, f_{n-3}), (t_{n-3}, f_{n-3})$

If $k = 1$, then the one-point Newton-Cotes rule is applied and we get

$$y_{n+1} = y_n + h_n f(t_n, y_n) \qquad h_n = t_{n+1} - t_n.$$

Thus the first-order AB method is the Euler method.

In the second-order Adams-Bashforth method, we set

$$p_{k-1}(t) = f_{n-1} + \frac{f_n - f_{n-1}}{h_{n-1}}(t - t_{n-1})$$

in (9.4). This is the linear interpolant of (t_{n-1}, f_{n-1}) and (t_n, f_n), and we obtain

$$\begin{aligned} \int_{t_n}^{t_{n+1}} f(t, y(t))dt \;&\approx\; \int_{t_n}^{t_{n+1}} \left(f_{n-1} + \frac{f_n - f_{n-1}}{h_{n-1}}(t - t_{n-1}) \right) dt \\ &=\; \frac{h_n}{2} \left(\frac{h_n + 2h_{n-1}}{h_{n-1}} f_n - \frac{h_n}{h_{n-1}} f_{n-1} \right). \end{aligned}$$

If $h_n = h_{n-1} = h$, then from (9.4)

$$y_{n+1} = y_n + \frac{h}{2}\left(3f_n - f_{n-1}\right).$$

The derivation of higher-order AB methods is analogous. A table of the first five Adams-Bashforth methods along with their respective local truncation errors is given in Figure 9.13. The derivation of the

Order	Step	LTE
1	$y_{n+1} = y_n + hf_n$	$\frac{h^2}{2}y^{(2)}(\eta)$
2	$y_{n+1} = y_n + \frac{h}{2}\left(3f_n - f_{n-1}\right)$	$\frac{5h^3}{12}y^{(3)}(\eta)$
3	$y_{n+1} = y_n + \frac{h}{12}\left(23f_n - 16f_{n-1} + 5f_{n-2}\right)$	$\frac{3h^4}{8}y^{(4)}(\eta)$
4	$y_{n+1} = y_n + \frac{h}{24}\left(55f_n - 59f_{n-1} + 37f_{n-2} - 9f_{n-3}\right)$	$\frac{251h^5}{720}y^{(5)}(\eta)$
5	$y_{n+1} = y_n + \frac{h}{720}\left(1901f_n - 2774f_{n-1} + 2616f_{n-2} - 1274f_{n-3} + 251f_{n-4}\right)$	$\frac{95h^6}{288}y^{(6)}(\eta)$

FIGURE 9.13 *Adams-Bashforth family*

LTEs for the AB methods is a straightforward computation that involves the Newton-Cotes error:

$$y(t_{n+1}) - y_n \;=\; \int_{t_n}^{t_{n+1}} \left(f(t, y_n(t)) - p_{k-1}(t)\right)dt.$$

9.3.2 Implementation

To facilitate experimentation with the AB method, here is a function that can carry out any of the methods specified in Figure 9.13:

```
  function [tnew,ynew,fnew] = ABstep(f,tc,yc,fvals,h,k)
% f is a handle that references a function of the form f(t,y)
%   where t is a scalar and y is a column d-vector.
%
% yc is an approximate solution to y'(t) = f(t,y(t)) at t=tc.
%
% fvals is an d-by-k matrix where fvals(:,i) is an approximation
%   to f(t,y) at t = tc +(1-i)h, i=1:k
%
% h    = the time step.
% k    = the order of the AB method used, 1<=k<=5.
% tnew = tc+h.
% ynew = an approximate solution at t=tnew.
% fnew = f(tnew,ynew).

if k==1,     ynew = yc + h*fvals;
elseif k==2, ynew = yc + (h/2)*(fvals*[3;-1]);
elseif k==3, ynew = yc + (h/12)*(fvals*[23;-16;5]);
elseif k==4, ynew = yc + (h/24)*(fvals*[55;-59;37;-9]);
elseif k==5, ynew = yc + (h/720)*(fvals*[1901;-2774;2616;-1274;251]);
end
tnew = tc+h;
fnew = f(tnew,ynew);
```

In the systems case, `fval` is a matrix and `ynew` is `yc` plus a matrix-vector product.

Note that k snapshots of $f(t, y)$ are required, and this is why Adams methods are called *multistep* methods. Because of this there is a "start-up" issue with the Adams-Bashforth method: How do we perform the first step when there is no "history"? There are several approaches to this, and care must be taken to ensure that the accuracy of the generated start-up values is consistent with the overall accuracy aims. For a kth order Adams framework we use a kth order Runge-Kutta method, to get $f_j = f(t_j, y_j)$, $j = 1{:}k-1$. See the function `ABStart`. Using `ABStart` we are able to formulate a fixed-step Adams-Bashforth solver:

```
  function [tvals,yvals] = FixedAB(f,t0,y0,h,k,n)
% Produces an approximate solution to the initial value problem
% y'(t) = f(t,y(t)), y(t0) = y0 using a strategy that is based upon a k-th order
% Adams-Bashforth method. Stepsize is fixed.
%
% f  = handle that references the function f.
% t0 = initial time.
% y0 = initial condition vector.
% h  = stepsize.
% k  = order of method. (1<=k<=5).
% n  = number of steps to be taken,
%
% tvals(j) = t0 + (j-1)h, j=1:n+1
% yvals(j,:) = approximate solution at t = tvals(j), j=1:n+1

[tvals,yvals,fvals] = ABStart(f,t0,y0,h,k);
tc = tvals(k);
yc = yvals(k,:)';
fc = fvals(:,k);

for j=k:n
   % Take a step and then update.
   [tc,yc,fc] = ABstep(f,tc,yc,fvals,h,k);
   tvals = [tvals tc];
   yvals = [yvals; yc'];
   fvals = [fc fvals(:,1:k-1)];
end
```

If we apply this algorithm to the model problem, $y' = -y, y(0) = 1$. (See Figure 9.14.) Notice that for the kth-order method, the error goes to zero as h^k, where $h = 1/n$.

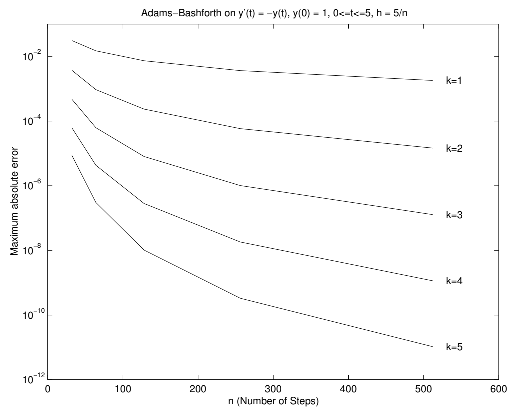


FIGURE 9.14 *kth order Adams-Bashforth error*

9.3.3 The Adams-Moulton Methods

The kth order *Adams-Moulton* (AM) method is just like the kth-order Adams-Bashforth method, but the points at which we interpolate the integrand in

$$y_{n+1} = y_n + \int_{t_n}^{t_{n+1}} p_{k-1}(t)dt$$

are "shifted" one time step into the future. In particular, the kth-order Adams-Moulton method uses a degree $k-1$ interpolant of the points (t_{n+1-j}, f_{n+1-j}), $j = 0{:}k-1$:

Order	Interpolant	AM Interpolation Points
1st	constant	(t_{n+1}, f_{n+1})
2nd	linear	$(t_{n+1}, f_{n+1}), (t_n, f_n)$
3rd	quadratic	$(t_{n+1}, f_{n+1}), (t_n, f_n), (t_{n-1}, f_{n-1})$
4th	cubic	$(t_{n+1}, f_{n+1}), (t_n, f_n), (t_{n-1}, f_{n-1}), (t_{n-2}, f_{n-2})$
5th	quartic	$(t_{n+1}, f_{n+1}), (t_n, f_n), (t_{n-1}, f_{n-1}), (t_{n-2}, f_{n-2}), (t_{n-3}, f_{n-3})$

For example, in the second-order Adams-Moulton method we set

$$p_{k-1}(t) = f_n + \frac{f_{n+1} - f_n}{h_n}(t - t_n),$$

Order	Step	LTE
1	$y_{n+1} = y_n + hf(t_{n+1}, y_{n+1})$	$-\frac{h^2}{2}y^{(2)}(\eta)$
2	$y_{n+1} = y_n + \frac{h}{2}\left(f(t_{n+1}, y_{n+1}) + f_n\right)$	$-\frac{h^3}{12}y^{(3)}(\eta)$
3	$y_{n+1} = y_n + \frac{h}{12}\left(5f(t_{n+1}, y_{n+1}) + 8f_n - f_{n-1}\right)$	$-\frac{h^4}{24}y^{(4)}(\eta)$
4	$y_{n+1} = y_n + \frac{h}{24}\left(9f(t_{n+1}, y_{n+1}) + 19f_n - 5f_{n-1} + f_{n-2}\right)$	$-\frac{19h^5}{720}y^{(5)}(\eta)$
5	$y_{n+1} = y_n + \frac{h}{720}\left(251f(t_{n+1}, y_{n+1}) + 646f_n - 264f_{n-1} + 106f_{n-2} - 19f_{n-3}\right)$	$-\frac{3h^6}{160}y^{(6)}(\eta)$

FIGURE 9.15 *The Adams-Moulton methods*

the linear interpolant of (t_n, f_n) and (t_{n+1}, f_{n+1}). We then obtain the approximation

$$\int_{t_n}^{t_{n+1}} f(t, y(t))dt \;\approx\; \int_{t_n}^{t_{n+1}} \left(f_n + \frac{f_{n+1} - f_n}{h_n}(t - t_n)\right) \;=\; \frac{h_n}{2}(f_n + f_{n+1}),$$

and thus

$$y_{n+1} = y_n + \frac{h_n}{2}(f(t, y_{n+1}) + f_n).$$

As in the backward Euler method, which is just the first order Adams-Moulton method, y_{n+1} is specified implicitly through a nonlinear equation. The higher-order Adams-Moulton methods are derived similarly, and in Figure 9.15 we specify the first five members in the family.

The LTE coefficient for any AM method is slightly smaller than the LTE coefficients for the corresponding AB method. Analogous to `ABstep`, we have

```
  function [tnew,ynew,fnew] = AMstep(f,tc,yc,fvals,h,k)
% Single step of the kth order Adams-Moulton method.
%
% f is a handle that references a function of the form f(t,y)
% where t is a scalar and y is a column d-vector.
%
% yc is an approximate solution to y'(t) = f(t,y(t)) at t=tc.
%
% fvals is an d-by-k matrix where fvals(:,i) is an approximation
% to f(t,y) at t = tc +(2-i)h, i=1:k.
%
% h is the time step.
%
% k is the order of the AM method used, 1<=k<=5.
%
% tnew=tc+h
% ynew is an approximate solution at t=tnew
% fnew = f(tnew,ynew).

if k==1,     ynew = yc + h*fvals;
elseif k==2, ynew = yc + (h/2)*(fvals*[1;1]);
elseif k==3, ynew = yc + (h/12)*(fvals*[5;8;-1]);
elseif k==4, ynew = yc + (h/24)*(fvals*[9;19;-5;1]);
elseif k==5, ynew = yc + (h/720)*(fvals*[251;646;-264;106;-19]);
end
tnew = tc+h;
fnew = f(tnew,ynew);
```

We could discuss methods for the solution of the nonlinear $F(z) = 0$ that defines y_{n+1}. However, we have other plans for the Adams-Moulton methods that circumvent this problem.

9.3.4 The Predictor-Corrector Idea

A very important framework for solving IVPs results when we couple an Adams-Bashforth method with an Adams-Moulton method of the same order. The idea is to *predict* y_{n+1} using an Adams-Bashforth method and then to *correct* its value using the corresponding Adams-Moulton method. In the second-order case, AB2 gives

$$y_{n+1}^{(P)} = y_n + \frac{h}{2}(3f_n - f_{n-1}),$$

which then is used in the right-hand side of the AM2 recipe to render

$$y_{n+1}^{(C)} = y_n + \frac{h}{2}\left(f(t_{n+1}, y_{n+1}^{(P)}) + f_n\right).$$

For general order we have developed a function

```
[tnew,yPred,fPred,yCorr,fCorr] = PCstep(f,tc,yc,fvals,h,k)
```

that implements this idea. It involves a simple combination of `ABStep` and `AMStep`:

```
[tnew,yPred,fPred] = ABstep(f,tc,yc,fvals,h,k);
[tnew,yCorr,fCorr] = AMstep(f,tc,yc,[fPred fvals(:,1:k-1)],h,k);
```

The repeated application of this function defines the fixed-step predictor-corrector framework:

```
  function [tvals,yvals] = FixedPC(f,t0,y0,h,k,n)
% Produces an approximate solution to the initial value problem
% y'(t) = f(t,y(t)), y(t0) = y0 using a strategy that is based upon a k-th order
% Adams Predictor-Corrector framework. Stepsize is fixed.
%
% f = handle that references the function f.
% t0 = initial time.
% y0 = initial condition vector.
% h  = stepsize.
% k  = order of method. (1<=k<=5).
% n  = number of steps to be taken,
%
% tvals(j) = t0 + (j-1)h, j=1:n+1
% yvals(j,:) = approximate solution at t = tvals(j), j=1:n+1

[tvals,yvals,fvals] = StartAB(f,t0,y0,h,k);
tc = tvals(k);
yc = yvals(:,k)';
fc = fvals(:,k);

for j=k:n
   % Take a step and then update.
   [tc,yPred,fPred,yc,fc] = PCstep(f,tc,yc,fvals,h,k);
   tvals = [tvals tc];
   yvals = [yvals; yc'];
   fvals = [fc fvals(:,1:k-1)];
end
```

The error associated with this method when applied to the model problem is given in Figure 9.16 on the next page.

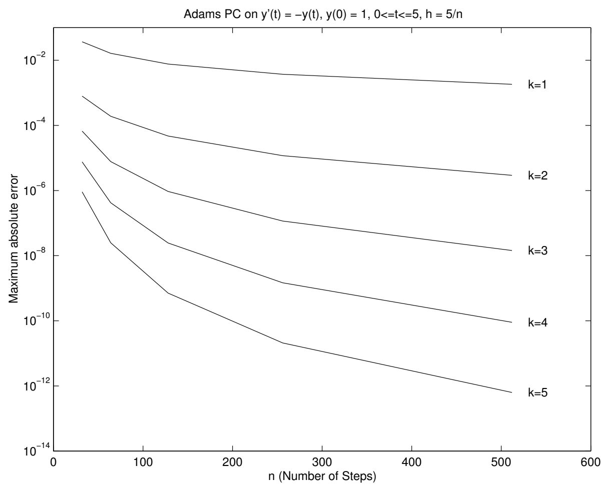


FIGURE 9.16 *kth Order predictor-corrector error*

Problems

P9.3.1 Write functions

```
[tvals,yvals] = AFixedAB(A,t0,y0,h,k,n)
[tvals,yvals] = AFixedAM(A,t0,y0,h,k,n)
```

that can be used to solve the IVP $y'(t) = Ay(t)$, $y(t_0) = y_0$, where A is a d-by-d matrix. In `AFixedAM` a linear system will have to be solved at each step. Get the factorization "out of the loop."

P9.3.2 Use `FixedAB` and `FixedPC` to solve the IVP described in problem P9.2.3. Explore the connections between step size, order, and the number of required function evaluations.

9.3.5 Stepsize Control

The idea behind error estimation in adaptive quadrature is to compute the integral in question in two ways, and then accept or reject the better estimate based on the observed discrepancies. The predictor-corrector framework presents us with a similar opportunity. The quality of $y_{n+1}^{(C)}$ can be estimated from $|y_{n+1}^{(C)} - y_{n+1}^{(P)}|$. If the error is too large, we can reduce the step. If the error is too small, then we can lengthen the step. Properly handled, we can use this mechanism to integrate the IVP across the interval of interest with steps that are as long as possible given a prescribed error tolerance. In this way we can compute the required solution, more or less minimizing the number of f evaluations. The MATLAB IVP solvers `ode23` and `ode45` are Runge-Kutta based and do just that. We develop a second-order adaptive step solver based on the second-order AB and AM methods.

Do we accept $y^{(C)}$ as our chosen y_{n+1}? If $\Delta = |y_{n+1}^{(P)} - y_{n+1}^{(C)}|$ is small, then our intuition tells us that $y_{n+1}^{(C)}$ is probably fairly good and worth accepting as our approximation to $y(t_{n+1})$. If not, there are two possibilities. We could refine $y_{n+1}^{(C)}$ through repeated application of the AM2 formula:

```
$y_{n+1} = y_{n+1}^{(C)}$
Repeat:
    $y_{n+1} = y_n + \frac{h}{2}\left(f(t_{n+1}, y_{n+1}) + f_n\right)$
```

A reasonable termination criterion might be to quit as soon as two successive iterates differ by a small amount. The goal of the iteration is to produce a solution to the AM2 equation. Alternatively, we could halve h and try another predict/correct step [i.e., produce an estimate y_{n+1} of $y(t_n + h/2)$]. The latter approach is more constructive because it addresses the primary reason for discrepancy between the predicted and corrected value: an overly long step h.

To implement a practical step size control process, we need to develop a heuristic for estimating the error in $y_{n+1}^{(c)}$ based on the discrepancy between it and $y_{n+1}^{(P)}$. The idea is to manipulate the LTE expressions

$$\begin{aligned} y(t_{n+1}) &= y_{n+1}^{(P)} + \frac{5}{12}h^3 y^{(3)}(\eta_1), \qquad \eta_1 \in [t_n, t_n + h] \\ y(t_{n+1}) &= y_{n+1}^{(C)} - \frac{1}{12}h^3 y^{(3)}(\eta_2), \qquad \eta_2 \in [t_n, t_n + h] \end{aligned}$$

We make the assumption that $y^{(3)}$ does *not* vary much across $[t_n, t_n + h]$. Subtracting the first equation from the second leads to approximation

$$|y_{n+1}^{(C)} - y_{n+1}^{(P)}| \approx \frac{1}{2}h^3 |y^{(3)}(\eta)|, \qquad \eta \in [t_n, t_n + h]$$

and so

$$|y_{n+1}^{(C)} - y(t_{n+1})| \approx \frac{1}{6}|y_{n+1}^{(C)} - y_{n+1}^{(P)}|.$$

This leads to the following framework for a second-order predictor-corrector scheme:

```
$y_{n+1}^{(P)} = y_n + \frac{h}{2}(3f_n - f_{n-1})$
$y_{n+1}^{(C)} = y_n + \frac{h}{2}\left(f(t_{n+1}, y_{n+1}^{(P)} + f_n\right)$
$\epsilon = \frac{1}{6}|y_{n+1}^{(C)} - y_{n+1}^{(P)}|$
```

```
If $\epsilon$ is too big, then
    reduce $h$ and try again.
Else if $\epsilon$ is about right, then
    set $y_{n+1} = y^{(C)}_{n+1}$ and keep $h$.
Else if $\epsilon$ is too small, then
    set $y_{n+1} = y^{(C)}_{n+1}$ and increase $h$.
```

The definitions of "too big," "about right," and "too small" are central. Here is one approach. Suppose we want the global error in the solution snapshots across $[t_0, t_{max}]$ to be less than δ. If it takes n_{max} steps to integrate across $[t_0, t_{max}]$, then we can heuristically guarantee this if

$$\sum_{n=1}^{n_{max}} \text{LTE}_n \leq \delta.$$

Thus if h_n is the length of the nth step, and

$$|\text{LTE}_n| \leq \frac{h_n\delta}{t_{max} - t_0},$$

then

$$\sum_{n=1}^{n_{max}} \text{LTE}_n \;\leq\; \sum_{n=1}^{n_{max}} \frac{h_n\delta}{t_{max} - t_0} \;\leq\; \delta.$$

This tells us when to accept a step. But if the estimated LTE is considerably smaller than the threshold, say

$$\epsilon \leq \frac{1}{10}\frac{\delta h}{t_{max} - t_0},$$

then it might be worth doubling h.

If the ϵ is too big, then our strategy is to halve h. But to carry out the predictor step with this step size, we need $f(t_n - h/2, y_{n-1/2})$ where $y_{n-1/2}$ is an estimate of $y(t_n - h/2)$. "Missing" values in in this setting can be generated by interpolation or by using (for example) an appropriate Runge-Kutta estimate.

We mention that the MATLAB IVP solver `ode113` implements an Adams-Bashforth-Moulton predictor-corrector framework.

Problems

P9.3.3 Derive an estimate for $|y^{(C)}_{n+1} - y(t_{n+1})|$ for the third-, fourth- and fifth-order predictor-corrector pairs.

M-Files and References

Script Files

`ShowTraj`	Shows family of solutions.
`ShowEuler`	Illustrates Euler method.
`ShowFixedEuler`	Plots error in fixed step Euler for y'=y, y(0)=1.
`ShowTrunc`	Shows effect of truncation error.
`EulerRoundoff`	Illustrates Euler in three-digit floating point.
`ShowAB`	Illustrates `FixedAB`.
`ShowPC`	Illustrates `FixedPC`.
`ShowRK`	Illustrates `FixedRK`.
`ShowMatIVPTools`	Illustrates `ode23` and `ode45` on a system.

Function Files

`FixedEuler`	Fixed step Euler method.
`ABStart`	Gets starting values for Adams methods.
`ABStep`	Adams-Bashforth step (order <= 5).
`FixedAB`	Fixed step size Adams-Bashforth.
`AMStep`	Adams-Moulton step (order <= 5).
`PCStep`	AB-AM predictor-corrector Step (order <= 5).
`FixedPC`	Fixed stepsize AB-AM predictor-corrector.
`RKStep`	Runge-Kutta step (order <= 5).
`FixedRK`	Fixed step size Runge-Kutta.
`Kepler`	For solving two-body IVP.
`f1`	The f function for the model problem y'=y.

References

C.W. Gear (1971). *Numerical Initial Value Problems in Ordinary Differential Equations*, Prentice Hall, Englewood Cliffs, NJ.

J. Lambert (1973). *Computational Methods in Ordinary Differential Equations*, John Wiley, New York.

J. Ortega and W. Poole (1981). *An Introduction to Numerical Methods for Differential Equations*, Pitman, Marshfield, MA.

L. Shampine and M. Gordon (1975). *Computer Solution of Ordinary Differential Equations: The Initial Value Problem*, Freeman, San Francisco.