Chapter 1

Getting Started with Matlab

The computational examples and exercises in this book have been computed using MATLAB, which is an interactive system designed specifically for scientific computation that is used widely in academia and industry. At its core, MATLAB contains an efficient, high level programming language and powerful graphical visualization tools which can be easily accessed through a development environment (that is, a graphical user interface containing various workspace and menu items). MATLAB has many advantages over computer languages such as C, C++, Fortran and Java. For example, when using the MATLAB programming language, essentially no declaration statements are needed for variables. In addition, MATLAB has a built-in extensive mathematical function library that contains such items as simple trigonometric functions, as well as much more sophisticated tools that can be used, for example, to compute difficult integrals. Some of these sophisticated functions are described as we progress through the book. MATLAB also provides additional *toolboxes* that are designed to solve specific classes of problems, such as for image processing. It should be noted that there is no free lunch; because MATLAB is an "interpreted" language, codes written in Fortran and C are usually more efficient for very large problems. (MATLAB may also be compiled.) Therefore, large production codes are usually written in one of these languages, in which case supplementary packages, such as the NAG or IMSL library, or free software from *netlib*, are recommended for scientific computing. However, because it is an excellent package for developing algorithms and problem solving environments, and it can be quite efficient when used properly, all computing in this book uses MATLAB.

We provide a very brief introduction to MATLAB. Though our discussion assumes the use of MATLAB 7.0 or higher, in most cases version 6.0 or 6.5 is sufficient. There are many good sources for more complete treatments on using MATLAB, both on-line, and as books. One excellent source is the *MATLAB Guide, 2nd ed.* by D.J. Higham and N.J. Higham published by SIAM Press, 2005. Another source that we highly recommend is MATLAB's built-in help system, which can be accessed once MATLAB is started. We explain how to access it in section 1.1. The remainder of the chapter then provides several examples that introduce various basic capabilities of MATLAB, such as graph plotting and writing functions.

1.1 Starting, Quitting, and Getting Help

The process by which you start MATLAB depends on your computer system; you may need to request specific commands from your instructor or system administrator. Generally, the process is as follows:

- On a PC with a Windows operating system, double-click the "MATLAB" shortcut icon on your Windows desktop. If there is no "MATLAB" icon on the desktop, you may bring up DOS (on Windows XP by going to the Command Prompt in Accessories) and entering `matlab` at the operating system prompt. Alternatively, you may search for the "MATLAB" icon in a subdirectory and click on it. Where it is to be found depends on how MATLAB was installed; in the simplest case with a default installation it is found in the `C:\$MATLAB` directory, where

`$MATLAB` is the name of the folder containing the MATLAB installation.

- On a Macintosh running OS X 10.1 or higher, there may be a "MATLAB" icon on the dock. If so, then clicking this icon should start MATLAB. If the "MATLAB" icon is not on the dock, then you need to find where it is located. Usually it is found in `/Applications/$MATLAB/`, or `/Applications/$MATLAB/bin/`, where `$MATLAB` is the name of the folder containing the MATLAB installation. Once you find the "MATLAB" icon, double clicking on it should start MATLAB.
- On Unix or Linux platforms, typically you enter `matlab` at the shell prompt. That is, you open a terminal, and enter the command `matlab`.

When you have been successful in getting MATLAB to start, then the development tool Graphical User Interface (GUI) should appear on the screen. Although there are slight differences (such as key stroke short cuts) between platforms, in general the GUI should have the same look and feel independently of the platform.

The *command window*, where you will do much of your work, contains a prompt:

```
>>
```

We can enter data and execute commands at this prompt. One very useful command is `doc`, which displays the "help browser". For example, entering the command

```
>> doc matlab
```

opens the help browser to a good location for first time MATLAB users to begin reading. Alternatively, you can pull down the *Help* menu, and let go on *MATLAB Help* or *Documentation*. We recommend that you read some of the information on these help pages now, but we also recommend returning periodically to read more as you gain experience using MATLAB.

Throughout this book we provide many examples using MATLAB. In all cases, we encourage readers to "play along" with the examples provided. While doing so, it may be helpful at times to use the `doc` command to find detailed information about various MATLAB commands and functions. For example,

```
>> doc plot
```

opens the help browser, and turns to a page containing detailed information on using the built-in `plot` function.

To exit MATLAB, you can pull down the *File* menu, and let go on *Quit MATLAB*. Alternatively, in the command window, you can use the `exit` command:

```
>> exit
```

1.2 Basics of Matlab

MATLAB derives its name from MATrix LABoratory because the primary object involved in any MATLAB computation is a *matrix*. A matrix A is an array of values, with a certain number of rows and columns that define the "dimension" of the matrix. For example, the array A given by

$$A = \begin{bmatrix} 0 & -6 & 8 & 1 \\ -2 & 5 & 5 & -3 \\ 7 & 8 & 0 & 3 \end{bmatrix}$$

is a matrix with 3 rows and 4 columns, and so is typically referred to as a 3×4 matrix. It is two-dimensional. The values in the matrix are by default all Double Precision numbers which will be discussed in more detail in the next chapter. This fact permits MATLAB to avoid using declarations but involves a possible overhead in memory usage and speed of computation. A matrix with only

one row (that is, a $1 \times n$ matrix) is often called a row vector, while a matrix with only one column (that is, an $n \times 1$ matrix) is called a column vector. For example if

$$x = \begin{bmatrix} -2 \\ 8 \\ 4 \\ 0 \\ 5 \end{bmatrix} \quad \text{and} \quad y = \begin{bmatrix} -3 & 6 & 3 & -4 \end{bmatrix},$$

then we can say that x is a 5×1 matrix, or that it is a column vector of length 5. Similarly, we can say that y is a 1×4 matrix, or that it is a row vector of length 4. If the shape is obvious from the context, then we may omit the words *row* or *column*, and just refer to the object as a vector.

MATLAB is very useful when solving problems whose computations involve matrices and vectors. We explore some of these basic *linear algebra* manipulations, as well as some basic features of MATLAB, through a series of examples.

Initializing Vectors. We can easily create row and/or column vectors in MATLAB. For example, the following two statements create the same row vector:

```
>> x = [1 2 3 4]
>> x = [1, 2, 3, 4]
```

Similarly, the following two statements create the same column vector:

```
>> x = [1
2
3
4]
>> x = [1; 2; 3; 4]
```

Rather than thinking of these structures as row and column vectors, we should think of them as 1×4 and 4×1 matrices, respectively. Observe that elements in a row may be separated by using either a blank space or by using a comma. Similarly, to indicate that a row has ended, we can use either a carriage return, or a semicolon.

Initializing Matrices. In general, we can create matrices with more than one row and column. The following three statements generate the same matrix:

```
>> A = [1 2 3 4
5 6 7 8
9 10 11 12]
>> A = [1 2 3 4; 5 6 7 8; 9 10 11 12]
>> A = [1, 2, 3, 4; 5, 6, 7, 8; 9, 10, 11, 12]
```

Matrix Arithmetic. Matrices can be combined (provided certain requirements on their dimensions are satisfied) using the operations `+, -, *` to form new matrices. Addition and subtraction of matrices is intuitive; to add or subtract two matrices, we simply add or subtract corresponding entries. The only requirement is that the two matrices have the same dimensions. For example, if

$$A = \begin{bmatrix} -3 & -1 \\ 3 & 2 \\ 3 & 0 \end{bmatrix}, \quad B = \begin{bmatrix} 7 & 3 \\ -2 & 2 \\ 3 & -1 \end{bmatrix}, \quad \text{and} \quad C = \begin{bmatrix} -3 & -2 & -2 \\ 5 & 8 & -2 \end{bmatrix}$$

then the MATLAB commands

```
>> D = A + B
>> E = B - A
```

produce the matrices

$$D = \begin{bmatrix} 4 & 2 \\ 1 & 4 \\ 6 & -1 \end{bmatrix} \quad \text{and} \quad E = \begin{bmatrix} 10 & 4 \\ -5 & 0 \\ 0 & -1 \end{bmatrix},$$

but the command

```
>> F = C + A
```

produces an error message because the matrices C and A do not have the same dimensions.

Matrix multiplication, which is less intuitive than addition and subtraction, can be defined using linear combinations of vectors. We begin with something that is intuitive, namely the product of a scalar (that is, a number) and a vector. In general, if c is a scalar and a is a vector with entries a_i, then ca is a vector with entries ca_i. For example,

$$\text{if} \quad \mathrm{c} = 9 \quad \text{and} \quad a = \begin{bmatrix} 3 \\ 4 \\ -2 \end{bmatrix} \quad \text{then} \quad \mathrm{c}a = 9 \begin{bmatrix} 3 \\ 4 \\ -2 \end{bmatrix} = \begin{bmatrix} 27 \\ 36 \\ -18 \end{bmatrix}.$$

Once multiplication by a scalar is defined, it is straight forward to form linear combinations of vectors. For example, if

$$\mathrm{c}_1 = 9,\ \mathrm{c}_2 = 3,\ \mathrm{c}_3 = 1, \quad \text{and} \quad a_1 = \begin{bmatrix} 3 \\ 4 \\ -2 \end{bmatrix},\ a_2 = \begin{bmatrix} 3 \\ 8 \\ 0 \end{bmatrix},\ a_3 = \begin{bmatrix} 4 \\ 0 \\ 4 \end{bmatrix},$$

then

$$\mathrm{c}_1 a_1 + \mathrm{c}_2 a_2 + \mathrm{c}_3 a_3 = 9 \begin{bmatrix} 3 \\ 4 \\ -2 \end{bmatrix} + 3 \begin{bmatrix} 3 \\ 8 \\ 0 \end{bmatrix} + 1 \begin{bmatrix} 4 \\ 0 \\ 4 \end{bmatrix} = \begin{bmatrix} 40 \\ 60 \\ -14 \end{bmatrix}.$$

is a linear combination of the vectors a_1, a_2 and a_3 with the scalars c_1, c_2 and c_3, respectively.

Now, suppose we define a matrix A and column vector x as

$$A = \begin{bmatrix} 3 & 3 & 4 \\ 4 & 8 & 0 \\ -2 & 0 & 4 \end{bmatrix}, \quad x = \begin{bmatrix} 9 \\ 3 \\ 1 \end{bmatrix},$$

then the matrix–vector product Ax is simply a compact way to represent a linear combination of the columns of the matrix A, where the scalars in the linear combination are the entries in the vector x. That is,

$$Ax = 9 \begin{bmatrix} 3 \\ 4 \\ -2 \end{bmatrix} + 3 \begin{bmatrix} 3 \\ 8 \\ 0 \end{bmatrix} + 1 \begin{bmatrix} 4 \\ 0 \\ 4 \end{bmatrix} = \begin{bmatrix} 40 \\ 60 \\ -14 \end{bmatrix}.$$

Thus, to form Ax, the number of columns in the matrix A must be the same as the number of entries in the vector x.

Finally we consider matrix-matrix multiplication. If A is an $m \times n$ matrix, and B is an $n \times p$ matrix (that is, the number of columns in A is the same as the number of rows in B), then the product $C = AB$ is an $m \times p$ matrix whose columns are formed by multiplying A by corresponding columns in B viewed as column vectors. For example, if

$$A = \begin{bmatrix} 3 & 3 & 4 \\ 4 & 8 & 0 \\ -2 & 0 & 4 \end{bmatrix} \quad \text{and} \quad B = \begin{bmatrix} 9 & -1 & -2 & 0 \\ 3 & 0 & 1 & -3 \\ 1 & 2 & -1 & 5 \end{bmatrix}$$

then

$$C = AB = \begin{bmatrix} 40 & 5 & -7 & 11 \\ 60 & -4 & 0 & -24 \\ -14 & 10 & 0 & 20 \end{bmatrix}.$$

So, the jth column of AB is the linear combination of the columns of A with scalars drawn from the jth column of B. For example, the 3rd column of AB is formed by taking the linear combination of the columns of A with scalars drawn from the 3rd column of B. Thus, the 3rd column of AB is $(-2) * a_1 + (1) * a_2 + (-1) * a_3$ where a_1, a_2 and a_3 denote, respectively, the 1st, 2nd, and 3rd columns of A.

The $*$ operator can be used in MATLAB to multiply scalars, vectors, and matrices, provided the dimension requirements are satisfied. For example, if we define

```
>> A = [1 2; 3 4]
>> B = [5 6; 7 8]
>> c = [1; -1]
>> r = [2 0]
```

and we enter the commands

```
>> C = A*B
>> y = A*c
>> z = A*r
>> w = r*A
```

we find that

$$C = \begin{bmatrix} 19 & 22 \\ 43 & 50 \end{bmatrix}, \quad y = \begin{bmatrix} -1 \\ -1 \end{bmatrix}, \quad w = \begin{bmatrix} 2 & 4 \end{bmatrix},$$

but an error message is displayed when trying to calculate `z = A*r` because it is not a legal linear algebra operation; the number of columns in A is not the same as the number of rows in r (that is, the inner matrix dimensions do not agree).

Suppressing Output. Note that each time we execute a statement in the MATLAB command window, the result is redisplayed in the same window. This action can be suppressed by placing a semicolon at the end of the statement. For example, if we enter

```
>> x = 10;
>> y = 3;
>> z = x*y
```

then only the result of the last statement, $z = 30$, is displayed in the command window.

Special Characters In the above examples we observe that the semicolon can be used for two different purposes. When used inside brackets `[ ]` it indicates the end of a row, but when used at the end of a statement, it suppresses display of the result in the command window.

The comma can also be used for two different purposes. When used inside brackets `[ ]` it separates entries in a row, but it can also be used to separate statements in a single line of code. For example, the previous example could have been written in one line as follows:

```
>> x = 10, y = 3, z = x*y
```

The values $x = 10$, $y = 3$ and $z = 30$, are displayed in the command window. If the commas are replaced by semicolons,

```
>> x = 10; y = 3; z = x*y
```

then only $z = 30$ is displayed in the command window.

The semicolon, comma and brackets are examples of certain special characters in MATLAB that are defined for specific purposes. For a full list of special characters, open MATLAB's documentation, which can be done using the MATLAB `Help` menu button, or by entering `doc` in the command window, and navigate to: `MATLAB > Language Fundamentals > Operators and Elementary Operations > MATLAB Operators and Special Characters.`

Transposing Matrices. The *transpose* operation is used in linear algebra to transform an $m \times n$ matrix into an $n \times m$ matrix by transforming rows to columns, and columns to rows. For example, if we have the matrices and vectors:

$$A = \begin{bmatrix} 1 & 2 & 5 \\ 3 & 4 & 6 \end{bmatrix}, \quad c = \begin{bmatrix} 1 \\ -1 \end{bmatrix}, \quad r = \begin{bmatrix} 2 & 0 \end{bmatrix},$$

then

$$A^T = \begin{bmatrix} 1 & 3 \\ 2 & 4 \\ 5 & 6 \end{bmatrix}, \quad c^T = \begin{bmatrix} 1 & -1 \end{bmatrix}, \quad r^T = \begin{bmatrix} 2 \\ 0 \end{bmatrix}.$$

where superscript T denotes transposition. In MATLAB, the single quote ' is used to perform the transposition operation. For example, consider the following matrices:

```
>> A = [1 2 5; 3 4 6];
>> c = [1; -1];
```

When we enter the commands

```
>> D = A'
>> s = c'*c
>> H = c*c'
```

we obtain

$$D = \begin{bmatrix} 1 & 3 \\ 2 & 4 \\ 5 & 6 \end{bmatrix}, \quad s = 2, \quad H = \begin{bmatrix} 1 & -1 \\ -1 & 1 \end{bmatrix}.$$

Other Array Operations. MATLAB supports certain array operations (not normally found in standard linear algebra books) that can be very useful in scientific computing applications. Some of these operations are:

```
.*    ./    .^
```

The *dot* indicates that the operation is to act on the matrices in an element by element (componentwise) way. For example,

$$\begin{bmatrix} 1 \\ 2 \\ 3 \\ 4 \end{bmatrix} .* \begin{bmatrix} 5 \\ 6 \\ 7 \\ 8 \end{bmatrix} = \begin{bmatrix} 5 \\ 12 \\ 21 \\ 32 \end{bmatrix}$$

$$\begin{bmatrix} 1 \\ 2 \\ 3 \\ 4 \end{bmatrix} .^ 3 = \begin{bmatrix} 1 \\ 8 \\ 27 \\ 64 \end{bmatrix}$$

The rules for the validity of these operations are different than for linear algebra. A full list of arithmetic operations, and the rules for their use in MATLAB, can be found by referring to the documentation (help), and navigating to: `MATLAB > Language Fundamentals > Operators and Elementary Operations > Arithmetic`.

Remark on the Problems. The problems in this chapter are designed to help you become more familiar with MATLAB and some of its capabilities. Some problems may include issues not explicitly discussed in the text, but a little exploration in MATLAB (that is, executing the statements and reading appropriate `doc` pages) should provide the necessary information to solve all of the problems.

Problem 1.2.1. *If* `z = [0 -1 2 4 -2 1 5 3]`, *and* `J = [5 2 1 6 3 8 4 7]`, *determine what is produced by the following sequences of* MATLAB *statements:*

```
>> x = z', A = x*x', s = x'*x, w = x*J,
>> length(x), length(z)
>> size(A), size(x), size(z), size(s)
```

Note that to save space, we have placed several statements on each line. Use `doc length` *and* `doc size`, *or search the* MATLAB *Help index, for further information on these commands.*

1.3 Matlab as a Scientific Computing Environment

So far we have used MATLAB as a sophisticated calculator. It is far more powerful than that, containing many programming constructs, such as flow control (if, for, while, etc.), and capabilities for writing functions, and creating structures, classes, objects, etc. Since this is not a MATLAB programming book, we do not discuss these capabilities in great detail. But many MATLAB programming issues are discussed, when needed, as we progress through the book. Here, we introduce a few concepts before beginning our study of scientific computing.

1.3.1 Initializing Vectors with Many Entries

Suppose we want to create a vector, `x`, containing the values $1, 2, \ldots, 100$. We could do this using a `for` loop:

```
n = 100;
for i = 1:n
  x(i) = i;
end
```

In this example, the notation `1:n` is used to create the list of integers $1, 2, 3, \ldots, n$. When MATLAB begins to execute this code, it does not know that `x` will be a row vector of length 100, but it has a very smart *memory manager* that creates space as needed. Forcing the memory manager to work hard can make codes very inefficient. Fortunately, if we know how many entries `x` will have, then we can help out the memory manager by first allocating space using the `zeros` function. For example:

```
n = 100;
x = zeros(1,n);
for i = 1:n
  x(i) = i;
end
```

In general, the function `zeros(m,n)` creates an $m \times n$ array containing all zeros. Thus, in our case, `zeros(1,n)` creates a $1 \times n$ array, that is a row vector with n entries.

A much simpler, and better way, to initialize this simple vector using one of MATLAB's *vector operations*, is as follows:

```
n = 100;
x = 1:n;
```

The colon operator is very useful! Let's see another example where it can be used to create a vector. Suppose we want to create a vector, x, containing n entries equally spaced between $a = 0$ and $b = 1$. The distance between each of the equally spaced points is given by $h = \dfrac{b-a}{n-1} = \dfrac{1}{n-1}$, and the vector, x, should therefore contain the entries:

$$0, \quad 0+h, \quad 0+2*h, \quad \cdots, \quad (i-1)*h, \quad \cdots, \quad 1$$

We can create a vector with these entries, using the colon operator, as follows:

```
n = 101;
h = 1 / (n-1);
x = 0:h:1;
```

or, if we do not need the variable `h` later in our program, we can use:

```
n = 101;
x = 0:1/(n-1):1;
```

We often want to create vectors like this in mathematical computations. Therefore, MATLAB provides a function `linspace` for it. In general, `linspace(a, b, n)` generates a vector of n equally spaced points between a and b. So, in our example with $a = 0$ and $b = 1$, we could instead use:

```
n = 101;
x = linspace(0, 1, n);
```

Note that, for the interval $[0, 1]$, choosing $n = 101$ produces a nice rational spacing between points, namely $h = 0.01$. That is,

$$x = \begin{bmatrix} 0 & 0.01 & 0.02 & \cdots & 0.98 & 0.99 & 1 \end{bmatrix}.$$

A lesson is to be learned from the examples in this subsection. Specifically, if we need to perform a fairly standard mathematical calculation, then it is often worth using the *search* facility in the *help browser* to determine if MATLAB already provides an optimized function for the calculation.

Problem 1.3.1. *Determine what is produced by the* MATLAB *statements:*

```
>> i = 1:10
>> j = 1:2:11
>> x = 5:-2:-3
```

For more information on the use of `":"`*, see* `doc colon`*.*

Problem 1.3.2. *If* `z = [0 -1 2 4 -2 1 5 3]`*, and* `J = [5 2 1 6 3 8 4 7]`*, determine what is produced by the following* MATLAB *statements:*

```
>> z(2:5)
>> z(J)
```

Problem 1.3.3. *Determine what is produced by the following* MATLAB *statements:*

```
>> A = zeros(2,5)
>> B = ones(3)
>> R = rand(3,2)
>> N = randn(3,2)
```

What is the difference between the functions `rand` *and* `randn`*? What happens if you repeat the statement* `R = rand(3,2)` *several times? Now repeat the following pair of commands several times:*

```
>> rng('default')
>> R = rand(3,2)
```

What do you observe? Note: The "up arrow" key can be used to recall statements previously entered in the command window.

Problem 1.3.4. *Determine what is produced by the* MATLAB *statements:*

```
>> x = linspace(1, 1000, 4)
>> y = logspace(0, 3, 4)
```

In each case, what is the spacing between the points?

Problem 1.3.5. *Given integers* `a` *and* `b`*, and a rational number* `h`*, determine a formula for* `n` *such that*

```
linspace(a, b, n) = [a, a+h, a+2h, ..., b]
```

1.3.2 Creating Plots

Suppose we want to plot the function $y = x^2 - \sqrt{x+3} + \cos 5x$ on the interval $-3 \le x \le 5$. The basic idea is first to plot several points, (x_i, y_i), and then to connect them together using lines. We can do this in MATLAB by creating a vector of x-coordinates, a vector of y-coordinates, and then using the MATLAB command `plot(x,y)` to draw the graph in a *figure* window. For example, we might consider the MATLAB code:

```
n = 81;
x = linspace(-3, 5, n);
y = zeros(1, n);
for i = 1:n
  y(i) = x(i)^2 - sqrt(x(i) + 3) + cos(5*x(i));
end
plot(x, y)
```

In this code we have used the `linspace` command to create the vector `x` efficiently, and we helped the memory manager by using the `zeros` command to pre-allocate space for the vector `y`. The `for` loop generates the entries of `y` one at time, and the `plot` command draws the graph.

MATLAB allows certain operations on arrays that can be used to shorten this code. For example, if `x` is a vector containing entries x_i, then:

- `x + 3` is a vector containing entries $x_i + 3$, and `sqrt(x + 3)` is a vector containing entries $\sqrt{x_i + 3}$.
- Similarly, `5*x` is a vector containing entries $5x_i$, and `cos(5*x)` is a vector containing entries $\cos(5x_i)$.
- Finally, recalling the previous section, we can use the *dot* operation `x.^2` to compute a vector containing entries x_i^2.

Using these properties, the `for` loop above may be replaced with a single *vector operation*:

```
n = 81;
y = zeros(1, n);
x = linspace(-3, 5, n);
y = x.^2 - sqrt(x + 3) + cos(5*x);
plot(x,y)
```

If you can use array operations instead of loops, then you should, as they are more efficient.

MATLAB has many more, very sophisticated, plotting capabilities. Three very useful commands are `axis`, `subplot`, and `hold`:

- `axis` is mainly used to scale the x and y-axes on the current plot as follows:
  `axis([xmin xmax ymin ymax])`
- `subplot` is mainly used to put several plots in a single figure window. Specifically,
  `subplot(m, n, p)`
  breaks the figure window into an "$m \times n$ matrix" of small axes, and selects the p^{th} set of axes for the current plot. The axes are counted along the top row of the figure window, then the second row, etc.
- `hold` allows you to overlay several plots on the same set of axes.

The following example illustrates how to use these commands.

Example 1.3.1. Chebyshev polynomials are used in a variety of engineering applications. The j^{th} Chebyshev polynomial $T_j(x)$ is defined by

$$T_j(x) = \cos(j \arccos(x)), \quad -1 \le x \le 1.$$

In section 4.1.4 we see that these strange objects are indeed polynomials.

(a) First we plot, in the same figure, the Chebyshev polynomials for $j = 1, 3, 5, 7$. This can be done by executing the following statements in the command window:

```
x = linspace(-1, 1, 201);
T1 = cos(acos(x));
T3 = cos(3*acos(x));
T5 = cos(5*acos(x));
T7 = cos(7*acos(x));
subplot(2,2,1), plot(x, T1)
subplot(2,2,2), plot(x, T3)
subplot(2,2,3), plot(x, T5)
subplot(2,2,4), plot(x, T7)
```

The resulting plot is shown in Fig. 1.1.

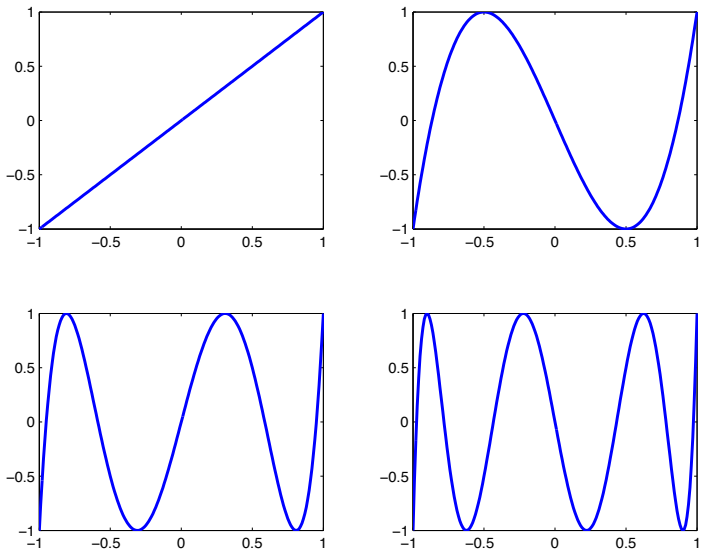


Figure 1.1: Using `subplot` in Example 1.3.1(a).

(b) When you use the `plot` command, MATLAB chooses (usually appropriately) default values for the axes. Here, it chooses precisely the domain and range of the Chebyshev polynomials. These can be changed using the `axis` command, for example:

```
subplot(2,2,1), axis([-1, 1, -2, 2])
subplot(2,2,2), axis([-2, 2, -1, 1])
subplot(2,2,3), axis([-2, 1, -1, 2])
subplot(2,2,4), axis([-2, 2, -2, 2])
```

The resulting plot is shown in Fig. 1.2.

(c) Finally, we can plot all of the polynomials on the same set of axes. This can be achieved as follows (here we use different colors, blue, red, green and cyan, for each):

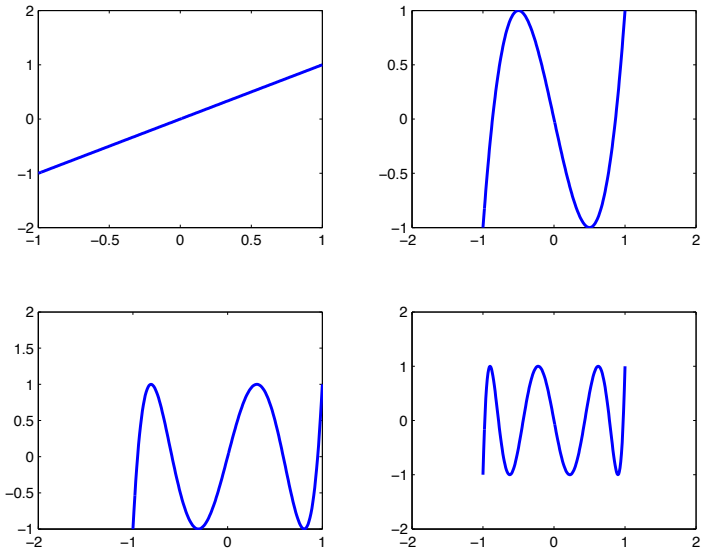


Figure 1.2: Using `axis` in Example 1.3.1(b).

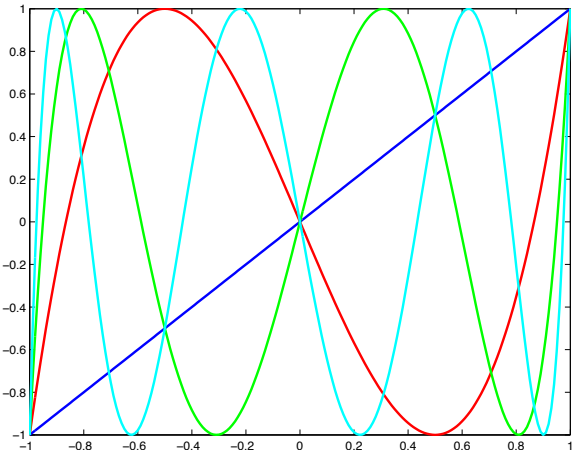


Figure 1.3: Using `hold` in Example 1.3.1(c).

```
subplot(1,1,1)
plot(x, T1, 'b')
hold on
plot(x, T3, 'r')
plot(x, T5, 'g')
plot(x, T7, 'c')
```

The resulting plot is shown in Fig. 1.3. (You should see the plot in color on your screen.)

Remarks on Matlab Figures. We have explained that `hold` can be used to overlay plots on the same axes, and `subplot` can be used to generate several different plots in the *same* figure. In some cases, it is preferable to generate several different plots in *different* figures, using the `figure` command. To clear a figure, so that a new set of plots can be drawn in it, use the `clf` command.

Problem 1.3.6. *Write* MATLAB *code that evaluates and plots the functions:*

(a) $y = 5\cos(3\pi x)$ *for 101 equally spaced points on the interval* $0 \le x \le 1$.

(b) $y = \dfrac{1}{1+x^2}$ *for 101 equally spaced points on the interval* $-5 \le x \le 5$.

(c) $y = \dfrac{\sin 7x - \sin 5x}{\cos 7x + \cos 5x}$ *using 200 equally spaced points on the interval* $-\pi/2 \le x \le \pi/2$. *Use the* `axis` *command to scale the plot so that* $-2 \le x \le 2$ *and* $-10 \le y \le 10$.

In each case, your code should not contain loops but should use arrays directly.

Problem 1.3.7. *A "fixed–point" of a function* $g(x)$ *is a point* x *that satisfies* $x = g(x)$. *An "educated guess" of the location of a fixed–point can be obtained by plotting* $y = g(x)$ *and* $y = x$ *on the same axes, and estimating the location of the intersection point. Use this technique to estimate the location of a fixed–point for* $g(x) = \cos x$.

Problem 1.3.8. *Use* MATLAB *to recreate the plot in Fig. 1.4. Hints: You will need the commands* `hold`, `xlabel`, `ylabel`, *and* `legend`. *The* $\pm$ *symbol in the legend can be created using* `\pm`.

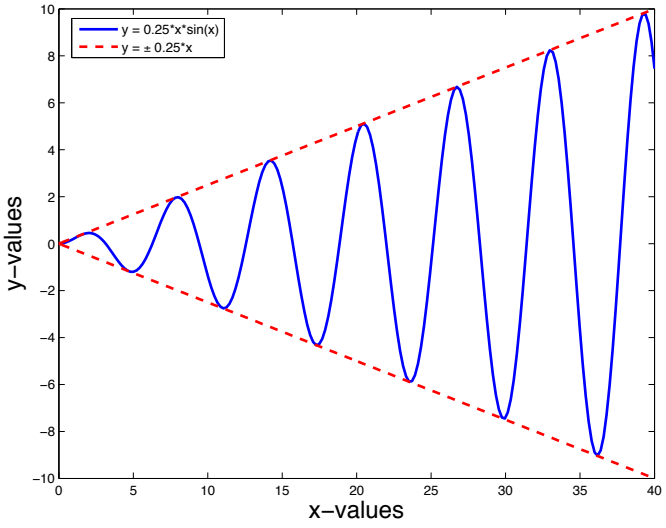


Figure 1.4: Example of a plot that uses MATLAB commands `hold`, `xlabel`, `ylabel`, and `legend`.

Problem 1.3.9. *Explain what happens when the following* MATLAB *code is executed.*

```
for k = 1:6
  x = linspace(0, 4*pi, 2^(k+1)+1);
  subplot(2,3,k), plot(x, sin(x))
  axis([0, 4*pi, -1.5, 1.5])
end
```

What happens if `subplot(2, 3, k)` *is changed to* `subplot(3, 2, k)`*? What happens if the* `axis` *statement is omitted?*

1.3.3 Script and Function M-Files

We introduce *script* and *function* M–files that should be used to write MATLAB programs.

- A *script* is a file containing MATLAB commands that are executed by using the *Run* icon (look for a green, right pointing triangle), under the *Editor* tab, or when the name of the file is entered at the prompt in the MATLAB command window. Scripts are convenient for conducting computational experiments that require entering many commands. However, care must be taken because variables in a script are global to the MATLAB session, and it is therefore easy to unintentionally change their values.

- A *function* is a file containing MATLAB commands that are executed when the function is called. The first line of a function must have the form:

  `function [`*out1, out2,...*`  ]  =` *FunctionName(in1, in2, ...)*

  By default, any data or variables created within the function are private. You can specify any number of inputs to the function, and if you want it to return results, then you can specify any number of outputs. Functions can call other functions, so you can write sophisticated programs as in any conventional programming language.

In each case, the program must be saved in a M–file; that is, a file with a `.m` extension. Any editor may be used to create an M–file, but we recommend using MATLAB's built-in editor, which can be opened in one of several ways. In addition to clicking on certain menu items, the editor can be opened using the `edit` command. For example, if we enter

```
edit ChebyPlots.m
```

in the command window, then the editor will open the file ChebyPlots.m, and we can begin entering and modifying MATLAB commands.

It is important to consider carefully how the script and function M–files are to be named. As mentioned above, they should all have names of the form:

FunctionName`.m` or *ScriptName*`.m`

The names should be descriptive, but it is also important to avoid using a name already taken be one of MATLAB's many built-in functions. If we happen to use the same name as one of these built-in functions, then MATLAB has a way of choosing which function to use, but such a situation can be very confusing. The command `exist` can be used to determine if a function (or variable) is already defined by a specific name and the command `which` can be used to determine its path. Note, for a function file the name of the function and the name of the M–file must be the same.

To illustrate how to write functions and scripts, we provide two examples.

Example 1.3.2. In this example we write a simple function, `PlotCircle.m`, that generates a plot of a circle with radius r centered at the origin:

```
function PlotCircle(r)
%
%         PlotCircle(r)
%
%  This function plots a circle of radius r centered at the origin.
%  If no input value for r is specified, the default value is chosen
%  as r = 1.  We check for too many inputs and for negative input
%  and report errors in both cases.
%
if nargin < 2
   if nargin == 0
     r = 1;
   elseif r <= 0
     error('The input value should be > 0.')
   end
   theta = linspace(0, 2*pi, 200);
   x = r*cos(theta);
   y = r*sin(theta);
   plot(x, y)
   axis([-2*r,2*r,-2*r,2*r])
   axis square
else
  error('Too many input values.')
end
```

We can use it to generate plots of circles. For example, if we enter

```
>> PlotCircle(2)
```

then a circle of radius 2 centered at the origin is plotted, as shown in Figure 1.5. Note, we have chosen to plot 200 points equally spaced in the variable `theta`. If the resulting plotted circle seems a little "ragged" increase the number of points.

This code introduces some new MATLAB commands that require a little explanation:

- The percent symbol, %, is used to denote a comment in the program. On any line anything after a % symbol is treated as comment. All programs (functions and scripts) should contain comments to explain their purpose, and if there are any input and/or output variables.

- `nargin` is a built-in MATLAB command that can be used to count the number of input arguments to a function. If we run this code using `PlotCircle(2)`, `PlotCircle(7)`, etc., then on entry to this function, the value of `nargin` is 1. However, if we run this code using the command `PlotCircle` (with no specified input argument), then on entry to this function, the value of `nargin` is 0. We also use the command `nargin` to check for too many input values.

- The conditional command `if` is used to execute a statement if the expression is true. First we check that there are not too many inputs. Then, we check if `nargin` is 0 (that is, the input value has not been specified). In the latter case, the code sets the radius r to the default value of 1. Note, the difference between the statements `x = 0` and `x == 0`; the former sets the value of x to 0, while the latter checks to see if x and 0 are equal. Observe that the `if` statements are "nested", and thus the `end` statements are correspondingly nested. Each `end` encountered corresponds to the most recent `if`. The indenting used (and produced automatically by the MATLAB editor) should help you follow the command structure of the program. We also use an `elseif` statement to make sure the input value for the radius is not negative. The `doc` command can be used to find more detailed information on `if` and related statements such as `else` and `elseif`.

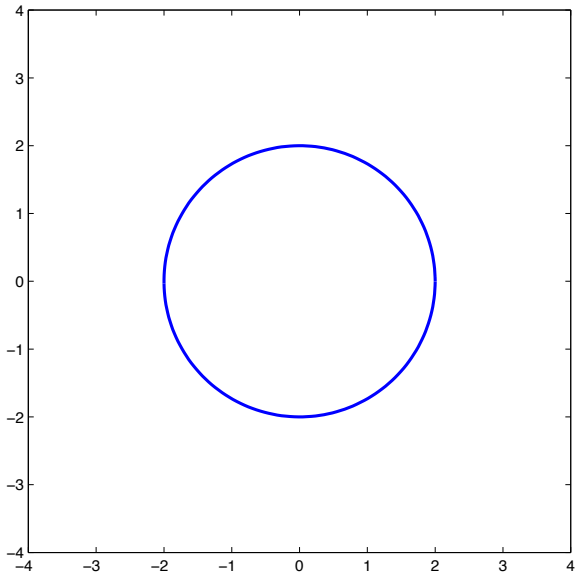


Figure 1.5: Figure created when entering `PlotCirle(2)`.

- `error` is a built-in MATLAB command. When this command is executed, the computation terminates, and the message between the single quotes is printed in the command window.

Example 1.3.3. Here, we illustrate how scripts can be used in combination with functions. Suppose that the concentration of spores of pollen per square centimeter are measured over a 15 day period, resulting in the following data:

day	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
pollen count	12	35	80	120	280	290	360	290	315	280	270	190	90	85	66

(a) First, we write a function that has as input a vector of data, and returns as output the mean and standard deviation of this data.

MATLAB has several built-in functions for statistical analysis. In particular, we can use `mean` and `std` to compute the mean and standard deviation, respectively. Thus, the function we write, named `GetStats.m`, is very simple:

```
function [m, s] = GetStats(x);
%
%   Compute the mean and standard deviation of entries
%   in a vector x.
%
%   Input:  x - vector (either row or column)
%
%   Output:  m - mean value of the elements in x.
%            s - standard deviation of the elements in x.
%
m = mean(x);
s = std(x);
```

```
%
%   Script:  PollenStats
%
%   This script shows a statistical analysis of measured pollen
%   counts on 15 consecutive days.
%

%
%  p is a vector containing the pollen count for each day.
%  d is a vector indicating the day (1 -- 15).
%
p = [12 35 80 120 280 290 360 290 315 280 270 190 90 85 66];
d = 1:length(p);
%
%  Get statistical data, and produce a plot of the results.
%
[m, s] = GetStats(p);
bar(d, p, 'b')
xlabel('Day of Measurement')
ylabel('Pollen Count')
hold on
plot([0, 16], [m, m], 'r')
plot([0, 16], [m-s, m-s], 'r--')
plot([0, 16], [m+s, m+s], 'r--')
text(16, m, '<-- m')
text(16, m-s, '<-- m - s')
text(16, m+s, '<-- m + s')
hold off
```

(b) We now write a script, `PollenStats.m`, to generate and display a statistical analysis of the pollen data.

Once the programs are written, we can run the script by simply entering its name in the command window:

```
>> PollenStats
```

The resulting plot is shown in Figure 1.6.

The script `PollenStats.m` uses several new commands. For detailed information on the specific uses of these commands, use `doc` (for example, `doc bar`). See if you can work out what is going on in the `plot` and `text` lines of `PollenStats.m`.

Problem 1.3.10. *Write a function M–file,* `PlotCircles.m`*, that accepts as input a vector,* `r`*, having positive entries, and plots circles of radius* `r(i)` *centered at the origin on the same axes. If no input value for* `r` *is specified, the default is to plot a single circle with radius* `r = 1`*. Test your function with* `r = 1:5`*. Hint:* MATLAB *commands that might be useful include* `max`, `min` *and* `any`*.*

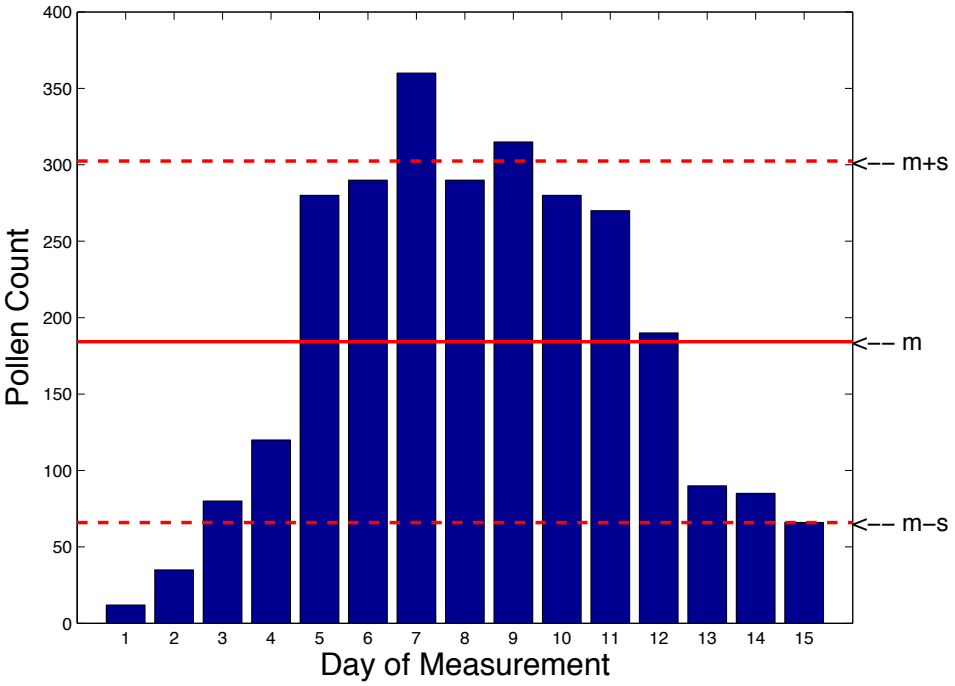


Figure 1.6: Bar graph summarizing the statistical analysis of pollen counts from Example 1.3.3

1.3.4 Creating Reports

Many problems in scientific computing (and in this book) require a combination of mathematical analysis, computational experiments, and a written summary of the results; that is, a report. For simple computer lab reports, a useful tool is MATLAB's `publish` command, which can be used to create a file containing some code and the results it produces. For example, if we enter the code for Example 1.3.1(c) into a script m-file called `Cheby_c.m`, and then use the *Publish* tab, we can create a PDF document that contains the code and the results it produces; see Figure 1.7.

By selecting *Edit Publishing Options ...* under the *Publish* tab you can change the type of format to save the published file. Default is html, but this can easily be changed to PDF, which is often much more convenient for reports and presentations.

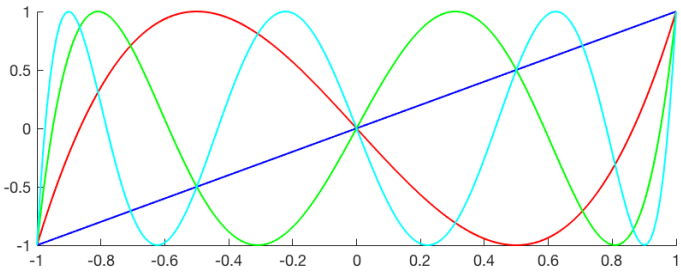
Other things you can do from *Edit Publishing Options ...* include:

- Save only the code, without executing it.
- Save only the result from executing the code, and not the code itself.
- Change the location where the published document is saved on your computer.

```
%
%  Script: Cheby_c
%
%  This creates the figure for the plotting example.  Here
%  we illustrate the use of the hold command.
%
%  Note that we make the line width a bit thicker and the
%  font size a bit larger to make for better displays.
%

x = linspace(-1, 1, 201);
T1 = cos(acos(x));
T3 = cos(3*acos(x));
T5 = cos(5*acos(x));
T7 = cos(7*acos(x));

figure(1), clf
axes('FontSize', 18), hold on
plot(x, T1, 'b', 'LineWidth', 2)
plot(x, T3, 'r', 'LineWidth', 2)
plot(x, T5, 'g', 'LineWidth', 2)
plot(x, T7, 'c', 'LineWidth', 2)
hold off
```



Published with MATLAB® R2017b

Figure 1.7: Example results produced using MATLAB's *Publish* tab.

More extensive reports should be prepared using software such as LaTeX. Plots created in MATLAB can be easily printed for inclusion in the report. Probably the easiest approach is to pull down the *File* menu on the desired MATLAB figure window, and let go on *Print*.

Another option is to save the plot to a file on your computer using the *Save as* option under the *File* menu on the figure window. MATLAB supports many types of file formats, including encapsulated postscript (EPS) and portable document format (PDF), as well as many image compression formats such as TIFF, JPEG and PNG. These files can also be created in the command window, or in any M–file, using the `print` command. For detailed information on creating such files, open the help browser to the associated reference page using the command `doc print`.

An example of a LaTeX document that includes a figure can easily be generated using the *Publish* tool by using *Edit Publishing Options ...* to change the file type to *latex*.

In addition to plots, it may be desirable to create tables of data for a report. Three useful commands that can be use for this purpose are `disp`, `sprintf` and `diary`. When used in combination, `disp` and `sprintf` can be used to write formatted output in the command window. The `diary` command can then be used to save the results in a file. For example, the following MATLAB code computes a compound interest table, with annual compounding:

```
a = 35000; n = 5;
r = 1:1:10;
rd = r/100;
t = a*(1 + rd).^n;
diary InterestInfo.dat
disp(' If a = 35,000 dollars is borrowed, to be paid in n = 5 years, then:')
disp(' ')
disp('      Interest rate, r        Total paid after 5 years')
disp('     ================================================')
for i = 1:length(r)
   disp(sprintf('            %5.2f                    %8.2f     ', r(i), t(i)))
end
diary off
```

If the above code is put into a script or function M–file, then when the code is executed, it prints the following table of data in the command window:

```
 If a = 35,000 dollars is borrowed, to be paid in n = 5 years, then:

      Interest rate, r        Total paid after 5 years
     ================================================
            1                    36785.35
            2                    38642.83
            3                    40574.59
            4                    42582.85
            5                    44669.85
            6                    46837.90
            7                    49089.31
            8                    51426.48
            9                    53851.84
           10                    56367.85
```

Here, the `diary` command is used to open a file named `InterestInfo.dat`, and any subsequent results displayed in the command window are automatically written in this file until it is closed with the `diary off` command. Thus, after execution, the above table is stored in the file `InterestInfo.dat`.

An alternative approach is to use the `fopen`, `fprintf` and `fclose` commands as follows:

```
a = 35000; n = 5;
r = 1:1:10;
rd = r/100;
t = a*(1 + rd).^n;
fid = fopen('InterestInfo.dat', 'w');
fprintf(fid, ' If a = 35,000 dollars is borrowed, to be paid in n = 5 years, then:\n\n');
fprintf(fid, '      Interest rate, r        Total paid after 5 years\n');
fprintf(fid, '     ================================================\n');
for i = 1:length(r)
   fprintf(fid, '            %5.2f                    %8.2f\n', r(i), t(i));
end
fclose(fid);
```

In this bit of MATLAB code,

- We use `fopen` to request that a file, called `InterestInfo.dat`, be opened for writing. MATLAB opens the file, and specifies it with a unique *file indentifier*, which we call `fid`.
- `fprintf` is then used to write data, text, etc., to the file. The commands `\n` indicate that the next printing command should begin on a new line.
- `fclose` is then used to close the file, so it can no longer be modified.

A word of warning about writing to files: If we create a script M–file containing the above code using the `sprintf`, `disp`, and `diary`, and we run that script several times, the file `InterestInfo.dat` will collect the results from each run. In many cases the first set of runs may be used to debug the code, or to get the formatted output to look good, and it is only the final set that we want to save. In this case, it is recommended that you comment out the lines of code containing the `diary` commands (that is, put the symbol % in front of the commands) until you are ready to make the final run.

On the other hand, if we create a script M–file containing the code with `fopen`, `fprintf` and `fclose`, and we run that script several times, the file `InterestInfo.dat` will contain only the results from the final run. This is because we used `'w'` for the file access type in the `fopen` command, which tells MATLAB to open the file for writing, and discard any existing contents. If we want to append results to an existing file, without discarding its current contents, then we should replace `'w'` with `'a'` in the `fopen` command. See `doc fopen` for more information.

1.3.5 Defining Mathematical Functions

In many computational science problems it is necessary to evaluate a function. For example, in order to numerically find the maximum of, say, $f(x) = 1/(1 + x^2)$, one way is to evaluate $f(x)$ at many different points. One approach is to write a function M–file, such as:

```
function f = Runge(x)
%
%     f = Runge(x);
%
%  This function evaluates Runge's function, f(x) = 1/(1 + x^2).
%
%  Input:  x - vector of x values
%
%  Output: f - vector of f(x) values.
%
f = 1 ./ (1 + x.*x);
```

To evaluate this function at, say, $x = 1.7$, we can use the MATLAB statement:

```
>> y = Runge(1.7)
```

Or, if we want to plot this function on the interval $-5 \leq x \leq 5$, we might use the MATLAB statements

```
>> x = linspace(-5, 5, 101);
>> y = Runge(x);
>> plot(x, y)
```

Although this approach works well, it is rather tedious to write a function M–file whose definition is just one line of code. An alternative approach is to use an *anonymous* function:

```
>> Runge = @(x) 1 ./ (1 + x.*x);
```

Here the notation `@(x)` explicitly states that `Runge` is a function of x, and the symbol `@` is referred to as a function handle.

Once `Runge` has been defined, we can evaluate it at, say, $x = 1.7$, using the MATLAB statement:

```
>> y = Runge(1.7);
```

Or, to plot the function on the interval $-5 \leq x \leq 5$, we might use the MATLAB statements above.

Note that it is always best to use array operators (such as `./` and `.*`) when possible, so that functions can be evaluated at arrays of x-values. For example, if we did not use array operations in the above code, that is, if we had used the statement `Runge = @(x) 1 / (1 + x*x)`, (or the same definition for `f` in the function `Runge`) then we could compute the single value `Runge(1.7)`, but `y = Runge(x)` would produce an error if `x` was a vector, such as that defined by `x = linspace(-5, 5, 101)`.

Anonymous functions provide a powerful and easy to use construct for defining mathematical functions. In general, if $f(x)$ can be defined with a single line of code, we use an anonymous function to define it, otherwise we use a function M–file.

Problem 1.3.11. *The* MATLAB *functions* `tic` *and* `toc` *can be used to find the (wall clock) time it takes to run a piece of code. For example, consider the following* MATLAB *statements:*

```
f = @(x) 1 ./ (1 + x.*x);
tic
for i = 1:n
  x = rand(n,1);
  y = f(x);
end
toc
```

When `tic` *is executed, the timer is started, and when* `toc` *is executed, the timer is stopped, and the time required to run the piece of code between the two statements is displayed in the* MATLAB *command window. Write a script M–file containing these statements. Replace the definition of $f(x)$ by a function M–file. Run the two codes for each of $n = 100, 200, 300, 400, 500$. What do you observe? Repeat this experiment. Do you observe any differences in the timings?*

Problem 1.3.12. *Repeat the experiment in Problem 1.3.11, but this time use*

$$f(x) = \frac{e^x + \sin \pi x}{x^2 + 7x + 4}\,.$$

Problem 1.3.13. *Write MATLAB code that will create the plot shown in the following Figure 1.8. Your code should be in either a MATLAB script or function m-file. You can use the code given in Example 1.3.2 as a template.*

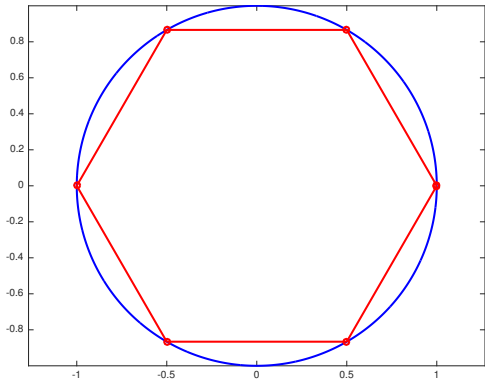


Figure 1.8: Circle with inscribed polygon.

Problem 1.3.14. *To plot a surface, you can use a combination of the MATLAB functions:*

- `linspace` *and* `meshgrid` *to generate a set of* (x, y) *coordinates, and*
- `mesh` *or* `contour` *to plot the surface.*

Read the document pages on these functions, and use them to plot the surfaces of the following (use both `mesh` *and* `contour`*):*

(a) $f(x, y) = (x^2 + 3y^2)e^{-x^2-y^2}$, $\quad -3 \le x \le 3, \quad -3 \le y \le 3$

(b) $g(x, y) = -3y/(x^2 + y^2 + 1)$, $\quad |x| \le 2, \quad |y| \le 4$

(c) $h(x, y) = |x| + |y|$, $\quad |x| \le 2, \quad |y| \le 1$