UNIT IV **APPLICATION OF MATLAB** **10 hrs**

Linear algebric equations – elementary solution method – matrix method for linear equation – Cramer's method – Statistics, Histogram and probability – normal distribution – random number generation – Interpolation – Analytical solution to differential equations – Numerical methods for differential equations.

Linear algebraic equations such as

$$5x - 2y = 13$$
$$7x + 3y = 24$$

occur in many engineering applications. For example, electrical engineers use them to predict the power requirements for circuits; civil, mechanical, and aerospace engineers use them to design structures and machines; chemical engineers use them to compute material balances in chemical processes; and industrial engineers apply them to design schedules and operations. The examples and homework problems in this chapter explore some of these applications. Linear algebraic equations can be solved by hand using pencil and paper, by calculator, or with software such as MATLAB. The choice depends on the circumstances. For equations with only two unknown variables, hand solution is easy and adequate. Some calculators can solve equation sets that have many variables. However, the greatest power and exibility is obtained by using software For example, MATLAB can obtain and plot equation solutions as we vary one or more parameters. Systematic solution methods have been developed for sets of linear equations. The conditions for the existence and uniqueness of solutions are then introduced.

Matrix Methods for Linear Equations

Sets of linear algebraic equations can be expressed as a single equation, using matrix notation. This standard and compact form is useful for expressing solutions and for developing software applications with an arbitrary number of variables. In this application, a vector is taken to be a column vector unless otherwise specified. Matrix notation enables us to represent multiple equations as a single matrix equation. For example, consider the following set.

$$2x1 + 9x2 = 5$$
$$3x1 - 4x2 = 7$$

This set can be expressed in vector-matrix form as

$$\begin{bmatrix} 2 & 9 \\ 3 & -4 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} 5 \\ 7 \end{bmatrix}$$

which can be represented in the following compact form

$$Ax = b \qquad (1)$$

where we have de ned the following matrices and vectors:

$$\mathbf{A} = \begin{bmatrix} 2 & 9 \\ 3 & -4 \end{bmatrix} \qquad \mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} \qquad \mathbf{b} = \begin{bmatrix} 5 \\ 7 \end{bmatrix}$$

In general, the set of *m* equations in *n* unknowns can be expressed in the form Equation (1), where **A** is *m* x *n*, **x** is *n* x 1, and **b** is *m* x 1.

Matrix Inverse
The solution of the scalar equation *ax* = *b* is *x* = *b/a* if *a* ≠ 0. The division operation of scalar algebra has an analogous operation in matrix algebra. For example, to solve the matrix equation (8.1.1) for **x**, we must somehow .divide. **b** by **A** The procedure for doing this is developed from the concept of a *matrix inverse*. The inverse of a matrix **A** is denoted by $\mathbf{A}^{-1}$ and has the property that

$$\mathbf{A}^{-1}\mathbf{A} = \mathbf{A}\mathbf{A}^{-1} = \mathbf{I}$$

where **I** is the identity matrix. Using this property, we multiply both sides of Equation (1) from the left by **A_1** to obtain **A_1Ax _A_1b.** Because **A_1Ax _Ix _x,**

The inverse of a matrix **A** is defined only if **A** is square and nonsingular. A matrix is *singular* if its determinant **|A|** is zero. If **A** is singular, then a unique solution does not exist. The MATLAB functions inv(A) and det(A) compute the inverse and the determinant of the matrix **A**. If the inv(A) function is applied to a singular matrix, MATLAB will issue a warning to that effect. An *ill-conditioned* set of equations is a set that is close to being singular. The ill-conditioned status depends on the accuracy with which the solution calculations are made. When internal numerical accuracy used by MATLAB is insufficient to obtain a solution, it prints the message warning that the matrix is close to singular and that the results might be inaccurate. For a 2 x 2 matrix **A,**

$$\mathbf{A} = \begin{bmatrix} a & b \\ c & d \end{bmatrix} \qquad \mathbf{A}^{-1} = \frac{1}{ad - bc}\begin{bmatrix} d & -b \\ -c & a \end{bmatrix}$$

where det(**A**) = $ad - bc$. Thus **A** is singular if $ad - bc = 0$.

Solve the following equations, using the matrix inverse.

$$2x_1 + 9x_2 = 5$$
$$3x_1 - 4x_2 = 7$$

■ Solution
The matrix **A** and the vector **b** are

$$\mathbf{A} = \begin{bmatrix} 2 & 9 \\ 3 & -4 \end{bmatrix} \qquad \mathbf{b} = \begin{bmatrix} 5 \\ 7 \end{bmatrix}$$

The session is

```
>>A = [2,9;3,-4]; b = [5;7];
>>x = inv(A)*b
x =
   2.3714
   0.0286
```

The solution is $x_1 = 2.3714$ and $x_2 = 0.0286$. MATLAB did not issue a warning, so the solution is unique.

The solution form **x** = $\mathbf{A}^{-1}\mathbf{b}$ is rarely applied in practice to obtain numerical solutions to sets of many equations, because calculation of the matrix inverse is likely to introduce greater numerical inaccuracy than the left division method to be introduced.

Existence and Uniqueness of Solutions

The matrix inverse method will warn us if a *unique* solution does not exist, but it does not tell us whether there is no solution or an infinite number of solutions. In addition, the method is limited to cases where the matrix **A** is square, that is, cases where the number of equations equals the number of unknowns. For this reason we now introduce a method that allows us to determine easily whether an equation set has a solution and whether it is unique. The method requires the concept of the *rank* of a matrix.
Consider the 3 x 3 determinant

$$|\mathbf{A}| = \begin{vmatrix} 3 & -4 & 1 \\ 6 & 10 & 2 \\ 9 & -7 & 3 \end{vmatrix} = 0$$

If we eliminate one row and one column in the determinant, we are left with a 2 x 2 determinant. Depending on which row and column we choose to eliminate, there are nine possible 2 x 2 determinants we can obtain. These are called *subdeterminants*. For example, if we eliminate the second row and third column, we obtain

$$\begin{vmatrix} 3 & -4 \\ 9 & -7 \end{vmatrix} = 3(-7) - 9(-4) = 15$$

Subdeterminants are used to de ne the *rank* of a matrix. The definition of matrix rank is as follows.

Definition of Matrix Rank.
An *m* x *n* matrix **A** has a *rank* $r \geq 1$ if and only if |**A**| contains a nonzero *r* x *r* determinant and every square subdeterminant with *r* + 1 or more rows is zero. For example, the rank of **A** is 2 because |**A**| = 0 while |**A**| contains at least one nonzero 2 x 2 subdeterminant. To determine the rank of a matrix **A** in MATLAB, type rank(A). If **A** is *n* x *n*, its rank is *n* if det(**A**) = 0. We can use the following test to determine if a solution exists to **Ax** =**b** and whether it is unique. The test requires that we rst form the *augmented matrix* [**A b**].

Existence and Uniqueness of Solutions.
The set **Ax** = **b** with *m* equations and *n* unknowns has solutions if and only if (1) rank(**A**) = rank([**A b**]). Let *r* = rank(**A**). If condition (1) is satis ed and if *r* =*n,* then the solution is unique. If condition (1) is satis ed but *r* < *n,* there are an in nite number of solutions, and *r* unknown variables can be expressed as linear combinations of the other *n* _ *r* unknown variables, whose values are arbitrary.

Homogeneous case.
The homogeneous set **Ax** = **0** is a special case in which **b** = **0.** For this case, rank(**A**) = rank([**A b**]) always, and thus the set always has the trivial solution **x** =**0.** A nonzero solution, in which at least one unknown is nonzero, exists if and only if rank(**A**) < *n.* If *m* < *n,* the homogeneous set always has a nonzero solution.
This test implies that if **A** is square and of dimension *n* x *n*, then rank([**A b**]) = rank(**A**), and a unique solution exists for any **b** if rank(**A**) = *n*.

The Left Division Method

MATLAB provides the *left division method* for solving the equation set **Ax** = **b.** This method is based on Gauss elimination. To use the left division method to solve for **x,** you type x = A\b. If |**A**| = 0 or if the number of equations does not equal the number of unknowns, then you need to use the other methods to be presented later.

Use the left division method to solve the following set.

$$
\begin{aligned}
3x_1 + 2x_2 - 9x_3 &= -65 \\
-9x_1 - 5x_2 + 2x_3 &= 16 \\
6x_1 + 7x_2 + 3x_3 &= 5
\end{aligned}
$$

■ **Solution**

The matrices **A** and **b** are

$$
\mathbf{A} = \begin{bmatrix} 3 & 2 & -9 \\ -9 & -5 & 2 \\ 6 & 7 & 3 \end{bmatrix} \qquad \mathbf{b} = \begin{bmatrix} -65 \\ 16 \\ 5 \end{bmatrix}
$$

The session is

```
>>A = [3,2,-9;-9,-5,2;6,7,3];
>>rank(A)
ans =
   3
```

Because **A** is 3 × 3 and rank(**A**) = 3, which is the number of unknowns, a unique solution exists. It is obtained by continuing the session as follows.

```
>>b = [-65;16;5];
>>x = A\b
x =
    2.0000
    -4.0000
    7.0000
```

This answer gives the vector **x**, which corresponds to the solution $x_1 = 2,\ x_2 = -4,\ x_3 = 7$.

For the solution **x** = $\mathbf{A}^{-1}\mathbf{b}$, vector **x** is proportional to the vector **b.** We can use this linearity property to obtain a more generally useful algebraic solution in cases where the right-hand sides are all multiplied by the same scalar. For example, suppose the matrix equation is **Ay** = **b***c*, where *c* is a scalar. The solution is **y** = $\mathbf{A}^{-1}\mathbf{b}c$ = **x***c*. Thus if we obtain the solution to **Ax** = **b,** the solution to **Ay** = **b***c* is given by **y** = **x***c*

Underdetermined Systems

An *underdetermined system* does not contain enough information to determine all the unknown variables, usually but not always because it has fewer equations than unknowns. Thus an in nite number of solutions can exist, with one or more of the unknowns dependent on the remaining unknowns. The left

division method works for square and nonsquare **A** matrices. However, if **A** is not square, the left division method can give answers that might be misinterpreted. We will show how to interpret MATLAB results correctly. When there are fewer equations than unknowns, the left division method might give a solution with some of the unknowns set equal to zero, but this is not the general solution. An in nite number of solutions might exist even when the number of equations equals the number of unknowns. This can occur when |**A**| = 0. For such systems the left division method generates an error message warning us that the matrix **A** is singular. In such cases the *pseudoinverse method* x = pinv(A)*b gives one solution, the *minimum norm solution*. In cases where there are an in nite number of solutions, the rref function can be used to express some of the unknowns in terms of the remaining unknowns, whose values are arbitrary.
An equation set can be underdetermined even though it has as many equations as unknowns. This can happen if some of the equations are not independent. Determining by hand whether all the equations are independent might not be easy, especially if the set has many equations, but it is easily done in MATLAB.

Statistics and Histograms

With MATLAB you can compute the *mean* (the average), the *mode* (the most frequently occurring value), and the *median* (the middle value) of a set of data. MATLAB provides the mean(x), mode(x), and median(x) functions to compute the mean, mode, and median of the data values stored in x, if x is a vector. However, if x is a matrix, a row vector is returned containing the mean (or mode or median) value of each column of x. These functions do not require the elements in x to be sorted in ascending or descending order. The way the data are spread around the mean can be described by a *histogram* plot. A *histogram* is a plot of the frequency of occurrence of data values versus the values themselves. It is a bar plot of the number of data values that occur within each range, with the bar centered in the middle of the range. To plot a histogram, you must group the data into subranges, called *bins.* The choice of the bin width and bin center can drastically change the shape of the histogram. If the number of data values is relatively small, the bin width cannot be small because some of the bins will contain no data and the resulting histogram might not usefully illustrate the distribution of the data. To obtain a histogram, rst sort the data values if they have has not yet been sorted (you can use the sort function here). Then choose the bin ranges and bin centers and count the number of values in each bin. Use the bar function to plot the number of values in each bin versus the bin centers as a bar chart. The function bar(x,y) creates a bar chart of y versus x. MATLAB also provides the hist command to generate a histogram. This command has several forms. Its basic form is hist(y), where y is a vector containing the data. This form aggregates the data into 10 bins evenly spaced between the minimum and maximum values in y. The second form is hist(y,n), where n is a user-speci ed scalar indicating the number of bins. The third form is hist(y,x), where x is a user-speci ed vector that determines the location of the bin centers; the bin widths are the distances between the centers. To ensure proper quality control, a thread manufacturer selects samples and tests them for breaking strength. Suppose that 20 thread samples are pulled until they break, and the breaking force is measured in newtons rounded off to integer values. The breaking force values recorded were 92, 94, 93, 96, 93, 94, 95, 96, 91, 93, 95, 95, 95, 92, 93, 94, 91, 94, 92, and 93. Plot the histogram of the data.

■ Solution

Store the data in the vector y, which is shown in the following script le. Because there are six outcomes (91, 92, 93, 94, 95, 96 N), we choose six bins. However, if you use hist(y,6), the bins will not be centered at 91, 92, 93, 94, 95, and 96. So use the form

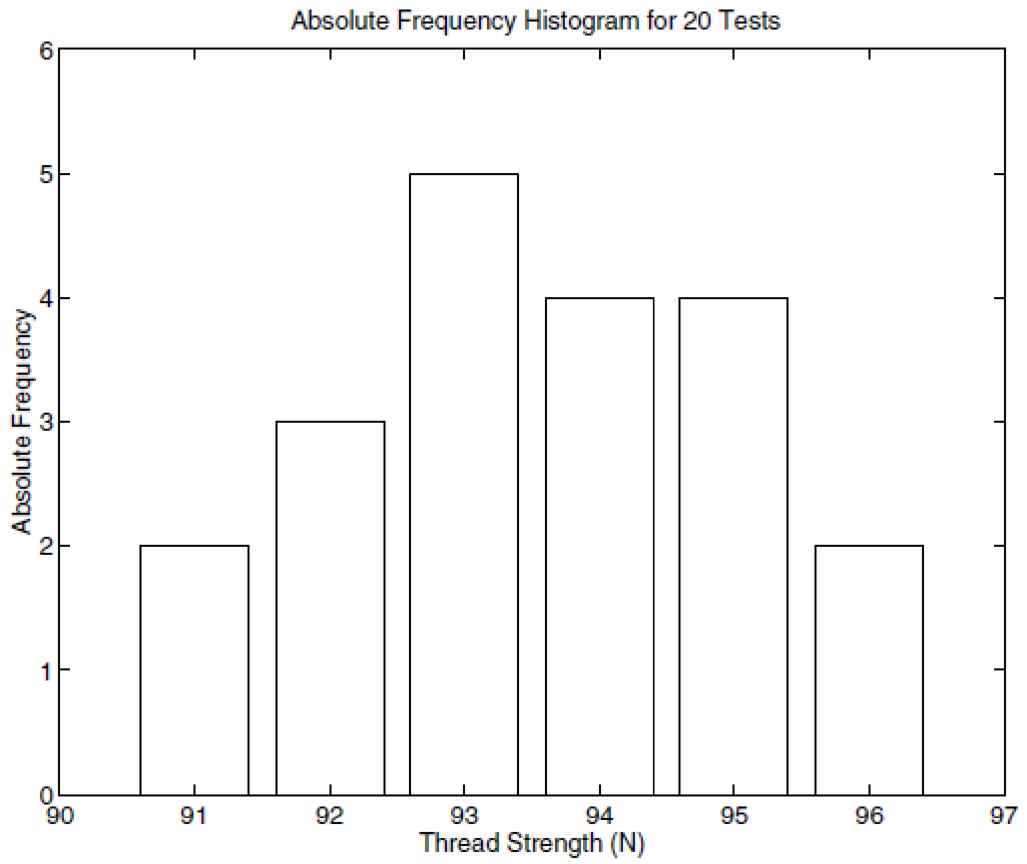


Figure 7.1–1 Histograms for 20 tests of thread strength.

hist(y,x), where x = 91:96. The following script le generates the histogram shown in Figure 7.1.1.

```
% Thread breaking strength data for 20 tests.
y = [92,94,93,96,93,94,95,96,91,93,...
95,95,95,92,93,94,91,94,92,93];
% The six possible outcomes are 91,92,93,94,95,96.
x = 91:96;
hist(y,x),axis([90 97 0 6]),ylabel('Absolute Frequency'),...
xlabel('Thread Strength (N)'),...
title('Absolute Frequency Histogram for 20 Tests')
```

The *absolute frequency* is the number of times a particular outcome occurs.
For example, in 20 tests these data show that a 95 occurred 4 times. The absolute frequency is 4, and its *relative frequency* is 4/20, or 20 percent of the time. When there is a large amount of data, you can avoid typing in every data value by rst aggregating the data. The following example shows how this is done using the ones function. The following data were generated by testing 100 thread samples.

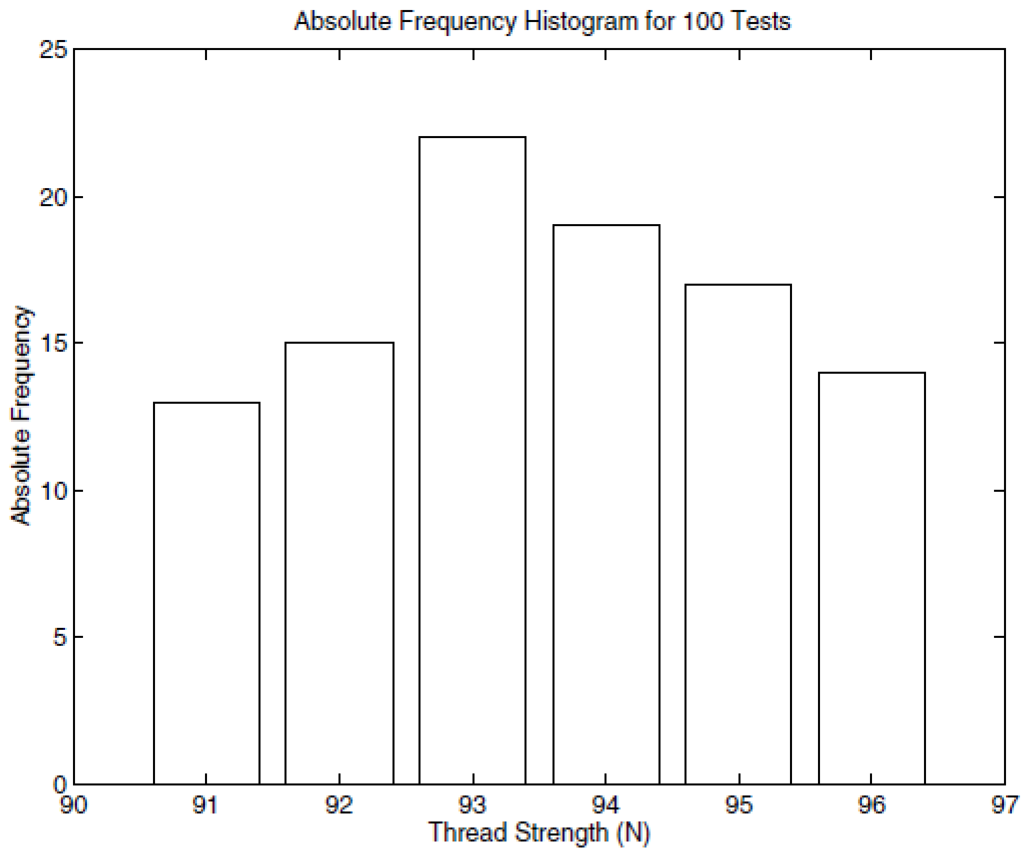


Figure 7.1–2 Absolute frequency histogram for 100 thread tests.

The number of times 91, 92, 93, 94, 95, or 96 N was measured is 13, 15, 22, 19, 17, and 14, respectively.

```
% Thread strength data for 100 tests.
y = [91*ones(1,13),92*ones(1,15),93*ones(1,22),...
94*ones(1,19),95*ones(1,17),96*ones(1,14)];
x = 91:96;
hist(y,x),ylabel('Absolute Frequency'),...
xlabel('Thread Strength (N)'),...
title('Absolute Frequency Histogram for 100 Tests')
```

The result appears in Figure 7.1.2. The hist function is somewhat limited in its ability to produce useful histograms. Unless all the outcome values are the same as the bin centers (as is the case with the thread examples), the graph produced by the hist function will not be satisfactory. This case occurs when you want to obtain a *relative* frequency histogram. In such cases you can use the bar function to generate the histogram. The following script le generates the relative frequency histogram for the 100 thread tests. Note that if you use the bar function, you must aggregate the data rst.

```
% Relative frequency histogram using the bar function.
tests = 100;
y = [13,15,22,19,17,14]/tests;
x = 91:96;
```

```
bar(x,y),ylabel('Relative Frequency'),...
xlabel('Thread Strength (N)'),...
title('Relative Frequency Histogram for 100 Tests')
```

The result appears in Figure 7.1.3. The fourth, fifth, and sixth forms of the hist function do not generate a plot, but are used to compute the frequency counts and bin locations. The bar function can then be used to plot the histogram. The syntax of the fourth form is [z,x] = hist(y), where z is the returned vector containing the frequency count and x is the returned vector containing the bin locations. The fifth and sixth forms are

[z,x] = hist(y,n) and [z,x] = hist(y,x).

In the latter case the returned vector x is the same as the user-supplied vector. The following script le shows how the sixth form can be used to generate a relative frequency histogram for the thread example with 100 tests.

```
tests = 100;
y = [91*ones(1,13),92*ones(1,15),93*ones(1,22),...
94*ones(1,19),95*ones(1,17),96*ones(1,14);
```

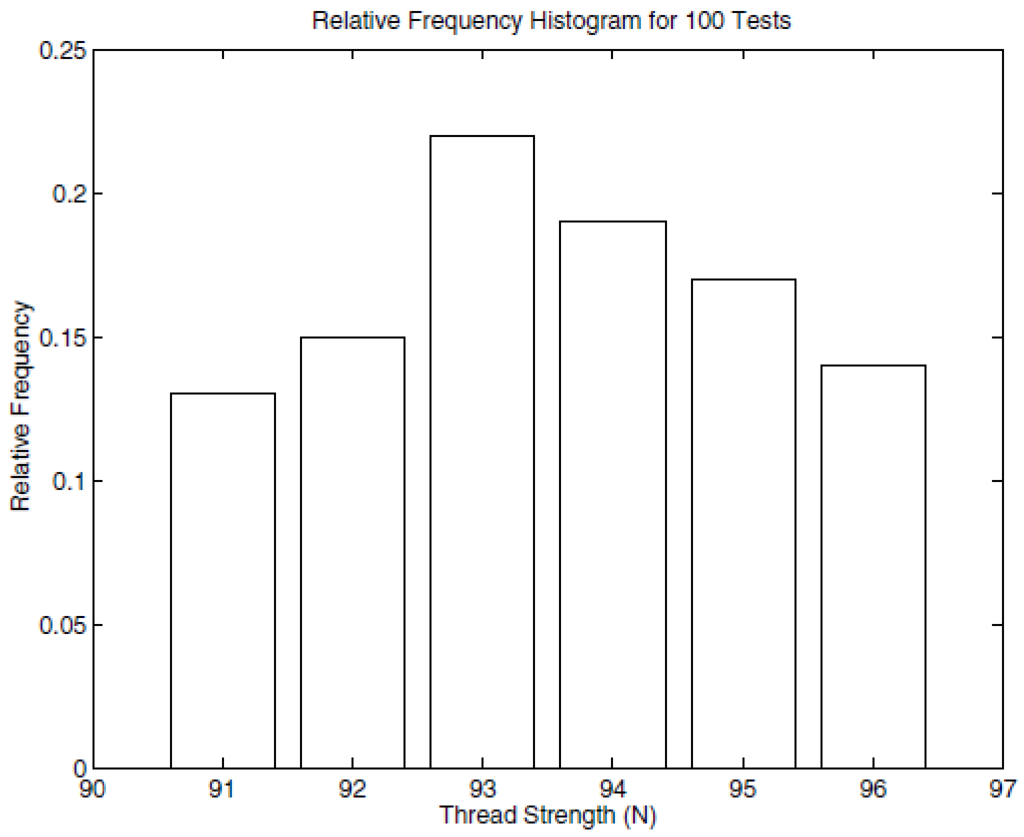


Figure 7.1–3 Relative frequency histogram for 100 thread tests.

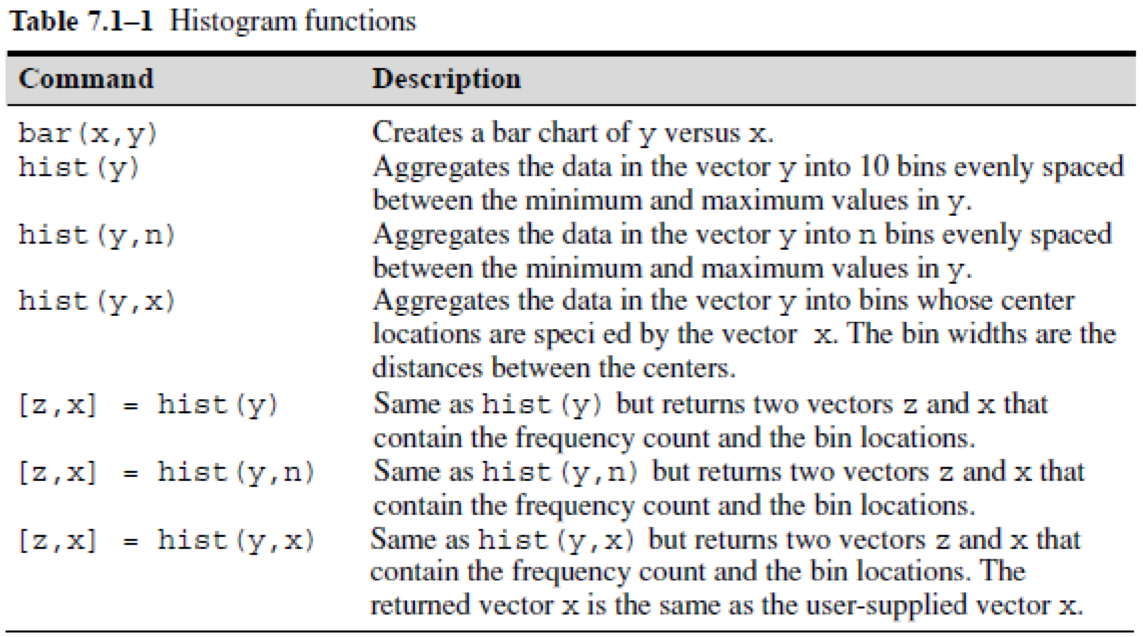


Table 7.1–1 Histogram functions

Command	Description
`bar(x,y)`	Creates a bar chart of `y` versus `x`.
`hist(y)`	Aggregates the data in the vector `y` into 10 bins evenly spaced between the minimum and maximum values in `y`.
`hist(y,n)`	Aggregates the data in the vector `y` into `n` bins evenly spaced between the minimum and maximum values in `y`.
`hist(y,x)`	Aggregates the data in the vector `y` into bins whose center locations are speci ed by the vector `x`. The bin widths are the distances between the centers.
`[z,x] = hist(y)`	Same as `hist(y)` but returns two vectors `z` and `x` that contain the frequency count and the bin locations.
`[z,x] = hist(y,n)`	Same as `hist(y,n)` but returns two vectors `z` and `x` that contain the frequency count and the bin locations.
`[z,x] = hist(y,x)`	Same as `hist(y,x)` but returns two vectors `z` and `x` that contain the frequency count and the bin locations. The returned vector `x` is the same as the user-supplied vector `x`.

```
x = 91:96;
[z,x] = hist(y,x);bar(x,z/tests),...
ylabel(‘Relative Frequency’),xlabel(‘Thread Strength(N)’),...
title(‘Relative Frequency Histogram for 100 Tests’)
```

The Normal Distribution

Rolling a die is an example of a process whose possible outcomes are a limited set of numbers, namely, the integers from 1 to 6. For such processes the probability is a function of a discrete-valued variable, that is, a variable having a limited number of values. For example, Table 7.2.1 gives the measured heights of 100 men 20 years of age. The heights were recorded to the nearest 1/2 in., so the height variable is discrete-valued.

Scaled Frequency Histogram

You can plot the data as a histogram using either the absolute or relative frequencies. However, another useful histogram uses data scaled so that the total area under the histogram.s rectangles is 1. This *scaled frequency histogram* is the absolute frequency histogram divided by the total area of that histogram. The area of each rectangle on the absolute frequency histogram equals the bin width times the absolute frequency for that bin. Because all the rectangles have the same width, the total area is the bin width times the sum of the absolute frequencies. The following M-file produces the scaled histogram shown in Figure 7.2.1.

Table 7.2–1 Height data for men 20 years of age

Height (in.)	Frequency	Height (in.)	Frequency
64	1	70	9
64.5	0	70.5	8
65	0	71	7
65.5	0	71.5	5
66	2	72	4
66.5	4	72.5	4
67	5	73	3
67.5	4	73.5	1
68	8	74	1
68.5	11	74.5	0
69	12	75	1
69.5	10		

```
% Absolute frequency data.
y_abs=[1,0,0,0,2,4,5,4,8,11,12,10,9,8,7,5,4,4,3,1,1,0,1];
binwidth = 0.5;
% Compute scaled frequency data.
area = binwidth*sum(y_abs);
y_scaled = y_abs/area;
```

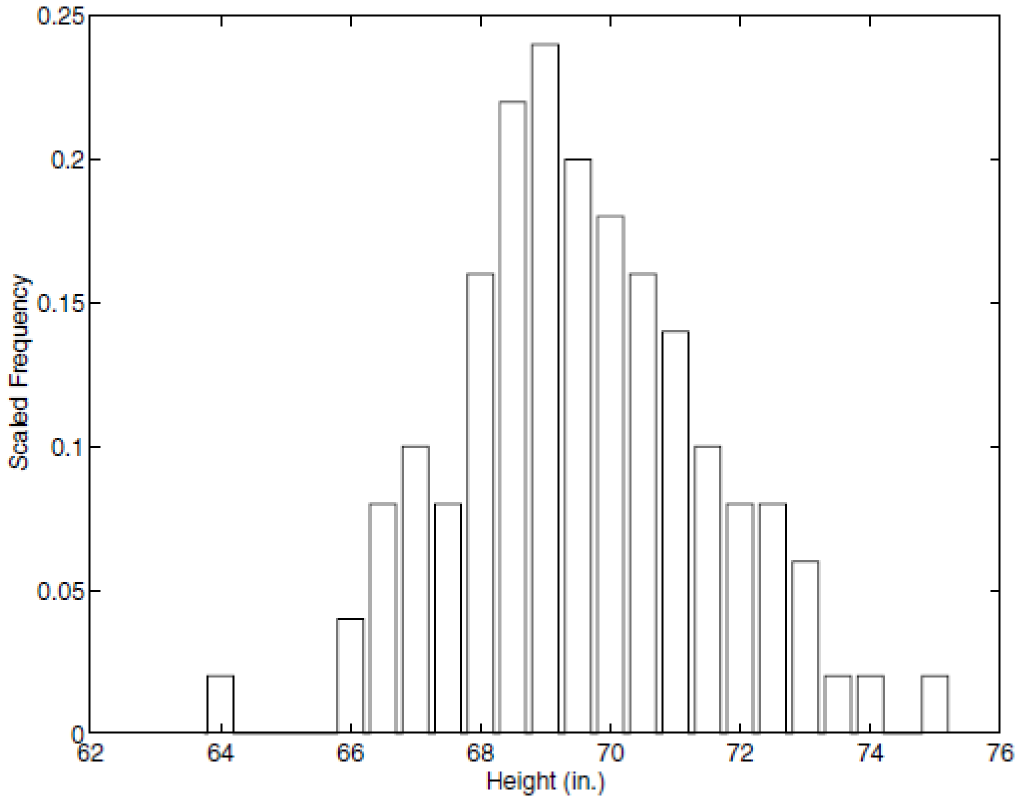


Figure 7.2–1 Scaled histogram of height data.

```
% De ne the bins.
bins = 64:binwidth:75;
% Plot the scaled histogram.
bar(bins,y_scaled),...
ylabel('Scaled Frequency'),xlabel('Height (in.)')
```

Because the total area under the scaled histogram is 1, the fractional area corresponding to a range of heights gives the probability that a randomly selected 20-year-old man will have a height in that range. For example, the heights of the scaled histogram rectangles corresponding to heights of 67 through 69 in. are 0.1, 0.08, 0.16, 0.22, and 0.24. Because the bin width is 0.5, the total area corresponding to these rectangles is (0.1_ 0.08 _ 0.16 _ 0.22 _ 0.24)(0.5) _ 0.4.
Thus 40 percent of the heights lie between 67 and 69 in.
You can use the cumsum function to calculate areas under the scaled frequency histogram and therefore to calculate probabilities. If x is a vector, cumsum(x) returns a vector the same length as x, whose elements are the sum of the previous elements. For example, if x =[2, 5, 3, 8], cumsum(x)= [2, 7, 10, 18]. If A is a matrix, cumsum(A) computes the cumulative sum of each row. The result is a matrix the same size as A.
After running the previous script, the last element of cumsum(y_scaled)* binwidth is 1, which is the area under the scaled frequency histogram. To compute the probability of a height lying between 67 and 69 in. (that is, above the 6th value up to the 11th value), type

```
>>prob = cumsum(y_scaled)*binwidth;
>>prob67_69 = prob(11)-prob(6)
```

The result is prob67_69 = 0.4000, which agrees with our previous calculation of 40 percent.

Continuous Approximation to the Scaled Histogram

For processes having an in nite number of possible outcomes, the probability is a function of a *continuous* variable and is plotted as a curve rather than as rectangles. It is based on the same concept as the scaled histogram; that is, the total area under the curve is 1, and the fractional area gives the probability of occurrence of a speci c range of outcomes. A probability function that describes many processes is the *normal* or *Gaussian* function, which is shown in Figure 7.2.2. This function is also known as the *bell-shaped curve*. Outcomes that can be described by this function are said to be *normally distributed.* The normal probability function is a two-parameter function; one parameter, μ, is the mean of the outcomes, and the other parameter, μ, is the *standard deviation.* The mean μ locates the peak of the curve and is the most likely value to occur. The width, or spread, of the curve is described by the parameter σ. Sometimes the term *variance* is used to describe the spread of the curve. The variance is the square of the standard deviation σ.

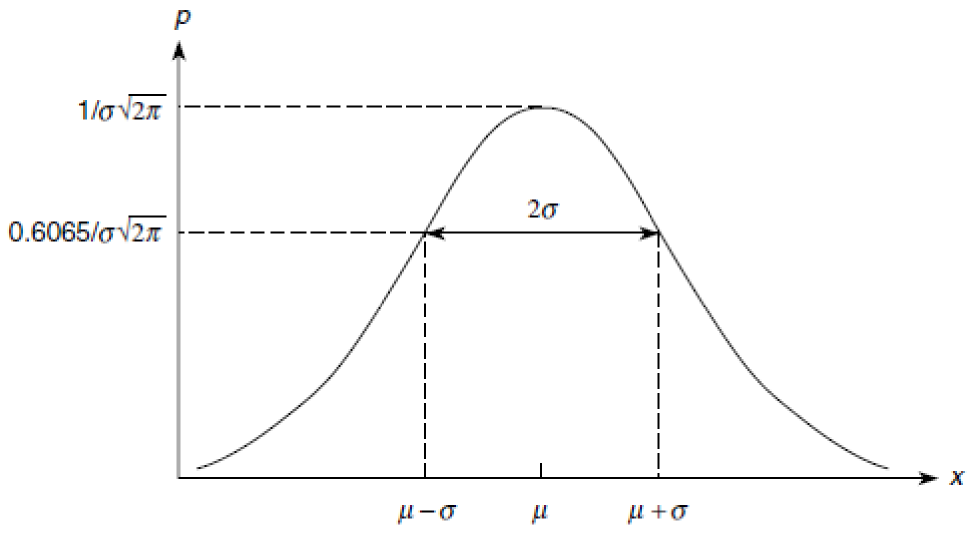


Figure 7.2–2 The basic shape of the normal distribution curve.

The normal probability function is described by the following equation:

$$p(x) = \frac{1}{\sigma\sqrt{2\pi}} e^{-(x-\mu)^2/2\sigma^2} \qquad (7.2\text{–}1)$$

It can be shown that approximately 68 percent of the area lies between the limits of $\mu - \sigma \le x \le \mu + \sigma$. Consequently, if a variable is normally distributed, there is a 68 percent chance that a randomly selected sample will lie within one standard deviation of the mean. In addition, approximately 96 percent of the area lies between the limits of $\mu - 2\sigma \le x \le \mu + 2\sigma$, and 99.7 percent, or practically 100 percent, of the area lies between the limits of $\mu - 3\sigma \le x \le \mu + 3\sigma$.

The functions `mean(x)`, `var(x)`, and `std(x)` compute the mean, variance, and standard deviation of the elements in the vector `x`.

Random Number Generation

We often do not have a simple probability distribution to describe the distribution of outcomes in many engineering applications. For example, the probability that a circuit consisting of many components will fail is a function of the number and the age of the components, but we often cannot obtain a function to describe the failure probability. In such cases we often resort to simulation to make predictions. The simulation program is executed many times, using a random set of numbers to represent the failure of one or more components, and the results are used to estimate the desired probability.

Uniformly Distributed Numbers

In a sequence of *uniformly distributed* random numbers, all values within a given interval are equally likely to occur. The MATLAB function rand generates random numbers uniformly distributed over the interval [0,1]. Type rand to obtain a single random number in the interval [0,1]. Typing rand again generates a different number because the MATLAB algorithm used for the rand function requires a .state. to start. MATLAB obtains this state from the computer.s CPU clock. Thus every time the rand function is used, a different result will be obtained. For example,

```
rand
ans =
0.6161
rand
ans =
0.5184
```

Type rand(n) to obtain an *nxn* matrix of uniformly distributed random numbers in the interval [0, 1]. Type rand(m,n) to obtain an *m* x *n* matrix of random numbers. For example, to create a 1x100 vector y having 100 random values in the interval [0, 1], type y = rand(1,100). Using the rand function this way is equivalent to typing rand 100 times. Even though there is a single call to the rand function, the rand functions calculation has the effect of using a different state to obtain each of the 100 numbers so that they will be random. Use Y = rand(m,n,p,...) to generate a multidimensional array Y having random elements. Typing rand(size(A)) produces an array of random entries that is the same size as A.

For example, the following script makes a random choice between two equally probable alternatives.

```
if rand < 0.5
disp(‘heads’)
else
disp(‘tails’)
end
```

To compare the results of two or more simulations, sometimes you will need to generate the same sequence of random numbers each time the simulation runs. To generate the same sequence, you must use the same state each time. The current state s of the uniform number generator can be obtained by typing
s = rand(‘twister’).
This returns a vector containing the current state of the uniform generator. To set the state of the generator to s, type rand(‘twister’,s). Typing rand(‘twister’,0) resets the generator to its initial state. Typing rand(‘twister’,j), for integer j, resets the generator to state j. Typing rand(‘twister’,sum(100*clock)) resets the generator to a different state each time. Table 7.3.1 summarizes these functions. The name ‘twister’ refers to the speci c algorithm used by MA TLAB to generate random numbers. In MATLAB Version 4, ‘seed’ was used instead of ‘twister’. In Versions 5 through 7.3, ‘state’ was used. Use ‘twister’ in Version 7.4 and later. The following session shows how to obtain the same sequence every time rand is called.

```
>>rand(‘twister’,0)
>>rand
ans =
0.5488
>>rand
ans =
0.7152
>>rand(‘twister’,0)
>>rand
ans =
0.5488
>>rand
```

```
ans =
0.7152
```
You need not start with the initial state to generate the same sequence. To show this, continue the above session as follows.
```
>>s = rand('twister');
>>rand('twister',s)
>>rand
ans =
0.6028
>>rand('twister',s)
>>rand
ans =
0.6028
```

Table 7.3–1 Random number functions

Command	Description
`rand`	Generates a single uniformly distributed random number between 0 and 1.
`rand(n)`	Generates an $n \times n$ matrix containing uniformly distributed random numbers between 0 and 1.
`rand(m,n)`	Generates an $m \times n$ matrix containing uniformly distributed random numbers between 0 and 1.
`s = rand('state')`	Returns a vector `s` containing the current state of the uniformly distributed generator.
`rand('twister',s)`	Sets the state of the uniformly distributed generator to `s`.
`rand('twister',0)`	Resets the uniformly distributed generator to its initial state.
`rand('twister',j)`	Resets the uniformly distributed generator to state `j`, for integer `j`.
`rand('twister',sum(100*clock))`	Resets the uniformly distributed generator to a different state each time it is executed.
`randn`	Generates a single normally distributed random number having a mean of 0 and a standard deviation of 1.
`randn(n)`	Generates an $n \times n$ matrix containing normally distributed random numbers having a mean of 0 and a standard deviation of 1.
`randn(m,n)`	Generates an $m \times n$ matrix containing normally distributed random numbers having a mean of 0 and a standard deviation of 1.
`s = randn('state')`	Like `rand('state')` but for the normally distributed generator.
`randn('state',s)`	Like `rand('state',s)` but for the normally distributed generator.
`randn('state',0)`	Like `rand('state',0)` but for the normally distributed generator.
`randn('state',j)`	Like `rand('state',j)` but for the normally distributed generator.
`randn('state',sum(100*clock))`	Like `rand('state',sum(100*clock))` but for the normally distributed generator.
`randperm(n)`	Generates a random permutation of the integers from `1` to `n`.

Interpolation

Paired data might represent a *cause and effect,* or *input-output relationship,* such as the current produced in a resistor as a result of an applied voltage, or a *time history,* such as the temperature of an object as a function of time. Another type of paired data represents a *pro le,* such as a road pro le (which shows the height of the road along its length). In some applications we want to estimate a variable .s value between the data points. This process is called *interpolation*. In other cases we might need to estimate the variable.s value outside the given data range. This process is called *extrapolation*. Interpolation and extrapolation are greatly aided by plotting the data. Such plots, some perhaps using logarithmic axes, often help to discover a functional description of the data. Suppose we have the following temperature measurements, taken once an hour starting at 7:00 A.M. The measurements at 8 and 10 A.M. are missing for some reason, perhaps because of equipment malfunction.

Time	7 A.M.	9 A.M.	11 A.M.	12 noon
Temperature (°F)	49	57	71	75

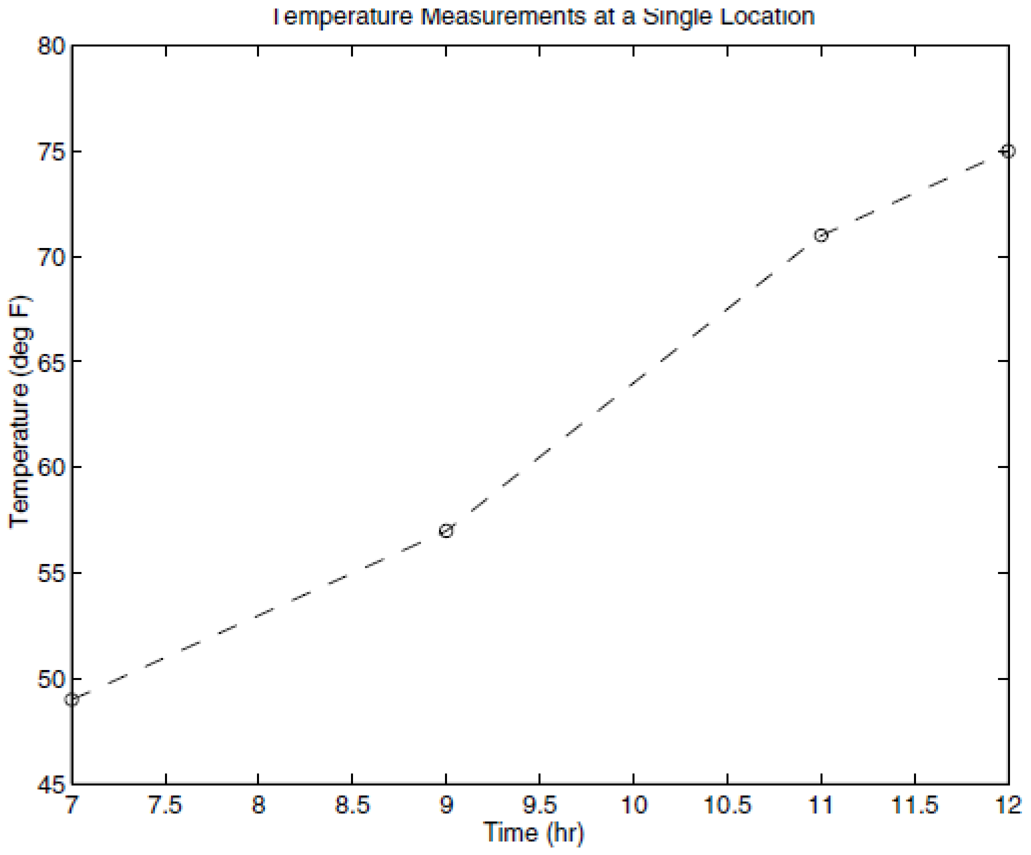


Figure 7.4–1 A plot of temperature data versus time.

A plot of these data is shown in Figure 7.4.1 with the data points connected by dashed lines. If we need to estimate the temperature at 10 A.M., we can read the value from the dashed line that connects the data points at 9 and 11 A.M. From the plot we thus estimate the temperature at 8 A.M. to be 53_F and at 10 A.M. to be 64_F. We have just performed *linear interpolation* on the data to obtain an *estimate* of the missing data. Linear interpolation is so named because it is equivalent to connecting the data points with a linear function (a straight line). Of course we have no reason to believe that the temperature follows the straight lines shown in the plot, and our estimate of 64_F will most likely be incorrect, but it might be close enough to be useful. Using straight lines to connect the data points is the simplest form of interpolation. Another function could be used if we have a good reason to do so. Later in this section we use polynomial functions to do the interpolation.
Linear interpolation in MATLAB is obtained with the interp1 and interp2 functions. Suppose that x is a vector containing the independent variable data and that y is a vector containing the dependent variable data. If x_int is a vector containing the value or values of the independent variable at which we wish to

estimate the dependent variable, then typing interp1(x,y,x_int) produces a vector the same size as x_int containing the interpolated values of y that correspond to x_int. For example, the following session produces an estimate of the temperatures at 8 and 10 A.M. from the preceding data. The vectors
x and y contain the times and temperatures, respectively.

```
>>x = [7, 9, 11, 12];
>>y = [49, 57, 71, 75];
>>x_int = [8, 10];
>>interp1(x,y,x_int)
ans =
53
64
```

You must keep in mind two restrictions when using the interp1 function. The values of the independent variable in the vector x must be in ascending order, and the values in the interpolation vector x_int must lie within the range of the values in x. Thus we cannot use the interp1 function to estimate the temperature at 6 A.M., for example. The interp1 function can be used to interpolate in a table of values by de ning y to be a matrix instead of a vector. For example, suppose that we now have temperature measurements at three locations and the measurements at 8 and 10 A.M. are missing for all three locations. The data are as follows:

	Temperature (°F)		
Time	**Location 1**	**Location 2**	**Location 3**
7 A.M.	49	52	54
9 A.M.	57	60	61
11 A.M.	71	73	75
12 noon	75	79	81

We de ne x as before, but now we de ne y to be a matrix whose three columns contain the second, third, and fourth columns of the preceding table. The following session produces an estimate of the temperatures at 8 and 10 A.M. at each location.

```
>>x = [7, 9, 11, 12]’;
>>y(:,1) = [49, 57, 71, 75]’;
>>y(:,2) = [52, 60, 73, 79]’;
>>y(:,3) = [54, 61, 75, 81]’;
>>x_int = [8, 10]’;
>>interp1(x,y,x_int)
ans =
     53.0000      56.0000      57.5000
     64.0000      65.5000      68.0000
```

Thus the estimated temperatures at 8 A.M. at each location are 53, 56, and 57.5°F, respectively. At 10 A.M. the estimated temperatures are 64, 65.5, and 68°F. From

this example we see that if the rst ar gument x in the interp1(x,y,x_int) function is a *vector* and the second argument y is a *matrix,* then the function interpolates between the rows of y and computes a matrix having the same number of columns as y and the same number of rows as the number of values in x_int.
Note that we need not de ne two separate vectors x and y. Rather, we can de ne a single matrix that contains the entire table. For example, by defining the matrix temp to be the preceding table, the session will look like this:

```
>>temp(:,1) = [7, 9, 11, 12]';
>>temp(:,2) = [49, 57, 71, 75]';
>>temp(:,3) = [52, 60, 73, 79]';
>>temp(:,4) = [54, 61, 75, 81]';
>>x_int = [8, 10]';
>>interp1(temp(:,1),temp(:,2:4),x_int)
ans =
53.0000 56.0000 57.5000
64.0000 65.5000 68.0000
```

Two-Dimensional Interpolation

Now suppose that we have temperature measurements at four locations at 7 A.M. These locations are at the corners of a rectangle 1 mi wide and 2 mi long. Assigning a coordinate system origin (0,0) to the rst location, the coordinates of the other locations are (1, 0), (1, 2), and (0, 2); see Figure 7.4.2. The temperature measurements are shown in the gure. The temperature is a function of two

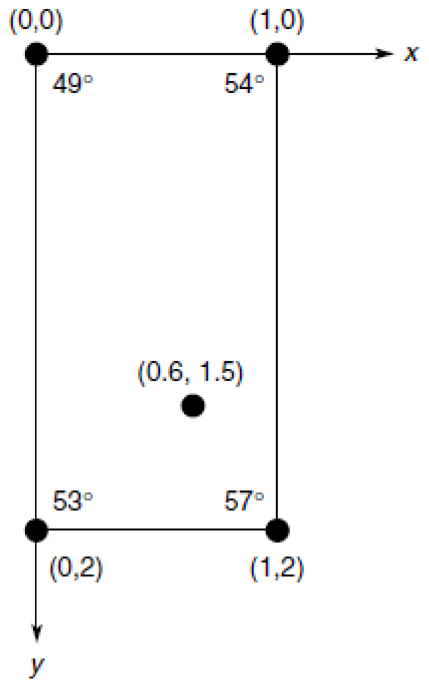


Figure 7.4–2 Temperature measurements at four locations.

variables, the coordinates *x* and *y*. MATLAB provides the interp2 function to interpolate functions of two variables. If the function is written as *z* _ *f*(*x*,*y*) and we wish to estimate the value of *z* for *x* _ *xi* and *y* _ *yi*, the syntax is

interp2(x,y,z,x_i,y_i).

Suppose we want to estimate the temperature at the point whose coordinates are (0.6, 1.5). Put the *x* coordinates in the vector x and the *y* coordinates in the vector y. Then put the temperature measurements in a matrix z such that going across a row represents an increase in *x* and going down a column represents an increase in *y*. The session to do this is as follows:

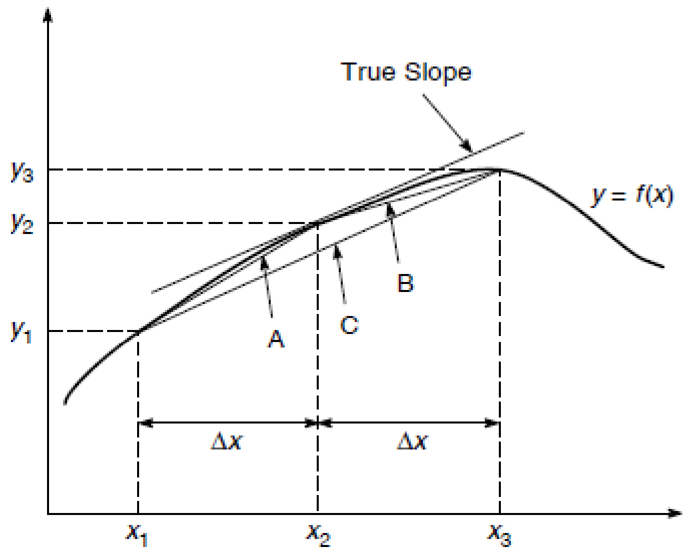
```
>>x = [0,1];
>>y = [0,2];
>>z = [49,54;53,57]
z =
49 54
53 57
>>interp2(x,y,z,0.6,1.5)
ans =
54.5500
```

Thus the estimated temperature is 54.55_F.
The syntax of the interp1 and interp2 functions is summarized in Table 7.4.1. MATLAB also provides the interpn function for interpolating multidimensional arrays.

Numerical Differentiation

The derivative of a function can be interpreted graphically as the slope of the function. This interpretation leads to various methods for computing the derivative of a set of data. Figure 9.2.1 shows three data points that represent a function *y*(*x*). Recall that the de nition of the derivative is

$$\frac{dy}{dx} = \lim_{\Delta x \to 0} \frac{\Delta y}{\Delta x}$$



The success of numerical differentiation depends heavily on two factors: the spacing of the data points and the scatter present in the data due to measurement error. The greater the spacing, the more dif cult it is to estimate the derivative. We assume here that the spacing between the measurements is regular; that is, *x*3 -*x*2= *x*2 - *x*1=Δ*x*. Suppose we want to estimate the derivative *dy*/*dx* at the point *x*2. The correct answer is the slope of the straight line passing through the point (*x*2, *y*2); but we do not have a second point on that line, so we cannot nd its slope. Therefore, we must estimate the slope by using nearby data points. One estimate can be obtained from the straight line labeled *A* in the figure. Its slope is

$$m_A = \frac{y_2 - y_1}{x_2 - x_1} = \frac{y_2 - y_1}{\Delta x}$$

This estimate of the derivative is called the *backward difference* estimate, and it is actually a better estimate of the derivative at *x* = *x*1 + (Δ*x*)/2 than at *x* = *x*2. Another estimate can be obtained from the straight line labeled *B*. Its slope is

$$m_B = \frac{y_3 - y_2}{x_3 - x_2} = \frac{y_3 - y_2}{\Delta x}$$

This estimate is called the *forward difference* estimate, and it is a better estimate of the derivative at *x* = *x*2 + (Δ*x*)_2 than at *x* = *x*2. Examining the plot, you might think that the average of these two slopes would provide a better estimate of the derivative at *x* = *x*2, because the average tends to cancel out the effects of measurement error. The average of *mA* and *mB* is

$$m_C = \frac{m_A + m_B}{2} = \frac{1}{2}\left(\frac{y_2 - y_1}{\Delta x} + \frac{y_3 - y_2}{\Delta x}\right) = \frac{y_3 - y_1}{2\,\Delta x}$$

This is the slope of the line labeled *C*, which connects the rst and third data points. This estimate of the derivative is called the *central difference* estimate

First-Order Differential Equations

In this section, we introduce numerical methods for solving rst-order dif ferential equations. In Section 9.4 we show how to extend the techniques to higherorder equations. An *ordinary differential equation* (ODE) is an equation containing ordinary derivatives of the dependent variable. An equation containing partial derivatives with respect to two or more independent variables is a *partial differential equation* (PDE). Solution methods for PDEs are an advanced topic, and we will not treat them in this text. In this chapter we limit ourselves to *initial-value problems* (IVPs). These are problems where the ODE must be solved for a given set of values speci ed at some initial time, which is usually taken to be *t* 0. Other types of ODE problems are discussed at the end of Section 9.6. It will be convenient to use the following abbreviated .dot. notation for derivatives.

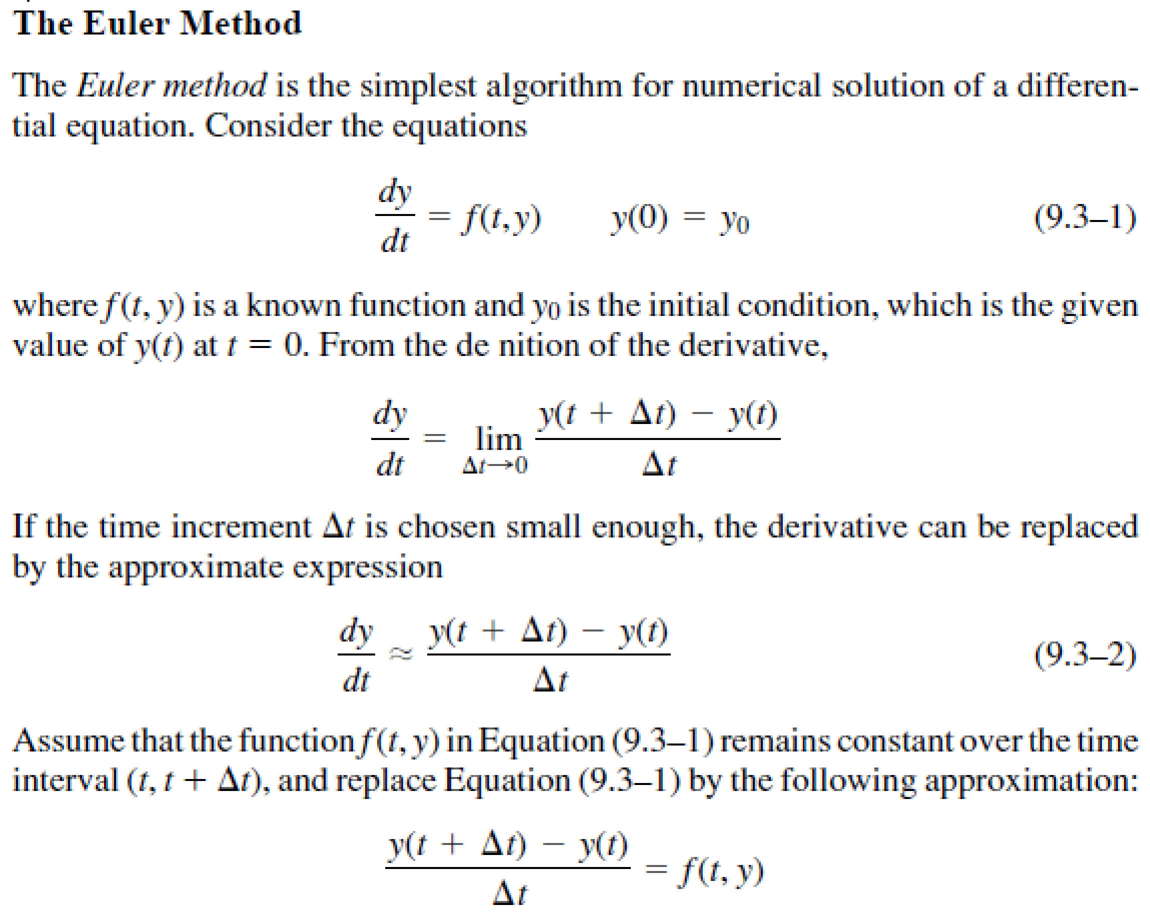
$$\dot{y}(t) = \frac{dy}{dt} \qquad \ddot{y}(t) = \frac{d^2y}{dt^2}$$

The *free response* of a differential equation, sometimes called the homogeneous solution or the initial response, is the solution for the case where there is no forcing function. The free response depends on the initial conditions. The *forced response* is the solution due to the forcing function when the initial conditions are zero. For linear differential equations, the complete or total response is the sum of the free and the forced responses. Nonlinear ODEs can be recognized by the fact that the dependent variable or its derivatives appear raised to a power or in a transcendental function. For example, the equations $\dot{y} = y^2$ and $\dot{y} = \cos y$ are nonlinear.

The essence of a numerical method is to convert the differential equation into a difference equation that can be programmed. Numerical algorithms differ partly as a result of the speci c procedure used to obtain the dif ference equations. It is important to understand the concept of .step size. and its effects on solution accuracy. To provide a simple introduction to these issues, we consider the simplest numerical methods, the *Euler method* and the *predictor-corrector method*.

The Euler Method

The *Euler method* is the simplest algorithm for numerical solution of a differential equation. Consider the equations



The Euler Method

The *Euler method* is the simplest algorithm for numerical solution of a differential equation. Consider the equations

$$\frac{dy}{dt} = f(t,y) \qquad y(0) = y_0 \tag{9.3–1}$$

where $f(t, y)$ is a known function and y_0 is the initial condition, which is the given value of $y(t)$ at $t = 0$. From the de nition of the derivative,

$$\frac{dy}{dt} = \lim_{\Delta t \to 0} \frac{y(t + \Delta t) - y(t)}{\Delta t}$$

If the time increment Δt is chosen small enough, the derivative can be replaced by the approximate expression

$$\frac{dy}{dt} \approx \frac{y(t + \Delta t) - y(t)}{\Delta t} \tag{9.3–2}$$

Assume that the function $f(t, y)$ in Equation (9.3–1) remains constant over the time interval $(t, t + \Delta t)$, and replace Equation (9.3–1) by the following approximation:

$$\frac{y(t + \Delta t) - y(t)}{\Delta t} = f(t, y)$$

or

$$y(t + \Delta t) = y(t) + f(t, y)\Delta t \tag{9.3–3}$$

The smaller Δt is, the more accurate are our two assumptions leading to Equation (9.3–3). This technique for replacing a differential equation with a difference equation is the *Euler method*. The increment Δt is called the *step size*.

Equation (9.3–3) can be written in more convenient form as

$$y(t_{k+1}) = y(t_k) + \Delta t f[t_k, y(t_k)] \tag{9.3–4}$$

where $t_{k+1} = t_k + \Delta t$. This equation can be applied successively at the times t_k by putting it in a `for` loop. The accuracy of the Euler method can be improved sometimes by using a smaller step size. However, very small step sizes require longer run times and can result in a large accumulated error due to roundoff effects.

Runge-Kutta Methods

The Taylor series representation forms the basis of several methods of solving differential equations, including the Runge-Kutta methods. The Taylor series may be used to represent the solution $y(t + h)$ in terms of $y(t)$ and its derivatives as follows.

$$y(t + h) = y(t) + h\dot{y}(t) + \frac{1}{2}h^2\ddot{y}(t) + \cdots \qquad (9.3\text{–}9)$$

The number of terms kept in the series determines its accuracy. The required derivatives are calculated from the differential equation. If these derivatives can be found, Equation (9.3–9) can be used to march forward in time. In practice, the high-order derivatives can be dif cult to calculate, and the series (9.3–9) is truncated at some term. The Runge-Kutta methods were developed because of the dif culty in computing the derivatives. These methods use several evaluations of the function $f(t, y)$ in a way that approximates the Taylor series. The number of terms in the series that is duplicated determines the order of the Runge-Kutta method. Thus, a fourth-order Runge-Kutta algorithm duplicates the Taylor series through the term involving h^4.

MATLAB ODE Solvers

In addition to the many variations of the predictor-corrector and Runge-Kutta algorithms that have been developed, there are more-advanced algorithms that use a variable step size. These "adaptive" algorithms use larger step sizes when the solution is changing more slowly. MATLAB provides several functions, called *solvers*, that implement the Runge-Kutta and other methods with variable step size. Two of these are the `ode45` and `ode15s` functions. The `ode45` function uses a combination of fourth- and fth-order Runge-Kutta methods. It is a general-purpose solver whereas `ode15s` is suitable for more-dif cult equations called "stiff" equations. These solvers are more than suf cient to solve the problems in this text. It is recommended that you try `ode45` rst. If the equation proves dif cult to solve (as indicated by a lengthy solution time or by a warning or error message), then use `ode15s`.

In this section we limit our coverage to rst-order equations. Solution of higher-order equations is covered in Section 9.4. When used to solve the equation $\dot{y} = f(t, y)$, the basic syntax is (using `ode45` as the example)

```
[t,y] = ode45(@ydot, tspan, y0)
```

where `@ydot` is the handle of the function le whose inputs must be t and y, and whose output must be a column vector representing dy/dt, that is, $f(t, y)$. The number of rows in this column vector must equal the order of the equation. The syntax for `ode15s` is identical. The function le `ydot` may also be speci ed by a character string (i.e., its name placed in single quotes), but use of the function handle is now the preferred approach.

The vector `tspan` contains the starting and ending values of the independent variable t, and optionally any intermediate values of t where the solution is desired. For example, if no intermediate values are speci ed, `tspan is [t0, tfinal]`, where `t0` and `tfinal` are the desired starting and ending values of the independent parameter *t*. As another example, using `tspan = [0, 5, 10]` tells MATLAB to nd the solution at $t = 5$ and at $t = 10$. You can solve equation backward in time by specifying `t0` to be greater than `tfinal`.

The parameter `y0` is the initial value y(0). The function le must have its rst two input arguments as `t` and `y` in that order, even for equations where $f(t, y)$ is not a function of *t*. You need not use array operations in the function le because the ODE solvers call the le with scalar values for the ar guments. The solvers may have an additional argument, `options`, which is discussed at the end of this section.

First consider an equation whose solution is known in closed form, so that we can make sure we are using the method correctly.

The model of the *RC* circuit shown in Figure 9.3–1 can be found from Kirchhoff's voltage law and conservation of charge. It is $RC\dot{y} + y = v(t)$. Suppose the value of *RC* is 0.1 s. Use a numerical method to nd the free response for the case where the applied voltage `y` is zero and the initial capacitor voltage is $y(0) = 2$ V. Compare the results with the analytical solution, which is $y(t) = 2e^{-10t}$.

Solution

The equation for the circuit becomes $0.1\dot{y} + y = 0$. First solve this for y: $\dot{y} = -10y$. Next de ne and save the following function le. Note that the order of the input arguments must be *t* and *y* even though *t* does not appear on the right-hand side of the equation.

```
function ydot = RC_circuit(t,y)
% Model of an RC circuit with no applied voltage.
ydot = -10*y;
```

The initial time is $t = 0$, so set `t0` to be 0. Here we know from the analytical solution that $y(t)$ will be close to 0 for $t \geq 0.5$ s, so we choose `tfinal` to be 0.5 s. In other problems we generally do not have a good guess for `tfinal`, so we must try several increasing values of `tfinal` until we see enough of the response on the plot.

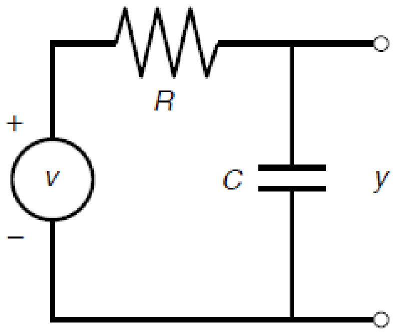
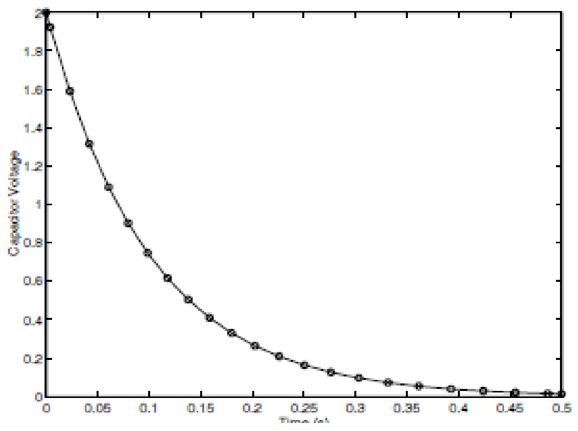


Figure 9.3–1 An *RC* circuit.



The function ode45 is called as follows, and the solution plotted along with the analytical solution y_true.

```
[t, y] = ode45(@RC_circuit, [0, 0.5], 2);
y_true = 2*exp(-10*t);
plot(t,y,'o',t,y_true), xlabel('Time(s)'),...
   ylabel('Capacitor Voltage')
```

Note that we need not generate the array t to evaluate y_true because t is generated by the ode45 function. The plot is shown in Figure 9.3–2. The numerical solution is marked by the circles, and the analytical solution is indicated by the solid line. Clearly the numerical solution gives an accurate answer. Note that the step size has been automatically selected by the ode45 function.

Higher-Order Differential Equations

To use the ODE solvers to solve an equation higher than order 1, you must rst write the equation as a set of rst-order equations. This is easily done. Consider the second-order equation.

$$5\ddot{y} + 7\dot{y} + 4y = f(t)$$

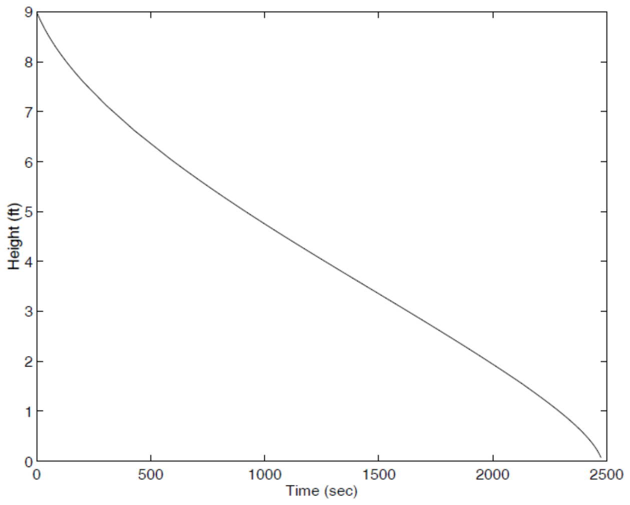


Figure 9.3–4 Plot of water height in a spherical tank.

Solve it for the highest derivative:

$$\ddot{y} = \frac{1}{5}f(t) - \frac{4}{5}y - \frac{7}{5}\dot{y} \qquad (9.4\text{–}2)$$

De ne two new variables x_1 and x_2 to be y and its derivative $\dot{y}$. That is, de ne $x_1 = y$ and $x_2 = \dot{y}$. This implies that

$$\dot{x}_1 = x_2$$

$$\dot{x}_2 = \frac{1}{5}f(t) - \frac{4}{5}x_1 - \frac{7}{5}x_2$$

This form is sometimes called the *Cauchy form* or the *state-variable form*.

Now write a function le that computes the values of $\dot{x}_1$ and $\dot{x}_2$ and stores them in a *column* vector. To do this, we must rst have a function speci ed for $f(t)$. Suppose that $f(t) = \sin t$. Then the required le is

```
function xdot = example_1(t,x)
% Computes derivatives of two equations
xdot(1) = x(2);
xdot(2) = (1/5)*(sin(t)-4*x(1)-7*x(2));
xdot = [xdot(1); xdot(2)];
```

Note that `xdot(1)` represents $\dot{x}_1$, `xdot(2)` represents $\dot{x}_2$, `x(1)` represents x_1, and `x(2)` represents x_2. Once you become familiar with the notation for the state-variable form, you will see that the previous code could be replaced with the following shorter form.

```
function xdot = example_1(t,x)
% Computes derivatives of two equations
xdot = [x(2); (1/5)*(sin(t)-4*x(1)-7*x(2))];
```

Suppose we want to solve Equation (9.4–1) for $0 \le t \le 6$ with the initial conditions $x(0) = 3$, $\dot{x}(0) = 9$. Then the initial condition for the *vector* `x` is `[3, 9]`. To use `ode45`, you type

```
[t, x] = ode45(@example_1, [0, 6], [3, 9]);
```

Each row in the vector `x` corresponds to a time returned in the column vector `t`. If you type `plot(t,x)`, you will obtain a plot of both x_1 and x_2 versus t. Note `x` is a matrix with two columns. The rst column contains the values of x_1 at the various times generated by the solver; the second column contains the values of x_2. Thus, to plot only x_1, type `plot(t,x(:,1))`. To plot only x_2, type `plot(t,x(:,2))`.

When we are solving nonlinear equations, sometimes it is possible to check the numerical results by using an approximation that reduces the equation to a linear one. The following example illustrates such an approach with a second-order equation.