# 2 Linear Equations and Eigensystems

When physical systems are modeled mathematically, they are sometimes described by linear equation systems or eigensystems, and in this chapter we will examine how such equation systems are solved. Linear equation systems can be expressed in terms of matrices and vectors, and we introduce some of the more important properties of vectors and matrices in Appendix A.

MATLAB is an ideal environment for studying linear algebra, including linear equation systems and eigenvalue problems, because MATLAB functions and operators can work directly on vectors and matrices. It is rich in functions and operators, which facilitate the manipulation of matrices. MATLAB originated as a set of linear algebra operators and functions based on the LINPACK (Dongarra et al., 1979) and EISPACK (Smith et al., 1976; Garbow et al., 1977) routines. These routines were developed specifically to solve linear equations and eigenvalue problems, respectively. In 2000, MATLAB began using the LAPACK library of linear algebra subroutines, which is the modern replacement for LINPACK and EISPACK.

## 2.1 Introduction

We start with a discussion of linear equation systems and defer discussion of eigensystems until Section 2.15. To illustrate how linear equation systems arise in the modeling of certain physical problems we will consider how current flows are calculated in a simple electrical network. The necessary equations can be developed using one of several techniques; here we use the loop-current method together with Ohm's law and Kirchhoff's voltage law. A *loop current* is assumed to circulate around each loop in the network. Thus, in the network given in Figure 2.1, the loop current $I_1$ circulates around the closed loop *abcd*. Note that the current $I_1 - I_2$ is assumed to flow in the link connecting $b$ to $c$. Ohm's law states that the voltage across an ideal resistor is proportional to the current flow through the resistor. For example, for the link connecting $b$ to $c$

$$V_{bc} = R_2(I_1 - I_2)$$

where $R_2$ is the value of the resistor in the link connecting $b$ to $c$. Kirchhoff's voltage law states that the algebraic sum of the voltages around a loop is zero. Applying these laws to the circuit *abcd* of Figure 2.1 we have

$$V_{ab} + V_{bc} + V_{cd} = V$$

**Numerical Methods Using MATLAB®**. DOI: 10.1016/B978-0-12-386942-5.00002-3

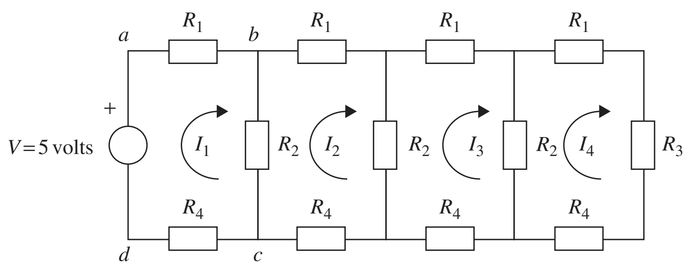

**FIGURE 2.1** Electrical network.

Substituting the product of current and resistance for voltage gives

$$R_1 I_1 + R_2(I_1 - I_2) + R_4 I_1 = V$$

We can repeat this process for each loop to obtain the following four equations:

$$\begin{aligned}(R_1 + R_2 + R_4)I_1 - R_2 I_2 &= V\\ (R_1 + 2R_2 + R_4)I_2 - R_2 I_1 - R_2 I_3 &= 0\\ (R_1 + 2R_2 + R_4)I_3 - R_2 I_2 - R_2 I_4 &= 0\\ (R_1 + R_2 + R_3 + R_4)I_4 - R_2 I_3 &= 0\end{aligned} \tag{2.1}$$

Letting $R_1 = R_4 = 1\Omega$, $R_2 = 2\Omega$, $R_3 = 4\Omega$, and $V = 5$ volts, (2.1) becomes

$$\begin{aligned}4I_1 - 2I_2 &= 5\\ -2I_1 + 6I_2 - 2I_3 &= 0\\ -2I_2 + 6I_3 - 2I_4 &= 0\\ -2I_3 + 8I_4 &= 0\end{aligned}$$

This is a system of linear equations in four variables, $I_1, \ldots, I_4$. In matrix notation it becomes

$$\begin{bmatrix} 4 & -2 & 0 & 0\\ -2 & 6 & -2 & 0\\ 0 & -2 & 6 & -2\\ 0 & 0 & -2 & 8\end{bmatrix}\begin{bmatrix} I_1\\ I_2\\ I_3\\ I_4\end{bmatrix} = \begin{bmatrix}5\\0\\0\\0\end{bmatrix} \tag{2.2}$$

This equation has the form $\mathbf{Ax} = \mathbf{b}$ where $\mathbf{A}$ is a square matrix of known coefficients, in this case relating to the values of the resistors in the circuit. The vector $\mathbf{b}$ is a vector of known coefficients, in this case the voltage applied to each current loop. The vector $\mathbf{x}$ is the vector of unknown currents. Although this set of equations can be solved by hand, the process is time consuming and error prone. Using MATLAB we simply enter matrix `A` and vector `b` and use the command `A\b` as follows:

```
>> A = [4 -2 0 0;-2 6 -2 0;0 -2 6 -2;0 0 -2 8];
>> b = [5 0 0 0].';
>> A\b

ans =
    1.5426
    0.5851
    0.2128
    0.0532
```

The sequence of operations that are invoked by this apparently simple command is examined in Section 2.3.

In many electrical networks the ideal resistors of Figure 2.1 are more accurately represented by electrical impedances. When a harmonic alternating current (AC) supply is connected to the network, electrical engineers represent the impedances by complex quantities. This is to account for the effect of capacitance and/or inductance. To illustrate this we will replace the 5 volt DC supply to the network of Figure 2.1 with a 5 volt AC supply and replace the ideal resistors $R_1, \ldots, R_4$ by impedances $Z_1, \ldots, Z_4$. Thus (2.1) becomes

$$\begin{aligned}(Z_1+Z_2+Z_4)I_1 - Z_2I_2 &= V\\ (Z_1+2Z_2+Z_4)I_2 - Z_2I_1 - Z_2I_3 &= 0\\ (Z_1+2Z_2+Z_4)I_3 - Z_2I_2 - Z_2I_4 &= 0\\ (Z_1+Z_2+Z_3+Z_4)I_4 - Z_2I_3 &= 0\end{aligned} \tag{2.3}$$

At the frequency of the 5 volt AC supply we will assume that $Z_1 = Z_4 = (1+0.5\jmath)$, $Z_2 = (2+0.5\jmath)$, and $Z_3 = (4+1\jmath)$, where $\jmath = \sqrt{-1}$. Electrical engineers prefer to use $\jmath$ rather than $\imath$ for $\sqrt{-1}$. This avoids any possible confusion with $I$ or $i$, which are normally used to denote the current in a circuit. Thus (2.3) becomes

$$\begin{aligned}(4+1.5\jmath)I_1 - (2+0.5\jmath)I_2 &= 5\\ -(2+0.5\jmath)I_1 + (6+2.0\jmath)I_2 - (2+0.5\jmath)I_3 &= 0\\ -(2+0.5\jmath)I_2 + (6+2.0\jmath)I_3 - (2+0.5\jmath)I_4 &= 0\\ -(2+0.5\jmath)I_3 + (8+2.5\jmath)I_4 &= 0\end{aligned}$$

This system of linear equations becomes, in matrix notation,

$$\begin{bmatrix}(4+1.5\jmath) & -(2+0.5\jmath) & 0 & 0\\ -(2+0.5\jmath) & (6+2.0\jmath) & -(2+0.5\jmath) & 0\\ 0 & -(2+0.5\jmath) & (6+2.0\jmath) & -(2+0.5\jmath)\\ 0 & 0 & -(2+0.5\jmath) & (8+2.5\jmath)\end{bmatrix}\begin{bmatrix}I_1\\ I_2\\ I_3\\ I_4\end{bmatrix} = \begin{bmatrix}5\\ 0\\ 0\\ 0\end{bmatrix} \tag{2.4}$$

Note that the coefficient matrix is now complex. This does not present any difficulty for MATLAB because the operation `A\b` works directly with both real and complex numbers. Thus

```
>> p = 4+1.5i; q = -2-0.5i;
>> r = 6+2i; s = 8+2.5i;
>> A = [p q 0 0;q r q 0;0 q r q;0 0 q s];
>> b = [5 0 0 0].';
>> A\b

ans =
   1.3008 - 0.5560i
   0.4560 - 0.2504i
   0.1530 - 0.1026i
   0.0361 - 0.0274i
```

Note that strictly we have no need to reenter the values in vector `b`, assuming that we have not cleared the memory, reassigned the vector `b`, or quit MATLAB. The answer shows that currents flowing in the network are complex. This means that there is a phase difference between the applied harmonic voltage and the currents flowing.

We will now begin a more detailed examination of linear equation systems.

## 2.2 Linear Equation Systems

In general, a linear equation system can be written in matrix form as

$$\mathbf{Ax} = \mathbf{b} \tag{2.5}$$

where $\mathbf{A}$ is an $n \times n$ matrix of known coefficients, $\mathbf{b}$ is a column vector of $n$ known coefficients, and $\mathbf{x}$ is the column vector of $n$ unknowns. We have already seen an example of this type of equation system in Section 2.1 where the matrix equation (2.2) is the matrix equivalent of the linear equations (2.1).

The equation system (2.5) is called homogeneous if $\mathbf{b} = \mathbf{0}$ and inhomogeneous if $\mathbf{b} \neq \mathbf{0}$. Before attempting to solve an equation system it is reasonable to ask if it has a solution and if so is it unique? A linear inhomogeneous equation system may be *consistent* and have one or an infinity of solutions or be *inconsistent* and have no solution. This is illustrated in Figure 2.2 for a system of three equations in three variables $x_1$, $x_2$, and $x_3$. Each equation represents a plane surface in the $x_1$, $x_2$, $x_3$ space. In Figure 2.2(a) the three planes have a common point of intersection. The coordinates of the point of intersection give the unique solution for the three equations. In Figure 2.2(b) the three planes intersect in a line. Any point on the line of intersection represents a solution so there is no unique solution but an infinite number of solutions satisfying the three equations. In Figure 2.2(c) two of the surfaces are parallel to each other and therefore they never intersect, while in Figure 2.2(d)

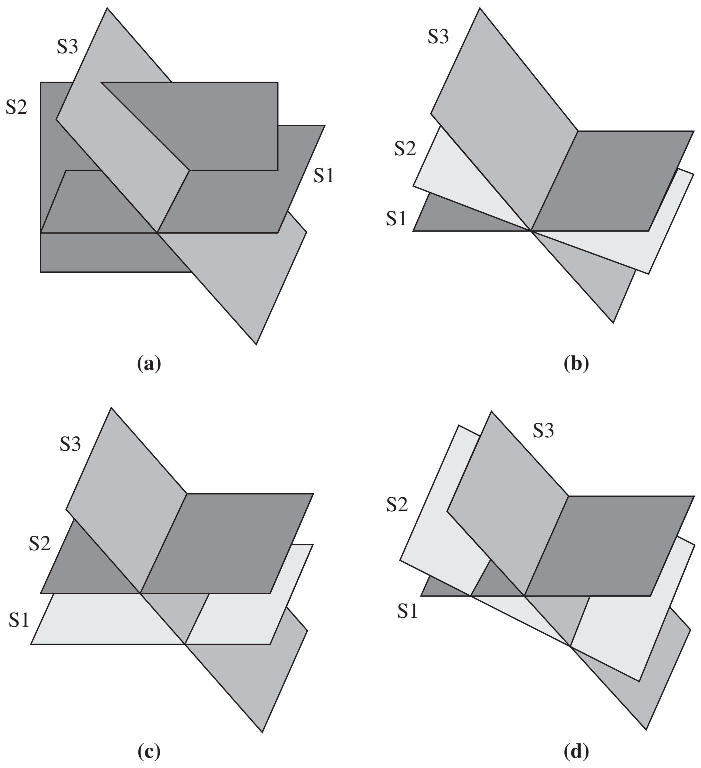

**FIGURE 2.2** Three intersecting planes representing three equations in three variables. (a) Three plane surfaces intersecting in a point. (b) Three plane surfaces intersecting in a line. (c) Three plane surfaces, two of which do not intersect. (d) Three plane surfaces intersecting in three lines.

the line of intersection of each pair of surfaces is different. In both of these cases there is no solution and the equations these surfaces represent are inconsistent.

To obtain an algebraic solution to the inhomogeneous equation system (2.5) we multiply both sides of (2.5) by a matrix called the inverse of **A**, denoted by $\mathbf{A}^{-1}$:

$$\mathbf{A}^{-1}\mathbf{A}\mathbf{x} = \mathbf{A}^{-1}\mathbf{b} \tag{2.6}$$

where $\mathbf{A}^{-1}$ is defined by

$$\mathbf{A}^{-1}\mathbf{A} = \mathbf{A}\,\mathbf{A}^{-1} = \mathbf{I} \tag{2.7}$$

and **I** is the identity matrix. Thus we obtain

$$\mathbf{x} = \mathbf{A}^{-1}\mathbf{b} \tag{2.8}$$

The standard algebraic formula for the inverse of **A** is

$$\mathbf{A}^{-1} = \text{adj}(\mathbf{A})/|\mathbf{A}| \tag{2.9}$$

where $|\mathbf{A}|$ is the determinant of **A** and adj(**A**) is the adjoint of **A**. The determinant and the adjoint of a matrix are defined in Appendix A. Equations (2.8) and (2.9) are algebraic statements allowing us to determine **x** but they do not provide an efficient means of solving the system because computing $\mathbf{A}^{-1}$ using (2.9) is extremely inefficient, involving order $(n+1)!$ multiplications where $n$ is the number of equations. However, (2.9) is theoretically important because it shows that if $|\mathbf{A}| = 0$ then **A** does not have an inverse. The matrix **A** is then said to be singular and a unique solution for **x** does not exist. Thus establishing that $|\mathbf{A}|$ is nonzero is one way of showing that an inhomogeneous equation system is a consistent system with a unique solution. It is shown in Sections 2.6 and 2.7 that (2.5) can be solved without formally determining the inverse of **A**.

An important concept in linear algebra is the rank of a matrix. For a square matrix, the rank is the number of independent rows or columns in the matrix. Independence can be explained as follows. The rows (or columns) of a matrix can clearly be viewed as a set of vectors. A set of vectors is said to be linearly independent if none of them can be expressed as a linear combination of any of the others. By linear combination we mean a sum of scalar multiples of the vectors. For example, the matrix

$$\begin{bmatrix} 1 & 2 & 3 \\ -2 & 1 & 4 \\ -1 & 3 & 4 \end{bmatrix} \text{ or } \begin{bmatrix} [1 & 2 & 3] \\ [-2 & 1 & 4] \\ [-1 & 3 & 7] \end{bmatrix} \text{ or } \begin{bmatrix} \begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix} & \begin{bmatrix} -2 \\ 1 \\ 4 \end{bmatrix} & \begin{bmatrix} -1 \\ 3 \\ 7 \end{bmatrix} \end{bmatrix}$$

has linearly *dependent* rows and columns. This is because row3 − row1 − row2 = 0 and column3 − 2(column2) + column1 = 0. There is only one equation relating the rows (or columns) and thus there are two independent rows (or columns). Hence this matrix has a rank of 2. Now consider

$$\begin{bmatrix} 1 & 2 & 3 \\ 2 & 4 & 6 \\ 3 & 6 & 9 \end{bmatrix}$$

Here row2 = 2(row1) and row3 = 3(row1). There are two equations relating the rows and hence only one row is independent and the matrix has a rank of 1. Note that the number of independent rows and columns in a square matrix is identical; that is, its row rank and column rank are equal. In general matrices may be nonsquare and the rank of an $m \times n$ matrix **A** is written rank(**A**). Matrix **A** is said to be of full rank if rank(**A**) = min($m$, $n$); otherwise rank(**A**) < min($m$, $n$) and **A** is said to be rank deficient. MATLAB provides the function `rank`, which works with both square and nonsquare matrices. In practice, MATLAB determines the rank of a matrix from its singular values; see Section 2.10.

For example, consider the following MATLAB statements:

```
>> D = [1 2 3;3 4 7;4 -3 1;-2 5 3;1 -7 6]

D =
     1     2     3
     3     4     7
     4    -3     1
    -2     5     3
     1    -7     6

>> rank(D)

ans =
     3
```

Thus `D` is of full rank since the rank equals the minimum size of the matrix.

A useful operation in linear algebra is the conversion of a matrix to its reduced row echelon form (RREF). The RREF is defined in Appendix A. In MATLAB we can use the `rref` function to compute the RREF of a matrix as follows:

```
>> rref(D)

ans =
     1     0     0
     0     1     0
     0     0     1
     0     0     0
     0     0     0
```

It is a property of the RREF of a matrix that the number of rows with at least one nonzero element equals the rank of the matrix. In this example we see that there are three rows in the RREF of the matrix containing a nonzero element, confirming that the matrix rank is 3. The RREF also allows us to determine whether a system has a unique solution or not.

We have discussed a number of important concepts relating to the nature of linear equations and their solutions. We now summarize the equivalencies between these concepts. Let $\mathbf{A}$ be an $n \times n$ matrix. If $\mathbf{Ax} = \mathbf{b}$ is consistent and has a unique solution, then all of the following statements are true:

$\mathbf{Ax} = \mathbf{0}$ has only the trivial solution $\mathbf{x} = \mathbf{0}$.

$\mathbf{A}$ is nonsingular and $\det(\mathbf{A}) \neq 0$.

The RREF of $\mathbf{A}$ is the identity matrix.

**A** has $n$ linearly independent rows and columns.

**A** has full rank, i.e., rank(**A**) $= n$.

In contrast, if $\mathbf{Ax} = \mathbf{b}$ is either inconsistent or consistent but with more than one solution, then all of the following statements are true:

$\mathbf{Ax} = \mathbf{0}$ has more than one solution.

**A** is singular and det(**A**) $= 0$.

The RREF of **A** contains at least one zero row.

**A** has linearly dependent rows and columns.

**A** is rank deficient, i.e., rank(**A**) $< n$.

So far we have only considered the case where there are as many equations as unknowns. Now we consider the cases where there are fewer or more equations than unknown variables.

If there are fewer equations than unknowns, then the system is said to be underdetermined. The equation system does not have a unique solution; it is either consistent with an infinity of solutions, or inconsistent with no solution. These conditions are illustrated by Figure 2.3. The diagram shows two plane surfaces in three-dimensional space, representing two equations in three variables. It is seen that the planes either intersect in a line so that the equations are consistent with an infinity of solutions represented by the line of intersection, or the surfaces do not intersect and the equations they represent are inconsistent.

Consider the following system of equations:

$$\begin{bmatrix} 1 & 2 & 3 & 4 \\ -4 & 2 & -3 & 7 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix} = \begin{bmatrix} 1 \\ 3 \end{bmatrix}$$

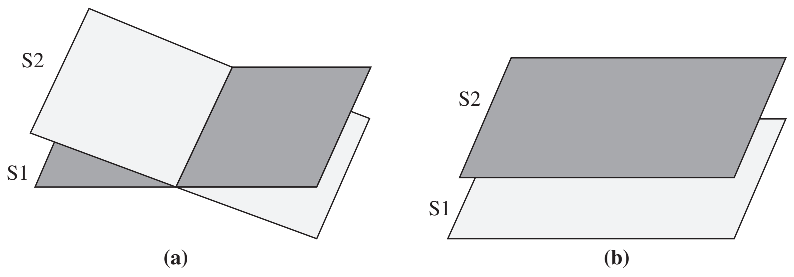

**FIGURE 2.3** Planes representing an underdetermined system of equations. (a) Two plane surfaces intersecting in a line. (b) Two plane surfaces which do not intersect.

This underdetermined system can be rearranged as follows:

$$\begin{bmatrix} 1 & 2 \\ -4 & 2 \end{bmatrix}\begin{bmatrix} x_1 \\ x_2 \end{bmatrix} + \begin{bmatrix} 3 & 4 \\ -3 & 7 \end{bmatrix}\begin{bmatrix} x_3 \\ x_4 \end{bmatrix} = \begin{bmatrix} 1 \\ 3 \end{bmatrix}$$

or

$$\begin{bmatrix} 1 & 2 \\ -4 & 2 \end{bmatrix}\begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} 1 \\ 3 \end{bmatrix} - \begin{bmatrix} 3 & 4 \\ -3 & 7 \end{bmatrix}\begin{bmatrix} x_3 \\ x_4 \end{bmatrix}$$

Thus we have reduced this to a system of two equations in two unknowns, provided values are assumed for $x_3$ and $x_4$. Thus the problem has an infinity of solutions, depending on the values chosen for $x_3$ and $x_4$.

If a system has more equations than unknowns, then the system is said to be overdetermined. Figure 2.4 shows four plane surfaces in three-dimensional space, representing four equations in three variables. Figure 2.4(a) shows all four planes intersecting in a single point so that the system of equations is consistent with a unique solution. Figure 2.4(b) shows all the planes intersecting in a line and this represents a consistent system with an infinity of solutions. Figure 2.4(d) shows planes that represent an inconsistent system of equations with no solution. In Figure 2.4(c) the planes do not intersect in a single point and so the system of equations is inconsistent. However, in this example the points of intersection of groups of three planes (i.e., (S1, S2, S3), (S1, S2, S4), (S1, S3, S4), and (S2, S3, S4)) are close to each other and a mean point of intersection could be determined and used as an approximate solution. This example of marginal inconsistency often arises because the coefficients in the equations are determined experimentally; if the coefficients were known exactly, it is likely that the equations would be consistent with a unique solution. Rather than accepting that the system is inconsistent we may ask what the best solution is that satisfies the equations approximately. In Sections 2.11 and 2.12 we deal with the problem of overdetermined and underdetermined systems in more detail.

## 2.3 Operators \ and / for Solving **Ax = b**

The purpose of this section is to introduce the reader to the MATLAB operator `\`. A detailed discussion of the algorithms behind its operation will be given in later sections. This operator is a very powerful one that provides a unified approach to the solution of many categories of linear equation systems. The operators `/` and `\` perform matrix "division" and have identical effects. Thus to solve $\mathbf{Ax} = \mathbf{b}$ we may write either `x=A\b` or `x'=b'/A'`. In the latter case the solution $\mathbf{x}$ is expressed as a row rather than a column vector. The operator `/` or `\`, when solving $\mathbf{Ax} = \mathbf{b}$, selects the appropriate algorithm dependent on the form of the matrix $\mathbf{A}$. These cases are outlined next:

- <u>if</u> $\mathbf{A}$ is a triangular matrix, the system is solved by back or forward substitution alone, described in Section 2.6.

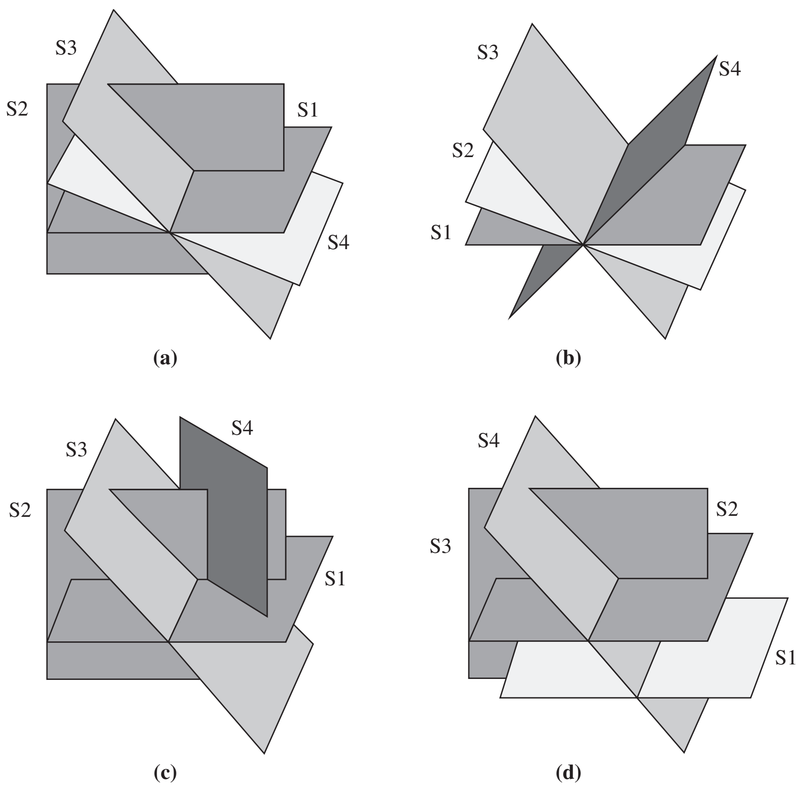

**FIGURE 2.4** Planes representing an overdetermined system of equations. (a) Four plane surfaces intersecting in a point. (b) Four plane surfaces intersecting in a line. (c) Four plane surfaces not intersecting at a single point; points of intersection of (S1, S2, S3) and (S1, S2, S4) are visible. (d) Four plane surfaces representing inconsistent equations.

- elseif **A** is a positive definite, square symmetric, or Hermitian matrix, Cholesky decomposition (described in Section 2.8) is applied. When **A** is sparse, Cholesky decomposition is preceded by a symmetric minimum degree preordering (described in Section 2.14).
- elseif **A** is a square matrix, general LU decomposition (described in Section 2.7) is applied. If **A** is sparse, this is preceded by a nonsymmetric minimum degree preordering (described in Section 2.14).
- elseif **A** is a full nonsquare matrix, QR decomposition (described in Section 2.9) is applied.
- elseif **A** is a sparse nonsquare matrix, it is augmented and then a minimum degree preordering is applied, followed by sparse Gaussian elimination (described in Section 2.14).

The MATLAB \ operator can also be used to solve $\mathbf{AX} = \mathbf{B}$ where $\mathbf{B}$ and the unknown $\mathbf{X}$ are $m \times n$ matrices. This could provide a simple method of finding the inverse of **A**. If we make

**B** the identity matrix **I** then we have

$$\mathbf{AX} = \mathbf{I}$$

and **X** must be the inverse of **A** since $\mathbf{AA}^{-1} = \mathbf{I}$. Thus in MATLAB we could determine the inverse of **A** by using the statement `A\eye(size(A))`. However, MATLAB provides the function `inv(A)` to find the inverse of a matrix. It is important to stress that the inverse of a matrix should only be determined if it is specifically required. If we require the solution of a set of linear equations it is more efficient to use the operators \ or /.

We now examine some cases to show how the \ operator works, beginning with the solution of a system where the system matrix is triangular. The experiment in this case examines the time taken by the operator \ to solve a system when it is full and then when the same system is converted to triangular form by zeroing appropriate elements to produce a triangular matrix. The script used for this experiment is

```
% e3s201.m
disp('   n    full-time  full-time/n^3   tri-time  tri-time/n^2');
A = [ ]; b = [ ];
for n = 2000:500:6000
    A = 100*rand(n); b = [1:n].';
    tic, x = A\b; t1 = toc;
    t1n = 5e9*t1/n^3;
    for i = 1:n
        for j = i+1:n
            A(i,j) = 0;
        end
    end
    tic, x = A\b; t2 = toc;
    t2n = 1e9*t2/n^2;
    fprintf('%6.0f %9.4f %12.4f %12.4f %11.4f\n',n,t1,t1n,t2,t2n)
end
```

The results for a series of randomly generated $n \times n$ matrices are as follows:

```
   n    full-time  full-time/n^3   tri-time  tri-time/n^2
  2000    1.7552       1.0970       0.0101      2.5203
  2500    3.3604       1.0753       0.0151      2.4151
  3000    5.4936       1.0173       0.0209      2.3275
  3500    8.5735       0.9998       0.0282      2.3001
  4000   12.6882       0.9913       0.0358      2.2393
  4500   17.5680       0.9639       0.0453      2.2392
  5000   24.8408       0.9936       0.0718      2.8703
```

Column 1 of this table gives the size of the square matrix, $n$. To demonstrate that the operator \ takes account of the triangular form, columns 2 and 3 contain the time taken and the time taken divided by $n^3$ and multiplied by the scaling factor $5 \times 10^9$ for the full matrix problem. Columns 4 and 5 give the time taken and the time taken divided by $n^2$ and multiplied by the scaling factor $1 \times 10^9$ for the triangular system. These interesting results show that for the full matrix the time taken by \ is approximately proportional to $n^3$ while for the triangular system the time taken by \ is approximately proportional to $n^2$. This is the expected result for simple back substitution. In addition we see a considerable reduction in the time taken to solve the system when the operator \ is used with a triangular system.

We now perform experiments to examine the effects of using the operator \ with positive definite symmetric systems. This is a more complex problem than those previously discussed and the script that follows implements this test. It is based on comparing the application of the \ operator to a positive definite system and a nonpositive definite system of equations. We can create a positive definite matrix by letting `A = M*M'`. Where `M` is any matrix, but in this case the matrix will be of random numbers. `A` will then be a positive definite matrix. To generate a nonpositive definite system we add a random matrix to the positive definite matrix and we compare the time required to solve the two forms of matrix. The script takes the form

```
% e3s202.m
disp(' n      time-pos   time-pos/n^3  time-npos   time-b/n^3');
for n = 100:100:1000
    A = [ ]; M = 100*randn(n,n);
    A = M*M'; b = [1:n].';
    tic, x = A\b; t1 = toc*1000;
    t1d = t1/n^3;
    A = A+rand(size(A));
    tic, x = A\b; t2 = toc*1000;
    t2d = t2/n^3;
    fprintf('%4.0f %10.4f %14.4e %11.4f %13.4e\n',n,t1,t1d,t2,t2d)
end
```

The result of running this script is

```
   n      time-pos   time-pos/n^3  time-npos   time-b/n^3
 100      0.9881    9.8811e-007     1.2085   1.2085e-006
 200      3.5946    4.4932e-007     3.0903   3.8629e-007
 300      5.0646    1.8758e-007     9.7878   3.6251e-007
 400     10.3890    1.6233e-007    20.4892   3.2014e-007
 500     18.0235    1.4419e-007    36.5653   2.9252e-007
 600     18.1892    8.4209e-008    37.7766   1.7489e-007
 700     26.5483    7.7400e-008    58.3854   1.7022e-007
 800     39.6402    7.7422e-008    79.4285   1.5513e-007
 900     58.5519    8.0318e-008   110.5409   1.5163e-007
1000     67.9078    6.7908e-008   130.2029   1.3020e-007
```

Column 1 of this table gives $n$, the size of the matrix. Column 2 gives the time multiplied by 1000 for the positive definite matrix and column 4 gives the time multiplied by 1000 for the nonpositive definite matrix. These results show that the time taken to determine the solution for the system is somewhat faster for the positive definite system. This is because the operator \ checks to see if the matrix is positive definite and if so uses the more efficient Cholesky decomposition. Columns 3 and 5 give the times divide by the size of the matrix cubed to illustrate that the processing time is approximately proportional to $n^3$.

The next test we perform examines how the operator \ succeeds with the very badly conditioned Hilbert matrix. The test gives the time taken to solve the system and the accuracy of the solution given by the Euclidean norm of the residuals, that is, norm($\mathbf{Ax} - \mathbf{b}$). For the definition of the norm see Appendix A, Section A.10. In addition, the test compares these results for the \ operator with the results obtained using the inverse, that is, $\mathbf{x} = \mathbf{A}^{-1}\mathbf{b}$. The script for this test is

```
% e3s203.m
disp(' n  time-slash acc-slash   time-inv   acc-inv   condition');
for n = 4:2:20
    A = hilb(n); b = [1:n].';
    tic, x = A\b; t1 = toc; t1 = t1*10000;
    nm1 = norm(b-A*x);
    tic, x = inv(A)*b; t2 = toc; t2 = t2*10000;
    nm2 = norm(b-A*x);
    c = cond(A);
    fprintf('%2.0f %10.4f %10.2e %8.4f %11.2e %11.2e \n',n,t1,nm1,t2,nm2,c)
end
```

This produces the following table of results:

```
 n  time-slash acc-slash   time-inv   acc-inv   condition
 4     1.6427  1.39e-013   0.8549   9.85e-014   1.55e+004
 6     0.9415  5.22e-012   0.7710   2.02e-009   1.50e+007
 8     1.1454  5.35e-010   0.8465   3.19e-006   1.53e+010
10     1.2627  3.53e-008   1.5477   2.47e-004   1.60e+013
12     1.9332  1.40e-006   1.5589   9.39e-001   1.74e+016
14     2.1958  3.36e-005   1.5924   3.39e+002   5.13e+017
16     2.3187  5.76e-006   1.6650   1.02e+002   4.52e+017
18     2.4836  5.25e-005   2.0589   2.31e+002   1.57e+018
20     2.4417  1.11e-005   2.0869   3.72e+002   2.57e+018
```

This output has been edited to remove warnings about the ill-conditioning of the matrix for $n >= 10$. Column 1 gives the size of the matrix. Columns 2 and 3 give the time taken multiplied by 10,000 and accuracy when using the \ operator. Columns 4 and 5 give the same information when using the `inv` function. Column 6 gives the condition number of the system matrix. When the condition number is large, the matrix is nearly singular and the equations are ill-conditioned. This is fully described in Section 2.4.

The results in the preceding table demonstrate convincingly the superiority of the \ operator over the `inv` function for solving a system of linear equations. It is considerably more accurate than using matrix inversion. However, it should be noted that the accuracy falls off as the matrix becomes increasingly ill-conditioned.

The MATLAB operator \ can also be used to solve under- and overdetermined systems. In this case the \ operator uses a least squares approximation, discussed in detail in Section 2.12.

## 2.4 Accuracy of Solutions and Ill-Conditioning

We now consider factors that affect the accuracy of the solution of $\mathbf{Ax} = \mathbf{b}$ and how any inaccuracies can be detected. A further discussion on the accuracy of the solution of this equation system is given in Appendix B, Section B.3. We begin with the following examples.

**Example 2.1**

Consider the following MATLAB statements:

```
>> A = [3.021 2.714 6.913;1.031 -4.273 1.121;5.084 -5.832 9.155]

A =
    3.0210    2.7140    6.9130
    1.0310   -4.2730    1.1210
    5.0840   -5.8320    9.1550

>> b = [12.648 -2.121 8.407].'

b =
   12.6480
   -2.1210
    8.4070

>> A\b

ans =
    1.0000
    1.0000
    1.0000
```

This result is correct and easily verified by substitution into the original equations.

**Example 2.2**

Consider Example 2.1 with `A(2,2)` changed from −4.2730 to −4.2750:

```
>> A(2,2) = -4.2750

A =
    3.0210    2.7140    6.9130
    1.0310   -4.2750    1.1210
    5.0840   -5.8320    9.1550

>> A\b

ans =
   -1.7403
    0.6851
    2.3212
```

Here we have a solution that is very different from that of Example 2.1, even though the only change in the equation system is less than 0.1% in coefficient `A(2,2)`.

The two examples just shown have dramatically different solutions because the coefficient matrix **A** is ill-conditioned. Ill-conditioning can be interpreted graphically by representing each of the equation systems by three plane surfaces, in the manner shown earlier in Figure 2.2. In an ill-conditioned system at least two of the surfaces will be almost parallel so that the point of intersection of the surfaces will be very sensitive to small changes in slope, caused by small changes in coefficient values.

A system of equations is said to be ill-conditioned if a relatively small change in the elements of the coefficient matrix **A** causes a relatively large change in the solution. Conversely a system of equations is said to be well-conditioned if a relatively small change in the elements of the coefficient matrix **A** causes a relatively small change in the solution. Clearly we require a measure of the condition of a system of equations. We know that a system of equations without a solution—the very worst condition possible—has a coefficient matrix with a determinant of zero. It is therefore tempting to think that the size of the determinant of **A** can be used as a measure of condition. However, if $\mathbf{Ax} = \mathbf{b}$ and **A** is an $n \times n$ diagonal matrix with each element on the leading diagonal equal to $s$, then **A** is perfectly conditioned, regardless of the value of $s$. But the determinant of **A** in this case is $s^n$. Thus, the size of the determinant of **A** is not a suitable measure of condition because in this example it changes with $s$ even though the condition of the system is constant.

Two of the functions MATLAB provides to estimate the condition of a matrix are `cond` and `rcond`. The function `cond` is a sophisticated function and is based on singular value decomposition, discussed in Section 2.10. For a perfect condition `cond` is unity but gives a large value for a matrix that is ill-conditioned. The function `rcond` is less reliable but usually faster. This function gives a value between zero and one. The smaller the value, the worse the conditioning. The reciprocal of `rcond` is usually of the same order of magnitude as `cond`. We now illustrate these points with two examples.

**Example 2.3**

Illustration of a perfectly conditioned system:

```
>> A = diag([20 20 20])
A =
    20     0     0
     0    20     0
     0     0    20

>> [det(A) rcond(A) cond(A)]

ans =
        8000            1            1
```

**Example 2.4**

Illustration of a badly conditioned system:

```
>> A = [1 2 3;4 5 6;7 8 9.000001];
>> format short e
>> [det(A) rcond(A) 1/rcond(A) cond(A)]

ans =
  -3.0000e-006   6.9444e-009   1.4400e+008   1.0109e+008
```

Note that the reciprocal of the `rcond` value is close to the value of `cond`. Using the MATLAB functions `cond` and `rcond` we now investigate the condition number of the Hilbert matrix (defined in Problem 2.1), using the script shown next:

```
% e3s204.m Hilbert matrix test.
disp('    n           cond              rcond      log10(cond)')
for n = 4:2:20
    A = hilb(n);
    fprintf('%5.0f %16.4e',n,cond(A));
    fprintf('%16.4e %10.2f\n',rcond(A),log10(cond(A)));
end
```

Running this script gives

```
     n          cond            rcond      log10(cond)
     4      1.5514e+004     3.5242e-005       4.19
     6      1.4951e+007     3.4399e-008       7.17
     8      1.5258e+010     2.9522e-011      10.18
    10      1.6025e+013     2.8286e-014      13.20
    12      1.7352e+016     2.6328e-017      16.24
    14      5.1317e+017     1.7082e-019      17.71
    16      4.5175e+017     4.6391e-019      17.65
    18      1.5745e+018     5.8371e-020      18.20
    20      2.5710e+018     1.9953e-019      18.41
```

This shows that the Hilbert matrix is ill-conditioned even for relatively small values of $n$, the size of the matrix. The last column of the preceding output gives the value of $\log_{10}$ of the condition number of the appropriate Hilbert matrix. This gives a rule of thumb estimate of the number of significant figures lost in solving an equation system with this matrix or inverting the matrix.

The Hilbert matrix of order $n$ was generated in the preceding script using the MATLAB function `hilb(n)`. Other important matrices with interesting structures and properties, such as the Hadamard matrix and the Wilkinson matrix, can be obtained using, in these cases, the MATLAB functions `hadamard(n)` and `wilkinson(n)`where `n` is the required size of the matrix. In addition, many other interesting matrices can be accessed using the `gallery` function. In almost every case we can choose the size of the matrix and in many cases we can also choose other parameters within the matrix. Example calls are

```
gallery('hanowa',6,4)
gallery('cauchy',6)
gallery('forsythe',6,8)
```

The next section begins the detailed examination of one of the algorithms used by the \ operator.

## 2.5 Elementary Row Operations

We now examine the operations that can usefully be carried out on each equation of a system of equations. Such a system will have the form

$$\begin{gathered}
a_{11}x_1 + a_{12}x_2 + \cdots + a_{1n}x_n = b_1 \\
a_{21}x_2 + a_{22}x_2 + \cdots + a_{2n}x_n = b_2 \\
\ldots\ldots\ldots\ldots\ldots \\
a_{n1}x_n + a_{n2}x_2 + \cdots + a_{nn}x_n = b_n
\end{gathered}$$

or in matrix notation

$$\mathbf{Ax} = \mathbf{b}$$

where

$$\mathbf{A} = \begin{bmatrix} a_{11} & a_{12}\dots & a_{1n} \\ a_{21} & a_{22}\dots & a_{2n} \\ \vdots & \vdots & \vdots \\ a_{n1} & a_{n2}\dots & a_{nn} \end{bmatrix} \quad \mathbf{b} = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{bmatrix} \quad \mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix}$$

**A** is called the coefficient matrix. Any operation performed on an equation must be applied to both its left and right sides. With this in mind it is helpful to combine the coefficient matrix **A** with the right side vector **b**:

$$\mathbf{A} = \begin{bmatrix} a_{11} & a_{12}\dots & a_{1n} & b_1 \\ a_{21} & a_{22}\dots & a_{2n} & b_2 \\ \vdots & \vdots & \vdots & \vdots \\ a_{n1} & a_{n2}\dots & a_{nn} & b_n \end{bmatrix}$$

This new matrix is called the augmented matrix and we will write it as [**A b**]. We have chosen to adopt this notation because it is consistent with MATLAB notation for combining **A** and **b**. Note that if **A** is an $n \times n$ matrix, then the augmented matrix is an $n \times (n+1)$ matrix. Each row of the augmented matrix holds all the coefficients of an equation and any operation must be applied to every element in the row. The three elementary row operations described in the following can be applied to individual equations in a system without altering the solution of the equation system. They are

1. Interchange the position of any two rows (i.e., equations).
2. Multiply a row (i.e., equation) by a nonzero scalar.
3. Replace a row by the sum of the row and a scalar multiple of another row.

These elementary row operations can be used to solve some important problems in linear algebra and we now discuss an application of them.

## 2.6 Solution of **Ax = b** by Gaussian Elimination

Gaussian elimination is an efficient way to solve equation systems, particularly those with a nonsymmetric coefficient matrix having a relatively small number of zero elements. The method depends entirely on using the three elementary row operations described in Section 2.5. Essentially the procedure is to form the augmented matrix for the system and then reduce the coefficient matrix part to an upper triangular form. To illustrate the

**Table 2.1** Gaussian Elimination to Transform an Augmented Matrix to Upper Triangular Form

| | | | | | |
|---|---|---|---|---|---|
| A1 | $\boxed{3}$ | 6 | 9 | 3 | **Stage 1**: Initial matrix |
| A2 | 2 | $(4+p)$ | 2 | 4 | |
| A3 | −3 | −4 | −11 | −5 | |
| A1 | 3 | 6 | 9 | 3 | **Stage 2**: Reduce col 1 of rows 2 and 3 to zero |
| B2 = A2 − 2(A1)/3 | 0 | $p$ | −4 | 2 | |
| B3 = A3 + 3(A1)/3 | 0 | 2 | −2 | −2 | |
| A1 | 3 | 6 | 9 | 3 | **Stage 3**: Interchange rows 2 and 3 |
| B3 | 0 | $\boxed{2}$ | −2 | −2 | |
| B2 | 0 | $p$ | −4 | 2 | |
| A1 | 3 | 6 | 9 | 3 | **Stage 4**: Reduce col 2 of row 3 to zero |
| B3 | 0 | 2 | −2 | −2 | |
| C3 = B2 − $p$(B3)/2 | 0 | 0 | $\boxed{(-4+p)}$ | $(2+p)$ | |

systematic use of the elementary row operations we consider the application of Gaussian elimination to solve the following equation system:

$$\begin{bmatrix} 3 & 6 & 9 \\ 2 & (4+p) & 2 \\ -3 & -4 & -11 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 3 \\ 4 \\ -5 \end{bmatrix} \tag{2.10}$$

where the value of $p$ is known. Table 2.1 shows the sequence of operations, beginning at Stage 1 with the augmented matrix. In Stage 1 the element in the first column of the first row (enclosed in a box in the table) is designated the pivot. We wish to make the elements of column 1 in rows 2 and 3 zero. To achieve this, we divide row 1 by the pivot and then add or subtract a suitable multiple of the modified row 1 to or from rows 2 and 3. The result of this is shown in Stage 2 of the table. We then select the next pivot. This is the element in the second column of the new second row, which in Stage 2 is equal to $p$. If $p$ is large, this does not present a problem, but if $p$ is small, then numerical problems may arise because we will be dividing all the elements of the new row 2 by this small quantity $p$. If $p$ is zero, then we have an impossible situation because we cannot divide by zero.

This difficulty is not related to ill-conditioning; indeed this particular equation system is quite well conditioned when $p$ is zero. To circumvent these problems the usual procedure is to interchange the row in question with the row containing the element of largest modulus in the column *below the pivot*. In this way we provide a new and larger pivot. This procedure is called partial pivoting. If we assume in this case that $p < 2$, then we interchange rows 2 and 3 as shown in Stage 3 of the table to replace $p$ by 2 as the pivot. From row 3 we now subtract row 2 divided by the pivot and multiply by a coefficient in order to make the element of column 2, row 3, zero. Thus in Stage 4 of the table the original coefficient matrix has been reduced to an upper triangular matrix. If, for example, $p = 0$, we obtain

$$3x_1 + 6x_2 + 9x_3 = 3 \tag{2.11}$$

$$2x_2 - 2x_3 = -2 \tag{2.12}$$

$$-4x_3 = 2 \tag{2.13}$$

We can now obtain the values of the unknowns $x_1$, $x_2$, and $x_3$ by a process called back substitution. We solve the equations in reverse order. Thus from (2.13), $x_3 = -0.5$. From (2.12), knowing $x_3$, we have $x_2 = -1.5$. Finally from (2.11), knowing $x_2$ and $x_3$, we have $x_1 = 5.5$.

It can be shown that the determinant of a matrix can be evaluated from the product of the elements on the main diagonal provided at Stage 3 in Table 2.1. This product must be multiplied by $(-1)^m$ where $m$ is the number of row interchanges used. For example, in the preceding problem, with $p = 0$, one row interchange is used so that $m = 1$ and the determinant of the coefficient matrix is given by $3 \times 2 \times (-4) \times (-1)^1 = 24$.

A method for solving a linear equation system that is closely related to Gaussian elimination is Gauss–Jordan elimination. The method uses the same elementary row operations but differs from Gaussian elimination because elements both below and above the leading diagonal are reduced to zero. This means that back substitution is avoided. For example, solving system (2.10) with $p = 0$ leads to the following augmented matrix:

$$\begin{bmatrix} 3 & 0 & 0 & 16.5 \\ 0 & 2 & 0 & -3.0 \\ 0 & 0 & -4 & 2.0 \end{bmatrix}$$

Thus $x_1 = 16.5/3 = 5.5$, $x_2 = -3/2 = -1.5$, and $x_3 = 2/-4 = -0.5$.

Gaussian elimination requires order $n^3/3$ multiplications followed by back substitution requiring order $n^2$ multiplications. Gauss–Jordan elimination requires order $n^3/2$ multiplications. Thus for large systems of equations (say $n > 10$), Gauss–Jordan elimination requires approximately 50% more operations than Gaussian elimination.

## 2.7 LU Decomposition

LU decomposition (or factorization) is a similar process to Gaussian elimination and is equivalent in terms of elementary row operations. The matrix **A** can be decomposed so that

$$\mathbf{A} = \mathbf{LU} \tag{2.14}$$

where **L** is a lower triangular matrix with a leading diagonal of ones and **U** is an upper triangular matrix. Matrix **A** may be real or complex. Compared with Gaussian elimination, LU decomposition has a particular advantage when the equation system we wish to solve, $\mathbf{Ax} = \mathbf{b}$, has more than one right side or when the right sides are not known in advance. This is because the factors **L** and **U** are obtained explicitly and they can be used for any right sides as they arise without recalculating **L** and **U**. Gaussian elimination does not determine **L** explicitly but rather forms $\mathbf{L}^{-1}\,\mathbf{b}$ so that all right sides must be known when the equation is solved.

The major steps required to solve an equation system by LU decomposition are as follows. Since $\mathbf{A} = \mathbf{LU}$, then $\mathbf{Ax} = \mathbf{B}$ becomes

$$\mathbf{LUx} = \mathbf{b}$$

where **b** is not restricted to a single column. Letting $\mathbf{y} = \mathbf{Ux}$ leads to

$$\mathbf{Ly} = \mathbf{b}$$

Because **L** is a lower triangular matrix this equation is solved efficiently by forward substitution. To find **x** we now solve

$$\mathbf{Ux} = \mathbf{y}$$

Because **U** is an upper triangular matrix, this equation can also be solved efficiently by back substitution.

We now illustrate the LU decomposition process by solving (2.10) with $p = 1$. We are not concerned with **b** and we do not form an augmented matrix. We proceed exactly as with Gaussian elimination (see Table 2.1), except that we keep a record of the elementary row operations performed at the $i$th stage in $\mathbf{T}^{(i)}$ and place the results of these operations in a matrix $\mathbf{U}^{(i)}$ rather than overwriting **A**.

We begin with the matrix

$$\mathbf{A} = \begin{bmatrix} 3 & 6 & 9 \\ 2 & 5 & 2 \\ -3 & -4 & -11 \end{bmatrix}$$

Following the same operations as used in Table 2.1, we will create a matrix $\mathbf{U}^{(1)}$ with zeros below the leading diagonal in the first column using the following elementary row operations:

$$\text{row 2 of } \mathbf{U}^{(1)} = \text{ row 2 of } \mathbf{A} - 2(\text{row 1 of} \mathbf{A})/3 \tag{2.15}$$

and

$$\text{row 3 of } \mathbf{U}^{(1)} = \text{ row 3 of } \mathbf{A} + 3(\text{row 1 of} \mathbf{A})/3 \tag{2.16}$$

Now **A** can be expressed as the product $\mathbf{T}^{(1)}\ \mathbf{U}^{(1)}$ as follows:

$$\begin{bmatrix} 3 & 6 & 9 \\ 2 & 5 & 2 \\ -3 & -4 & -11 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 2/3 & 1 & 0 \\ -1 & 0 & 1 \end{bmatrix} \begin{bmatrix} 3 & 6 & 9 \\ 0 & 1 & -4 \\ 0 & 2 & -2 \end{bmatrix}$$

Note that row 1 of **A** and row 1 of $\mathbf{U}^{(1)}$ are identical. Thus row 1 of $\mathbf{T}^{(1)}$ has a unit entry in column 1 and zero elsewhere. The remaining rows of $\mathbf{T}^{(1)}$ are determined from (2.15) and (2.16). For example, row 2 of $\mathbf{T}^{(1)}$ is derived by rearranging (2.15); thus

$$\text{row 2 of } \mathbf{A} = \text{ row 2 of } \mathbf{U}^{(1)} + 2(\text{row 1 of} \mathbf{A})/3 \tag{2.17}$$

or

$$\text{row 2 of } \mathbf{A} = 2(\text{row 1 of } \mathbf{U}^{(1)})/3 + \text{row 2 of } \mathbf{U}^{(1)} \tag{2.18}$$

since row 1 of $\mathbf{U}^{(1)}$ is identical to row 1 of **A**. Hence row 2 of $\mathbf{T}^{(1)}$ is [2/3 1 0].

We now move to the next stage of the decomposition process. In order to bring the largest element of column 2 in $\mathbf{U}^{(1)}$ onto the leading diagonal we must interchange rows 2 and 3. Thus $\mathbf{U}^{(1)}$ becomes the product $\mathbf{T}^{(2)}\ \mathbf{U}^{(2)}$ as follows:

$$\begin{bmatrix} 3 & 6 & 9 \\ 0 & 1 & -4 \\ 0 & 2 & -2 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} 3 & 6 & 9 \\ 0 & 2 & -2 \\ 0 & 1 & -4 \end{bmatrix}$$

Finally, to complete the process of obtaining an upper triangular matrix we make

$$\text{row 3 of } \mathbf{U} = \text{ row 3 of } \mathbf{U}^{(2)} - (\text{row 2 of } \mathbf{U}^{(2)})/2$$

Hence $\mathbf{U}^{(2)}$ becomes the product $\mathbf{T}^{(3)}\ \mathbf{U}$ as follows:

$$\begin{bmatrix} 3 & 6 & 9 \\ 0 & 2 & -2 \\ 0 & 1 & -4 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 1/2 & 1 \end{bmatrix} \begin{bmatrix} 3 & 6 & 9 \\ 0 & 2 & -2 \\ 0 & 0 & -3 \end{bmatrix}$$

Thus $\mathbf{A} = \mathbf{T}^{(1)}\ \mathbf{T}^{(2)}\ \mathbf{T}^{(3)}\ \mathbf{U}$, implying that $\mathbf{L} = \mathbf{T}^{(1)}\ \mathbf{T}^{(2)}\ \mathbf{T}^{(3)}$ as follows:

$$\begin{bmatrix} 1 & 0 & 0 \\ 2/3 & 1 & 0 \\ -1 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 1/2 & 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 2/3 & 1/2 & 1 \\ -1 & 1 & 0 \end{bmatrix}$$

Note that owing to the row interchanges **L** is not strictly a lower triangular matrix but it can be made so by interchanging rows.

MATLAB implements LU factorization by using the function `lu` and may produce a matrix that is not strictly a lower triangular matrix. However, a permutation matrix **P** may be produced, if required, such that **LU** = **PA** with **L** lower triangular.

We now show how the MATLAB function `lu` deals with the preceding example:

```
>> A = [3 6 9;2 5 2;-3 -4 -11]

A =
     3     6     9
     2     5     2
    -3    -4   -11
```

To obtain the **L** and **U** matrices, we must use that MATLAB facility of assigning two parameters simultaneously as follows:

```
>> [L1 U] = lu(A)

L1 =
    1.0000         0         0
    0.6667    0.5000    1.0000
   -1.0000    1.0000         0

U =
     3     6     9
     0     2    -2
     0     0    -3
```

Note that the `L1` matrix is not in lower triangular form, although its true form can easily be deduced by interchanging rows 2 and 3 to form a triangle. To obtain a true lower triangular matrix we must assign three parameters as follows:

```
>> [L U P] = lu(A)

L =
    1.0000         0         0
   -1.0000    1.0000         0
    0.6667    0.5000    1.0000

U =
     3     6     9
     0     2    -2
     0     0    -3

P =
     1     0     0
     0     0     1
     0     1     0
```

In the preceding output `P` is the permutation matrix such that `L*U = P*A` or `P'*L*U = A`. Thus `P'*L` is equal to `L1`.

The MATLAB operator \ determines the solution of $\mathbf{Ax} = \mathbf{b}$ using LU factorization. As an example of an equation system with multiple right sides we solve $\mathbf{AX} = \mathbf{B}$ where

$$\mathbf{A} = \begin{bmatrix} 3 & 4 & -5 \\ 6 & -3 & 4 \\ 8 & 9 & -2 \end{bmatrix} \quad \text{and} \quad \mathbf{B} = \begin{bmatrix} 1 & 3 \\ 9 & 5 \\ 9 & 4 \end{bmatrix}$$

Performing LU decomposition such that $\mathbf{LU} = \mathbf{A}$ gives

$$\mathbf{L} = \begin{bmatrix} 0.375 & -0.064 & 1 \\ 0.750 & 1 & 0 \\ 1 & 0 & 0 \end{bmatrix} \quad \text{and} \quad \mathbf{U} = \begin{bmatrix} 8 & 9 & -2 \\ 0 & -9.75 & 5.5 \\ 0 & 0 & -3.897 \end{bmatrix}$$

Thus $\mathbf{LY} = \mathbf{B}$ is given by

$$\begin{bmatrix} 0.375 & -0.064 & 1 \\ 0.750 & 1 & 0 \\ 1 & 0 & 0 \end{bmatrix} \begin{bmatrix} y_{11} & y_{12} \\ y_{21} & y_{22} \\ y_{31} & y_{32} \end{bmatrix} = \begin{bmatrix} 1 & 3 \\ 9 & 5 \\ 9 & 4 \end{bmatrix}$$

We note that implicitly we have two systems of equations, which when separated can be written

$$\mathbf{L} \begin{bmatrix} y_{11} \\ y_{21} \\ y_{31} \end{bmatrix} = \begin{bmatrix} 1 \\ 9 \\ 9 \end{bmatrix} \quad \text{and} \quad \mathbf{L} \begin{bmatrix} y_{12} \\ y_{22} \\ y_{32} \end{bmatrix} = \begin{bmatrix} 3 \\ 5 \\ 4 \end{bmatrix}$$

In this example $\mathbf{L}$ is not strictly a lower triangular matrix owing to the reordering of the rows. However, the solution of this equation is still found by forward substitution. For example, $1y_{11} = b_{31} = 9$, so that $y_{11} = 9$. Then $0.75y_{11} + 1y_{21} = b_{21} = 9$. Hence $y_{21} = 2.25$, and so on. The complete $\mathbf{Y}$ matrix is

$$\mathbf{Y} = \begin{bmatrix} 9.000 & 4.000 \\ 2.250 & 2.000 \\ -2.231 & 1.628 \end{bmatrix}$$

Finally, solving $\mathbf{UX} = \mathbf{Y}$ by back substitution gives

$$\mathbf{X} = \begin{bmatrix} 1.165 & 0.891 \\ 0.092 & -0.441 \\ 0.572 & -0.418 \end{bmatrix}$$

The MATLAB function `det` determines the determinant of a matrix using LU factorization as follows. Since $\mathbf{A} = \mathbf{LU}$ then $|\mathbf{A}| = |\mathbf{L}|\ |\mathbf{U}|$. The elements of the leading diagonal of $\mathbf{L}$ are all ones so that $|\mathbf{L}| = 1$. Since **U** is upper triangular, its determinant is the product of the elements of its leading diagonal. Thus, taking account of row interchanges, the appropriately signed product of the diagonal elements of **U** gives the determinant.

## 2.8 Cholesky Decomposition

Cholesky decomposition or factorization is a form of triangular decomposition that can only be applied to positive definite symmetric or positive definite Hermitian matrices. A symmetric or Hermitian matrix **A** is said to be positive definite if $\mathbf{x}^\top \mathbf{A}\mathbf{x} > 0$ for any nonzero **x**. A more useful definition of a positive definite matrix is one that has all eigenvalues greater than zero. The eigenvalue problem is discussed in Section 2.15. If **A** is symmetric or Hermitian, we can write

$$\mathbf{A} = \mathbf{P}^\top \mathbf{P} \ \ (\text{or } \mathbf{A} = \mathbf{P}^H \mathbf{P} \text{ when } \mathbf{A} \text{ is Hermitian}) \tag{2.19}$$

where **P** is an upper triangular matrix. The algorithm computes **P** row by row by equating coefficients of each side of (2.19). Thus $p_{11}, p_{12}, p_{13}, \ldots, p_{22}, p_{23}, \ldots$ are determined in sequence, ending with $p_{nn}$. Coefficients on the leading diagonal of **P** are computed from expressions that involve determining a square root. For example,

$$p_{22} = \sqrt{a_{22} - p_{12}^2}$$

A property of positive definite matrices is that the term under the square root is always positive and so the square root will be real. Furthermore, row interchanges are not required because the dominant coefficients will always be on the main diagonal. The whole process requires only about half as many multiplications as LU decomposition. Cholesky factorization is implemented for positive definite symmetric matrices in MATLAB by the function `chol`. For example, consider the Cholesky factorization of the following positive definite Hermitian matrix:

```
>> A = [2 -i 0;i 2 0;0 0 3]

A =
   2.0000                 0 - 1.0000i        0
        0 + 1.0000i  2.0000                  0
        0                 0             3.0000

>> P = chol(A)

P =
   1.4142                 0 - 0.7071i        0
        0            1.2247                  0
        0                 0             1.7321
```

When the operator \ detects a symmetric positive definite or Hermitian positive definite system matrix, it solves $\mathbf{Ax} = \mathbf{b}$ using the following sequence of operations. $\mathbf{A}$ is factorized into $\mathbf{P}^\top\mathbf{P}$, and $\mathbf{y}$ is set to $\mathbf{Px}$; then $\mathbf{P}^\top\mathbf{y} = \mathbf{b}$. The algorithm solves for $\mathbf{y}$ by forward substitution since $\mathbf{P}^\top$ is a lower triangular matrix. Then $\mathbf{x}$ can be determined from $\mathbf{y}$ by backward substitution since $\mathbf{P}$ is an upper triangular matrix. We can illustrate the steps in this process by the following example:

$$\mathbf{A} = \begin{bmatrix} 2 & 3 & 4 \\ 3 & 6 & 7 \\ 4 & 7 & 10 \end{bmatrix} \text{ and } \mathbf{b} = \begin{bmatrix} 2 \\ 4 \\ 8 \end{bmatrix}$$

Then by Cholesky factorization

$$\mathbf{P} = \begin{bmatrix} 1.414 & 2.121 & 2.828 \\ 0 & 1.225 & 0.817 \\ 0 & 0 & 1.155 \end{bmatrix}$$

Now since $\mathbf{P}^\top\mathbf{y} = \mathbf{b}$, solving for $\mathbf{y}$ by forward substitution gives

$$\mathbf{y} = \begin{bmatrix} 1.414 \\ 0.817 \\ 2.887 \end{bmatrix}$$

Finally, solving $\mathbf{Px} = \mathbf{y}$ by back substitution gives

$$\mathbf{x} = \begin{bmatrix} -2.5 \\ -1.0 \\ 2.5 \end{bmatrix}$$

We now compare the performance of the operator \ with the function `chol`. Clearly their performance should be similar in the case of a positive definite matrix. To generate a symmetric positive define matrix in the following script, we multiply a matrix by its transpose:

```
% e3s205.m
disp('  n       time-backslash  time-chol');
for n = 300:100:1300
    A = [ ]; M = 100*randn(n,n);
    A = M*M'; b = [1:n].';
    tic, x = A\b; t1 = toc;
    tic, R = chol(A);
    v = R.'\b; x = R\b;
    t2 = toc;
    fprintf('%4.0f %14.4f %13.4f \n',n,t1,t2)
end
```

Running this script gives

```
    n       time-backslash  time-chol
   300          0.0053        0.0073
   400          0.0105        0.0115
   500          0.0182        0.0216
   600          0.0176        0.0197
   700          0.0263        0.0281
   800          0.0368        0.0385
   900          0.0510        0.0519
  1000          0.0666        0.0668
  1100          0.0862        0.0869
  1200          0.1113        0.1065
  1300          0.1449        0.1438
```

The similarity in performance of the function `chol` and the operator \ is borne out by the preceding table. In this table, column 1 is the size of the matrix and column 2 gives the time taken using the \ operator. Column 3 gives the time taken using Cholesky decomposition to solve the same problem.

Cholesky factorization *can* be applied to a symmetric matrix that is not positive definite but the process does not possess the numerical stability of the positive definite case. Furthermore, one or more rows in **P** may be purely imaginary. For example,

$$\text{If } \mathbf{A} = \begin{bmatrix} 1 & 2 & 3 \\ 2 & -5 & 9 \\ 3 & 9 & 4 \end{bmatrix} \text{ then } \mathbf{P} = \begin{bmatrix} 1 & 2 & 3 \\ 0 & 3\imath & -\imath \\ 0 & 0 & 2\imath \end{bmatrix}$$

This is not implemented in MATLAB.

## 2.9 QR Decomposition

We have seen how a square matrix can be decomposed or factorized into the product of a lower and an upper triangular matrix by the use of elementary row operations. An alternative decomposition is into an upper triangular matrix and an orthogonal matrix if **A** is real or a unitary matrix if **A** is complex. This is called QR decomposition. Thus

$$\mathbf{A} = \mathbf{Q}\,\mathbf{R}$$

where **R** is the upper triangular matrix and **Q** is the orthogonal, or the unitary matrix. If **Q** is orthogonal, $\mathbf{Q}^{-1} = \mathbf{Q}^\top$ and if **Q** is unitary, $\mathbf{Q}^{-1} = \mathbf{Q}^{\mathrm{H}}$. The preceding are very useful properties.

There are several procedures that provide QR decomposition; here we present Householder's method. To decompose a real matrix, Householder's method begins by defining a matrix **P**:

$$\mathbf{P} = \mathbf{I} - 2\mathbf{w}\mathbf{w}^\top \tag{2.20}$$

**P** is symmetrical and providing $\mathbf{w}^\top\mathbf{w} = 1$, **P** is also orthogonal. The orthogonality can easily be verified by expanding the product $\mathbf{P}^\top\mathbf{P} = \mathbf{P}\mathbf{P}$ as follows:

$$\begin{aligned}\mathbf{P}\mathbf{P} &= \left(\mathbf{I} - 2\mathbf{w}\mathbf{w}^\top\right)\left(\mathbf{I} - 2\mathbf{w}\mathbf{w}^\top\right)\\ &= \mathbf{I} - 4\mathbf{w}\mathbf{w}^\top + 4\mathbf{w}\mathbf{w}^\top\left(\mathbf{w}\mathbf{w}^\top\right) = \mathbf{I}\end{aligned}$$

To decompose **A** into **QR**, we begin by forming the vector $\mathbf{w}_1$ from the coefficients of the first column of **A** as follows:

$$\mathbf{w}_1^\top = \mu_1\left[(a_{11} - s_1)\;\; a_{21}\;\; a_{31} \ldots a_{n1}\right]$$

where

$$\mu_1 = \frac{1}{\sqrt{2s_1(s_1 - a_{11})}} \text{ and } s_1 = \pm\left(\sum_{j=1}^{n} a_{j1}^2\right)^{1/2}$$

By substituting for $\mu_1$ and $s_1$ in $\mathbf{w}_1$ it can be verified that the necessary orthogonality condition, $\mathbf{w}_1^\top\mathbf{w}_1 = 1$, is satisfied. Substituting $\mathbf{w}_1$ into (2.20) we generate an orthogonal matrix $\mathbf{P}^{(1)}$.

The matrix $\mathbf{A}^{(1)}$ is now created from the product $\mathbf{P}^{(1)}\mathbf{A}$. It can easily be verified that all elements in the first column of $\mathbf{A}^{(1)}$ are zero except for the element on the leading diagonal, which is equal to $s_1$. Thus

$$\mathbf{A}^{(1)} = \mathbf{P}^{(1)}\mathbf{A} = \begin{bmatrix} s_1 & + & \cdots & + \\ 0 & + & \cdots & + \\ \vdots & \vdots & & \vdots \\ 0 & + & \cdots & + \\ 0 & + & \cdots & + \end{bmatrix}$$

In the matrix $\mathbf{A}^{(1)}$, $+$ indicates a nonzero element.

We now begin the second stage of the orthogonalization process by forming $\mathbf{w}_2$ from the coefficients of the second column of $\mathbf{A}^{(1)}$:

$$\mathbf{w}_2^\top = \mu_2\left[0\;\; \left(a_{22}^{(1)} - s_2\right)\;\; a_{32}^{(1)}\;\; a_{42}^{(1)} \cdots a_{n2}^{(1)}\right]$$

where $a_{ij}$ are the coefficients of **A** and

$$\mu_2 = \frac{1}{\sqrt{2s_2(s_2 - a_{22}^{(1)})}} \text{ and } s_2 = \pm\left(\sum_{j=2}^{n} (a_{j2}^{(1)})^2\right)^{1/2}$$

Then the orthogonal matrix $\mathbf{P}^{(2)}$ is generated from

$$\mathbf{P}^{(2)} = \mathbf{I} - 2\mathbf{w}_2\mathbf{w}_2^\top$$

The matrix $\mathbf{A}^{(2)}$ is then created from the product $\mathbf{P}^{(2)}\mathbf{A}^{(1)}$ as follows:

$$\mathbf{A}^{(2)} = \mathbf{P}^{(2)}\mathbf{A}^{(1)} = \mathbf{P}^{(2)}\mathbf{P}^{(1)}\mathbf{A} = \begin{bmatrix} s_1 & + & \cdots & + \\ 0 & s_2 & \cdots & + \\ \vdots & \vdots & & \vdots \\ 0 & 0 & \cdots & + \\ 0 & 0 & \cdots & + \end{bmatrix}$$

Note that $\mathbf{A}^{(2)}$ has zero elements in its first two columns except for the elements on and above the leading diagonal. We can continue this process $n-1$ times until we obtain an upper triangular matrix $\mathbf{R}$. Thus

$$\mathbf{R} = \mathbf{P}^{(n-1)} \ldots \mathbf{P}^{(2)}\mathbf{P}^{(1)}\mathbf{A} \tag{2.21}$$

Note that since $\mathbf{P}^{(i)}$ is orthogonal, the product $\mathbf{P}^{(n-1)} \ldots \mathbf{P}^{(2)}\mathbf{P}^{(1)}$ is also orthogonal.

We wish to determine the orthogonal matrix $\mathbf{Q}$ such that $\mathbf{A} = \mathbf{QR}$. Thus $\mathbf{R} = \mathbf{Q}^{-1}\mathbf{A}$ or $\mathbf{R} = \mathbf{Q}^\top\mathbf{A}$. Hence, from (2.21),

$$\mathbf{Q}^\top = \mathbf{P}^{(n-1)} \ldots \mathbf{P}^{(2)}\mathbf{P}^{(1)}$$

Apart from the signs associated with the columns of $\mathbf{Q}$ and the rows of $\mathbf{R}$, the decomposition is unique. These signs are dependent on whether the positive or negative square root is taken in determining $s_1$, $s_2$, and so on. Complete decomposition of the matrix requires $2n^3/3$ multiplications and $n$ square roots. To illustrate this procedure consider the decomposition of the matrix

$$\mathbf{A} = \begin{bmatrix} 4 & -2 & 7 \\ 6 & 2 & -3 \\ 3 & 4 & 4 \end{bmatrix}$$

Thus

$$s_1 = \surd\left(4^2 + 6^2 + 3^2\right) = 7.8102$$

$$\mu_1 = 1/\surd[2 \times 7.8102 \times (7.8102 - 4)] = 0.1296$$

$$\mathbf{w}_1^\top = 0.1296[(4 - 7.8102) \;\; 6 \;\; 3] = [-0.4939 \;\; 0.7777 \;\; 0.3889]$$

Using (2.20) we generate $\mathbf{P}^{(1)}$ and hence $\mathbf{A}^{(1)}$ as follows:

$$\mathbf{P}^{(1)} = \begin{bmatrix} 0.5121 & 0.7682 & 0.3841 \\ 0.7682 & -0.2097 & -0.6049 \\ 0.3841 & -0.6049 & 0.6976 \end{bmatrix}$$

$$\mathbf{A}^{(1)} = \mathbf{P}^{(1)}\mathbf{A} = \begin{bmatrix} 7.8102 & 2.0486 & 2.8168 \\ 0 & -4.3753 & 3.5873 \\ 0 & 0.8123 & 7.2936 \end{bmatrix}$$

Note that we have reduced the elements of the first column of $\mathbf{A}^{(1)}$ below the leading diagonal to zero. We continue with the second stage:

$$s_2 = \surd\left\{(-4.3753)^2 + 0.8123^2\right\} = 4.4501$$

$$\mu_2 = 1/\surd\{2 \times 4.4501 \times (4.4501 + 4.3753)\} = 0.1128$$

$$\mathbf{w}_2^\top = 0.1128[0 \;\; (-4.3753 - 4.4501) \;\; 0.8123] = [0 \;\; -0.9958 \;\; 0.0917]$$

$$\mathbf{P}^{(2)} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & -0.9832 & 0.1825 \\ 0 & 0.1825 & 0.9832 \end{bmatrix}$$

$$\mathbf{R} = \mathbf{A}^{(2)} = \mathbf{P}^{(2)}\mathbf{A}^{(1)} = \begin{bmatrix} 7.8102 & 2.0486 & 2.8168 \\ 0 & 4.4501 & -2.1956 \\ 0 & 0 & 7.8259 \end{bmatrix}$$

Note that we have now reduced the first two columns of $\mathbf{A}^{(2)}$ below the leading diagonal to zero. This completes the process to determine the upper triangular matrix $\mathbf{R}$. Finally we determine the orthogonal matrix $\mathbf{Q}$ as follows:

$$\mathbf{Q} = \left(\mathbf{P}^{(2)}\mathbf{P}^{(1)}\right)^\top = \begin{bmatrix} 0.5121 & -0.6852 & 0.5179 \\ 0.7682 & 0.0958 & -0.6330 \\ 0.3841 & 0.7220 & 0.5754 \end{bmatrix}$$

It is not necessary for the reader to carry out the preceding calculations since MATLAB provides the function `qr` to carry out this decomposition. For example,

```
>> A = [4 -2 7;6 2 -3;3 4 4]

A =
     4    -2     7
     6     2    -3
     3     4     4
```

```
>> [Q R] = qr(A)

Q =
   -0.5121    0.6852    0.5179
   -0.7682   -0.0958   -0.6330
   -0.3841   -0.7220    0.5754

R =
   -7.8102   -2.0486   -2.8168
         0   -4.4501    2.1956
         0         0    7.8259
```

One advantage of QR decomposition is that it can be applied to nonsquare matrices, decomposing an $m \times n$ matrix into an $m \times m$ orthogonal matrix and an $m \times n$ upper triangular matrix. Note that if $m > n$, the decomposition is not unique.

## 2.10 Singular Value Decomposition

The singular value decomposition (SVD) of an $m \times n$ matrix $\mathbf{A}$ is given by

$$\mathbf{A} = \mathbf{USV}^\top \text{ (or } \mathbf{A} = \mathbf{USV}^\mathrm{H} \text{ if } \mathbf{A} \text{ is complex)}$$

where $\mathbf{U}$ is an orthogonal $m \times m$ matrix and $\mathbf{V}$ is an orthogonal $n \times n$ matrix. If $\mathbf{A}$ is complex then $\mathbf{U}$ and $\mathbf{V}$ are unitary matrices. In all cases $\mathbf{S}$ is a real diagonal $m \times n$ matrix. The elements of the leading diagonal of this matrix are called the singular values of $\mathbf{A}$. Normally they are arranged in decreasing value so that $s_1 > s_2 > \cdots > s_n$. Thus

$$\mathbf{S} = \begin{bmatrix} s_1 & 0 & \ldots & 0 \\ 0 & s_2 & \ldots & 0 \\ \vdots & \vdots & & \vdots \\ 0 & 0 & \ldots & s_n \\ 0 & 0 & \ldots & 0 \\ \vdots & \vdots & & \vdots \\ 0 & 0 & \ldots & 0 \end{bmatrix}$$

The singular values are the nonnegative square roots of the eigenvalues of $\mathbf{A}^\top\mathbf{A}$. Because $\mathbf{A}^\top\mathbf{A}$ is symmetric or Hermitian these eigenvalues are real and nonnegative so that the singular values are also real and nonnegative. Algorithms for computing the SVD of a matrix are given by Golub and Van Loan (1989).

The SVD of a matrix has several important applications. In Section 2.2 we introduced the reduced row echelon form of a matrix and explained how the MATLAB function `rref` gives information from which the rank of a matrix can be deduced. However, rank can be more effectively determined from the SVD of a matrix since its rank is equal to its number of nonzero singular values. Thus for a $5 \times 5$ matrix of rank 3, $s_4$, and $s_5$ are zero. In practice, rather than counting the nonzero singular values, MATLAB determines rank from the SVD by counting the number of singular values greater than some tolerance value. This is a more realistic approach to determining rank than counting any nonzero value, however small.

To illustrate how singular value decomposition helps us to examine the properties of a matrix we will use the MATLAB function `svd` to carry out a singular value decomposition and compare it with the function `rref`. Consider the following example in which a Vandermonde matrix is created using the MATLAB function `vander`. The Vandermonde matrix is known to be ill-conditioned. SVD allows us to examine the nature of this ill-conditioning. In particular, a zero or a very small singular value indicates rank deficiency and this example shows that the singular values are becoming relatively close to this condition. In addition SVD allows us to compute the condition number of the matrix. In fact, the MATLAB function `cond` uses SVD to compute the condition number and this gives the same values as obtained by dividing the largest singular value by the smallest singular value. Additionally, the Euclidean norm of the matrix is supplied by the first singular value. Comparing the SVD with the RREF process in the following script, we see that using the MATLAB functions `rref` and `rank` give the rank of this special Vandermonde matrix as 5 but tell us nothing else. There is no warning that the matrix is badly conditioned.

```
>> c = [1 1.01 1.02 1.03 1.04];
>> V = vander(c)

V =
    1.0000    1.0000    1.0000    1.0000    1.0000
    1.0406    1.0303    1.0201    1.0100    1.0000
    1.0824    1.0612    1.0404    1.0200    1.0000
    1.1255    1.0927    1.0609    1.0300    1.0000
    1.1699    1.1249    1.0816    1.0400    1.0000

>> format long
>> s = svd(V)

s =
   5.210367051037899
   0.101918335876689
   0.000699698839445
   0.000002352380295
   0.000000003294983
```

```
>> norm(V)

ans =
   5.210367051037899

>> cond(V)

ans =
    1.581303246763933e+009

>> s(1)/s(5)

ans =
    1.581303246763933e+009

>> rank(V)

ans =
     5

>> rref(V)

ans =
     1     0     0     0     0
     0     1     0     0     0
     0     0     1     0     0
     0     0     0     1     0
     0     0     0     0     1
```

The following example is very similar to the preceding one but the Vandermonde matrix has now been generated to be rank deficient. The smallest singular value, although not zero, is zero to machine precision and `rank` returns the value of 4.

```
>> c = [1 1.01 1.02 1.03 1.03];
>> V = vander(c)

V =
    1.0000    1.0000    1.0000    1.0000    1.0000
    1.0406    1.0303    1.0201    1.0100    1.0000
    1.0824    1.0612    1.0404    1.0200    1.0000
    1.1255    1.0927    1.0609    1.0300    1.0000
    1.1255    1.0927    1.0609    1.0300    1.0000
```

```
>> format long e
>> s = svd(V)

s =
    5.187797954424026e+000
    8.336322098941414e-002
    3.997349250042135e-004
    8.462129966456217e-007
                         0

>> format short
>> rank(V)

ans =
     4

>> rref(V)

ans =
    1.0000         0         0         0   -0.9424
         0    1.0000         0         0    3.8262
         0         0    1.0000         0   -5.8251
         0         0         0    1.0000    3.9414
         0         0         0         0         0

>> cond(V)

ans =
   Inf
```

The `rank` function does allow the user to vary the tolerance. However, tolerance should be used with care since the `rank` function counts the number of singular values greater than tolerance and this gives the rank of the matrix. If tolerance is very small (i.e., smaller than the machine precision), the rank may be miscounted.

## 2.11 The Pseudo-Inverse

Here we discuss the pseudo-inverse and in Section 2.12 apply it to solve over- and under-determined systems.

If **A** is an $m \times n$ rectangular matrix, then the system

$$\mathbf{Ax} = \mathbf{b} \tag{2.22}$$

cannot be solved by inverting **A**, since **A** is not a square matrix. Assuming an equation system with more equations than variables (i.e., $m > n$), then by premultiplying (2.22) by $\mathbf{A}^\top$ we can convert the system matrix to a square matrix as follows:

$$\mathbf{A}^\top\mathbf{A}\mathbf{x} = \mathbf{A}^\top\mathbf{b}$$

The product $\mathbf{A}^\top\mathbf{A}$ is square and, provided it is nonsingular, it can be inverted to give the solution to (2.22):

$$\mathbf{x} = \left(\mathbf{A}^\top\mathbf{A}\right)^{-1}\mathbf{A}^\top\mathbf{b} \tag{2.23}$$

Let

$$\mathbf{A}^+ = \left(\mathbf{A}^\top\mathbf{A}\right)^{-1}\mathbf{A}^\top \tag{2.24}$$

The matrix $\mathbf{A}^+$ is called the Moore–Penrose pseudo-inverse of **A** or just the pseudo-inverse. Thus the solution of (2.22) is

$$\mathbf{x} = (\mathbf{A}^+)\,\mathbf{b} \tag{2.25}$$

This definition of the pseudo-inverse, $\mathbf{A}^+$, requires **A** to have full rank. If **A** is full rank and $m > n$, then rank($\mathbf{A}$) $= n$. Now rank($\mathbf{A}^\top\mathbf{A}$) $=$ rank($\mathbf{A}$) and hence rank($\mathbf{A}^\top\mathbf{A}$) $= n$. Since $\mathbf{A}^\top\mathbf{A}$ is an $n \times n$ array, $\mathbf{A}^\top\mathbf{A}$ is automatically full rank and $\mathbf{A}^+$ is then a unique $m \times n$ array. If **A** is rank deficient, then $\mathbf{A}^\top\mathbf{A}$ is rank deficient and cannot be inverted.

If **A** is square and nonsingular, then $\mathbf{A}^+ = \mathbf{A}^{-1}$. If **A** is complex then

$$\mathbf{A}^+ = \left(\mathbf{A}^\mathrm{H}\mathbf{A}\right)^{-1}\mathbf{A}^\mathrm{H} \tag{2.26}$$

where $\mathbf{A}^\mathrm{H}$ is the conjugate transpose, described in Appendix A, Section A.6. The product $\mathbf{A}^\top\mathbf{A}$ has a condition number, which is the square of the condition number of **A**. This has implications for the computations involved in $\mathbf{A}^+$.

It can be shown that the pseudo-inverse has the following properties:

1. $\mathbf{A}(\mathbf{A}^+)\mathbf{A} = \mathbf{A}$
2. $(\mathbf{A}^+)\mathbf{A}(\mathbf{A}^+) = \mathbf{A}^+$
3. $(\mathbf{A}^+)\mathbf{A}$ and $\mathbf{A}(\mathbf{A}^+)$ are symmetrical matrices.

We must now consider the situation that pertains when **A** of (2.22) is $m \times n$ with $m < n$; that is, an equation system with more variables than equations. If **A** is full rank, then rank($\mathbf{A}$) $= m$. Now rank($\mathbf{A}^\top\mathbf{A}$)$=$rank($\mathbf{A}$) and hence rank($\mathbf{A}^\top\mathbf{A}$) $= m$. Since $\mathbf{A}^\top\mathbf{A}$ is an $n \times n$ matrix, $\mathbf{A}^\top\mathbf{A}$ is rank deficient and cannot be inverted, even though **A** is full rank. We can avoid this problem by recasting (2.22) as follows:

$$\mathbf{A}\mathbf{x} = \left(\mathbf{A}\mathbf{A}^\top\right)\left(\mathbf{A}\mathbf{A}^\top\right)^{-1}\mathbf{b}$$

and hence

$$\mathbf{x} = \mathbf{A}^\top \left(\mathbf{A}\mathbf{A}^\top\right)^{-1} \mathbf{b}$$

Thus

$$\mathbf{x} = (\mathbf{A}^+)\,\mathbf{b}$$

where $\mathbf{A}^+ = \mathbf{A}^\top(\mathbf{A}\mathbf{A}^\top)^{-1}$ and is the pseudo-inverse. Note that $\mathbf{A}\mathbf{A}^\top$ is an $m \times m$ array with rank $m$ and can thus be inverted.

It has been shown that if **A** is rank deficient, (2.24) cannot be used to determine the pseudo-inverse of **A**. This doesn't mean that the pseudo-inverse does not exist; it always exists but we must use a different method to evaluate it. When **A** is rank deficient, or close to rank deficient, $\mathbf{A}^+$ is best calculated from the singular value decomposition (SVD) of **A**. If **A** is real, the SVD of **A** is $\mathbf{U}\mathbf{S}\mathbf{V}^\top$ where **U** is an orthogonal $m \times m$ matrix, **V** is an orthogonal $n \times n$ matrix, and **S** is a $n \times m$ matrix of singular values. Thus the SVD of $\mathbf{A}^\top$ is $\mathbf{V}\mathbf{S}^\top\mathbf{U}^\top$ so that

$$\mathbf{A}^\top\mathbf{A} = (\mathbf{V}\mathbf{S}^\top\mathbf{U}^\top)(\mathbf{U}\mathbf{S}\mathbf{V}^\top) = \mathbf{V}\mathbf{S}^\top\mathbf{S}\mathbf{V}^\top \text{ since } \mathbf{U}^\top\mathbf{U} = \mathbf{I}$$

Hence

$$\begin{aligned}\mathbf{A}^+ &= (\mathbf{V}\mathbf{S}^\top\mathbf{S}\mathbf{V}^\top)^{-1}\mathbf{V}\mathbf{S}^\top\mathbf{U}^\top = \mathbf{V}^{-\top}(\mathbf{S}^\top\mathbf{S})^{-1}\mathbf{V}^{-1}\mathbf{V}\mathbf{S}^\top\mathbf{U}^\top \\ &= \mathbf{V}(\mathbf{S}^\top\mathbf{S})^{-1}\mathbf{S}^\top\mathbf{U}^\top\end{aligned} \tag{2.27}$$

We note that $\mathbf{V}^{-\top} = (\mathbf{V}^\top)^{-1} = (\mathbf{V}^\top)^\top = \mathbf{V}$ because by orthogonality $\mathbf{V}\mathbf{V}^\top = \mathbf{I}$. Since **V** is an $m \times m$ matrix, **U** is an $n \times n$ matrix, and **S** is an $n \times m$ matrix, then (2.27) is conformable (i.e., matrix multiplication is possible); see Appendix A, Section A.5.

Consider now the case when **A** is rank deficient. In this situation $\mathbf{S}^\top\,\mathbf{S}$ cannot be inverted because of the very small or zero singular values. To deal with this problem we take only the $r$ nonzero singular values of the matrix so that **S** is an $r \times r$ matrix where $r$ is the rank of **A**. To make the multiplications of (2.27) conformable we take the first $r$ columns of **V** and the first $r$ rows of $\mathbf{U}^\top$—that is, the first $r$ columns of **U**. This is illustrated in the second of following examples in which the pseudo-inverse of **A** is determined.

**Example 2.5**

Consider the following matrix:

$$\mathbf{A} = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 9 \\ 5 & 6 & 7 \\ -2 & 3 & 1 \end{bmatrix}$$

Computing the pseudo-inverse of **A** using a MATLAB implementation of (2.24) we have

```
>> A = [1 2 3;4 5 9;5 6 7;-2 3 1];
>> rank(A)

ans =
     3
```

We note that A is full rank. Thus

```
>> A_cross = inv(A.'*A)*A.'

A_cross =
   -0.0747   -0.1467    0.2500   -0.2057
   -0.0378   -0.2039    0.2500    0.1983
    0.0858    0.2795   -0.2500   -0.0231
```

The MATLAB function pinv provides this result directly and with greater accuracy.

```
A*A_cross*A

ans =
    1.0000    2.0000    3.0000
    4.0000    5.0000    9.0000
    5.0000    6.0000    7.0000
   -2.0000    3.0000    1.0000

>> A*A_cross

ans =
    0.1070    0.2841    0.0000    0.1218
    0.2841    0.9096    0.0000   -0.0387
    0.0000    0.0000    1.0000   -0.0000
    0.1218   -0.0387   -0.0000    0.9834

>> A_cross*A

ans =
    1.0000    0.0000    0.0000
    0.0000    1.0000    0.0000
   -0.0000   -0.0000    1.0000
```

Note that these calculations verify that A*A_cross*A equals A and that both A*A_cross and A_cross*A are symmetrical.

**Example 2.6**
Consider the following rank deficient matrix:

$$\mathbf{G} = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 9 \\ 7 & 11 & 18 \\ -2 & 3 & 1 \\ 7 & 1 & 8 \end{bmatrix}$$

Using MATLAB, we have

```
>> G = [1 2 3;4 5 9;7 11 18;-2 3 1;7 1 8]

G =
     1     2     3
     4     5     9
     7    11    18
    -2     3     1
     7     1     8

>> rank(G)

ans =
     2
```

Note that `G` has a rank of 2 (i.e., it is rank deficient) and we cannot use (2.24) to determine its pseudo-inverse. We now find the SVD of `G`:

```
>> [U S V] = svd(G)

U =
   -0.1381    0.0839    0.9724   -0.0044   -0.1681
   -0.4115    0.0215    0.0539   -0.6081    0.6764
   -0.8258    0.2732   -0.2165    0.0607   -0.4392
   -0.0524    0.5650    0.0366    0.6373    0.5201
   -0.3563   -0.7737    0.0572    0.4695    0.2253

S =
   26.8394         0         0
         0    6.1358         0
         0         0    0.0000
         0         0         0
         0         0         0
```

```
V =
   -0.3709   -0.7274   -0.5774
   -0.4445    0.6849   -0.5774
   -0.8154   -0.0425    0.5774
```

We now select the two significant singular values for use in the subsequent computation:

```
>> SS = S(1:2,1:2)

SS =
   26.8394         0
         0    6.1358
```

To make the multiplication conformable we use only the first two columns of `U` and `V`.

```
>> G_cross = V(:,1:2)*inv(SS.'*SS)*SS.'*U(:,1:2).'

G_cross =
   -0.0080    0.0031   -0.0210   -0.0663    0.0966
    0.0117    0.0092    0.0442    0.0639   -0.0805
    0.0036    0.0124    0.0232   -0.0023    0.0162
```

This result can be obtained directly using the `pinv` function, which is based on the singular value decomposition of `G`.

```
>> G*G_cross

ans =
    0.0261    0.0586    0.1369    0.0546   -0.0157
    0.0586    0.1698    0.3457    0.0337    0.1300
    0.1369    0.3457    0.7565    0.1977    0.0829
    0.0546    0.0337    0.1977    0.3220   -0.4185
   -0.0157    0.1300    0.0829   -0.4185    0.7256

>> G_cross*G

ans =
    0.6667   -0.3333    0.3333
   -0.3333    0.6667    0.3333
    0.3333    0.3333    0.6667
```

Note that `G*G_cross` and `G_cross*G` are symmetric.

In the following section we will apply these methods to solve over- and underdetermined systems and discuss the meaning of the solution.

## 2.12 Over- and Underdetermined Systems

We will begin by examining examples of overdetermined systems, that is, systems of equations in which there are more equations than unknown variables.

Although overdetermined systems may have a unique solution, most often we are concerned with equation systems that are generated from experimental data, which can lead to a relatively small degree of inconsistency between the equations. For example, consider the following overdetermined system of linear equations:

$$\begin{aligned} x_1 + x_2 &= 1.98 \\ 2.05x_1 - x_2 &= 0.95 \\ 3.06x_1 + x_2 &= 3.98 \\ -1.02x_1 + 2x_2 &= 0.92 \\ 4.08x_1 - x_2 &= 2.90 \end{aligned} \tag{2.28}$$

Figure 2.5 shows that (2.28) is such a system; the lines do not intersect in a point, although there is a point that *nearly* satisfies all the equations.

We would like to choose the best point of all in the region defined by the intersections. One criterion for doing this is that the chosen solution should minimize the sum of squares of the residual errors (or residuals) of the equations. For example, consider the equation system (2.28). Letting $r_1, \ldots, r_5$ be the residuals, then

$$\begin{aligned} x_1 + x_2 - 1.98 &= r_1 \\ 2.05x_1 - x_2 - 0.95 &= r_2 \\ 3.06x_1 + x_2 - 3.98 &= r_3 \\ -1.02x_1 + 2x_2 - 0.92 &= r_4 \\ 4.08x_1 - x_2 - 2.90 &= r_5 \end{aligned}$$

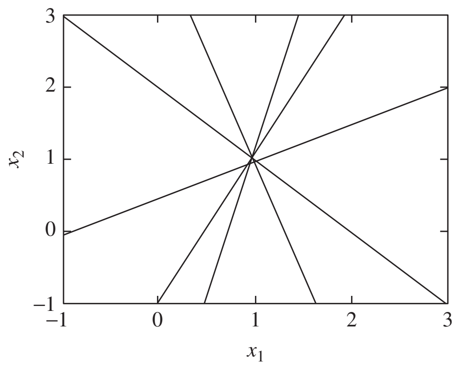

**FIGURE 2.5** Plot of inconsistent equation system (2.28).

In this case the sum of the residuals squared is given by

$$S = \sum_{i=1}^{5} r_i^2 \tag{2.29}$$

We wish to minimize $S$ and we can do this by making

$$\frac{\partial S}{\partial x_k} = 0, \quad k = 1, 2$$

Now

$$\frac{\partial S}{\partial x_k} = \sum_{i=1}^{5} 2r_i \frac{\partial r_i}{\partial x_k}, \quad k = 1, 2$$

and thus

$$\sum_{i=1}^{5} r_i \frac{\partial r_i}{\partial x_k} = 0, \quad k = 1, 2 \tag{2.30}$$

It can be shown that minimizing the sum of the squares of the equation residuals using (2.30) gives an identical solution to that given by the pseudo-inverse method of solving the equation system.

When solving a set of overdetermined equations, determining the pseudo-inverse of the system matrix is only part of the process and normally we do not require this interim result. The MATLAB operator \ solves overdetermined systems automatically. Thus the operator may be used to solve any linear equation system.

In the following example we compare the results obtained using the operator \ and using the pseudo-inverse for solving (2.28). The MATLAB script is

```
% e3s206.m
A = [1 1;2.05 -1;3.06 1;-1.02 2;4.08 -1];
b = [1.98;0.95;3.98;0.92;2.90];
x = pinv(A)*b
norm_pinv = norm(A*x-b)
x = A\b
norm_op = norm(A*x-b)
```

Running this script gives the following numeric output:

```
x =
    0.9631
    0.9885
```

```
norm_pinv =
    0.1064

x =
    0.9631
    0.9885

norm_op =
    0.1064
```

Here both the MATLAB operator \ and the function `pinv` have provided the same "best-fit" solution for the inconsistent set of equations. Figure 2.6 shows the region where these equations intersect in greater detail than Figure 2.5. The symbol "+" indicates the MATLAB solution, which lies in this region. The norm of $\mathbf{Ax} - \mathbf{b}$ is the square root of the sum of the squares of the residuals and provides a measure of how well the equations are satisfied.

The MATLAB operator \ does not solve an overdetermined system by using the pseudo-inverse, as given in (2.24). Instead, it solves (2.22) directly by QR decomposition. QR decomposition can be applied to both square and rectangular matrices providing the number of rows is greater than the number of columns. For example, applying the MATLAB function `qr` to solve the overdetermined system (2.28) we have

```
>> A = [1 1;2.05 -1;3.06 1;-1.02 2;4.08 -1];
>> b = [1.98 0.95 3.98 0.92 2.90].';
>> [Q R] = qr(A)

Q =
   -0.1761    0.4123   -0.7157   -0.2339   -0.4818
   -0.3610   -0.2702    0.0998    0.6751   -0.5753
   -0.5388    0.5083    0.5991   -0.2780   -0.1230
    0.1796    0.6839   -0.0615    0.6363    0.3021
   -0.7184   -0.1756   -0.3394    0.0857    0.5749

R =
   -5.6792    0.7237
         0    2.7343
         0         0
         0         0
         0         0
```

In the equation $\mathbf{Ax} = \mathbf{b}$ we have replaced $\mathbf{A}$ by $\mathbf{QR}$ so that $\mathbf{QRx} = \mathbf{b}$. Let $\mathbf{Rx} = \mathbf{y}$. Thus we have $\mathbf{y} = \mathbf{Q}^{-1}\mathbf{b} = \mathbf{Q}^{\top}\mathbf{b}$ since $\mathbf{Q}$ is orthogonal. Once $\mathbf{y}$ is determined we can efficiently determine

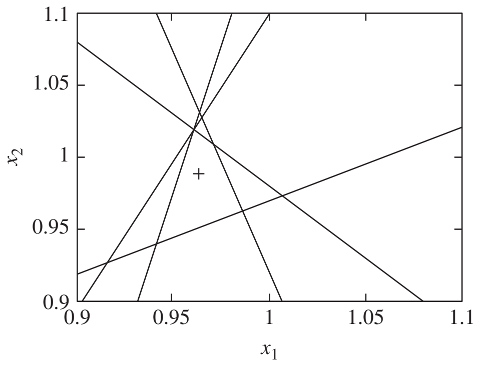

**FIGURE 2.6** Plot of inconsistent equation system (2.28) showing the region of intersection of the equations, where + indicates the "best" solution.

**x** by back substitution since **R** is upper triangular. Thus, continuing the previous example,

```
>> y = Q.'*b

y =
   -4.7542
    2.7029
    0.0212
   -0.0942
   -0.0446
```

Using the second row of **R** and the second row of **y** we can determine $x_2$. From the first row of **R** and the first row of **y** we can determine $x_1$ since $x_2$ is known. Thus

$$\begin{aligned} -5.6792x_1 + 0.7237x_2 &= -4.7542 \\ 2.7343x_2 &= 2.7029 \end{aligned}$$

give $x_1 = 0.9631$ and $x_2 = 0.9885$, as before. The MATLAB operator \ implements this sequence of operations.

We now consider a case where the coefficient matrix of the overdetermined system is rank deficient. The following example is rank deficient and represents a system of parallel lines.

$$\begin{aligned} x_1 + 2x_2 &= 1.00 \\ x_1 + 2x_2 &= 1.03 \\ x_1 + 2x_2 &= 0.97 \\ x_1 + 2x_2 &= 1.01 \end{aligned}$$

In MATLAB this becomes

```
>> A = [1 2;1 2;1 2;1 2]

A =
     1     2
     1     2
     1     2
     1     2

>> b = [1 1.03 0.97 1.01].'

b =
    1.0000
    1.0300
    0.9700
    1.0100

>> y = A\b
Warning: Rank deficient, rank = 1,  tol =   3.552714e-015.

y =
         0
    0.5012

>> norm(y)

ans =
    0.5012
```

The user is warned that this system is rank deficient. We have solved the system using the \ operator and now solve it using the `pinv` function as follows:

```
>> x = pinv(A)*b

x =
    0.2005
    0.4010

>> norm(x)

ans =
    0.4483
```

We see that when the `pinv` function and the \ operator are applied to rank deficient systems, the `pinv` function gives the solution with the smallest Euclidean norm; see Appendix A, Section A.10. Clearly there is no unique solution to this system since it represents a set of parallel lines.

We now turn to the problem of underdetermined systems. Here there is insufficient information to determine a unique solution. For example, consider the equation system

$$\begin{aligned} x_1 + 2x_2 + 3x_3 + 4x_4 &= 1 \\ -5x_1 + 3x_2 + 2x_3 + 7x_4 &= 2 \end{aligned}$$

Expressing these equations in MATLAB we have

```
>> A = [1 2 3 4;-5 3 2 7];
>> b = [1 2].';
>> x1 = A\b

x1 =
   -0.0370
         0
         0
    0.2593

>> x2 = pinv(A)*b

x2 =
   -0.0780
    0.0787
    0.0729
    0.1755
```

We calculate the norms:

```
>> norm(x1)

ans =
    0.2619

>> norm(x2)

ans =
    0.2199
```

The first solution, `x1`, is a solution which satisfies the system; the second solution, `x2`, satisfies the system of equations but also gives the solution with the minimum norm.

The definition of the Euclidean or 2-norm of a vector is the square root of the sum of the squares of the elements of the vector; see Appendix A, Section A.10. The shortest distance between a point in space and the origin is given by Pythagoras's theorem as the square root of the sum of squares of the coordinates of the point. Thus the Euclidean norm of a vector, which is a point on a line, surface, or hypersurface, may be interpreted geometrically as the distance between this point and the origin. The vector with the minimum norm must be the point on the line, surface, or hypersurface that is closest to the origin. The line joining this vector to the origin must be perpendicular to the line, surface, or hypersurface. Giving the minimum norm solution has the advantage that, whereas there are an infinite number of solutions to an underdetermined problem, there is only one minimum norm solution. This provides a uniform result.

To complete the discussion of over- and underdetermined systems we consider the use of the `lsqnonneg` function, which solves the nonnegative least squares problem. This solves the problem of finding a solution **x** to the linear equation system

$$\mathbf{Ax} = \mathbf{b} \text{ subject to } \mathbf{x} \geq \mathbf{0}$$

where **A** and **b** must be real. This is equivalent to the problem of finding the vector **x** that minimizes norm($\mathbf{Ax} - \mathbf{b}$) subject to $\mathbf{x} \geq \mathbf{0}$.

We can call the MATLAB function `lsqnonneg` for a specific problem using the statement

```
x = lsqnonneg(A,b)
```

where `A` corresponds to **A** in our definition and `b` to **b**. The solution is given by `x`. Consider the example which follows. Solve

$$\begin{bmatrix} 1 & 1 & 1 & 1 & 0 \\ 1 & 2 & 3 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \end{bmatrix} = \begin{bmatrix} 7 \\ 12 \end{bmatrix}$$

subject to $x_i \geq 0$, $i = 1, 2, \ldots, 5$. In MATLAB this becomes

```
>> A = [1 1 1 1 0;1 2 3 0 1];
>> b = [7 12].';
```

Solving this system gives

```
>> x = lsqnonneg(A,b)

x =
     0
     0
     4
     3
     0
```

We can solve this using \ but this will not ensure nonnegative values for `x`.

```
>> x2 = A\b

x2 =
         0
         0
    4.0000
    3.0000
         0
```

In this case we do obtain a nonnegative solution but this is fortuitous.

The following example illustrates how the `lsqnonneg` function forces a nonnegative solution that best satisfies the equations:

$$\begin{bmatrix} 3.0501 & 4.8913 \\ 3.2311 & -3.2379 \\ 1.6068 & 7.4565 \\ 2.4860 & -0.9815 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} 2.5 \\ 2.5 \\ 0.5 \\ 2.5 \end{bmatrix} \tag{2.31}$$

```
>> A = [3.0501 4.8913;3.2311 -3.2379; 1.6068 7.4565;2.4860 -0.9815];
>> b = [2.5 2.5 0.5 2.5].';
```

We can compute the solution using \ or the `lsqnonneg` function:

```
>> x1 = A\b

x1 =
    0.8307
   -0.0684
```

```
>> x2 = lsqnonneg(A,b)

x2 =
    0.7971
         0

>> norm(A*x1-b)

ans =
    0.7040

>> norm(A*x2-b)

ans =
    0.9428
```

Thus, the best fit is given by using the operator `\`, but if we require all components of the solution to be nonnegative, then we must use the `lsqnonneg` function.

## 2.13 Iterative Methods

Except in special circumstances it is unlikely that any function or script developed by the user will outperform a function or operator that is an integral part of MATLAB. Thus we cannot expect to develop a function that will determine the solution of $\mathbf{Ax} = \mathbf{b}$ more efficiently than by using the MATLAB operation `A\b`. However, we describe iterative methods here for the sake of completeness.

Iterative methods of solution are developed as follows. We begin with a system of linear equations

$$\begin{array}{ccccc} a_{11}x_1+ & a_{12}x_2+ & \ldots & +a_{1n}x_n & = b_1 \\ a_{21}x_1+ & a_{22}x_2+ & \ldots & +a_{2n}x_n & = b_2 \\ \vdots & \vdots & & \vdots & \vdots \\ a_{n1}x_1+ & a_{n2}x_2+ & \ldots & +a_{nn}x_n & = b_n \end{array}$$

These can be rearranged to give

$$\begin{aligned} x_1 &= \left(b_1 - a_{12}x_2 - a_{13}x_3 - \ldots - a_{1n}x_n\right)/a_{11} \\ x_2 &= (b_2 - a_{21}x_1 - a_{23}x_3 - \ldots - a_{2n}x_n/a_{22} \\ &\vdots \qquad \vdots \qquad \vdots \qquad \vdots \\ x_n &= \left(b_n - a_{n1}x_1 - a_{n2}x_2 - \ldots - a_{n,n-1}x_{n-1}\right)/a_{nn} \end{aligned} \tag{2.32}$$

If we assume initial values for $x_i$, where $i = 1, \ldots, n$, and substitute these values into the right side of the preceding equations, we may determine new values for the $x_i$ from (2.32). The iterative process is continued by substituting these values of $x_i$ into the right side of the equations, and so on. There are several variants of the process. For example, we can use old values of $x_i$ in the right side of the equations to determine *all* the new values of $x_i$ in the left side of the equation. This is called Jacobi or simultaneous iteration. Alternatively, we may use a new value of $x_i$ in the right side of the equation as soon as it is determined, to obtain the other values of $x_i$ in the right side. For example, once a new value of $x_1$ is determined from the first equation of (2.32), it is used in the second equation, together with the old $x_3, \ldots, x_n$ to determine $x_2$. This is called Gauss–Seidel or cyclic iteration.

The conditions for convergence for this type of iteration are

$$|a_{ii}| >> \sum_{j=1,\ j \neq i}^{n} |a_{ij}| \quad \text{for} \ i = 1,\ 2, \ldots, n$$

Thus these iterative methods are only guaranteed to work when the coefficient matrix is diagonally dominant. An iterative method based on conjugate gradients for the solution of systems of linear equations is discussed in Chapter 8.

## 2.14 Sparse Matrices

Sparse matrices arise in many problems of science and engineering—for example, in linear programming and the analysis of structures. Indeed, most large matrices that arise in the analysis of physical systems are sparse and the recognition of this fact makes the solution of linear systems with millions of coefficients feasible. The aim of this section is to give a brief description of the extensive sparse matrix facilities available in MATLAB and to give practical illustrations of their value through examples. For background information on how MATLAB implements the concept of sparsity, see Gilbert et al. (1992).

It is difficult to give a simple quantitative answer to the question: When is a matrix sparse? A matrix is sparse if it contains a high proportion of zero elements. However, this is significant only if the sparsity is of such an extent that we can utilize this feature to reduce the computation time and storage facilities required for operations used on such matrices. One major way in which time can be saved in dealing with sparse matrices is to avoid unnecessary operations on zero elements.

MATLAB *does not automatically treat a matrix as sparse* and the sparsity features of MATLAB are not introduced until invoked. Thus the user determines whether a matrix is in the sparse class or the full class. If the user considers a matrix to be sparse and wants to use this fact to advantage, the matrix must first be converted to sparse form. This is achieved by using the function `sparse`. Thus `b = sparse(a)` converts the matrix `a` to sparse form and subsequent MATLAB operations will take account of this sparsity. If we wish to return this matrix to full form, we simply use `c = full(b)`. However, the sparse function can also be used directly to generate sparse matrices.

It is important to note that binary operators `*`, `+`, `-`, `/`, and `\` produce sparse results if *both* operands are sparse. Thus the property of sparsity may survive a long sequence of matrix operations. In addition, such functions as `chol(A)` and `lu(A)` produce sparse results if the matrix `A` is sparse. However, in mixed cases, where one operand is sparse and the other is full, the result is generally a full matrix. Thus the property of sparsity may be inadvertently lost. Notice in particular that `eye(n)` is not in the sparse class of matrices in MATLAB but a sparse identity matrix can be created using `speye(n)`. Thus the latter should be used in manipulations with sparse matrices.

We will now introduce some of the key MATLAB functions for dealing with sparse matrices, describe their use and, where appropriate, give examples of their application. The simplest MATLAB function that helps in dealing with sparsity is the function `nnz(a)`, which provides the number of nonzero elements in a given matrix `a`, regardless of whether it is sparse or full. A function that enables us to examine whether a given matrix has been defined or has been propagated as sparse is the function `issparse(a)`, which returns the value 1 if the matrix `a` is sparse or 0 if it is not sparse. The function `spy(a)` allows the user to view the structure of a given matrix `a` by displaying symbolically only its nonzero elements; see Figure 2.7 later in the chapter for examples.

Before we can illustrate the action of these and other functions, it is useful to generate some sparse matrices. This is easily done using a different form of the `sparse` function. This time the function is supplied with the location of the nonzero entries in the matrix, the value of these entries, the size of the sparse matrix, and the space allocated for the nonzero entries. This function call takes the form `sparse(i, j, nzvals, m, n, nzmax)`. This generates an `m` $\times$ `n` matrix and allocates the nonzero values in the vector `nzvals` to the positions in the matrix given by the vectors `i` and `j`. The row position is given by `i` and the column position by `j`. Space is allocated for `nzmax` nonzeros. Since all but one parameter is optional, there are many forms of this function. We cannot give examples of all these forms but the following cases illustrate its use.

```
>> colpos = [1 2 1 2 5 3 4 3 4 5];
>> rowpos = [1 1 2 2 2 4 4 5 5 5];
>> value = [12 -4 7 3 -8 -13 11 2 7 -4];
>> A = sparse(rowpos,colpos,value,5,5)
```

These statements give the following output:

```
A =
   (1,1)        12
   (2,1)         7
   (1,2)        -4
   (2,2)         3
   (4,3)       -13
   (5,3)         2
   (4,4)        11
   (5,4)         7
   (2,5)        -8
   (5,5)        -4
```

We see that a $5 \times 5$ sparse matrix with 10 nonzero elements has been generated with the required coefficient values in the required positions. This sparse matrix can be converted to a full matrix as follows:

```
>> B = full(A)

B =
    12    -4     0     0     0
     7     3     0     0    -8
     0     0     0     0     0
     0     0   -13    11     0
     0     0     2     7    -4
```

This is the equivalent full matrix. Now the following statements test to see if the matrices `A` and `B` are in the sparse class and give the number of nonzeros they contain.

```
>> [issparse(A) issparse(B) nnz(A) nnz(B)]

ans =
     1     0    10    10
```

Clearly these functions give the expected results. Since `A` is a member of the class of sparse matrices, the value of `issparse(A)` is 1. However, although `B` looks sparse, it is not *stored* as a sparse matrix and hence is not in the class of sparse matrices within the MATLAB environment. The next example shows how to generate a large $5000 \times 5000$ sparse matrix and compares the time required to solve a linear system of equations involving this sparse matrix with the time required for the equivalent full matrix. The script for this is

```
% e3s207.m Generates a sparse triple diagonal matrix
n = 5000;
rowpos = 2:n; colpos = 1:n-1;
values = 2*ones(1,n-1);
Offdiag = sparse(rowpos,colpos,values,n,n);
A = sparse(1:n,1:n,4*ones(1,n),n,n);
A = A+Offdiag+Offdiag.';
%generate full matrix
B = full(A);
```

```
%generate arbitrary right hand side for system of equations
rhs = [1:n].';
tic, x = A\rhs; f1 = toc;
tic, x = B\rhs; f2 = toc;
fprintf('Time to solve sparse matrix = %8.5f\n',f1);
fprintf('Time to solve  full  matrix = %8.5f\n',f2);
```

This provides the following results:

```
Time to solve sparse matrix =  0.00051
Time to solve  full  matrix =  5.74781
```

In this example there is a major reduction in the time taken to solve the system when using the sparse class of matrix. We now perform a similar exercise, this time to determine the `lu` decomposition of a $5000 \times 5000$ matrix:

```
% e3s208.m
n = 5000;
offdiag = sparse(2:n,1:n-1,2*ones(1,n-1),n,n);
A = sparse(1:n,1:n,4*ones(1,n),n,n);
A = A+offdiag+offdiag';
%generate full matrix
B = full(A);
%generate arbitrary right hand side for system of equations
rhs = [1:n]';
tic, lu1 = lu(A); f1 = toc;
tic, lu2 = lu(B); f2 = toc;
fprintf('Time for sparse LU = %8.4f\n',f1);
fprintf('Time for  full  LU = %8.4f\n',f2);
```

The time taken to solve the systems is

```
Time for sparse LU =   0.0056
Time for  full  LU =   9.6355
```

Again this provides a considerable reduction in the time taken.

An alternative way to generate sparse matrices is to use the functions `sprandn` and `sprandsym`. These provide random sparse matrices and random sparse symmetric matrices, respectively. The call

```
A = sprandn(m,n,d)
```

produces an $m \times n$ random matrix with normally distributed nonzero entries of density `d`. The density is the proportion of the nonzero entries to the total number of entries in the matrix. Thus `d` must be in the range 0 to 1. To produce a symmetric random matrix with normally distributed nonzero entries of density `d`, we use

```
A = sprandsys(n,d)
```

Examples of calls of these functions are given by

```
>> A = sprandn(5,5,0.25)

A =
   (2,1)      -0.4326
   (3,3)      -1.6656
   (5,3)      -1.1465
   (4,4)       0.1253
   (5,4)       1.1909
   (4,5)       0.2877

>> B = full(A)

B =
         0         0         0         0         0
   -0.4326         0         0         0         0
         0         0   -1.6656         0         0
         0         0         0    0.1253    0.2877
         0         0   -1.1465    1.1909         0

>> As = sprandsym(5,0.25)

As =
   (3,1)       0.3273
   (1,3)       0.3273
   (5,3)       0.1746
   (5,4)      -0.0376
   (3,5)       0.1746
   (4,5)      -0.0376
   (5,5)       1.1892

>> Bs = full(As)

Bs =
         0         0    0.3273         0         0
         0         0         0         0         0
    0.3273         0         0         0    0.1746
         0         0         0         0   -0.0376
         0         0    0.1746   -0.0376    1.1892
```

An alternative call for `sprandsym` is given by

```
A = sprandsym(n,density,r)
```

If `r` is a scalar, then this produces a random sparse symmetric matrix with a condition number equal to 1/`r`. Remarkably, if `r` is a vector of length *n*, a random sparse matrix with eigenvalues equal to the elements of `r` is produced. Eigenvalues are discussed in Section 2.15. A positive definite matrix has all positive eigenvalues and consequently such a matrix can be generated by choosing each of the *n* elements of `r` to be positive. An example of this form of call is

```
>> Apd = sprandsym(6,0.4,[1 2.5 6 9 2 4.3])

Apd =
   (1,1)        1.0058
   (2,1)       -0.0294
   (4,1)       -0.0879
   (1,2)       -0.0294
   (2,2)        8.3477
   (4,2)       -1.9540
   (3,3)        5.4937
   (5,3)       -1.3300
   (1,4)       -0.0879
   (2,4)       -1.9540
   (4,4)        3.1465
   (3,5)       -1.3300
   (5,5)        2.5063
   (6,6)        4.3000

>> Bpd = full(Apd)

Bpd =
    1.0058   -0.0294         0   -0.0879         0         0
   -0.0294    8.3477         0   -1.9540         0         0
         0         0    5.4937         0   -1.3300         0
   -0.0879   -1.9540         0    3.1465         0         0
         0         0   -1.3300         0    2.5063         0
         0         0         0         0         0    4.3000
```

This provides an important method for generating test matrices with required properties since, by providing a list of eigenvalues with a range of values, we can produce positive definite matrices that are very badly conditioned.

We now return to examine further the value of using sparsity. The reasons for the very high level of improvement in computing efficiency when using the \ operator, illustrated in the example at the beginning of this section, are complex. The process includes a special preordering of the columns of the matrix. This special preordering, called *minimum degree ordering*, is used in the case of the \ operator. This preordering takes different forms depending on whether the matrix is symmetric or nonsymmetric. The aim of any preordering is to reduce the amount of *fill-in* from any subsequent matrix operations. Fill-in is the introduction of additional nonzero elements.

We can examine this preordering process using the `spy` function and the function `symamd`, which implements *symmetric minimum degree ordering* in MATLAB. The function is automatically applied when working on matrices that belong to the class of sparse matrices for the standard functions and operators of MATLAB. However, if we are required to use this preordering in nonstandard applications, then we may use the `symmmd` function. The following examples illustrate the use of this function.

We first consider the simple process of multiplication applied to a full and a sparse matrix. The sparse multiplication uses the minimum degree ordering. The following script generates a sparse matrix, obtains a minimum degree ordering for it, and then examines the result of multiplying the matrix by itself transposed. This is compared with the same operations carried out on the full matrix, and the time required for each operation is compared.

```
% e3s209.m
% generate a sparse matrix
n = 3000;
offdiag = sparse(2:n,1:n-1,2*ones(1,n-1),n,n);
offdiag2 = sparse(4:n,1:n-3,3*ones(1,n-3),n,n);
offdiag3 = sparse(n-5:n,1:6,7*ones(1,6),n,n);
A = sparse(1:n,1:n,4*ones(1,n),n,n);
A = A+offdiag+offdiag'+offdiag2+offdiag2'+offdiag3+offdiag3';
A = A*A.';
% generate full matrix
B = full(A);
m_order = symamd(A);
tic
spmult = A(m_order,m_order)*A(m_order,m_order).';
flsp = toc;
tic, fulmult = B*B.'; flful = toc;
fprintf('Time for sparse mult = %6.4f\n',flsp)
fprintf('Time for  full  mult = %6.4f\n',flful)
```

Running this script results in the following output:

```
Time for sparse mult = 0.0184
Time for  full  mult = 3.8359
```

We now perform a similar experiment to the preceding but for a more complex numerical process than multiplication. In the script that follows we examine LU decomposition. We consider the result of using a minimum degree ordering on the LU decomposition process by comparing the performance of the `lu` function with and without a preordering. The script has the form

```
% e3s210.m
% generate a sparse matrix
n = 100;
offdiag = sparse(2:n,1:n-1,2*ones(1,n-1),n,n);
offdiag2 = sparse(4:n,1:n-3,3*ones(1,n-3),n,n);
offdiag3 = sparse(n-5:n,1:6,7*ones(1,6),n,n);
A = sparse(1:n,1:n,4*ones(1,n),n,n);
A = A+offdiag+offdiag'+offdiag2+offdiag2'+offdiag3+offdiag3';
A = A*A.';
A1 = flipud(A);
A = A+A1;
n1 = nnz(A)
B = full(A); %generate full matrix
m_order = symamd(A);
tic, lud = lu(A(m_order,m_order)); flsp = toc;
n2 = nnz(lud)
tic, fullu = lu(B); flful = toc;
n3 = nnz(fullu)
subplot(2,2,1), spy(A,'k');
title('Original matrix')
subplot(2,2,2), spy(A(m_order,m_order),'k')
title('Ordered matrix')
subplot(2,2,3), spy(fullu,'k')
title('LU decomposition,unordered matrix')
subplot(2,2,4), spy(lud,'k')
title('LU decomposition, ordered matrix')
fprintf('Time for sparse lu = %6.4f\n',flsp)
fprintf('Time for  full  lu = %6.4f\n',flful)
```

Running this script gives

```
n1 =
        2096


n2 =
        1307
```

```
n3 =
        4465

Time for sparse lu = 0.0013
Time for  full  lu = 0.0047
```

As expected, by using a sparse operation we achieve a reduction in the time taken to determine the LU decomposition. Figure 2.7 shows the original matrix with 2096 nonzero elements, the reordered matrix (which has the same number of nonzeros), and the LU decomposition structure both with and without minimum degree ordering. Notice that the number of nonzeros in the LU matrices with preordering is 1307 and without is 4465. Thus there is a large increase in the number of nonzero elements in the LU matrices without preordering. In contrast, LU decomposition of the preordered matrix has produced fewer nonzeros than the original matrix. The reduction of fill-in is an important feature of sparse numerical processes and may ultimately lead to great saving in computational effort. Note that if the size of the matrices is increased from $100 \times 100$ to $3000 \times 3000$ then the output from the preceding script is

```
n1 =
       65896

n2 =
       34657
```

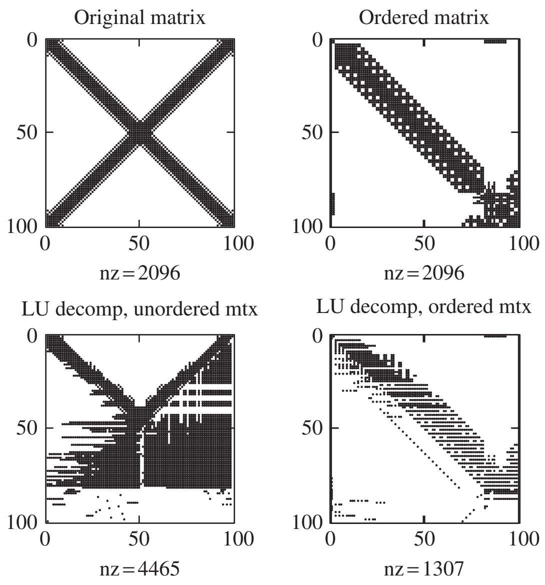

**FIGURE 2.7** Effect of minimum degree ordering on LU decomposition.

```
n3 =
      526810

Time for sparse lu = 0.0708
Time for  full  lu = 2.3564
```

Here we obtain a more substantial reduction by using sparse operations.

The MATLAB function `symamd` provides a minimum degree ordering for symmetric matrices. For nonsymmetric matrices MATLAB provides the function `colmmd`, which gives the column minimum degree ordering for nonsymmetric matrices. An alternative ordering, which is used to reduce bandwidth, is the reverse Cuthill-McKee ordering. This is implemented in MATLAB by the function `symrcm`. The execution of the statement `p = symrcm(A)` provides the permutation vector `p` to produce the required preordering and `A(p,p)` is the reordered matrix.

We have shown that in general taking account of sparsity will provide savings in floating-point operations. However, these savings fall off as the matrices on which we are operating become less sparse, as the following example illustrates.

```
% e3s211.m
n = 1000;  b = 1:n;
disp('   density    time_sparse   time_full');
for density = 0.004:0.003:0.039
    A = sprandsym(n,density)+0.1*speye(n);
    density = density+1/n;
    tic, x = A\b'; f1 = toc;
    B = full(A);
    tic, y = B\b'; f2 = toc;
    fprintf('%10.4f %12.4f %12.4f\n',density,f1,f2);
end
```

In the preceding script a diagonal of elements has been added to the randomly generated sparse matrix. This is done to ensure that each row of the matrix contains a nonzero element; otherwise, the matrix may be singular. Adding this diagonal modifies the density. If the original $n \times n$ matrix has a density of $d$, then, assuming that this matrix has only zeros on the diagonal, the modified density is $d + 1/n$.

```
   density    time_sparse   time_full
    0.0050       0.0204       0.1907
    0.0080       0.0329       0.1318
    0.0110       0.0508       0.1332
    0.0140       0.0744       0.1399
    0.0170       0.0892       0.1351
    0.0200       0.1064       0.1372
    0.0230       0.1179       0.1348
    0.0260       0.1317       0.1381
    0.0290       0.1444       0.1372
    0.0320       0.1516       0.1369
    0.0350       0.1789       0.1404
    0.0380       0.1627       0.1450
```

This output shows that the advantage of using a sparse class of matrix diminishes as the density increases.

Another application where sparsity is important is in solving the least squares problem. This problem is known to be ill-conditioned and hence any saving in computational effort is particularly beneficial. This is directly implemented by using `A\b` where `A` is nonsquare and sparse. To illustrate the use of the \ operator with sparse matrices and compare its performance when no account is taken of sparsity, we use the following script:

```
% e3s212.m
% generate a sparse triple diagonal matrix
n = 1000;
rowpos = 2:n;  colpos = 1:n-1;
values = ones(1,n-1);
offdiag = sparse(rowpos,colpos,values,n,n);
A = sparse(1:n,1:n,4*ones(1,n),n,n);
A = A+offdiag+offdiag';
%Now generate a sparse least squares system
Als = A(:,1:n/2);
%generate full matrix
Cfl = full(Als);
rhs = 1:n;
tic, x = Als\rhs'; f1 = toc;
tic, x = Cfl\rhs'; f2 = toc;
fprintf('Time for sparse least squares solve = %8.4f\n',f1)
fprintf('Time for  full  least squares solve = %8.4f\n',f2)
```

This provides the following results:

```
Time for sparse least squares solve =   0.0023
Time for  full  least squares solve =   0.2734
```

Again we see the advantage of using sparsity.

We have not covered all aspects of sparsity or described all the related functions. However, we hope this section has provided a helpful introduction to this difficult but important and valuable development of MATLAB.

## 2.15 The Eigenvalue Problem

Eigenvalue problems arise in many branches of science and engineering. For example, the vibration characteristics of structures are determined from the solution of an algebraic eigenvalue problem. Here we consider a particular example of a system of masses and springs shown in Figure 2.8. The equations of motion for this system are

$$\begin{aligned} m_1\ddot{q}_1 + (k_1+k_2+k_4)\,q_1 - k_2q_2 - k_4q_3 &= 0 \\ m_2\ddot{q}_2 - k_2q_1 + (k_2+k_3)\,q_2 - k_3q_3 &= 0 \\ m_3\ddot{q}_3 - k_4q_1 - k_3q_2 + (k_3+k_4)\,q_3 &= 0 \end{aligned} \tag{2.33}$$

where $m_1$, $m_2$, and $m_3$ are the system masses and $k_1, \ldots, k_4$ are the spring stiffnesses. If we assume an harmonic solution for each coordinate, then $q_i(t) = u_i \exp(\jmath\omega t)$ where $\jmath = \sqrt{-1}$, for $i = 1$, 2, and 3. Hence $d^2q_i/dt^2 = -\omega^2 u_i \exp(\jmath\omega t)$. Substituting in (2.33) and canceling the common factor $\exp(j\omega t)$ gives

$$\begin{aligned} -\omega^2 m_1u_1 + (k_1+k_2+k_4)\,u_1 - k_2u_2 - k_4u_3 &= 0 \\ -\omega^2 m_2u_2 - k_2u_1 + (k_2+k_3)\,u_2 - k_3u_3 &= 0 \\ -\omega^2 m_3u_3 - k_4u_1 - k_3u_2 + (k_3+k_4)\,u_3 &= 0 \end{aligned} \tag{2.34}$$

If $m_1 = 10$ kg, $m_2 = 20$ kg, $m_3 = 30$ kg, $k_1 = 10$ kN/m, $k_2 = 20$ kN/m, $k_3 = 25$ kN/m, and $k_4 = 15$ kN/m, then (2.34) becomes

$$\begin{aligned} -\omega^2 10u_1 + 45000u_1 - 20000u_2 - 15000u_3 &= 0 \\ -\omega^2 20u_1 - 20000u_1 + 45000u_2 - 25000u_3 &= 0 \\ -\omega^2 30u_1 - 15000u_1 - 25000u_2 + 40000u_3 &= 0 \end{aligned}$$

This can be expressed in matrix notation as

$$-\omega^2\mathbf{Mu} + \mathbf{Ku} = \mathbf{0} \tag{2.35}$$

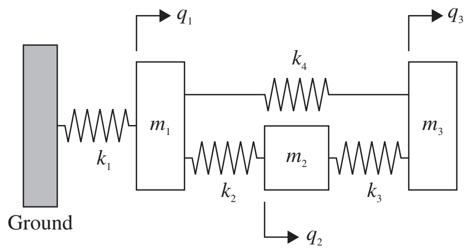

**FIGURE 2.8** Mass-spring system with three degrees of freedom.

where

$$\mathbf{M} = \begin{bmatrix} 10 & 0 & 0 \\ 0 & 20 & 0 \\ 0 & 0 & 30 \end{bmatrix} \text{kg} \quad \text{and} \quad \mathbf{K} = \begin{bmatrix} 45 & -20 & -15 \\ -20 & 45 & -25 \\ -15 & -25 & 40 \end{bmatrix} \text{kN/m}$$

Equation (2.35) can be rearranged in a variety of ways. For example, it can be written

$$\mathbf{Mu} = \lambda\mathbf{Ku} \quad \text{where} \quad \lambda = \frac{1}{\omega^2} \tag{2.36}$$

This is an algebraic eigenvalue problem and solving it determines values for **u** and $\lambda$. MATLAB provides a function `eig` to solve the eigenvalue problem. To illustrate its use we apply it to the solution of (2.35).

```
>> M = [10 0 0;0 20 0;0 0 30];
>> K = 1000*[45 -20 -15;-20 45 -25;-15 -25 40];
>> lambda = eig(M,K).'

lambda =
    0.0002    0.0004    0.0073

>> omega = sqrt(1./lambda)

omega =
   72.2165   52.2551   11.7268
```

This result tells us that the system of Figure 2.8 vibrates with natural frequencies 11.72, 52.25, and 72.21 rad/s. In this example we have chosen not to determine `u`. We will discuss further the use of the function `eig` in Section 2.17.

Having provided an example of an eigenvalue problem, we consider the standard form of this problem:

$$\mathbf{Ax} = \lambda\mathbf{x} \tag{2.37}$$

This equation is an algebraic eigenvalue problem where **A** is a given $n \times n$ matrix of coefficients, **x** is an unknown column vector of $n$ elements, and $\lambda$ is an unknown scalar. Equation (2.37) can alternatively be written as

$$(\mathbf{A} - \lambda\mathbf{I})\mathbf{x} = \mathbf{0} \tag{2.38}$$

Our aim is to discover the values of **x**, called the characteristic vectors or eigenvectors, and the corresponding values of $\lambda$, called the characteristic values or eigenvalues. The values of $\lambda$ that satisfy (2.38) are given by the roots of the equation

$$|\mathbf{A} - \lambda\mathbf{I}| = 0 \tag{2.39}$$

These values of $\lambda$ are such that $(\mathbf{A}-\lambda\mathbf{I})$ is singular. Since (2.38) is a homogeneous equation, nontrivial solutions exist for these values of $\lambda$. Evaluation of the determinant (2.39) leads to an $n$th-degree polynomial in $\lambda$, which is called the characteristic polynomial. This characteristic polynomial has $n$ roots, some of which may be repeated, giving the $n$ values of $\lambda$. In MATLAB we can create the coefficients of the characteristic polynomial using the function `poly`, and the roots of the resulting characteristic polynomial can be found using the function `roots`. For example, if

$$\mathbf{A}=\begin{bmatrix}1 & 2 & 3\\ 4 & 5 & -6\\ 7 & -8 & 9\end{bmatrix}$$

then we have

```
>> A = [1 2 3;4 5 -6;7 -8 9];
>> p = poly(A)

p =
    1.0000  -15.0000  -18.0000  360.0000
```

Hence the characteristic equation is $\lambda^3-15\lambda^2-18\lambda+360=0$. To find the roots of this we use the statement

```
>> roots(p).'

ans =
   14.5343   -4.7494    5.2152
```

We can verify this result using the function `eig`:

```
>> eig(A).'

ans =
   -4.7494    5.2152   14.5343
```

Having obtained the eigenvalues we can substitute back into (2.38) to obtain linear equations for the characteristic vectors:

$$(\mathbf{A}-\lambda_i\mathbf{I})\mathbf{x}=\mathbf{0}\quad i=1,2,\ldots,n \tag{2.40}$$

This homogeneous system provides $n$ nontrivial solutions for $\mathbf{x}$. However, the use of (2.39) and (2.40) is not a practical means of solving eigenvalue problems.

We now consider the properties of eigensolutions where the system matrix is real. If $\mathbf{A}$ is a real symmetric matrix the eigenvalues of $\mathbf{A}$ are real, but not necessarily positive, and the corresponding eigenvectors are also real. In addition, if $\lambda_i$, $\mathbf{x}_i$ and $\lambda_j$, $\mathbf{x}_j$ satisfy the eigenvalue problem (2.37) and $\lambda_i$ and $\lambda_j$ are distinct, then

$$\mathbf{x}_i^\top \mathbf{x}_j = 0 \quad i \neq j \tag{2.41}$$

and

$$\mathbf{x}_i^\top \mathbf{A}\mathbf{x}_j = 0 \quad i \neq j \tag{2.42}$$

Equations (2.41) and (2.42) are called the orthogonality relationships. Note that if $i = j$, then in general $\mathbf{x}_i^\top \mathbf{x}_i$ and $\mathbf{x}_i^\top \mathbf{A}\mathbf{x}_i$ are not zero. The vector $\mathbf{x}_i$ includes an arbitrary scalar multiplier because the vector multiplies both sides of (2.37). Hence the product $\mathbf{x}_i^\top \mathbf{x}_i$ must be arbitrary. However, if the arbitrary scalar multiplier is adjusted so that

$$\mathbf{x}_i^\top \mathbf{x}_i = 1 \tag{2.43}$$

then

$$\mathbf{x}_i^\top \mathbf{A}\mathbf{x}_i = \lambda_i \tag{2.44}$$

and the eigenvectors are then said to be *normalized*. Sometimes the eigenvalues are not distinct and the eigenvectors associated with these equal or repeated eigenvalues are not necessarily orthogonal. If $\lambda_i = \lambda_j$ and the other eigenvalues, $\lambda_k$, are distinct, then

$$\left.\begin{aligned} \mathbf{x}_i^\top \mathbf{x}_k = 0 \\ \mathbf{x}_j^\top \mathbf{x}_k = 0 \end{aligned}\right\} \quad k = 1, 2, \ldots, n, \quad k \neq i, \quad k \neq j \tag{2.45}$$

For consistency we can choose to make $\mathbf{x}_i^\top \mathbf{x}_j = 0$. When $\lambda_i = \lambda_j$, the eigenvectors $\mathbf{x}_i$ and $\mathbf{x}_j$ are not unique and a linear combination of them (i.e., $\alpha\mathbf{x}_i + \gamma\mathbf{x}_j$, where $\alpha$ and $\gamma$ are arbitrary constants), also satisfies the eigenvalue problem.

Let us now consider the case where $\mathbf{A}$ is real but not symmetric. A pair of related eigenvalue problems can arise as follows:

$$\mathbf{A}\mathbf{x} = \lambda\mathbf{x} \tag{2.46}$$

$$\mathbf{A}^\top \mathbf{y} = \beta\mathbf{y} \tag{2.47}$$

and (2.47) can be transposed to give

$$\mathbf{y}^\top \mathbf{A} = \beta\mathbf{y}^\top \tag{2.48}$$

The vectors $\mathbf{x}$ and $\mathbf{y}$ are called the right and left vectors of $\mathbf{A}$, respectively. The equations $|\mathbf{A} - \lambda\mathbf{I}| = 0$ and $|\mathbf{A}^\top - \beta\mathbf{I}| = 0$ must have the same solutions for $\lambda$ and $\beta$ because the determinant of a matrix and the determinant of its transpose are equal. Thus the eigenvalues of $\mathbf{A}$ and $\mathbf{A}^\top$ are identical but the eigenvectors $\mathbf{x}$ and $\mathbf{y}$ will, in general, differ from each other.

The eigenvalues and eigenvectors of a nonsymmetric real matrix are either real or pairs of complex conjugates. If $\lambda_i$, $\mathbf{x}_i$, $\mathbf{y}_i$ and $\lambda_j$, $\mathbf{x}_j$, $\mathbf{y}_j$ are solutions that satisfy the eigenvalue problems of (2.46) and (2.47) and $\lambda_i$ and $\lambda_j$ are distinct, then

$$\mathbf{x}_i^\top \mathbf{x}_j = 0 \quad i \neq j \tag{2.49}$$

and

$$\mathbf{x}_i^\top \mathbf{A}\mathbf{x}_j = 0 \quad i \neq j \tag{2.50}$$

Equations (2.49) and (2.50) are called the biorthogonal relationships. As with (2.43) and (2.44) if, in these equations, $i = j$, then in general $\mathbf{y}_i^\top \mathbf{x}_i$ and $\mathbf{y}_i^\top \mathbf{A}\mathbf{x}_i$ are not zero. The eigenvectors $\mathbf{x}_i$ and $\mathbf{y}_i$ include arbitrary scaling factors and so the product of these vectors will also be arbitrary. However, if the vectors are adjusted so that

$$\mathbf{y}_i^\top \mathbf{x}_i = 1 \tag{2.51}$$

then

$$\mathbf{y}_i^\top \mathbf{A}\mathbf{x}_i = \lambda_i \tag{2.52}$$

We cannot, in these circumstances, describe either $\mathbf{x}_i$ or $\mathbf{y}_i$ as normalized. The vectors still include an arbitrary scale factor; only their product is uniquely chosen.

## 2.16 Iterative Methods for Solving the Eigenvalue Problem

The first of two simple iterative procedures described here determines the dominant or largest eigenvalue. The method, which is called the power method or matrix iteration, can be used on both symmetric and nonsymmetric matrices. However, for a nonsymmetric matrix the user must be alert to the possibility that there is not a single real dominant eigenvalue value but a complex conjugate pair. Under these conditions simple iteration does not converge.

Consider the eigenvalue problem defined by (2.37) and let the vector $\mathbf{u}_0$ be an initial trial solution. The vector $\mathbf{u}_0$ is an unknown linear combination of all the eigenvectors of the system provided they are linearly independent. Thus

$$\mathbf{u}_0 = \sum_{i=1}^{n} \alpha_i \mathbf{x}_i \tag{2.53}$$

where $\alpha_i$ are unknown coefficients and $\mathbf{x}_i$ are the unknown eigenvectors. Let the iterative scheme be

$$\mathbf{u}_1 = \mathbf{A}\mathbf{u}_0,\ \mathbf{u}_2 = \mathbf{A}\mathbf{u}_1, \ldots, \mathbf{u}_p = \mathbf{A}\mathbf{u}_{p-1} \tag{2.54}$$

Substituting (2.53) into the sequence (2.54) we have

$$\begin{aligned}
\mathbf{u}_1 &= \sum_{i=1}^{n} \alpha_i \mathbf{A}\mathbf{x}_i = \sum_{i=1}^{n} \alpha_i \lambda_i \mathbf{x}_i \quad \text{since } \mathbf{A}\mathbf{x}_i = \lambda_i \mathbf{x}_i \\
\mathbf{u}_2 &= \sum_{i=1}^{n} \alpha_i \lambda_i \mathbf{A}\mathbf{x}_i = \sum_{i=1}^{n} \alpha_i \lambda_i^2 \mathbf{x}_i \\
&\cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots \\
\mathbf{u}_p &= \sum_{i=1}^{n} \alpha_i \lambda_i^{p-1} \mathbf{A}\mathbf{x}_i = \sum_{i=1}^{n} \alpha_i \lambda_i^{p} \mathbf{x}_i
\end{aligned} \tag{2.55}$$

The final equation can be rearranged as follows:

$$\mathbf{u}_p = \lambda_1^p \left[ \alpha_1 \mathbf{x}_1 + \sum_{i=2}^{n} \alpha_i \left( \frac{\lambda_i}{\lambda_1} \right)^p \mathbf{x}_i \right] \tag{2.56}$$

It is the accepted convention that the $n$ eigenvalues of a matrix are numbered such that

$$|\lambda_1| > |\lambda_2| > \cdots > |\lambda_n|$$

Hence

$$\left[ \frac{\lambda_i}{\lambda_1} \right]^p$$

tends to zero as $p$ tends to infinity for $i = 2, 3, \ldots, n$. As $p$ becomes large, we have from (2.56):

$$\mathbf{u}_p \Rightarrow \lambda_1^p \alpha_1 \mathbf{x}_1$$

Thus $\mathbf{u}_p$ becomes proportional to $\mathbf{x}_1$ and the ratio between corresponding components of $\mathbf{u}_p$ and $\mathbf{u}_{p-1}$ tends to $\lambda_1$.

The algorithm is not usually implemented exactly as described previously because problems could arise due to numeric overflows. Usually, after each iteration, the resulting trial vector is normalized by dividing it by its largest element, thereby reducing the largest element in the vector to unity. This can be expressed mathematically as

$$\left.\begin{aligned}
\mathbf{v}_p &= \mathbf{A}\mathbf{u}_p \\
\mathbf{u}_{p+1} &= \left( \frac{1}{\max(\mathbf{v}_p)} \right) \mathbf{v}_p
\end{aligned}\right\} \quad p = 0, 1, 2, \ldots \tag{2.57}$$

where $\max(\mathbf{v}_p)$ is the element of $\mathbf{v}_p$ with the maximum modulus. The pair of equations (2.57) are iterated until convergence is achieved. This modification to the algorithm does not affect the rate of convergence of the iteration. In addition to preventing the buildup of very large numbers, the modification described before has the added advantage that it is now much easier to decide at what stage the iteration should be terminated.

Post-multiplying the coefficient matrix **A** by one of its eigenvectors gives the eigenvector multiplied by the corresponding eigenvalue. Thus, when we stop the iteration because $\mathbf{u}_{p+1}$ is sufficiently close to $\mathbf{u}_p$ to ensure convergence, max($\mathbf{v}_p$) will be an estimate of the eigenvalue.

The rate of convergence of the iteration is primarily dependent on the distribution of the eigenvalues; the smaller the ratios $|\lambda_i/\lambda_1|$, where $i = 2, 3, \ldots, n$, the faster the convergence. The following MATLAB function `eigit` implements the iterative method to find the dominant eigenvalue and the associated eigenvector.

```
function [lam u iter] = eigit(A,tol)
% Solves EVP to determine dominant eigenvalue and associated vector
% Sample call: [lam u iter] = eigit(A,tol)
% A is a square matrix, tol is the accuracy
% lam is the dominant eigenvalue, u is the associated vector
% iter is the number of iterations required
[n n] = size(A);
err = 100*tol;
u0 = ones(n,1);  iter = 0;
while err>tol
    v = A*u0;
    u1 = (1/max(v))*v;
    err = max(abs(u1-u0));
    u0 = u1;  iter = iter+1;
end
u = u0;  lam = max(v);
```

We now apply this method to find the dominant eigenvalue and corresponding vector for the following eigenvalue problem.

$$\begin{bmatrix} 1 & 2 & 3 \\ 2 & 5 & -6 \\ 3 & -6 & 9 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \lambda \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} \tag{2.58}$$

```
>> A = [1 2 3;2 5 -6;3 -6 9];
>> [lam u iterations] = eigit(A,1e-8)

lam =
   13.4627


u =
    0.1319
   -0.6778
    1.0000


iterations =
    18
```

The dominant eigenvalue, to eight decimal places, is 13.46269899.

Iteration can also be used to determine the smallest eigenvalue of a system. The eigenvalue problem $\mathbf{Ax} = \lambda\mathbf{x}$ can be rearranged to give

$$\mathbf{A}^{-1}\mathbf{x} = (1/\lambda)\,\mathbf{x}$$

Here iteration will converge to the largest value of $1/\lambda$—that is, the smallest value of $\lambda$. However, as a general rule, matrix inversion should be avoided, particularly in large systems.

We have seen that direct iteration of $\mathbf{Ax} = \lambda\mathbf{x}$ leads to the largest or dominant eigenvalue. A second iterative procedure, called inverse iteration, provides a powerful method of determining subdominant eigensolutions. Consider again the eigenvalue problem of (2.37). Subtracting $\mu\mathbf{x}$ from both sides of this equation we have

$$(\mathbf{A} - \mu\mathbf{I})\,\mathbf{x} = (\lambda - \mu)\,\mathbf{x} \tag{2.59}$$

$$(\mathbf{A} - \mu\mathbf{I})^{-1}\,\mathbf{x} = \left(\frac{1}{\lambda - \mu}\right)\mathbf{x} \tag{2.60}$$

Consider the iterative scheme that begins with a trial vector $\mathbf{u}_0$. Then, using the equivalent of (2.57), we have

$$\left.\begin{aligned} \mathbf{v}_s &= (\mathbf{A} - \mu\mathbf{I})^{-1}\,\mathbf{u}_s \\ \mathbf{u}_{s+1} &= \left(\frac{1}{\max(\mathbf{v}_s)}\right)\mathbf{v}_s \end{aligned}\right\}\quad s = 0, 1, 2, \ldots \tag{2.61}$$

Iteration will lead to the largest value of $1/(\lambda - \mu)$—that is, the smallest value of $(\lambda - \mu)$. The smallest value of $(\lambda - \mu)$ implies that the value of $\lambda$ will be the eigenvalue closest to $\mu$ and $\mathbf{u}$ will have converged to the eigenvector $\mathbf{x}$ corresponding to this particular eigenvalue. Thus, by a suitable choice of $\mu$, we have a procedure for finding subdominant eigensolutions.

Iteration is terminated when $\mathbf{u}_{s+1}$ is sufficiently close to $\mathbf{u}_s$. When convergence is complete

$$\frac{1}{\lambda - \mu} = \max(\mathbf{v}_s)$$

Thus the value of $\lambda$ nearest to $\mu$ is given by

$$\lambda = \mu + \frac{1}{\max(\mathbf{v}_s)} \tag{2.62}$$

The rate of convergence is fast, provided the chosen value of $\mu$ is close to an eigenvalue. If $\mu$ is equal to an eigenvalue, then $(\mathbf{A} - \mu\mathbf{I})$ is singular. In practice this seldom presents difficulties because it is unlikely that $\mu$ would be chosen, by chance, to exactly equal an eigenvalue. However, if $(\mathbf{A} - \mu\mathbf{I})$ is singular then we have confirmation that the eigenvalue is known to a very high precision. The corresponding eigenvector can then be obtained by changing $\mu$ by a small quantity and iterating to determine the eigenvector.

Although inverse iteration can be used to find the eigensolutions of a system about which we have no previous knowledge, it is more usual to use inverse iteration to refine the approximate eigensolution obtained by some other technique. In practice $(\mathbf{A}-\mu\mathbf{I})^{-1}$ is not formed explicitly; instead, $(\mathbf{A}-\mu\mathbf{I})$ is usually decomposed into the product of a lower and an upper triangular matrix. Explicit matrix inversion is avoided and is replaced by two efficient substitution procedures. In the simple MATLAB implementation of this procedure shown next, the operator \ is used to avoid matrix inversion.

```
function [lam u iter] = eiginv(A,mu,tol)
% Determines eigenvalue of A closest to mu with a tolerance tol.
% Sample call: [lam u] = eiginv(A,mu,tol)
% lam is the eigenvalue and u the corresponding eigenvector.
[n,n] = size(A);
err = 100*tol;
B = A-mu*eye(n,n);
u0 = ones(n,1);
iter = 0;
while err>tol
    v = B\u0; f = 1/max(v);
    u1 = f*v;
    err = max(abs(u1-u0));
    u0 = u1; iter = iter+1;
end
u = u0; lam = mu+f;
```

We now apply this function to find the eigenvalue of (2.58) nearest to 4 and the corresponding eigenvector.

```
>> A = [1 2 3;2 5 -6;3 -6 9];
>> [lam u iterations] = eiginv(A,4,1e-8)

lam =
    4.1283

u =
    1.0000
    0.8737
    0.4603

iterations =
     6
```

The eigenvalue closest to 4 is 4.12827017 to eight decimal places. The functions `eigit` and `eiginv` should be used with care when solving large-scale eigenvalue problems since convergence is not always guaranteed and in adverse circumstances may be slow.

We now discuss the MATLAB function `eig` in some detail.

## 2.17 The MATLAB Function **eig**

There are many algorithms available to solve the eigenvalue problem. The method chosen is influenced by many factors such as the form and size of the eigenvalue problem, whether or not it is symmetric, whether it is real or complex, whether or not only the eigenvalues are required, and whether all or only some of the eigenvalues and vectors are required.

We now describe the algorithms that are used in the MATLAB function `eig`. This MATLAB function can be used in several forms and, in the process, makes use of different algorithms. The different forms are as follows:

1. `lambda = eig(a)`
2. `[u lambda] = eig(a)`
3. `lambda = eig(a,b)`
4. `[u lambda]=eig(a,b)`

where `lambda` is a vector of eigenvalues in (1) and (3) and a diagonal matrix with the eigenvalues on the diagonal in (2) and (4). In these latter cases `u` is a matrix, the columns of which are the eigenvectors.

For real matrices the MATLAB function `eig(a)` proceeds as follows. If **A** is a general matrix, it is first reduced to Hessenberg form using Householder's transformation method. A Hessenberg matrix has zeros everywhere below the diagonal except for the first subdiagonal. If **A** is a symmetric matrix, the transform creates a tridiagonal matrix. Then the eigenvalues and eigenvectors of the real upper Hessenberg matrix are found by the iterative application of the QR procedure. The QR procedure involves decomposing the Hessenberg matrix into an upper triangular and a unitary matrix. The method is as follows:

1. $k = 0$.
2. Decompose $\mathbf{H}_k$ into $\mathbf{Q}_k$ and $\mathbf{R}_k$ such that $\mathbf{H}_k = \mathbf{Q}_k\mathbf{R}_k$ where $\mathbf{H}_k$ is a Hessenberg or tridiagonal matrix.
3. Compute $\mathbf{H}_{k+1} = \mathbf{R}_k\mathbf{Q}_k$. The estimates of the eigenvalues equal diag($\mathbf{H}_{k+1}$).
4. Check the accuracy of the eigenvalues. If the process has not converged, $k = k + 1$; repeat from (2).

The values on the leading diagonal of $\mathbf{H}_k$ tend to the eigenvalues. The following script uses the MATLAB function `hess` to convert the original matrix to the Hessenberg form, followed by the iterative application of the `qr` function to determine the eigenvalues of a symmetric matrix. Note that in this script we have iterated 10 times rather than use a formal test for convergence since the purpose of the script is merely to illustrate the functioning of the iterative application of the QR procedure.

```
% e3s213.m
A = [5 4 1 1;4 5 1 1; 1 1 4 2;1 1 2 4];
H1 = hess(A);
for i = 1:10
    [Q R] = qr(H1);
    H2 = R*Q;  H1 = H2;
    p = diag(H1)';
    fprintf('%2.0f %8.4f %8.4f',i,p(1),p(2))
    fprintf('%8.4f %8.4f\n',p(3),p(4))
end
```

Running this script gives

```
 1   1.0000   8.3636  6.2420   2.3944
 2   1.0000   9.4940  5.4433   2.0627
 3   1.0000   9.8646  5.1255   2.0099
 4   1.0000   9.9655  5.0329   2.0016
 5   1.0000   9.9913  5.0084   2.0003
 6   1.0000   9.9978  5.0021   2.0000
 7   1.0000   9.9995  5.0005   2.0000
 8   1.0000   9.9999  5.0001   2.0000
 9   1.0000  10.0000  5.0000   2.0000
10   1.0000  10.0000  5.0000   2.0000
```

The iteration converges to the values 1, 10, 5, and 2, which are the correct values. This QR iteration could be applied directly to the full matrix **A** but in general it would be inefficient. We have not given details of how the eigenvectors are computed.

When there are two real or complex arguments in the MATLAB function `eig`, the QZ algorithm is used instead of the QR algorithm. The QZ algorithm (Golub and Van Loan, 1989) has been modified to deal with the complex case. When `eig` is called using a single complex matrix `A` then the algorithm works by applying the QZ algorithm to `eig(A,eye(size(A)))`. The QZ algorithm begins by noting that there exists a unitary **Q** and **Z** such that $\mathbf{Q}^{\mathrm{H}}\mathbf{AZ} = \mathbf{T}$ and $\mathbf{Q}^{\mathrm{H}}\mathbf{BZ} = \mathbf{S}$ are both upper triangular. This is called generalized Schur decomposition. Providing $s_{kk}$ is not zero then the eigenvalues are computed from the ratio $t_{kk}/s_{kk}$, where $k = 1, 2, \ldots, n$. The following script demonstrates that the ratios of the diagonal elements of the **T** and **S** matrices give the required eigenvalues.

```
% e3s214.m
A = [10+2i 1 2;1-3i 2 -1;1 1 2];
b = [1 2-2i -2;4 5 6;7+3i 9 9];
[T S Q Z V] = qz(A,b);
r1 = diag(T)./diag(S)
r2 = eig(A,b)
```

Running this script gives

```
r1 =
   1.6154 + 2.7252i
  -0.4882 - 1.3680i
   0.1518 + 0.0193i

r2 =
   1.6154 + 2.7252i
  -0.4882 - 1.3680i
   0.1518 + 0.0193i
```

Schur decomposition is closely related to the eigenvalue problem. The MATLAB function `schur(a)` produces an upper triangular matrix **T** with real eigenvalues on its diagonal and complex eigenvalues in $2 \times 2$ blocks on the diagonal. Thus **A** can be written

$$\mathbf{A} = \mathbf{U}\mathbf{T}\mathbf{U}^{\mathrm{H}}$$

where **U** is a unitary matrix such that $\mathbf{U}^{\mathrm{H}}\mathbf{U} = \mathbf{I}$. The following script shows the similarity between Schur decomposition and the eigenvalues of a given matrix.

```
% e3s215.m
A = [4 -5 0 3;0 4 -3 -5;5 -3 4 0;3 0 5 4];
T = schur(A), lam = eig(A)
```

Running this script gives

```
T =
   12.0000    0.0000   -0.0000   -0.0000
         0    1.0000   -5.0000   -0.0000
         0    5.0000    1.0000   -0.0000
         0         0         0    2.0000

lam =
  12.0000
   1.0000 + 5.0000i
   1.0000 - 5.0000i
   2.0000
```

We can readily identify the four eigenvalues in the matrix `T`. The following script compares the performance of the `eig` function when solving various classes of problem.

```
% e3s216.m
disp('      real1    realsym1     real2    realsym2     comp1      comp2')
for n = 100:50:500
    A = rand(n); C = rand(n);
    S = A+C*i;
    T = rand(n)+i*rand(n);
    tic, [U,V] = eig(A); f1 = toc;
    B = A+A.'; D = C+C.';
    tic, [U,V] = eig(B); f2 = toc;
    tic, [U,V] = eig(A,C); f3 = toc;
    tic, [U,V] = eig(B,D); f4 = toc;
    tic, [U,V] = eig(S); f5 = toc;
    tic, [U,V] = eig(S,T); f6 = toc;
    fprintf('%12.3f %10.3f %10.3f %10.3f %10.3f %10.3f\n',f1,f2,f3,f4,f5,f6);
end
```

This script gives the time taken (in seconds) to carry out the various operations. The output is as follows:

```
real1    realsym1     real2    realsym2     comp1      comp2
0.042      0.009      0.063      0.061      0.039      0.037
0.067      0.014      0.086      0.090      0.067      0.106
0.129      0.028      0.228      0.184      0.116      0.200
0.182      0.046      0.430      0.425      0.186      0.432
0.270      0.073      0.729      0.724      0.279      0.782
0.371      0.104      1.277      1.257      0.373      1.232
0.514      0.154      2.006      2.103      0.538      2.104
0.708      0.205      3.055      3.097      0.698      2.919
0.946      0.278      4.403      4.187      0.901      4.344
```

In some circumstances not all the eigenvalues and eigenvectors are required. For example, in a complex engineering structure, modeled with many hundreds of degrees of freedom, we may only require the first 15 eigenvalues, giving the natural frequencies of the model, and the corresponding eigenvectors. MATLAB provides the function `eigs`, which finds a small number of eigenvalues, such as those with the largest amplitude, the largest or smallest real or imaginary part, and so on. This function is particularly useful when seeking a small number of eigenvalues of very large sparse matrices. Eigenvalue reduction algorithms are used to reduce the size of eigenvalue problem (for example, Guyan, 1965) but still allow selected eigenvalues to be computed to an acceptable level of accuracy.

MATLAB also includes the facility to find the eigenvalues of a sparse matrix. The following script compares the number of floating-point operations required to find the eigenvalues of a matrix treated as sparse with the corresponding number required to find the eigenvalues of the corresponding full matrix.

```
% e3s217.m
% generate a sparse triple diagonal matrix
n = 2000;
rowpos = 2:n; colpos = 1:n-1;
values = ones(1,n-1);
offdiag = sparse(rowpos,colpos,values,n,n);
A = sparse(1:n,1:n,4*ones(1,n),n,n);
A = A+offdiag+offdiag.';
% generate full matrix
B = full(A);
tic, eig(A); sptim = toc;
tic, eig(B); futim = toc;
fprintf('Time for sparse eigen solve = %8.6f\n',sptim)
fprintf('Time for  full  eigen solve = %8.6f\n',futim)
```

The results from running this script are as follows:

```
Time for sparse eigen solve = 0.349619
Time for  full  eigen solve = 3.000229
```

Clearly there is a significant savings in time.

## 2.18 Summary

We have described many of the important algorithms related to computational matrix algebra and shown how the power of MATLAB can be used to illustrate the application of these algorithms in a revealing way. We have shown how to solve over- and underdetermined systems and eigenvalue problems. We have drawn the attention of the reader to the importance of sparsity in linear systems and demonstrated its significance. The scripts provided should help readers to develop their own applications.

In Chapter 9 we show how the symbolic toolbox can be usefully applied to solve some problems in linear algebra.

## Problems

**2.1.** An $n \times n$ Hilbert matrix, $\mathbf{A}$, is defined by

$$a_{ij} = 1/(i+j-1) \quad \text{for} \quad i,j = 1,2,\ldots,\ n$$

Find the inverse of $\mathbf{A}$ and the inverse of $\mathbf{A}^\top\mathbf{A}$ for $n = 5$. Then, noting that

$$(\mathbf{A}^\top\mathbf{A})^{-1} = \mathbf{A}^{-1}(\mathbf{A}^{-1})^\top$$

find the inverse of $\mathbf{A}^\top\mathbf{A}$ using this result for $n = 3,\ 4,\ 5,$ and 6. Compare the accuracy of the two results by using the inverse Hilbert function `invhilb` to find the exact inverse using $(\mathbf{A}^\top\mathbf{A})^{-1} = \mathbf{A}^{-1}(\mathbf{A}^{-1})^\top$. *Hint*: Compute norm($\mathbf{P} - \mathbf{R}$) and norm($\mathbf{Q} - \mathbf{R}$) where $\mathbf{P} = (\mathbf{A}^\top\mathbf{A})^{-1}$ and $\mathbf{Q} = \mathbf{A}^{-1}(\mathbf{A}^{-1})^\top$ and $\mathbf{R}$ is the exact `value` of $\mathbf{Q}$ obtained by using the `invhilb` function. .

**2.2.** Find the condition number of $\mathbf{A}^\top\mathbf{A}$ where $\mathbf{A}$ is an $n \times n$ Hilbert matrix, defined in Problem 2.1, for $n = 3,\ 4,\ldots,6$. How do these results relate to the results of Problem 2.1?

**2.3.** It can be proved that the series $(\mathbf{I} - \mathbf{A})^{-1} = \mathbf{I} + \mathbf{A} + \mathbf{A}^2 + \mathbf{A}^3 + \cdots$, where $\mathbf{A}$ is an $n \times n$ matrix, converges if the eigenvalues of $\mathbf{A}$ are all less than unity. The following $n \times n$ matrix satisfies this condition if $a + 2b < 1$ and $a$ and $b$ are positive:

$$\begin{bmatrix} a & b & 0 & \ldots & 0 & 0 & 0 \\ b & a & b & \ldots & 0 & 0 & 0 \\ \vdots & \vdots & \vdots & & \vdots & \vdots & \vdots \\ 0 & 0 & 0 & \ldots & b & a & b \\ 0 & 0 & 0 & \ldots & 0 & b & a \end{bmatrix}$$

Experiment with this matrix for various values of $n$, $a$, and $b$ to illustrate that the series converges under the condition stated.

**2.4.** Use the function `eig` to find the eigenvalues of the following matrix:

$$\begin{bmatrix} 2 & 3 & 6 \\ 2 & 3 & -4 \\ 6 & 11 & 4 \end{bmatrix}$$

Then use the `rref` function on the matrix $(\mathbf{A} - \lambda\mathbf{I})$, taking $\lambda$ equal to any of the eigenvalues. Solve the resulting equations by hand to obtain the eigenvector of the matrix. *Hint*: Note that an eigenvector is the solution of $(\mathbf{A} - \lambda\mathbf{I})\mathbf{x} = \mathbf{0}$ for $\lambda$ equal to a specific eigenvalue. Assume an arbitrary value for $x_3$.

**2.5.** For the system given in Problem 2.3, find the eigenvalues, assuming both full and sparse forms with $n = 10:10:30$. Compare your results with the exact solution given by

$$\lambda_k = a + 2b\cos\left\{k\pi/(n+1)\right\}, \ k = 1, 2, \ldots$$

**2.6.** Find the solution of the overdetermined system that follows using `pinv`, `qr`, and the \ operator.

$$\begin{bmatrix} 2.0 & -3.0 & 2.0 \\ 1.9 & -3.0 & 2.2 \\ 2.1 & -2.9 & 2.0 \\ 6.1 & 2.1 & -3.0 \\ -3.0 & 5.0 & 2.1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 1.01 \\ 1.01 \\ 0.98 \\ 4.94 \\ 4.10 \end{bmatrix}$$

**2.7.** Write a script to generate $\mathbf{E} = \{1/(n+1)\}\mathbf{C}$ where

$$\begin{aligned} c_{ij} &= i(n-i+1) && \text{if } i = j \\ &= c_{i,j-1} - i && \text{if } j > i \\ &= c_{ji} && \text{if } j < i \end{aligned}$$

Having generated $\mathbf{E}$ for $n = 5$, solve $\mathbf{Ex} = \mathbf{b}$ where $\mathbf{b} = [1:n]^\top$ by

**(a)** Using the \ operator

**(b)** Using the `lu` function and solving $\mathbf{Ux} = \mathbf{y}$ and $\mathbf{Ly} = \mathbf{b}$

**2.8.** Determine the inverse of $\mathbf{E}$ of Problem 2.7 for $n = 20$ and 50. Compare with the exact inverse, which is a matrix with 2 along the main diagonal and $-1$ along the upper and lower subdiagonals and zero elsewhere.

**2.9.** Determine the eigenvalues of $\mathbf{E}$ defined in Problem 2.7 for $n = 20$ and 50. The exact eigenvalues for this system are given by $\lambda_k = 1/[2 - 2\cos\{k\pi/(n+1)\}]$ where $k = 1, \ldots, n$.

**2.10.** Determine the condition number of $\mathbf{E}$ of Problem 2.7, using the MATLAB function `cond`, for $n = 20$ and 50. Compare your results with the theoretical expression for the condition number, which is $4n^2/\pi^2$.

**2.11.** Find the eigenvalues and the left and right eigenvectors using the MATLAB function `eig` for the matrix

$$\mathbf{A} = \begin{bmatrix} 8 & -1 & -5 \\ -4 & 4 & -2 \\ 18 & -5 & -7 \end{bmatrix}$$

**2.12.** For the following matrix $\mathbf{A}$, using `eigit`, `eiginv`, determine

**(a)** The largest eigenvalue

**(b)** The eigenvalue nearest 100

**(c)** The smallest eigenvalue

$$\mathbf{A} = \begin{bmatrix} 122 & 41 & 40 & 26 & 25 \\ 40 & 170 & 25 & 14 & 24 \\ 27 & 26 & 172 & 7 & 3 \\ 32 & 22 & 9 & 106 & 6 \\ 31 & 28 & -2 & -1 & 165 \end{bmatrix}$$

**2.13.** Given that

$$\mathbf{A} = \begin{bmatrix} 1 & 2 & 2 \\ 5 & 6 & -2 \\ 1 & -1 & 0 \end{bmatrix} \quad \text{and} \quad \mathbf{B} = \begin{bmatrix} 2 & 0 & 1 \\ 4 & -5 & 1 \\ 1 & 0 & 0 \end{bmatrix}$$

and defining **C** by

$$\mathbf{C} = \begin{bmatrix} \mathbf{A} & \mathbf{B} \\ \mathbf{B} & \mathbf{A} \end{bmatrix}$$

verify using `eig` that the eigenvalues of **C** are given by a combination of the eigenvalues of $\mathbf{A}+\mathbf{B}$ and $\mathbf{A}-\mathbf{B}$.

**2.14.** Write a MATLAB script to generate the matrix

$$\mathbf{A} = \begin{bmatrix} n & n-1 & n-2 & \dots & 2 & 1 \\ n-1 & n-1 & n-2 & \dots & 2 & 1 \\ n-2 & n-2 & n-2 & \dots & 2 & 1 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ 2 & 2 & 2 & \dots & 2 & 1 \\ 1 & 1 & 1 & \dots & 1 & 1 \end{bmatrix}$$

The eigenvalues of this matrix are given by the formula

$$\lambda_i = \frac{1}{2}\left[1 - \cos\frac{(2i-1)\pi}{2n+1}\right]^{-1}, \quad i = 1, 2 \dots, n$$

Taking $n = 5$ and $n = 50$ and using the MATLAB function `eig`, find the largest and smallest eigenvalues. Verify your results are correct using the preceding formula.

**2.15.** Taking $n = 10$, find the eigenvalues of the matrix

$$\mathbf{A} = \begin{bmatrix} 1 & 0 & 0 & \dots & 0 & 1 \\ 0 & 1 & 0 & \dots & 0 & 2 \\ 0 & 0 & 1 & \dots & 0 & 3 \\ \vdots & \vdots & \vdots & \dots & \vdots & \vdots \\ 0 & 0 & 0 & \dots & 1 & n-1 \\ 1 & 2 & 3 & \dots & n-1 & n \end{bmatrix}$$

using `eig`. As an alternative, find the eigenvalues of **A** by first generating the characteristic polynomial using `poly` for the matrix **A** and then using `roots` to find the roots of the resulting polynomial. What conclusions do you draw from these results?

**2.16.** For the matrix given in Problem 2.12, use `eig` to find the eigenvalues. Then find the eigenvalues of **A** by first generating the characteristic polynomial for **A** using `poly` and then using `roots` to find the roots of the resulting polynomial. Use `sort` to compare the results of the two approaches. What conclusions do you draw from these results?

**2.17.** For the matrix given in Problem 2.14, taking $n = 10$, show that the trace is equal to the sum of the eigenvalues and the determinant is equal to the product of the eigenvalues. Use the MATLAB functions `det`, `trace`, and `eig`.

**2.18.** The matrix **A** is defined as follows:

$$\mathbf{A} = \begin{bmatrix} 2 & -1 & 0 & 0 & \dots & 0 \\ -1 & 2 & -1 & 0 & \dots & 0 \\ 0 & -1 & 2 & -1 & \dots & 0 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ 0 & 0 & \dots & -1 & 2 & -1 \\ 0 & 0 & \dots & 0 & -1 & 2 \end{bmatrix}$$

The condition number for this matrix takes the form $c = pn^q$ where $n$ is the size of the matrix, $c$ is the condition number, and $p$ and $q$ are constants. By computing the condition number for the matrix **A** for $n = 5:5:50$ using the MATLAB function `cond`, fit the function $pn^q$ to the set of results you produce. *Hint*: Take logs of both sides of the equation for $c$ and solve the system of overdetermined equations using the \ operator.

**2.19.** An approximation for the inverse of $(\mathbf{I} - \mathbf{A})$ where **I** is an $n \times n$ unit matrix and **A** is an $n \times n$ matrix is given by

$$(\mathbf{I} - \mathbf{A})^{-1} = \mathbf{I} + \mathbf{A} + \mathbf{A}^2 + \mathbf{A}^3 + \cdots$$

This series only converges and the approximation is only valid if the maximum eigenvalue of **A** is less than 1. Write a MATLAB function `invapprox(A,k)` that obtains an approximation to $(\mathbf{I}-\mathbf{A})^{-1}$ using $k$ terms of the given series. The function must find all eigenvalues of A using the MATLAB function `eig`. If the largest eigenvalue is greater than one then a message will be output indicating that the method fails. Otherwise, the function will compute an approximation to $(\mathbf{I}-\mathbf{A})^{-1}$ using $k$ terms of the series expansion given. Taking $k=4$, test the function on the matrices:

$$\begin{bmatrix} 0.2 & 0.3 & 0 \\ 0.3 & 0.2 & 0.3 \\ 0 & 0.3 & 0.2 \end{bmatrix} \quad \text{and} \quad \begin{bmatrix} 1.0 & 0.3 & 0 \\ 0.3 & 1.0 & 0.3 \\ 0 & 0.3 & 1.0 \end{bmatrix}$$

Use the `norm` function to compare the accuracy of the inverse of the matrix $(\mathbf{I}-\mathbf{A})$ found using the MATLAB `inv` function and the function `invapprox(A,k)` for $k=4$, 8, 16.

**2.20.** The system of equations $\mathbf{Ax}=\mathbf{b}$, where **A** is a matrix of $m$ rows and $n$ columns, **x** is an $n$ element column vector, and **b** is an $m$ element column vector, is said to be underdetermined if $n > m$. The direct use of the MATLAB `inv` function to solve this system fails since the matrix **A** is not square. However, multiplying both sides of the equation by $\mathbf{A}^\top$ gives

$$\mathbf{A}^\top\mathbf{Ax}=\mathbf{A}^\top\mathbf{b}$$

$\mathbf{A}^\top\mathbf{A}$ is a square matrix and the MATLAB `inv` function can now be used to solve the system. Write a MATLAB function to use this result to solve underdetermined systems. The function should allow the input of the **b** vector and the **A** matrix, form the necessary matrix products, and use the MATLAB `inv` function to solve the system. The accuracy of the solution should be checked using the MATLAB `norm` function to measure the difference between **Ax** and **b**. The function must also include the direct use of the MATLAB \ symbol to solve the same underdetermined linear system, again with a check on the accuracy of the solution that uses the MATLAB `norm` function to measure the difference between **Ax** and **b**. The function should take the form `udsys(A,b)` and return the solutions given by the different methods and the norms produced by the two methods. Test your program by using it to solve the underdetermined system of linear equations $\mathbf{Ax}=\mathbf{b}$ where

$$\mathbf{A}=\begin{bmatrix} 1 & -2 & -5 & 3 \\ 3 & 4 & 2 & -7 \end{bmatrix} \quad \text{and} \quad \mathbf{b}=\begin{bmatrix} -10 \\ 20 \end{bmatrix}$$

What conclusions do you draw regarding the two methods by comparing the norms that the two methods produce?

**2.21.** An orthogonal matrix **A** is defined as a square matrix such that the product of the matrix and its transpose equals the unit matrix or

$$\mathbf{A}\mathbf{A}^\top = \mathbf{I}$$

Use MATLAB to verify that the following matrices are orthogonal:

$$\mathbf{B} = \begin{bmatrix} \frac{1}{\sqrt{3}} & \frac{1}{\sqrt{6}} & -\frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{3}} & \frac{-2}{\sqrt{6}} & 0 \\ \frac{1}{\sqrt{3}} & \frac{1}{\sqrt{6}} & \frac{1}{\sqrt{2}} \end{bmatrix}$$

$$\mathbf{C} = \begin{bmatrix} \cos(\pi/3) & \sin(\pi/3) \\ -\sin(\pi/3) & \cos(\pi/3) \end{bmatrix}$$

**2.22.** Write MATLAB scripts to implement both the Gauss–Seidel and the Jacobi method and use them to solve, with an accuracy of 0.000005, the equation system $\mathbf{Ax} = \mathbf{b}$ where the elements of **A** are

$$\begin{aligned} &a_{ii} = -4 \\ &a_{ij} = 2 \text{ if } |i-j| = 1 \\ &a_{ij} = 0 \text{ if } |i-j| \geq 2 \quad \text{where} \quad i,j = 1,2,\ldots,10 \end{aligned}$$

and

$$\mathbf{b}^\top = [2\ 3\ 4\ \ldots\ 11]$$

Use initial values of $x_i = 0,\ i = 1,2,\ldots,10$. (You might also like to experiment with other initial values.) Check your results by solving the system using the MATLAB \ operator.