2

The Solution of Linear Systems

In many branches of engineering and science it is desirable to be able to mathematically determine the state of a system based on a set of physical relationships. These physical relationships may be determined from characteristics such as circuit topology, mass, weight, or force, to name a few. For example, the injected currents, network topology, and branch impedances govern the voltages at each node of a circuit. In many cases, the relationship between the known, or input, quantities and the unknown, or output, states is a linear relationship. Therefore, a linear system may be generically modeled as

$$Ax = b \tag{2.1}$$

where b is the $n \times 1$ vector of known quantities, x is the $n \times 1$ unknown state vector, and A is the $n \times n$ matrix that relates x to b. For the time being, it will be assumed that the matrix A is invertible, or non-singular; thus each vector b will yield a unique corresponding vector x. Thus the matrix A^{-1} exists and

$$x^* = A^{-1}b \tag{2.2}$$

is the unique solution to Equation (2.1).

The natural approach to solving Equation (2.1) is to directly calculate the inverse of A and multiply it by the vector b. One method to calculate A^{-1} is to use *Cramer's* rule:

$$A^{-1}(i,j) = \frac{1}{det(A)} \left(A_{ij}\right)^T \quad \text{for } i = 1, \ldots, n, j = 1, \ldots, n \tag{2.3}$$

where $A^{-1}(i,j)$ is the ijth entry of A^{-1} and A_{ij} is the cofactor of each entry a_{ij} of A. This method requires the calculation of $(n+1)$ determinants, which results in $2(n+1)!$ multiplications to find A^{-1}! For large values of n, the calculation requirement grows too rapidly for computational tractability; thus alternative approaches have been developed.

Basically, there are two approaches to solving Equation (2.1):

- *Direct methods*, or elimination methods, find the exact solution (within the accuracy of the computer) through a finite number of arithmetic operations. The solution x of a direct method would be completely accurate were it not for computer roundoff errors.

- *Iterative methods*, on the other hand, generate a sequence of (hopefully) progressively improving approximations to the solution based on the application of the same computational procedure at each step. The iteration is terminated when an approximate solution is obtained having some prespecified accuracy or when it is determined that the iterates are not improving.

The choice of solution methodology usually relies on the structure of the system under consideration. Certain systems lend themselves more amenably to one type of solution method versus the other. In general, direct methods are best for full matrices, whereas iterative methods are better for matrices that are large and sparse. But, as with most generalizations, there are notable exceptions to this rule of thumb.

2.1 Gaussian Elimination

An alternate method for solving Equation (2.1) is to solve for x without calculating A^{-1} explicitly. This approach is a *direct method* of linear system solution, since x is found directly. One common direct method is the method of *Gaussian elimination*. The basic idea behind Gaussian elimination is to use the first equation to eliminate the first unknown from the remaining equations. This process is repeated sequentially for the second unknown, the third unknown, etc., until the elimination process is completed. The nth unknown is then calculated directly from the input vector b. The unknowns are then recursively substituted back into the equations until all unknowns have been calculated.

Gaussian elimination is the process by which the augmented $n \times (n+1)$ matrix

$$[A \mid b]$$

is converted to the $n \times (n+1)$ matrix

$$[I \mid b^*]$$

through a series of elementary row operations, where

$$\begin{aligned} Ax &= b \\ A^{-1}Ax &= A^{-1}b \\ Ix &= A^{-1}b = b^* \\ x^* &= b^* \end{aligned}$$

Thus, if a series of elementary row operations exist that can transform the matrix A into the identity matrix I, then the application of the same set of

elementary row operations will also transform the vector b into the solution vector x^*.

An elementary row operation consists of one of three possible actions that can be applied to a matrix:

- interchange any two rows of the matrix
- multiply any row by a constant
- take a linear combination of rows and add it to another row

The elementary row operations are chosen to transform the matrix A into an upper triangular matrix that has ones on the diagonal and zeros in the subdiagonal positions. This process is known as the *forward elimination* step. Each step in the forward elimination can be obtained by successively premultiplying the matrix A by an elementary matrix ξ, where ξ is the matrix obtained by performing an elementary row operation on the identity matrix.

Example 2.1
Find a sequence of elementary matrices that, when applied to the following matrix, will produce an upper triangular matrix.

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

Solution 2.1 To upper triangularize the matrix, the elementary row operations will need to systematically zero out each column below the diagonal. This can be achieved by replacing each row of the matrix below the diagonal with the difference of the row itself and a constant times the diagonal row, where the constant is chosen to result in a zero sum in the column under the diagonal. Therefore, row 2 of A is replaced by (row 2 minus 2(row 1)) and the elementary matrix is

$$\xi_1 = \begin{bmatrix} 1 & 0 & 0 & 0 \\ -2 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

and

$$\xi_1 A = \begin{bmatrix} 1 & 3 & 4 & 8 \\ 0 & -5 & -6 & -13 \\ 4 & 3 & 5 & 8 \\ 9 & 2 & 7 & 4 \end{bmatrix}$$

Note that all rows except row 2 remain the same and row 2 now has a 0 in the column under the first diagonal. Similarly, the two elementary matrices that complete the elimination of the first column are

$$\xi_2 = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ -4 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$\xi_3 = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ -9 & 0 & 0 & 1 \end{bmatrix}$$

and

$$\xi_3\xi_2\xi_1 A = \begin{bmatrix} 1 & 3 & 4 & 8 \\ 0 & -5 & -6 & -13 \\ 0 & -9 & -11 & -24 \\ 0 & -25 & -29 & -68 \end{bmatrix} \tag{2.4}$$

The process is now applied to the second column to zero out everything below the second diagonal and scale the diagonal to one. Therefore,

$$\xi_4 = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & -\frac{9}{5} & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$\xi_5 = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & -\frac{25}{5} & 0 & 1 \end{bmatrix}$$

$$\xi_6 = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & -\frac{1}{5} & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Similarly,

$$\xi_6\xi_5\xi_4\xi_3\xi_2\xi_1 A = \begin{bmatrix} 1 & 3 & 4 & 8 \\ 0 & 1 & \frac{6}{5} & \frac{13}{5} \\ 0 & 0 & -\frac{1}{5} & -\frac{3}{5} \\ 0 & 0 & 1 & -3 \end{bmatrix} \tag{2.5}$$

Similarly,

$$\xi_7 = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 5 & 1 \end{bmatrix}$$

$$\xi_8 = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & -5 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

yielding

$$\xi_8\xi_7\xi_6\xi_5\xi_4\xi_3\xi_2\xi_1 A = \begin{bmatrix} 1 & 3 & 4 & 8 \\ 0 & 1 & \frac{6}{5} & \frac{13}{5} \\ 0 & 0 & 1 & 3 \\ 0 & 0 & 0 & -6 \end{bmatrix} \tag{2.6}$$

Finally

$$\xi_9 = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & -\frac{1}{6} \end{bmatrix}$$

and

$$\xi_9\xi_8\xi_7\xi_6\xi_5\xi_4\xi_3\xi_2\xi_1 A = \begin{bmatrix} 1 & 3 & 4 & 8 \\ 0 & 1 & \frac{6}{5} & \frac{13}{5} \\ 0 & 0 & 1 & 3 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{2.7}$$

which completes the upper triangularization process. ■

Once an upper triangular matrix has been achieved, the solution vector x^* can be found by successive substitution (or *back substitution*) of the states.

Example 2.2

Using the upper triangular matrix of Example 2.1, find the solution to

$$\begin{bmatrix} 1 & 3 & 4 & 8 \\ 2 & 1 & 2 & 3 \\ 4 & 3 & 5 & 8 \\ 9 & 2 & 7 & 4 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix} = \begin{bmatrix} 1 \\ 1 \\ 1 \\ 1 \end{bmatrix}$$

Solution 2.2 Note that the product of a series of lower triangular matrices is lower triangular; therefore, the product

$$W = \xi_9\xi_8\xi_7\xi_6\xi_5\xi_4\xi_3\xi_2\xi_1 \tag{2.8}$$

is lower triangular. Since the application of the elementary matrices to the matrix A results in an upper triangular matrix, then

$$WA = U \tag{2.9}$$

where U is the upper triangular matrix that results from the forward elimination process. Premultiplying Equation (2.1) by W yields

$$WAx = Wb \tag{2.10}$$

$$Ux = Wb \tag{2.11}$$

$$= b' \tag{2.12}$$

where $Wb = b'$.

From Example 2.1:

$$W = \begin{bmatrix} 1 & 0 & 0 & 0 \\ \frac{2}{5} & -\frac{1}{5} & 0 & 0 \\ 2 & 9 & -5 & 0 \\ \frac{1}{6} & \frac{14}{6} & -\frac{5}{6} & -\frac{1}{6} \end{bmatrix}$$

and

$$b' = W \begin{bmatrix} 1 \\ 1 \\ 1 \\ 1 \end{bmatrix} = \begin{bmatrix} 1 \\ \frac{1}{5} \\ 6 \\ \frac{3}{2} \end{bmatrix}$$

Thus

$$\begin{bmatrix} 1 & 3 & 4 & 8 \\ 0 & 1 & \frac{6}{5} & \frac{13}{5} \\ 0 & 0 & 1 & 3 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix} = \begin{bmatrix} 1 \\ \frac{1}{5} \\ 6 \\ \frac{3}{2} \end{bmatrix} \tag{2.13}$$

By inspection, $x_4 = \frac{3}{2}$. The third row yields

$$x_3 = 6 - 3x_4 \tag{2.14}$$

Substituting the value of x_4 into Equation (2.14) yields $x_3 = \frac{3}{2}$. Similarly,

$$x_2 = \frac{1}{5} - \frac{6}{5}x_3 - \frac{13}{5}x_4 \tag{2.15}$$

and substituting x_3 and x_4 into Equation (2.15) yields $x_2 = -\frac{11}{2}$. Solving for x_1 in a similar manner produces

$$x_1 = 1 - 3x_2 - 4x_3 - 8x_4 \tag{2.16}$$

$$= -\frac{1}{2} \tag{2.17}$$

Thus

$$\begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix} = \frac{1}{2} \begin{bmatrix} -1 \\ -11 \\ 3 \\ 3 \end{bmatrix} \blacksquare$$

The solution methodology of successively substituting values of x back into the equation as they are found gives rise to the name of *back substitution* for

this step of Gaussian elimination. Therefore, Gaussian elimination consists of two main steps: forward elimination and back substitution. Forward elimination is the process of transforming the matrix A into triangular factors. Back substitution is the process by which the unknown vector x is found from the input vector b and the factors of A. Gaussian elimination also provides the framework under which the LU factorization process is developed.

2.2 LU Factorization

The forward elimination step of Gaussian elimination produces a series of upper and lower triangular matrices that are related to the A matrix as given in Equation (2.9). The matrix W is a lower triangular matrix and U is an upper triangular matrix with ones on the diagonal. Recall that the inverse of a lower triangular matrix is also a lower triangular matrix; therefore, if

$$L \triangleq W^{-1}$$

then

$$A = LU$$

The matrices L and U give rise to the name of the factorization/elimination algorithm known as "LU factorization." In fact, given any nonsingular matrix A, there exists some permutation matrix P (possibly $P = I$) such that

$$LU = PA \tag{2.18}$$

where U is upper triangular with unit diagonals, L is lower triangular with nonzero diagonals, and P is a matrix of ones and zeros obtained by rearranging the rows and columns of the identity matrix. Once a proper matrix P is chosen, this factorization is unique [8]. Once P, L, and U are determined, then the system

$$Ax = b \tag{2.19}$$

can be solved expeditiously. Premultiplying Equation (2.19) by the matrix P yields

$$PAx = Pb = b' \tag{2.20}$$

$$LUx = b' \tag{2.21}$$

where b' is just a rearrangement of the vector b. Introducing a "dummy" vector y such that

$$Ux = y \tag{2.22}$$

thus

$$Ly = b' \tag{2.23}$$

Consider the structure of Equation (2.23):

$$\begin{bmatrix} l_{11} & 0 & 0 & \cdots & 0 \\ l_{21} & l_{22} & 0 & \cdots & 0 \\ l_{31} & l_{32} & l_{33} & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ l_{n1} & l_{n2} & l_{n3} & \cdots & l_{nn} \end{bmatrix} \begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ \vdots \\ y_n \end{bmatrix} = \begin{bmatrix} b'_1 \\ b'_2 \\ b'_3 \\ \vdots \\ b'_n \end{bmatrix}$$

The elements of the vector y can be found by straightforward substitution:

$$\begin{aligned} y_1 &= \frac{b'_1}{l_{11}} \\ y_2 &= \frac{1}{l_{22}} (b'_2 - l_{21} y_1) \\ y_3 &= \frac{1}{l_{33}} (b'_3 - l_{31} y_1 - l_{32} y_2) \\ &\vdots \\ y_n &= \frac{1}{l_{nn}} \left(b'_n - \sum_{j=1}^{n-1} l_{nj} y_j \right) \end{aligned}$$

After the vector y has been found, then x can be easily found from

$$\begin{bmatrix} 1 & u_{12} & u_{13} & \cdots & u_{1n} \\ 0 & 1 & u_{23} & \cdots & u_{2n} \\ 0 & 0 & 1 & \cdots & u_{3n} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \vdots & 1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ \vdots \\ x_n \end{bmatrix} = \begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ \vdots \\ y_n \end{bmatrix}$$

Similarly, the solution vector x can be found by backward substitution:

$$\begin{aligned} x_n &= y_n \\ x_{n-1} &= y_{n-1} - u_{n-1,n} x_n \\ x_{n-2} &= y_{n-2} - u_{n-2,n} x_n - u_{n-2,n-1} x_{n-1} \\ &\vdots \\ x_1 &= y_1 - \sum_{j=2}^{n} u_{1j} x_j \end{aligned}$$

The value of LU factorization is that, once A is factored into the upper and lower triangular matrices, the solution for the solution vector x is straightforward. Note that the inverse to A is never explicitly found.

Several methods for computing the LU factors exist and each method has its advantages and disadvantages. One common factorization approach is

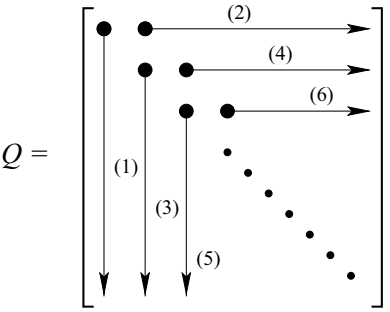

FIGURE 2.1
Order of calculating columns and rows of Q

known as *Crout's* algorithm for finding the LU factors [8]. Let the matrix Q be defined as

$$Q \triangleq L + U - I = \begin{bmatrix} l_{11} & u_{12} & u_{13} & \cdots & u_{1n} \\ l_{21} & l_{22} & u_{23} & \cdots & u_{2n} \\ l_{31} & l_{32} & l_{33} & \cdots & u_{3n} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ l_{n1} & l_{n2} & l_{n3} & \cdots & l_{nn} \end{bmatrix} \tag{2.24}$$

Crout's algorithm computes the elements of Q first by column and then row, as shown in Figure 2.1. Each element q_{ij} of Q depends only on the a_{ij} entry of A and previously computed values of Q.

Crout's Algorithm for Computing LU from A

1. Initialize Q to the zero matrix. Let $j = 1$.

2. Complete the jth column of Q (jth column of L) as

$$q_{kj} = a_{kj} - \sum_{i=1}^{j-1} q_{ki} q_{ij} \quad \text{for } k = j, \ldots, n \tag{2.25}$$

3. If $j = n$, then stop.

4. Assuming that $q_{jj} \neq 0$, complete the jth row of Q (jth row of U) as

$$q_{jk} = \frac{1}{q_{jj}} \left(a_{jk} - \sum_{i=1}^{j-1} q_{ji} q_{ik} \right) \quad \text{for } k = j+1, \ldots, n \tag{2.26}$$

5. Set $j = j + 1$. Go to step 2.

Once the LU factors are found, then the dummy vector y can be found by forward substitution.

Forward Substitution

$$y_k = \frac{1}{q_{kk}} \left(b_k - \sum_{j=1}^{k-1} q_{kj} y_j \right) \quad \text{for } k = 1, \ldots, n \tag{2.27}$$

Similarly, the solution vector x can be found by backward substitution.

Backward Substitution:

$$x_k = y_k - \sum_{j=k+1}^{n} q_{kj} x_j \quad \text{for } k = n, n-1, \ldots, 1 \tag{2.28}$$

One measure of the computation involved in the LU factorization process is to count the number of multiplications and divisions required to find the solution, since these are both floating point operations. Computing the jth column of Q (jth column of L) requires

$$\sum_{j=1}^{n} \sum_{k=j}^{n} (j-1)$$

multiplications and divisions. Similarly, computing the jth row of Q (jth row of U) requires

$$\sum_{j=1}^{n-1} \sum_{k=j+1}^{n} j$$

multiplications and divisions. The forward substitution step requires

$$\sum_{j=1}^{n} j$$

and the backward substitution step requires

$$\sum_{j=1}^{n} (n-j)$$

multiplications and divisions. Taken together, the LU factorization procedure requires

$$\frac{1}{3} \left(n^3 - n \right)$$

and the substitution steps require n^2 multiplications and divisions. Therefore, the whole process of solving the linear system of Equation (2.1) requires a total of

$$\frac{1}{3}\left(n^3 - n\right) + n^2 \tag{2.29}$$

multiplications and divisions. Compare this to the requirements of Cramer's rule, which requires $2(n+1)!$ multiplications and divisions. Obviously, for a system of any significant size, it is far more computationally efficient to use LU factorization and forward/backward substitution to find the solution x.

Example 2.3
Using LU factorization with forward and backward substitution, find the solution to the system of Example 2.2.

Solution 2.3 The first step is to find the LU factors of the A matrix:

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

Starting with $j = 1$, Equation (2.25) indicates that the elements of the first column of Q are identical to the elements of the first column of A. Similarly, according to Equation (2.26), the first row of Q becomes

$$q_{12} = \frac{a_{12}}{q_{11}} = \frac{3}{1} = 3$$
$$q_{13} = \frac{a_{13}}{q_{11}} = \frac{4}{1} = 4$$
$$q_{14} = \frac{a_{14}}{q_{11}} = \frac{8}{1} = 8$$

Thus, for $j = 1$, the Q matrix becomes

$$Q = \begin{bmatrix} 1 & 3 & 4 & 8 \\ 2 & & & \\ 4 & & & \\ 9 & & & \end{bmatrix}$$

For $j = 2$, the second column and row of Q below and to the right of the diagonal, respectively, will be calculated. For the second column of Q:

$$q_{22} = a_{22} - q_{21}q_{12} = 1 - (2)(3) = -5$$
$$q_{32} = a_{32} - q_{31}q_{12} = 3 - (4)(3) = -9$$
$$q_{42} = a_{42} - q_{41}q_{12} = 2 - (9)(3) = -25$$

Each element of Q uses the corresponding element of A and elements of Q that have been previously computed. Note also that the inner indices of the products are always the same and the outer indices are the same as the indices of the element being computed. This holds true for both column and row calculations. The second row of Q is computed

$$q_{23} = \frac{1}{q_{22}}\left(a_{23} - q_{21}q_{13}\right) = \frac{1}{-5}\left(2 - (2)(4)\right) = \frac{6}{5}$$
$$q_{24} = \frac{1}{q_{22}}\left(a_{24} - q_{21}q_{14}\right) = \frac{1}{-5}\left(3 - (2)(8)\right) = \frac{13}{5}$$

After $j = 2$, the Q matrix becomes

$$Q = \begin{bmatrix} 1 & 3 & 4 & 8 \\ 2 & -5 & \frac{6}{5} & \frac{13}{5} \\ 4 & -9 & & \\ 9 & -25 & & \end{bmatrix}$$

Continuing on for $j = 3$, the third column of Q is calculated

$$q_{33} = a_{33} - \left(q_{31}q_{13} + q_{32}q_{23}\right) = 5 - \left((4)(4) + (-9)\frac{6}{5}\right) = -\frac{1}{5}$$
$$q_{43} = a_{43} - \left(q_{41}q_{13} + q_{42}q_{23}\right) = 7 - \left((9)(4) + (-25)\frac{6}{5}\right) = 1$$

and the third row of Q becomes

$$q_{34} = \frac{1}{q_{33}}\left(a_{34} - \left(q_{31}q_{14} + q_{32}q_{24}\right)\right)$$
$$= (-5)\left(8 - \left((4)(8) + (-9)\left(\frac{13}{5}\right)\right)\right) = 3$$

yielding

$$Q = \begin{bmatrix} 1 & 3 & 4 & 8 \\ 2 & -5 & \frac{6}{5} & \frac{13}{5} \\ 4 & -9 & -\frac{1}{5} & 3 \\ 9 & -25 & 1 & \end{bmatrix}$$

Finally, for $j = 4$, the final diagonal element is found:

$$q_{44} = a_{44} - \left(q_{41}q_{14} + q_{42}q_{24} + q_{43}q_{34}\right)$$
$$= 4 - \left((9)(8) + (-25)\left(\frac{13}{5}\right) + (3)(1)\right) = -6$$

Thus

$$Q = \begin{bmatrix} 1 & 3 & 4 & 8 \\ 2 & -5 & \frac{6}{5} & \frac{13}{5} \\ 4 & -9 & -\frac{1}{5} & 3 \\ 9 & -25 & 1 & -6 \end{bmatrix}$$

$$L = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 2 & -5 & 0 & 0 \\ 4 & -9 & -\frac{1}{5} & 0 \\ 9 & -25 & 1 & -6 \end{bmatrix}$$

$$U = \begin{bmatrix} 1 & 3 & 4 & 8 \\ 0 & 1 & \frac{6}{5} & \frac{13}{5} \\ 0 & 0 & 1 & 3 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

One method of checking the correctness of the solution is to check if $LU = A$, which in this case it does.

Once the LU factors have been found, then the next step in the solution process is forward elimination using the L matrix and the b vector to find the dummy vector y. Using forward substitution to solve $Ly = b$ for y:

$$\begin{aligned} y_1 &= \frac{b_1}{L_{11}} = \frac{1}{1} = 1 \\ y_2 &= \frac{(b_2 - L_{21}y_1)}{L_{22}} = \frac{(1-(2)(1))}{-5} = \frac{1}{5} \\ y_3 &= \frac{(b_3 - (L_{31}y_1 + L_{32}y_2))}{L_{33}} = (-5)\left(1 - \left((4)(1) + (-9)\frac{1}{5}\right)\right) = 6 \\ y_4 &= \frac{(b_4 - (L_{41}y_1 + L_{42}y_2 + L_{43}y_3))}{L_{44}} \\ &= \frac{\left(1 - \left((9)(1) + (-25)\left(\frac{1}{5}\right) + (1)(6)\right)\right)}{-6} = \frac{3}{2} \end{aligned}$$

Thus

$$y = \begin{bmatrix} 1 \\ \frac{1}{5} \\ 6 \\ \frac{3}{2} \end{bmatrix}$$

Similarly, backward substitution is then applied to $Ux = y$ to find the solution vector x:

$$\begin{aligned} x_4 &= y_4 = \frac{3}{2} \\ x_3 &= y_3 - U_{34}x_4 = 6 - (3)\left(\frac{3}{2}\right) = \frac{3}{2} \\ x_2 &= y_2 - (U_{24}x_4 + U_{23}x_3) = \frac{1}{5} - \left(\left(\frac{13}{5}\right)\left(\frac{3}{2}\right) + \left(\frac{6}{5}\right)\left(\frac{3}{2}\right)\right) = -\frac{11}{2} \\ x_1 &= y_1 - (U_{14}x_4 + U_{13}x_3 + U_{12}x_2) \\ &= 1 - \left((8)\left(\frac{3}{2}\right) + (4)\left(\frac{3}{2}\right) + (3)\left(-\frac{11}{2}\right)\right) = -\frac{1}{2} \end{aligned}$$

yielding the final solution vector

$$x = \frac{1}{2}\begin{bmatrix} -1 \\ -11 \\ 3 \\ 3 \end{bmatrix}$$

which is the same solution found by Gaussian elimination and backward substitution in Example 2.2. A quick check to verify the correctness of the solution is to substitute the solution vector x back into the linear system $Ax = b$. ■

2.2.1 LU Factorization with Partial Pivoting

The LU factorization process presented assumes that the diagonal element is nonzero. Not only must the diagonal element be nonzero, it must be of the same order of magnitude as the other nonzero elements. Consider the solution of the following linear system

$$\begin{bmatrix} 10^{-10} & 1 \\ 2 & 1 \end{bmatrix}\begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} 1 \\ 5 \end{bmatrix} \tag{2.30}$$

By inspection, the solution to this linear system is

$$x_1 \approx 2$$
$$x_2 \approx 1$$

The LU factors for A are

$$L = \begin{bmatrix} 10^{-10} & 0 \\ 2 & \left(1 - 2 \times 10^{10}\right) \end{bmatrix}$$
$$U = \begin{bmatrix} 1 & 10^{10} \\ 0 & 1 \end{bmatrix}$$

Applying forward elimination to solve for the dummy vector y yields

$$y_1 = 10^{10}$$
$$y_2 = \frac{\left(5 - 2 \times 10^{10}\right)}{\left(1 - 2 \times 10^{10}\right)} \approx 1$$

Back substituting y into $Ux = y$ yields

$$x_2 = y_2 \approx 1$$
$$x_1 = 10^{10} - 10^{10} x_2 \approx 0$$

The solution for x_2 is correct, but the solution for x_1 is considerably off. Why did this happen? The problem with the equations arranged the way they are

in Equation (2.30) is that 10^{-10} is too near zero for most computers. However, if the equations are rearranged such that

$$\begin{bmatrix} 2 & 1 \\ 10^{-10} & 1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} 5 \\ 1 \end{bmatrix} \tag{2.31}$$

then the LU factors become

$$L = \begin{bmatrix} 2 & 0 \\ 10^{-10} & \left(1 - \frac{1}{2} \times 10^{-10}\right) \end{bmatrix}$$
$$U = \begin{bmatrix} 1 & \frac{1}{2} \\ 0 & 1 \end{bmatrix}$$

The dummy vector y becomes

$$y_1 = \frac{5}{2}$$
$$y_2 = \frac{\left(1 - \frac{5}{2} \times 10^{-10}\right)}{\left(1 - \frac{1}{2} \times 10^{-10}\right)} \approx 1$$

and by back substitution, x becomes

$$x_2 \approx 1$$
$$x_1 \approx \frac{5}{2} - \frac{1}{2}(1) = 2$$

which is the solution obtained by inspection of the equations. Therefore, even though the diagonal entry may not be exactly zero, it is still good practice to rearrange the equations such that the largest magnitude element lies on the diagonal. This process is known as *pivoting* and gives rise to the permutation matrix P of Equation (2.18).

Since Crout's algorithm computes the Q matrix by column and row with increasing index, only *partial pivoting* can be used, that is, only the rows of Q (and correspondingly A) can be exchanged. The columns must remain static. To choose the best pivot, the column beneath the jth diagonal (at the jth step in the LU factorization) is searched for the element with the largest absolute value. The corresponding row and the jth row are then exchanged. The pivoting strategy may be succinctly expressed as

Partial Pivoting Strategy

1. At the jth step of LU factorization, choose the kth row as the exchange row such that
$$|q_{jj}| = max\,|q_{kj}| \text{ for } k = j, \ldots, n \tag{2.32}$$
2. Exchange rows and update A, P, and Q correspondingly.

The permutation matrix P is composed of ones and zeros and is obtained as the product of a series of elementary permutation matrices $P^{j,k}$ which represent the exchange of rows j and k. The elementary permutation matrix $P^{j,k}$, shown in Figure 2.2, is obtained from the identify matrix by interchanging rows j and k. A pivot is achieved by the premultiplication of a properly chosen $P^{j,k}$. Since this is only an interchange of rows, the order of the unknown vector does not change.

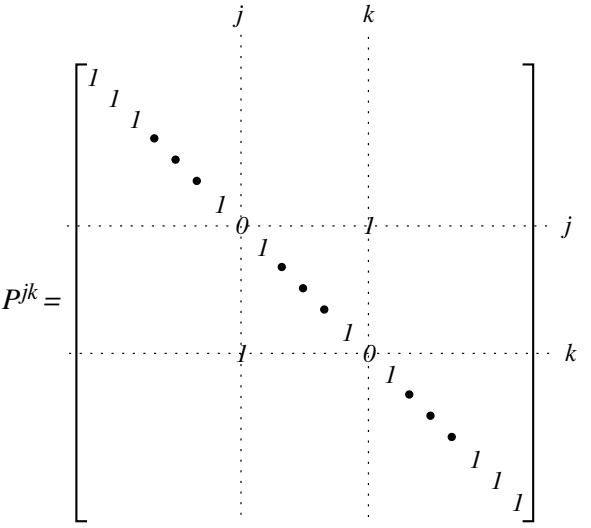

FIGURE 2.2
Elementary permutation matrix $P^{j,k}$

Example 2.4
Repeat Example 2.3 using partial pivoting.

Solution 2.4 The A matrix is repeated here for convenience.

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

For $j = 1$, the first column of Q is exactly the first column of A. Applying the pivoting strategy of Equation (2.32), the q_{41} element has the largest magnitude of the first column; therefore, rows four and one are exchanged. The

elementary permutation matrix $P^{1,4}$ is

$$P^{1,4} = \begin{bmatrix} 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix}$$

The corresponding A matrix becomes

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

and Q at the $j = 1$ step

$$Q = \begin{bmatrix} 9 & \frac{2}{9} & \frac{7}{9} & \frac{4}{9} \\ 2 & & & \\ 4 & & & \\ 1 & & & \end{bmatrix}$$

At $j = 2$, the calculation of the second column of Q yields

$$Q = \begin{bmatrix} 9 & \frac{2}{9} & \frac{7}{9} & \frac{4}{9} \\ 2 & \frac{5}{9} & & \\ 4 & \frac{19}{9} & & \\ 1 & \frac{25}{9} & & \end{bmatrix}$$

Searching the elements in the jth column below the diagonal, the fourth row of the jth (i.e., second) column once again yields the largest magnitude. Therefore, rows two and four must be exchanged, yielding the elementary permutation matrix $P^{2,4}$

$$P^{2,4} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \end{bmatrix}$$

Similarly, the updated A is

$$\begin{bmatrix} 9 & 2 & 7 & 4 \\ 1 & 3 & 4 & 8 \\ 4 & 3 & 5 & 8 \\ 2 & 1 & 2 & 3 \end{bmatrix}$$

which yields the following Q

$$Q = \begin{bmatrix} 9 & \frac{2}{9} & \frac{7}{9} & \frac{4}{9} \\ 1 & \frac{25}{9} & \frac{29}{25} & \frac{68}{25} \\ 4 & \frac{19}{9} & & \\ 2 & \frac{5}{9} & & \end{bmatrix}$$

For $j = 3$, the calculation of the third column of Q yields

$$Q = \begin{bmatrix} 9 & \frac{2}{9} & \frac{7}{9} & \frac{4}{9} \\ 1 & \frac{25}{9} & \frac{29}{25} & \frac{68}{25} \\ 4 & \frac{19}{9} & -\frac{14}{25} & \\ 2 & \frac{5}{9} & -\frac{1}{5} & \end{bmatrix}$$

In this case, the diagonal element has the largest magnitude, so no pivoting is required. Continuing with the calculation of the 3rd row of Q yields

$$Q = \begin{bmatrix} 9 & \frac{2}{9} & \frac{7}{9} & \frac{4}{9} \\ 1 & \frac{25}{9} & \frac{29}{25} & \frac{68}{25} \\ 4 & \frac{19}{9} & -\frac{14}{25} & -\frac{12}{14} \\ 2 & \frac{5}{9} & -\frac{1}{5} & \end{bmatrix}$$

Finally, calculating q_{44} yields the final Q matrix

$$Q = \begin{bmatrix} 9 & \frac{2}{9} & \frac{7}{9} & \frac{4}{9} \\ 1 & \frac{25}{9} & \frac{29}{25} & \frac{68}{25} \\ 4 & \frac{19}{9} & -\frac{14}{25} & -\frac{12}{14} \\ 2 & \frac{5}{9} & -\frac{1}{5} & \frac{3}{7} \end{bmatrix}$$

The permutation matrix P is found by multiplying together the two elementary permutation matrices:

$$\begin{aligned} P &= P^{2,4} P^{1,4} I \\ &= \begin{bmatrix} 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \end{bmatrix} \end{aligned}$$

The results can be checked to verify that $PA = LU$. The forward and backward substitution steps are carried out on the modified vector $b' = Pb$. ■

2.2.2 LU Factorization with Complete Pivoting

An alternate LU factorization that allows complete pivoting is *Gauss'* method. In this approach, two permutation matrices are developed: one for row exchange as in partial pivoting, and a second matrix for column exchange. In this approach, the LU factors are found such that

$$P_1 A P_2 = LU \tag{2.33}$$

Therefore, to solve the linear system of equations $Ax = b$ requires that a slightly different approach be used. As with partial pivoting, the permutation matrix P_1 premultiplies the linear system:

$$P_1 A x = P_1 b = b' \tag{2.34}$$

Now, define a new vector z such that

$$x = P_2 z \tag{2.35}$$

Then substituting Equation (2.35) into Equation (2.34) yields

$$\begin{aligned} P_1 A P_2 z &= P_1 b = b' \\ LUz &= b' \end{aligned} \tag{2.36}$$

where Equation (2.36) can be solved using forward and backward substitution for z. Once z is obtained, then the solution vector x follows from Equation (2.35).

In complete pivoting, both rows and columns may be interchanged to place the largest element (in magnitude) on the diagonal at each step in the LU factorization process. The pivot element is chosen from the remaining elements below and to the right of the diagonal.

Complete Pivoting Strategy

1. At the jth step of LU factorization, choose the pivot element such that

$$|q_{jj}| = max\,|q_{kl}| \text{ for } k = j, \ldots, n, \text{and } l = j, \ldots, n \tag{2.37}$$

2. Exchange rows and update A, P, and Q correspondingly.

Gauss' Algorithm for Computing LU from A

1. Initialize Q to the zero matrix. Let $j = 1$.

2. Set the jth column of Q (jth column of L) to the jth column of the reduced matrix $A^{(j)}$, where $A^{(1)} = A$, and

$$q_{kj} = a_{kj}^{(j)} \text{ for } k = j, \ldots, n \tag{2.38}$$

3. If $j = n$, then stop.

4. Assuming that $q_{jj} \neq 0$, set the jth row of Q (jth row of U) as

$$q_{jk} = \frac{a_{jk}^{(j)}}{q_{jj}} \text{ for } k = j + 1, \ldots, n \tag{2.39}$$

5. Update $A^{(j+1)}$ from $A^{(j)}$ as

$$a_{ik}^{(j+1)} = a_{ik}^{(j)} - q_{ij} q_{jk} \text{ for } i = j + 1, \ldots, n, \text{and } k = j + 1, \ldots, n \tag{2.40}$$

6. Set $j = j + 1$. Go to step 2.

This factorization algorithm gives rise to the same number of multiplications and divisions as Crout's algorithm for LU factorization. Crout's algorithm uses each entry of the A matrix only once, whereas Gauss' algorithm updates the A matrix each time. One advantage of Crout's algorithm over Gauss' algorithm is each element of the A matrix is used only once. Since each q_{jk} is a function of a_{jk} and then a_{jk} is never used again, the element q_{jk} can be written *over* the a_{jk} element. Therefore, rather than having to store two $n \times n$ matrices in memory (A and Q), only one matrix is required.

Crout's and Gauss' algorithms are only two of numerous algorithms for LU factorization. Other methods include Doolittle and bifactorization algorithms [21], [26], [54]. Most of these algorithms require similar numbers of multiplications and divisions and only differ slightly in performance when implemented on traditional serial computers. However, these algorithms differ considerably when factors such as memory access, storage, and parallelization are considered. Consequently, it is wise to choose the factorization algorithm to fit the application and the computer architecture upon which it will be implemented.

Note that, as Q is constructed, there are now entries that may appear in the matrix structure of Q that were not originally present in the structure of A. Entries that change from an initial zero to a nonzero value during the LU factorization are called "fill-ins" or *fills* for short. The number and placement of fills play an important role in multiple applications, including preconditioning and sparse computation.

Example 2.5

For the matrix A given below, find the Q matrix and identify the fills.

$$A = \begin{bmatrix} 10 & 1 & 0 & 3 & 0 & 0 & 0 & 5 & 0 & 0 \\ 2 & 9 & 0 & 0 & 0 & 0 & 5 & 0 & 0 & 2 \\ 0 & 0 & 21 & 5 & 7 & 0 & 0 & 0 & 0 & 4 \\ 4 & 0 & 1 & 18 & 8 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 4 & 7 & 25 & 4 & 1 & 0 & 0 & 2 \\ 0 & 0 & 0 & 0 & 3 & 14 & 9 & 0 & 0 & 0 \\ 0 & 1 & 4 & 0 & 2 & 3 & 12 & 1 & 1 & 0 \\ 1 & 0 & 5 & 0 & 0 & 0 & 5 & 10 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 6 & 0 & 20 & 0 \\ 0 & 2 & 3 & 0 & 4 & 0 & 0 & 0 & 0 & 35 \end{bmatrix}$$

Solution 2.5

The resulting Q is given below. The fills are indicated by the boxed numbers. By comparing Q with A, it can be seen that there are 24 fills, or entries in Q that were previously zero in A.

$$Q = \begin{bmatrix}
10.0000 & 0.1000 & 0 & 0.3000 & 0 & 0 & 0 & 0.5000 & 0 & 0 \\
2.0000 & 8.8000 & 0 & -0.0682 & 0 & 0 & 0.5682 & -0.1136 & 0 & 0.2273 \\
0 & 0 & 21.0000 & 0.2381 & 0.3333 & 0 & 0 & 0 & 0 & 0.1905 \\
4.0000 & -0.4000 & 1.0000 & 16.5346 & 0.4637 & 0 & 0.0137 & -0.1237 & 0 & -0.0060 \\
0 & 0 & 4.0000 & 6.0476 & 20.8625 & 0.1917 & 0.0439 & 0.0359 & 0 & 0.0611 \\
0 & 0 & 0 & 0 & 3.0000 & 13.4248 & 0.6606 & -0.0080 & 0 & -0.0137 \\
0 & 1.0000 & 4.0000 & -0.8842 & 1.0766 & 2.7936 & 9.5513 & 0.1034 & 0.1047 & -0.1070 \\
1.0000 & -0.1000 & 5.0000 & -1.4973 & -0.9724 & 0.1864 & 4.9970 & 8.8229 & -0.0593 & -0.0388 \\
0 & 0 & 0 & 0 & 0 & 0 & 6.0000 & -0.6207 & 19.3350 & 0.0320 \\
0 & 2.0000 & 3.0000 & -0.5779 & 3.2680 & -0.6266 & -0.8581 & 0.1223 & 0.0971 & 33.6722
\end{bmatrix}$$

■

2.3 Condition Numbers and Error Propagation

The Gaussian elimination and LU factorization algorithms are considered direct methods because they calculate the solution vector $x^* = A^{-1}b$ in a finite number of steps without an iterative refinement. On a computer with infinite precision, direct methods would yield the exact solution x^*. However, since computers have finite precision, the solution obtained has limited accuracy. The *condition number* of a matrix is a useful measure for determining the level of accuracy of a solution. The condition number of the matrix A is generally defined as

$$\kappa(A) = \sqrt{\frac{\lambda_{max}}{\lambda_{min}}} \tag{2.41}$$

where λ_{max} and λ_{min} denote the largest and smallest eigenvalues of the matrix $A^T A$. These eigenvalues are real and nonnegative regardless of whether the eigenvalues of A are real or complex.

The condition number of a matrix is a measure of the linear independence of the eigenvectors of the matrix. A singular matrix has at least one zero eigenvalue and contains at least one degenerate row (i.e., the row can be expressed as a linear combination of other rows). The identity matrix, which gives rise to the most linearly independent eigenvectors possible and has every eigenvalue equal to one, has a condition number of 1. If the condition number of a matrix is much greater than one, then the matrix is said to be *ill conditioned*. The larger the condition number, the more sensitive the solution process is to slight perturbations in the elements of A and the more numerical error likely to be contained in the solution.

Because of numerical error introduced into the solution process, the computed solution $\tilde{x}$ of Equation (2.1) will differ from the exact solution x^* by a finite amount Δx. Other errors, such as approximation, measurement, or round-off error, may be introduced into the matrix A and vector b. Gaussian elimination produces a solution that has roughly

$$t \log_{10} \beta - \log_{10} \kappa(A) \tag{2.42}$$

correct decimal places in the solution, where t is the bit length of the mantissa ($t = 24$ for a typical 32-bit binary word), β is the base ($\beta = 2$ for binary operations), and κ is the condition number of the matrix A. One interpretation of Equation (2.42) is that the solution will lose about $\log_{10} \kappa$ digits of accuracy during Gaussian elimination (and consequently LU factorization). Based upon the known accuracy of the matrix entries, the condition number, and the machine precision, the accuracy of the numerical solution $\tilde{x}$ can be predicted [39].

2.4 Stationary Iterative Methods

Stationary iterative methods, also known as relaxation methods, are iterative in nature and produce a sequence of vectors that ideally converge to the solution $x^* = A^{-1}b$. Relaxation methods can be incorporated into the solution of Equation (2.1) in several ways. In all cases, the principal advantage of using an iterative method stems from not requiring a direct solution of a large system of linear equations and from the fact that the relaxation methods permit the simulator to exploit the latent portions of the system (those portions which are relatively unchanging at the present time) effectively. In addition, with the advent of parallel-processing technology, relaxation methods lend themselves more readily to parallel implementation than do direct methods. The two most common stationary methods are the Jacobi and the Gauss–Seidel methods [61].

These relaxation methods may be applied for the solution of the linear system

$$Ax = b \tag{2.43}$$

A general approach to relaxation methods is to define a *splitting matrix* M such that Equation (2.43) can be rewritten in equivalent form as

$$Mx = (M - A)\, x + b \tag{2.44}$$

This splitting leads to the iterative process

$$Mx^{k+1} = (M - A)\, x^k + b \quad k = 1, \ldots, \infty \tag{2.45}$$

where k is the iteration index. This iteration produces a sequence of vectors $x^1, x^2, \ldots$ for a given initial guess x^0. Various iterative methods can be developed by different choices of the matrix M. The objective of a relaxation method is to choose the splitting matrix M such that the sequence is easily computed and the sequence converges rapidly to a solution.

Let A be split into $L + D + U$, where L is strictly lower triangular, D is a diagonal matrix, and U is strictly upper triangular. Note that these matrices

are different from the L and U obtained from LU factorization. The vector x can then be solved for in an iterative manner using the Jacobi relaxation method,

$$x^{k+1} = -D^{-1}\left((L+U)\,x^k - b\right) \tag{2.46}$$

or identically in scalar form,

$$x_i^{k+1} = -\sum_{j\neq i}^{n}\left(\frac{a_{ij}}{a_{ii}}\right)x_j^k + \frac{b_i}{a_{ii}} \quad 1 \leq i \leq n,\ k \geq 0 \tag{2.47}$$

In the Jacobi relaxation method, all of the updates of the approximation vector x^{k+1} are obtained by using only the components of the previous approximation vector x^k. Therefore, this method is also sometimes called the method of simultaneous displacements.

The Gauss–Seidel relaxation method is similar:

$$x^{k+1} = -\left(L+D\right)^{-1}\left(Ux^k - b\right) \tag{2.48}$$

or in scalar form

$$x_i^{k+1} = -\sum_{j=1}^{i-1}\left(\frac{a_{ij}}{a_{ii}}\right)x_j^{k+1} - \sum_{j=i+1}^{n}\left(\frac{a_{ij}}{a_{ii}}\right)x_j^k + \frac{b_i}{a_{ii}} \quad 1 \leq i \leq n,\ k \geq 0 \tag{2.49}$$

The Gauss–Seidel method has the advantage that each new update x_i^{k+1} relies only on previously computed values at that iteration: $x_1^{k+1}, x_2^{k+1}, \ldots, x_{i-1}^{k+1}$. Since the states are updated one by one, the new values can be stored in the same locations held by the old values, thus reducing the storage requirements.

Since relaxation methods are iterative, it is essential to determine under what conditions they are guaranteed to converge to the exact solution

$$x^* = A^{-1}b \tag{2.50}$$

It is well known that a necessary and sufficient condition for the Jacobi relaxation method to converge given any initial guess x_0 is that all eigenvalues of

$$M_J \triangleq -D^{-1}\left(L+U\right) \tag{2.51}$$

must lie within the unit circle in the complex plane [61]. Similarly, the eigenvalues of

$$M_{GS} \triangleq -\left(L+D\right)^{-1}U \tag{2.52}$$

must lie within the unit circle in the complex plane for the Gauss–Seidel relaxation algorithm to converge for any initial guess x_0. In practice, these conditions are difficult to confirm. There are several more general conditions that are easily confirmed under which convergence is guaranteed. In particular, if A is strictly diagonally dominant, then both the Jacobi and Gauss–Seidel methods are guaranteed to converge to the exact solution.

The initial vector x_0 can be arbitrary; however, if a good guess of the solution is available, it should be used for x_0 to produce more rapid convergence to within some predefined tolerance.

In general, the Gauss–Seidel method converges faster than the Jacobi for most classes of problems. If A is lower-triangular, the Gauss–Seidel method will converge in one iteration to the exact solution, whereas the Jacobi method will take n iterations. The Jacobi method has the advantage, however, that, at each iteration, each x_i^{k+1} is independent of all other x_j^{k+1} for $j \neq i$. Thus the computation of all x_i^{k+1} can proceed in parallel. This method is therefore well suited to parallel processing [40].

Both the Jacobi and Gauss–Seidel methods can be generalized to the block-Jacobi and block-Gauss–Seidel methods where A is split into block matrices $L + D + U$, where D is block diagonal and L and U are lower- and upper-block triangular, respectively. The same necessary and sufficient convergence conditions exist for the block case as for the scalar case, that is, the eigenvalues of M_J and M_{GS} must lie within the unit circle in the complex plane.

Example 2.6

Solve

$$\begin{bmatrix} -10 & 2 & 3 & 6 \\ 0 & -9 & 1 & 4 \\ 2 & 6 & -12 & 2 \\ 3 & 1 & 0 & -8 \end{bmatrix} x = \begin{bmatrix} 1 \\ 2 \\ 3 \\ 4 \end{bmatrix} \tag{2.53}$$

for x using (1) the Gauss–Seidel method, and (2) the Jacobi method.

Solution 2.6 The Gauss–Seidel method given in Equation (2.49) with the initial vector $x = [0\ 0\ 0\ 0]$ leads to the following updates:

k	x_1	x_2	x_3	x_4
1	0.0000	0.0000	0.0000	0.0000
2	−0.1000	−0.2222	−0.3778	−0.5653
3	−0.5969	−0.5154	−0.7014	−0.7883
4	−0.8865	−0.6505	−0.8544	−0.9137
5	−1.0347	−0.7233	−0.9364	−0.9784
6	−1.1126	−0.7611	−0.9791	−1.0124
7	−1.1534	−0.7809	−1.0014	−1.0301
8	−1.1747	−0.7913	−1.0131	−1.0394
9	−1.1859	−0.7968	−1.0193	−1.0443
10	−1.1917	−0.7996	−1.0225	−1.0468
11	−1.1948	−0.8011	−1.0241	−1.0482
12	−1.1964	−0.8019	−1.0250	−1.0489
13	−1.1972	−0.8023	−1.0255	−1.0492
14	−1.1976	−0.8025	−1.0257	−1.0494
15	−1.1979	−0.8026	−1.0259	−1.0495
16	−1.1980	−0.8027	−1.0259	−1.0496

The Gauss–Seidel iterates have converged to the solution

$$x = [-1.1980 \ -0.8027 \ -1.0259 \ -1.0496]^T$$

From Equation (2.47) and using the initial vector $x = [0\ 0\ 0\ 0]$, the following updates are obtained for the Jacobi method:

k	x_1	x_2	x_3	x_4
1	0.0000	0.0000	0.0000	0.0000
2	−0.1000	−0.2222	−0.2500	−0.5000
3	−0.5194	−0.4722	−0.4611	−0.5653
4	−0.6719	−0.5247	−0.6669	−0.7538
5	−0.8573	−0.6314	−0.7500	−0.8176
6	−0.9418	−0.6689	−0.8448	−0.9004
7	−1.0275	−0.7163	−0.8915	−0.9368
8	−1.0728	−0.7376	−0.9355	−0.9748
9	−1.1131	−0.7594	−0.9601	−0.9945
10	−1.1366	−0.7709	−0.9810	−1.0123
11	−1.1559	−0.7811	−0.9936	−1.0226
12	−1.1679	−0.7871	−1.0037	−1.0311
13	−1.1772	−0.7920	−1.0100	−1.0363
14	−1.1832	−0.7950	−1.0149	−1.0404
15	−1.1877	−0.7974	−1.0181	−1.0431
16	−1.1908	−0.7989	−1.0205	−1.0451
17	−1.1930	−0.8001	−1.0221	−1.0464
18	−1.1945	−0.8009	−1.0233	−1.0474
19	−1.1956	−0.8014	−1.0241	−1.0480
20	−1.1963	−0.8018	−1.0247	−1.0485
21	−1.1969	−0.8021	−1.0250	−1.0489
22	−1.1972	−0.8023	−1.0253	−1.0491
23	−1.1975	−0.8024	−1.0255	−1.0492
24	−1.1977	−0.8025	−1.0257	−1.0494
25	−1.1978	−0.8026	−1.0258	−1.0494

The Jacobi iterates have converged to the same solution as the Gauss–Seidel method. The error in the iterates is shown in Figure 2.3 on a semilog scale, where the error is defined as the maximum $|(x_i^k - x_i^*)|$ for all $i = 1, \ldots, 4$. Both the Gauss–Seidel and the Jacobi methods exhibit *linear convergence*, but the Gauss–Seidel converges with a steeper slope and will therefore reach the convergence tolerance sooner for the same initial condition. ■

Example 2.7

Repeat Example 2.2 using the Jacobi iterative method.

Solution 2.7 Repeating the solution procedure of Example 2.6 yields the

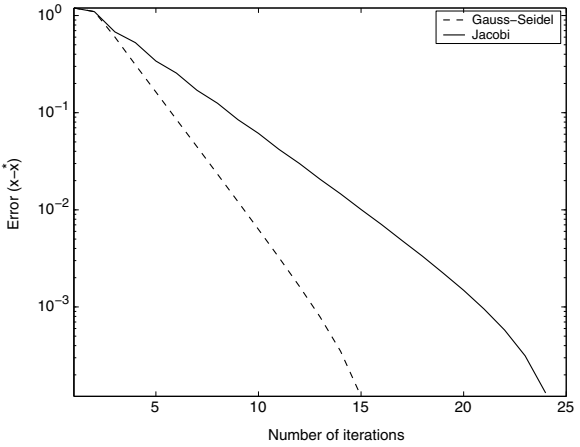

FIGURE 2.3
Convergence rates of the Gauss–Seidel and Jacobi methods

following iterations for the Jacobi method:

k	x_1	x_2	x_3	x_4
1	0	0	0	0
2	1.0000	1.0000	0.2000	0.2500
3	−4.8000	−2.1500	−1.6000	−2.8500
4	36.6500	22.3500	9.8900	14.9250
5	−225.0100	−136.8550	−66.4100	−110.6950

Obviously these iterates are not converging. To understand why they are diverging, consider the iterative matrix for the Jacobi matrix:

$$M_J = -D^{-1}(L+U) = \begin{bmatrix} 0.00 & -3.00 & -4.00 & -8.00 \\ -2.00 & 0.00 & -2.00 & -3.00 \\ -0.80 & -0.60 & 0.00 & -1.60 \\ -2.25 & -0.50 & -1.75 & 0.00 \end{bmatrix}$$

The eigenvalues of M_J are

$$\begin{bmatrix} -6.6212 \\ 4.3574 \\ 1.2072 \\ 1.0566 \end{bmatrix}$$

which are all greater than one and lie outside the unit circle. Therefore, the Jacobi method will not converge to the solution regardless of choice of initial condition and cannot be used to solve the system of Example 2.2. ■

If the largest eigenvalue of the iterative matrix M_J or M_{GS} is less than,

but almost, unity, then the convergence may proceed very slowly. In this case it is desirable to introduce a weighting factor ω that will improve the rate of convergence. From

$$x^{k+1} = -\left(L + D\right)^{-1}\left(Ux^k - b\right) \tag{2.54}$$

it follows that

$$x^{k+1} = x^k - D^{-1}\left(Lx^{k+1} + \left(D + U\right)x^k - b\right) \tag{2.55}$$

A new iterative method can be defined with the weighting factor ω such that

$$x^{k+1} = x^k - \omega D^{-1}\left(Lx^{k+1} + \left(D + U\right)x^k - b\right) \tag{2.56}$$

This method is known as the *successive overrelaxation (SOR)* method with relaxation coefficient $\omega > 0$. This method takes the form of a weighted average between the previous iterate and the computed Gauss–Seidel iterate successively for each component. Note that, if the relaxation iterates converge, they converge to the solution $x^* = A^{-1}b$. One necessary condition for the SOR method to be convergent is that $0 < \omega < 2$ [29]. The idea is to choose ω such that the convergence of the iterates is accelerated. The calculation of the optimal value for ω is difficult, except in a few simple cases. The optimal value is usually determined through trial and error, but analysis shows that, for systems larger than $n = 30$, the optimal SOR can be more than forty times faster than the Jacobi method [29]. The improvement in the speed of convergence often increases as n increases.

If $\omega = 1$, the SOR method simplifies to the Gauss–Seidel method. In principle, given the spectral radius ρ of the Jacobi iteration matrix, ω can be determined

$$\omega_{opt} = \frac{2}{1 + \sqrt{1 - \rho^2}} \tag{2.57}$$

but this calculation is typically not computationally efficient.

One further modification can be added to the SOR method to achieve a *symmetric successive overrelaxation (SSOR)* method. This is essentially the combination of a forward SOR step combined with a backward SOR step:

$$x^{k+1} = B_1 B_2 x^k + \omega(2 - \omega)(D - \omega U)^{-1} D (D - \omega L)^{-1} b \tag{2.58}$$

where

$$B_1 = (D - \omega U)^{-1}(\omega L + (1 - \omega)D) \tag{2.59}$$

$$B_2 = (D - \omega L)^{-1}(\omega U + (1 - \omega)D) \tag{2.60}$$

This method is analogous to a two-step SOR applied to a symmetric matrix. In the first step, the unknowns are updated in forward order, and in the second step, or backward step, the unknowns are updated in reverse order.

2.5 Conjugate Gradient Methods

In this section, the Krylov subspace method is described for solving linear systems. The conjugate gradient method is a nonstationary iterative method. Nonstationary methods differ from stationary methods in that the computations do not use an iteration matrix and typically involve information that changes at each iteration. Krylov subspace methods define a space such that the kth Krylov subspace $\mathcal{K}_k$ is

$$\mathcal{K}_k = \text{span}\left(r_0, Ar_0, \ldots, A^{k-1}r_0\right) \text{ for } k \geq 1 \tag{2.61}$$

where the residual is defined asG

$$r_k = b - Ax_k \tag{2.62}$$

Krylov subspace methods generate vector sequences of iterates (i.e., successive approximations to the solution), residuals corresponding to the iterates, and search directions used in updating the iterates and residuals.

A common Krylov iterative method for solving $Ax = b$ is the *conjugate gradient* method. The conjugate gradient (CG) method was originally intended as a direct method, but has been adopted as an iterative method and has generally superseded the Jacobi–Gauss–Seidel–SOR family of methods [27].

The conjugate gradient method is intended to solve symmetric positive definite systems of equations. Recall that the matrix A is symmetric positive definite if $A = A^T$ and

$$x^T Ax > 0 \text{ for all } x \neq 0$$

The CG method can be considered a minimization method for the function

$$E(x) = \|Ax - b\|^2 \tag{2.63}$$

along a succession of rays. Therefore, the kth iterate x_k of the CG iteration minimizes

$$\phi(x) = \frac{1}{2}x^T Ax - x^T b \tag{2.64}$$

over $x_0 + \mathcal{K}_k$.

One attractive feature of this method is that it is guaranteed to converge in at most n steps (neglecting round-off error) if the A matrix is positive definite. The conjugate gradient method is most frequently used instead of Gaussian elimination if the A matrix is very large and sparse, in which case the solution may be obtained in less than n steps. This is especially true if the A matrix is well conditioned. If the matrix is ill conditioned, then round-off errors may prevent the algorithm from obtaining a sufficiently accurate solution after n steps.

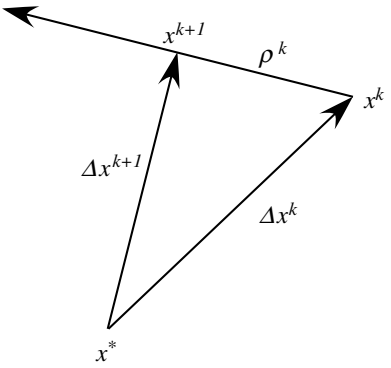

FIGURE 2.4
The conjugate gradient method

In the conjugate gradient method, a succession of search directions ρ_k is employed and a parameter α_k is computed such that $f\left(x^k - \alpha_k \rho_k\right)$ is minimized along the ρ_k direction. Upon setting x^{k+1} equal to $x^k - \alpha_k \rho_k$, the new search direction is found. As the conjugate gradient method progresses, each error function is associated with a specific ray, or orthogonal expansion. Therefore, the conjugate gradient method is reduced to the process of generating the orthogonal vectors and finding the proper coefficients to represent the desired solution. The conjugate gradient method is illustrated in Figure 2.4. Let $x^\star$ denote the exact (but unknown) solution, x^k an approximate solution, and $\Delta x^k = x^k - x^\star$. Given any search direction ρ^k, the minimal distance from the line to $x^\star$ is found by constructing Δx^{k+1} perpendicular to x^k. Since the exact solution is unknown, the residual is made to be perpendicular to ρ^k. Regardless of how the new search direction is chosen, the norm of the residual will not increase.

All Krylov iterative methods for solving $Ax = b$ define an iterative process such that

$$x^{k+1} = x^k + \alpha_{k+1} \rho_{k+1} \tag{2.65}$$

where x^{k+1} is the updated value, α_k is the step length, and ρ_k defines the direction $\in R^n$ in which the algorithm moves to update the estimate.

Let the residual, or mismatch, vector at step k be given by

$$r_k = Ax^k - b \tag{2.66}$$

and the error function given by

$$E_k\left(x^k\right) = \left\|Ax^k - b\right\|^2 \tag{2.67}$$

Once the search direction ρ_{k+1} is determined, α_{k+1} can be computed from

the minimization property of the iteration, in which

$$\frac{d\phi\left(x_k + \alpha\rho_{k+1}\right)}{d\alpha} = 0 \tag{2.68}$$

for $\alpha = \alpha_{k+1}$. Equation (2.68) can be written as

$$\rho_{k+1}^T A x_k + \alpha\rho_{k+1}^T A\rho_{k+1} - \rho_{k+1}^T b = 0 \tag{2.69}$$

Then the coefficient that minimizes the error function at step $k+1$ is

$$\begin{aligned}\alpha_{k+1} &= \frac{\rho_{k+1}^T\left(b - Ax_k\right)}{\rho_{k+1}^T A\rho_{k+1}} \\ &= \frac{\rho_{k+1}^T r_k}{\rho_{k+1}^T A\rho_{k+1}} \\ &= \frac{\left\|A^T r_k\right\|^2}{\left\|A\rho_{k+1}\right\|^2}\end{aligned}$$

This has the geometric interpretation of minimizing E_{k+1} along the ray defined by ρ_{k+1}. Further, an improved algorithm is one that seeks the minimum of E_{k+1} in a plane spanned by two direction vectors, such that

$$x^{k+1} = x^k + \alpha_{k+1}\left(\rho_{k+1} + \beta_{k+1}\sigma_{k+1}\right) \tag{2.70}$$

where the rays ρ_{k+1} and σ_{k+1} span a plane in R^n. The process of selecting direction vectors and coefficients to minimize the error function E_{k+1} is optimized when the chosen vectors are orthogonal, such that

$$\langle A\rho_{k+1}, A\sigma_{k+1}\rangle = 0 \tag{2.71}$$

where $\langle\cdot\rangle$ denotes inner product. Vectors that satisfy the orthogonality condition of Equation (2.71) are said to be mutually conjugate with respect to the operator $A^T A$, where A^T is the conjugate transpose of A. One method of choosing appropriate vectors is to choose σ_{k+1} as a vector orthogonal to ρ_k, thus eliminating the need to specify two orthogonal vectors at each step. While this simplifies the procedure, there is now an implicit recursive dependence for generating the ρ vectors.

Conjugate Gradient Algorithm for Solving $Ax = b$

Initialization: Let $k = 1$, and

$$r_0 = Ax^0 - b \tag{2.72}$$

$$\rho_0 = \|r_0\|^2 \tag{2.73}$$

While $\|r_k\| \geq \varepsilon$,

$$\sigma = r_{k-1} \text{ if } k-1 = 0, \text{ else } \beta = \frac{\rho_{k-1}}{\rho_{k-2}} \text{ and } \sigma = r_{k-1} + \beta\sigma \tag{2.74}$$

$$w = A\sigma \tag{2.75}$$
$$\alpha = \frac{\rho_{k-1}}{\sigma^T w} \tag{2.76}$$
$$x = x + \alpha\sigma \tag{2.77}$$
$$r_k = r_{k-1} - \alpha w \tag{2.78}$$
$$\rho_k = \|r_k\|^2 \tag{2.79}$$
$$k = k + 1 \tag{2.80}$$

For an arbitrary symmetric positive definite matrix A, the conjugate gradient method will produce a solution in at most n steps (neglecting round-off error). This is a direct consequence of the fact that the n direction vectors $\rho_0, \rho_1, \ldots$ span the solution space. Finite step termination is a significant advantage of the conjugate gradient method over other iterative methods such as relaxation methods.

Note that the matrix A itself need not be formed or stored. The only requirement is a routine for producing matrix-vector products. Krylov space methods are often called *matrix-free* methods for this reason. Each iteration requires only a single matrix-vector product (to compute $w = A\sigma$) and two scalar products ($\sigma^T w$ and $\|r_k\|^2$).

Example 2.8
Use the conjugate gradient method to solve the following system of equations:

$$\begin{bmatrix} 22 & 2 & 8 & 5 & 3 \\ 2 & 19 & 6 & 6 & 2 \\ 8 & 6 & 27 & 7 & 4 \\ 5 & 6 & 7 & 24 & 0 \\ 3 & 2 & 4 & 0 & 9 \end{bmatrix} x = \begin{bmatrix} 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{bmatrix}$$

with

$$x^0 = [0\ 0\ 0\ 0\ 0]$$

Solution 2.8 It can be easily verified that the matrix A is symmetric and positive definite.

Following the order of the conjugate gradient algorithm yields

Initialization

$$r_0 = [1\ 2\ 3\ 4\ 5]^T$$
$$\rho_0 = 55$$

k=1

$$\sigma = [1\ 2\ 3\ 4\ 5]^T$$

$$w = \begin{bmatrix} 85 \\ 92 \\ 149 \\ 134 \\ 64 \end{bmatrix}$$

$$\alpha = 0.0350$$

$$x^1 = \begin{bmatrix} 0.0350 \\ 0.0700 \\ 0.1050 \\ 0.1399 \\ 0.1749 \end{bmatrix}$$

$$r_1 = \begin{bmatrix} -1.9739 \\ -1.2188 \\ -2.2131 \\ -0.6883 \\ 2.7608 \end{bmatrix}$$

$$\rho_1 = 18.3756$$

$$\|r_1\| = 4.2867$$

k=2

$$\beta = 0.3341$$

$$\sigma = \begin{bmatrix} -1.6398 \\ -0.5506 \\ -1.2108 \\ 0.6481 \\ 4.4313 \end{bmatrix}$$

$$w = \begin{bmatrix} -30.3291 \\ -8.2550 \\ -26.8518 \\ -4.4238 \\ 29.0180 \end{bmatrix}$$

$$\alpha = 0.0865$$

$$x^2 = \begin{bmatrix} -0.1068 \\ 0.0224 \\ 0.0003 \\ 0.1960 \\ 0.5581 \end{bmatrix}$$

$$r_2 = \begin{bmatrix} 0.6486 \\ -0.5050 \\ 0.1087 \\ -0.3058 \\ 0.2517 \end{bmatrix}$$

$$\rho_2 = 0.8444$$

$$\|r_2\| = 0.9189$$

Similarly

k	3	4	5
β_k	0.0460	0.0041	0.0001
σ_k	$\begin{bmatrix} 0.5732 \\ -0.5303 \\ 0.0531 \\ -0.2760 \\ 0.4553 \end{bmatrix}$	$\begin{bmatrix} 0.0212 \\ -0.0146 \\ -0.0346 \\ 0.0389 \\ -0.0081 \end{bmatrix}$	$\begin{bmatrix} 0.0002 \\ 0.0003 \\ -0.0002 \\ -0.0001 \\ 0.0001 \end{bmatrix}$
w	$\begin{bmatrix} 11.9610 \\ -9.3567 \\ 2.7264 \\ -6.5682 \\ 4.9692 \end{bmatrix}$	$\begin{bmatrix} 0.3306 \\ -0.2250 \\ -0.6121 \\ 0.7098 \\ -0.1767 \end{bmatrix}$	$\begin{bmatrix} 0.0024 \\ 0.0042 \\ -0.0026 \\ -0.0019 \\ 0.0010 \end{bmatrix}$
α	0.0526	0.0566	0.0713
x^k	$\begin{bmatrix} -0.0766 \\ -0.0056 \\ 0.0031 \\ 0.1815 \\ 0.5821 \end{bmatrix}$	$\begin{bmatrix} -0.0754 \\ -0.0064 \\ 0.0011 \\ 0.1837 \\ 0.5816 \end{bmatrix}$	$\begin{bmatrix} -0.0754 \\ -0.0064 \\ 0.0011 \\ 0.1837 \\ 0.5816 \end{bmatrix}$
ρ_k	0.0034	0.0000	0.0000
r_k	$\begin{bmatrix} 0.0189 \\ -0.0124 \\ -0.0348 \\ 0.0400 \\ -0.0099 \end{bmatrix}$	$\begin{bmatrix} 0.0002 \\ 0.0003 \\ -0.0002 \\ -0.0001 \\ 0.0001 \end{bmatrix}$	$\begin{bmatrix} -0.0000 \\ -0.0000 \\ -0.0000 \\ -0.0000 \\ -0.0000 \end{bmatrix}$
$\|r_k\|$	0.0585	0.0004	0.0000

The iterations converged in five iterations, as the algorithm guarantees. Note that, depending on the size of the convergence criterion ϵ, it is possible to have convergence after four iterations, as the norm of $r_4 = 0.0004$ is relatively small at the conclusion of the fourth iteration. ■

The conjugate gradient method is more numerically competitive for matrices that are very large and sparse or that have a special structure that cannot be easily handled by LU factorization. In some cases, the speed of convergence of the conjugate gradient method can be improved by *preconditioning*. As seen with the Gauss–Seidel and Jacobi iteration, the convergence rate of iterative algorithms is closely related to the eigenvalue spectrum of the iterative matrix. Consequently, a scaling or matrix transformation that converts

the original system of equations into one with a better eigenvalue spectrum may significantly improve the rate of convergence. This procedure is known as preconditioning and is discussed in Section 2.7.

2.6 Generalized Minimal Residual Algorithm

If the matrix A is neither symmetric nor positive definite, then the term

$$\langle A\rho_{k+1}, A\sigma_{k+1}\rangle$$

is not guaranteed to be zero and the search vectors are not mutually orthogonal. Mutual orthogonality is required to generate a basis of the solution space. Hence this basis must be explicitly constructed. The extension of the conjugate gradient method, called the generalized minimal residual algorithm (GMRES), minimizes the norm of the residual in a subspace spanned by the set of vectors

$$r^0,\ Ar^0,\ A^2r^0,\ \ldots,\ A^{k-1}r^0$$

where vector r^0 is the initial residual $r^0 = \|b - Ax^0\|$, and the kth approximation to the solution is chosen from this space. This subspace, a *Krylov subspace*, is made orthogonal by the well-known Gram–Schmidt procedure, known as the Arnoldi process when applied to a Krylov subspace [41]. At each step k, the GMRES algorithm applies the Arnoldi process to a set of k orthonormal basis vectors for the kth Krylov subspace to generate the next basis vector. Arnoldi methods are described in greater detail in Section 7.3.

The kth iteration of the GMRES method is the solution to the least squares problem

$$\text{minimize}_{x\in x_0+\mathcal{K}_k}\ \|b - Ax\| \tag{2.81}$$

At each step, the algorithm multiplies the previous Arnoldi vector v_j by A and then orthonormalizes the resulting vector w_j against all previous v_i's. The columns $V = [v_1,\ v_2, \ldots, v_k]$ form an orthonormal basis for the Krylov subspace and H is the orthogonal projection of A onto this space.

An orthogonal matrix triangularization such as the Arnoldi method consists in determining an $n \times n$ orthogonal matrix Q such that

$$Q^T = \begin{bmatrix} R \\ 0 \end{bmatrix} \tag{2.82}$$

where R is an $m \times m$ upper triangular matrix R. Then the solution process reduces to solving the triangular system $Rx = Py$, where P consists of the first m rows of Q.

To clear one element at a time to upper triangularize a matrix, Given's rotation can be applied. Given's rotation is a transformation based on the matrix

$$G_{jk} = \begin{bmatrix} 1 \ldots & 0 & \ldots & 0 & \ldots & 0 \\ \vdots & \ddots & \vdots & & & \vdots \\ 0 \ldots & \text{cs} & \ldots & \text{sn} & \ldots & 0 \\ \vdots & & \vdots & \ddots & & \vdots \\ 0 \ldots & -\text{sn} & \ldots & \text{cs} & \ldots & 0 \\ \vdots & & \vdots & & \ddots & \vdots \\ 0 \ldots & 0 & \ldots & 0 & \ldots & 1 \end{bmatrix} \tag{2.83}$$

where properly chosen $\text{cs} = \cos(\phi)$ and $\text{sn} = \sin(\phi)$ (for some rotation angle ϕ) can be used to zero out the element A_{kj}. The orthogonal matrix G_{jk} rotates the vector $(c, -s)$, which makes an angle of $-\phi$ with the x-axis, such that it overlaps the x-axis. Given's rotations are used to remove single nonzero elements of matrices in reduction to triangular form.

One of the difficulties with the GMRES methods is that, as k increases, the number of vectors requiring storage increases as k and the number of multiplications as $\frac{1}{2}k^2n$ (for an $n \times n$ matrix). To remedy this difficulty, the algorithm can be applied iteratively, i.e., it can be restarted every m steps, where m is some fixed integer parameter.

GMRES Algorithm for Solving $Ax = b$

Initialization:

$$\begin{aligned} r_0 &= b - Ax^0 \\ e_1 &= [1\ 0\ 0\ \ldots\ 0]^T \\ v_1 &= r_0/\|r_0\| \\ s &= \|r_0\|e_1 \\ k &= 1 \\ \text{cs} &= [0\ 0\ 0\ \ldots\ 0]^T \\ \text{sn} &= [0\ 0\ 0\ \ldots\ 0]^T \end{aligned}$$

While $\|r_k\| \geq \varepsilon$ and $k \leq k_{\max}$, set

1. $H(j,k) = (Av_k)^T v_j, \quad j = 1, \ldots, k$
2. $v_{k+1} = Av_k - \sum_{j=1}^{k} H(j,k)v_j$
3. $H(k+1,k) = \|v_{k+1}\|$
4. $v_{k+1} = v_{k+1}/\|v_{k+1}\|$
5. Givens rotation:

(a)

$$\begin{bmatrix} H(j,k) \\ H(j+1,k) \end{bmatrix} = \begin{bmatrix} \text{cs}(j) & \text{sn}(j) \\ -\text{sn}(j) & \text{cs}(j) \end{bmatrix} \begin{bmatrix} H(j,k) \\ H(j+1,k) \end{bmatrix}, \quad j = 1, \ldots, k-1$$

(b)

$$\text{cs}(k) = \frac{H(k,k)}{\sqrt{H(k+1,k)^2 + H(k,k)^2}}$$
$$\text{sn}(k) = \frac{H(k+1,k)}{\sqrt{H(k+1,k)^2 + H(k,k)^2}}$$

(c) Approximate residual norm

$$\alpha = \text{cs}(k)s(k)$$
$$s(k+1) = -\text{sn}(k)s(k)$$
$$s(k) = \alpha$$
$$\text{error} = |s(k+1)|$$

(d) Set

$$H(k,k) = \text{cs}(k)H(k,k) + \text{sn}(k)H(k+1,k)$$
$$H(k+1,k) = 0$$

6. If error $\leq \varepsilon$

   (a) Solve $Hy = s$ for y

   (b) Calculate $x = x - Vy$

   (c) Method has converged. Return.

   Otherwise $k = k + 1$

Example 2.9
Repeat Example 2.6 using the GMRES method.

Solution 2.9 The problem of Example 2.6 is repeated here for convenience. Solve

$$\begin{bmatrix} -10 & 2 & 3 & 6 \\ 0 & -9 & 1 & 4 \\ 2 & 6 & -12 & 2 \\ 3 & 1 & 0 & -8 \end{bmatrix} x = \begin{bmatrix} 1 \\ 2 \\ 3 \\ 4 \end{bmatrix} \tag{2.84}$$

with $x^0 = [0\ 0\ 0\ 0]^T$. Let $\varepsilon = 10^{-3}$.

k=1 Solving the Arnoldi process yields

$$H = \begin{bmatrix} -4.0333 & 0 & 0 & 0 \\ 6.2369 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

$$V = \begin{bmatrix} 0.1826 & 0.9084 & 0 & 0 \\ 0.3651 & 0.2654 & 0 & 0 \\ 0.5477 & -0.0556 & 0 & 0 \\ 0.7303 & -0.3181 & 0 & 0 \end{bmatrix}$$

Applying Given's rotation:

At $k = 1$, the H matrix is not updated; therefore,

$$\text{cs}(k = 1) = -0.5430$$
$$\text{sn}(k = 1) = 0.8397$$

The new H matrix becomes

$$H = \begin{bmatrix} 7.4274 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

$$s = \begin{bmatrix} -2.9743 & -4.5993 & 0 & 0 \end{bmatrix}^T$$

Since error ($= |s(2)| = 4.5993$) is greater than ε, $k = k + 1$ and repeat.

k=2 Solving the Arnoldi process yields

$$H = \begin{bmatrix} 7.4274 & 2.6293 & 0 & 0 \\ 0 & -12.5947 & 0 & 0 \\ 0 & 1.9321 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

$$V = \begin{bmatrix} 0.1826 & 0.9084 & 0.1721 & 0 \\ 0.3651 & 0.2654 & -0.6905 & 0 \\ 0.5477 & -0.0556 & 0.6728 & 0 \\ 0.7303 & -0.3181 & -0.2024 & 0 \end{bmatrix}$$

Applying Given's rotation yields

$$H = \begin{bmatrix} 7.4274 & -12.0037 & 0 & 0 \\ 0 & 4.6314 & 0 & 0 \\ 0 & 1.9321 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

$$\text{cs}(k=2) = 0.9229$$
$$\text{sn}(k=2) = 0.3850$$

Updating H

$$H = \begin{bmatrix} 7.4274 & -12.0037 & 0 & 0 \\ 0 & 5.0183 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$
$$s = \begin{bmatrix} -2.9743 & -4.2447 & 1.7708 & 0 \end{bmatrix}^T$$

Since error ($= |s(3)| = 1.7708$) is greater than ε, $k = k + 1$ and repeat.

k=3 Solving the Arnoldi process yields

$$H = \begin{bmatrix} 7.4274 & -12.0037 & -3.8697 & 0 \\ 0 & 5.0183 & -0.2507 & 0 \\ 0 & 0 & -13.1444 & 0 \\ 0 & 0 & 2.6872 & 0 \end{bmatrix}$$
$$V = \begin{bmatrix} 0.1826 & 0.9084 & 0.1721 & 0.3343 \\ 0.3651 & 0.2654 & -0.6905 & -0.5652 \\ 0.5477 & -0.0556 & 0.6728 & -0.4942 \\ 0.7303 & -0.3181 & -0.2024 & 0.5697 \end{bmatrix}$$

Applying Given's rotation yields

$$H = \begin{bmatrix} 7.4274 & -12.0037 & 1.8908 & 0 \\ 0 & 5.0183 & -1.9362 & 0 \\ 0 & 0 & -13.4346 & 0 \\ 0 & 0 & 2.6872 & 0 \end{bmatrix}$$

$$\text{cs}(k=3) = -0.9806$$
$$\text{sn}(k=3) = 0.1961$$

$$H = \begin{bmatrix} 7.4274 & -12.0037 & 1.8908 & 0 \\ 0 & 5.0183 & -1.9362 & 0 \\ 0 & 0 & 13.7007 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$
$$s = \begin{bmatrix} -2.9743 & -4.2447 & -1.7364 & -0.3473 \end{bmatrix}^T$$

Since error ($= |s(4)| = 0.3473$) is greater than ε, $k = k + 1$ and repeat.

k=4 Solving the Arnoldi process yields

$$H = \begin{bmatrix} 7.4274 & -12.0037 & 1.8908 & 1.4182 \\ 0 & 5.0183 & -1.9362 & 0.5863 \\ 0 & 0 & 13.7007 & -1.4228 \\ 0 & 0 & 0 & -9.2276 \end{bmatrix}$$

$$V = \begin{bmatrix} -0.1826 & -0.9084 & -0.1721 & -0.3343 & 0.7404 \\ -0.3651 & -0.2654 & 0.6905 & 0.5652 & 0.2468 \\ -0.5477 & 0.0556 & -0.6728 & 0.4942 & 0.6032 \\ -0.7303 & 0.3181 & 0.2024 & -0.5697 & 0.1645 \end{bmatrix}$$

Applying Given's rotation yields

$$\begin{aligned} \text{cs}(k=4) &= 1.0000 \\ \text{sn}(k=4) &= 0.0000 \end{aligned}$$

$$H = \begin{bmatrix} 7.4274 & -12.0037 & 1.8908 & -0.2778 \\ 0 & 5.0183 & -1.9362 & -1.9407 \\ 0 & 0 & 13.7007 & -1.0920 \\ 0 & 0 & 0 & 9.1919 \end{bmatrix}$$

$$s = \begin{bmatrix} -2.9743 & -4.2447 & -1.7364 & -0.3473 & 0.0000 \end{bmatrix}^T$$

Since error $(= |s(5)| = 0)$, the iteration has converged.

Solving for y from $Hy = s$ yields

$$y = \begin{bmatrix} -1.8404 \\ -0.9105 \\ -0.1297 \\ -0.0378 \end{bmatrix} \tag{2.85}$$

Note that, since H is upper triangular, y can be found quickly using forward elimination.
Solving for x from

$$x = x - Vy \tag{2.86}$$

yields

$$x = \begin{bmatrix} -1.1981 \\ -0.8027 \\ -1.0260 \\ -1.0496 \end{bmatrix} \tag{2.87}$$

which is the same as the previous example. ■

2.7 Preconditioners for Iterative Methods

As noted previously, the rate of convergence of iterative methods depends on the condition number of the iteration matrix. Therefore, it may be advantageous to transform the linear system of equations into an equivalent system (i.e., one that has the same solution) that has more favorable properties. The matrix, or matrices, that perform this transformation are called *preconditioners.*

For example, for any nonsingular preconditioner matrix M, the transformed system

$$M^{-1}Ax = M^{-1}b \tag{2.88}$$

has the same solution as the original system $Ax = b$. Furthermore, if M is chosen such that it approximates A, then the resulting condition number of M^1A is much improved. There is, of course, a trade-off between the reduction in computation achieved by the improved convergence properties and the amount of computation required to find the matrix M^{-1}.

In addition, note that, even if A is symmetric, M^1A will most likely be nonsymmetric, and methods that require symmetry (such as the conjugate gradient method) will fail to converge. For this reason, the matrix M is often split such that

$$M = M_1M_2 \tag{2.89}$$

The preconditioned system becomes

$$M_1^{-1}AM_2^{-1}\left(M_2x\right) = M_1^{-1} \tag{2.90}$$

The matrix M_1 is called the *left preconditioner* and M_2 is the *right preconditioner.* If $M_1^T = M_2$, then $M_1^{-1}AM_2^{-1}$ is symmetric. This leads directly to the ability to transform any iterative method into an equivalent iterative method by replacing the initial residual r_0 by $M_1^{-1}r_0$ and replacing the final solution x_k by $M_2^{-1}x_k$.

2.7.1 Jacobi

Just as stationary methods are based on a splitting of the A matrix such that $A = L + U + D$, preconditioners for stationary methods can be based on the splitting matrices. The simplest preconditioner consists of just the diagonal D of the A matrix:

$$m_{i,j}\begin{cases} a_{i,i} \text{ if } i = j \\ 0 \quad \text{otherwise} \end{cases} \tag{2.91}$$

This is known as the (point) Jacobi preconditioner. It can be split such that $m_1(i,i) = m_2(i,i) = \sqrt{a_{i,i}}$. The inverses of the matrices are straightforward to calculate. In the interest of computational efficiency, the inverse of these matrices is usually stored so that repeated divisions are not required.

The Jacobi preconditioner can be easily generalized to the *block-Jacobi* preconditioner in which small blocks along the diagonal are kept rather than individual points. Block preconditioning is useful in cases where the A matrix has blocks of coupled variables with sparse interconnections between blocks.

2.7.2 Symmetric Successive Overrelaxation

A preconditioner matrix based on the SSOR method can be defined

$$M = \frac{1}{\omega\,(2-\omega)}(D+\omega L)D^{-1}(D+\omega U) \tag{2.92}$$

The constant coefficient $\frac{1}{\omega(2-\omega)}$ only has the effect of scaling the equations of the preconditioned system. Furthermore, if this M matrix is used as a preconditioner, it is not necessary to choose ω as carefully as for the underlying fixed-point iteration.

2.7.3 Symmetric Gauss–Seidel

Taking $\omega = 1$ in the SSOR preconditioner leads to the symmetric Gauss–Seidel iteration

$$M = (D+L)D^{-1}(D+U) \tag{2.93}$$

Note that the preconditioned systems (2.88) or (2.90) may be a full system even if the original A matrix was sparse due to M^{-1}. Even if M is sparse, M^{-1} may be full. Trying to find M^{-1} directly may adversely impact the computational efficiency of the solution method. This must be taken into account when selecting a particular solution technique. Quite often, recasting the problem as a series of matrix-vector products can replace finding the inverse directly.

2.7.4 Incomplete LU Factorization

Since it is desired to find a preconditioner matrix M^{-1} that approximates A^{-1}, it may be reasonable to combine direct methods and iterative methods to derive the preconditioner matrix. LU factorization produces matrices L and U from which

$$U^{-1}L^{-1} = A^{-1} \tag{2.94}$$

Since L and U are lower and upper triangular, respectively, they may be computationally more efficient. An LU factorization is called *incomplete* if, during the factorization process, one or more fill elements are ignored. This has the impact of maintaining the underlying sparsity of the original matrix.

If all of the fills that result from the factorization process are neglected, this is referred to as the ILU(0) case, corresponding to incomplete LU factorization of level 0.

Example 2.10
Repeat Example 2.5 using the ILU(0) method. Compare the resulting Q matrix for both cases.

Solution 2.10 The incomplete LU factorization method results in the factors shown below. Note their similarity to the LU factors of Example 2.5.

$$Q = \begin{bmatrix} 10.0000 & 0.1000 & 0 & 0.3000 & 0 & 0 & 0 & 0.5000 & 0 & 0 \\ 2.0000 & 8.8000 & 0 & 0 & 0 & 0 & 0.5682 & 0 & 0 & 0.2273 \\ 0 & 0 & 21.0000 & 0.2381 & 0.3333 & 0 & 0 & 0 & 0 & 0.1905 \\ 4.0000 & 0 & 1.0000 & 16.5619 & 0.4629 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 4.0000 & 6.0476 & 20.8672 & 0.1917 & 0.0479 & 0 & 0 & 0.0593 \\ 0 & 0 & 0 & 0 & 3.0000 & 13.4249 & 0.6597 & 0 & 0 & 0 \\ 0 & 1.0000 & 4.0000 & 0 & 0.6667 & 2.8722 & 9.5051 & 0.1052 & 0.1052 & 0 \\ 1.0000 & 0 & 5.0000 & 0 & 0 & 0 & 5.0000 & 8.9740 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 6.0000 & 0 & 19.3688 & 0 \\ 0 & 2.0000 & 3.0000 & 0 & 3.0000 & 0 & 0 & 0 & 0 & 33.7960 \end{bmatrix}$$

■

Modifications to the ILU method focus on different methods of identifying which fills to include in the LU factors. Several methods are presented in [42].

2.7.5 Graph Based

A recent development in preconditioners is to use the underlying graph structure of the matrix to drive the structure of the preconditioner M. A graph-based preconditioning method is designed to be applied to symmetric systems that can be represented by graphs [30]. In this approach, matrix A is represented by a graph $G = (V, E, \omega)$ with each nonzero entry of the matrix representing a connection in the power system. To increase the speed of computation, it is beneficial to reduce the density of A. Graph theory has a natural approach to accomplishing this by removing trivial edges in graphs through graph sparsification [30] [34].

The preconditioner is derived from the low-stretch spanning tree of the matrix. To find the low-stretch spanning tree, the system matrix A is represented by a graph with m representing the number of edges in the graph or nonzero entries in the matrix. The concept of stretch is critical for constructing a good spanning tree preconditioner [6]. In order to have a spanning tree that serves as an effective preconditioner, the off-tree edges must have an average stretch δ over a spanning tree in order for the spanning tree to be an $O(\delta m)$-approximation of the graph [30]. There exists a unique "detour" path in the tree between vertices u and v, for every edge $e(u, v)$. Stretch is defined as the distortion caused by the detour required by taking the tree path. The stretch of the edges in the tree is given by

$$\text{stretch}(e) = \frac{\sum_{i=1}^{k} w'(e_i)}{w'(e)} \tag{2.95}$$

The denominator of the stretch expression contains the distance between the vertices (u, v) in the graph, and the numerator sums the distances of the edges

along the unique path in the tree connecting the vertices (u, v), with distance equaling the inverse of the edge weight $w' = 1/e$.

The spanning tree generates a preconditioner with a provably small condition number between the original graph G and the approximate graph $\tilde{G}$, preserving the spectral properties of the system. The approximate graph $\tilde{G}$, contains a nearly linear number of edges that approximates the given graph derived from A. Most of the current progress in this area is focused on various methods to contruct the spanning trees [51] [28].

Example 2.11
Solve the system of Example 2.5 using the following preconditioners with the GMRES method.

1. Jacobi
2. SSOR with $\omega = 1.0$
3. ILU(0)

Let $b = [1 \quad 1 \quad \ldots 1]^T$ and x^0 be the zero vector. Compare the number of iterations required in each method. Use a convergence tolerance of 10^{-5}.

Solution 2.5 The matrix A is repeated here for convenience.

$$A = \begin{bmatrix} 10 & 1 & 0 & 3 & 0 & 0 & 0 & 5 & 0 & 0 \\ 2 & 9 & 0 & 0 & 0 & 0 & 5 & 0 & 0 & 2 \\ 0 & 0 & 21 & 5 & 7 & 0 & 0 & 0 & 0 & 4 \\ 4 & 0 & 1 & 18 & 8 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 4 & 7 & 25 & 4 & 1 & 0 & 0 & 2 \\ 0 & 0 & 0 & 0 & 3 & 14 & 9 & 0 & 0 & 0 \\ 0 & 1 & 4 & 0 & 2 & 3 & 12 & 1 & 1 & 0 \\ 1 & 0 & 5 & 0 & 0 & 0 & 5 & 10 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 6 & 0 & 20 & 0 \\ 0 & 2 & 3 & 0 & 4 & 0 & 0 & 0 & 0 & 35 \end{bmatrix}$$

1. For the Jacobi preconditioner, M will be the diagonal of the A matrix.

$$M = D = \begin{bmatrix} 10 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 9 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 21 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 18 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 25 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 14 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 12 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 10 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 20 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 35 \end{bmatrix}$$

2. For the SSOR preconditioner with $\omega = 1$,

$$M = (D + L)\, D^{-1}\, (D + U)$$

where

$$L = \begin{bmatrix} 0&0&0&0&0&0&0&0&0&0 \\ 2&0&0&0&0&0&0&0&0&0 \\ 0&0&0&0&0&0&0&0&0&0 \\ 4&0&1&0&0&0&0&0&0&0 \\ 0&0&4&7&0&0&0&0&0&0 \\ 0&0&0&0&3&0&0&0&0&0 \\ 0&1&4&0&2&3&0&0&0&0 \\ 1&0&5&0&0&0&5&0&0&0 \\ 0&0&0&0&0&0&6&0&0&0 \\ 0&2&3&0&4&0&0&0&0&0 \end{bmatrix} \text{ and } U = \begin{bmatrix} 0&1&0&3&0&0&0&5&0&0 \\ 0&0&0&0&0&0&0&5&0&0&2 \\ 0&0&0&5&7&0&0&0&0&4 \\ 0&0&0&0&8&0&0&0&0&0 \\ 0&0&0&0&0&0&4&1&0&0&2 \\ 0&0&0&0&0&0&0&9&0&0&0 \\ 0&0&0&0&0&0&0&0&1&1&0 \\ 0&0&0&0&0&0&0&0&0&0&0 \\ 0&0&0&0&0&0&0&0&0&0&0 \\ 0&0&0&0&0&0&0&0&0&0&0 \end{bmatrix}$$

with

$$M = \begin{bmatrix} 10.0000 & 1.0000 & 0 & 3.0000 & 0 & 0 & 0 & 5.0000 & 0 & 0 \\ 2.0000 & 9.2000 & 0 & 0.6000 & 0 & 0 & 5.0000 & 1.0000 & 0 & 2.0000 \\ 0 & 0 & 21.0000 & 5.0000 & 7.0000 & 0 & 0 & 0 & 0 & 4.0000 \\ 4.0000 & 0.4000 & 1.0000 & 19.4381 & 8.3333 & 0 & 0 & 2.0000 & 0 & 0.1905 \\ 0 & 0 & 4.0000 & 7.9524 & 29.4444 & 4.0000 & 1.0000 & 0 & 0 & 2.7619 \\ 0 & 0 & 0 & 0 & 3.0000 & 14.4800 & 9.1200 & 0 & 0 & 0.2400 \\ 0 & 1.0000 & 4.0000 & 0.9524 & 3.3333 & 3.3200 & 14.5641 & 1.0000 & 1.0000 & 1.1441 \\ 1.0000 & 0.1000 & 5.0000 & 1.4905 & 1.6667 & 0 & 5.0000 & 10.9167 & 0.4167 & 0.9524 \\ 0 & 0 & 0 & 0 & 0 & 0 & 6.0000 & 0.5000 & 20.5000 & 0 \\ 0 & 2.0000 & 3.0000 & 0.7143 & 5.0000 & 0.6400 & 1.2711 & 0 & 0 & 36.3359 \end{bmatrix}$$

3. The incomplete LU factorization method results in the pseudo LU factors:

$$L = \begin{bmatrix} 10.0000 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 2.0000 & 8.8000 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 21.0000 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 4.0000 & 0 & 1.0000 & 16.5619 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 4.0000 & 6.0476 & 20.8672 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 3.0000 & 13.4249 & 0 & 0 & 0 & 0 \\ 0 & 1.0000 & 4.0000 & 0 & 0.6667 & 2.8722 & 9.5051 & 0 & 0 & 0 \\ 1.0000 & 0 & 5.0000 & 0 & 0 & 0 & 5.0000 & 8.9740 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 6.0000 & 0 & 19.3688 & 0 \\ 0 & 2.0000 & 3.0000 & 0 & 3.0000 & 0 & 0 & 0 & 0 & 33.7960 \end{bmatrix}$$

$$U = \begin{bmatrix} 1.0000 & 0.1000 & 0 & 0.3000 & 0 & 0 & 0 & 0.5000 & 0 & 0 \\ 0 & 1.0000 & 0 & 0 & 0 & 0 & 0.5682 & 0 & 0 & 0.2273 \\ 0 & 0 & 1.0000 & 0.2381 & 0.3333 & 0 & 0 & 0 & 0 & 0.1905 \\ 0 & 0 & 0 & 1.0000 & 0.4629 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1.0000 & 0.1917 & 0.0479 & 0 & 0 & 0.0593 \\ 0 & 0 & 0 & 0 & 0 & 1.0000 & 0.6597 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1.0000 & 0.1052 & 0.1052 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1.0000 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1.0000 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1.0000 \end{bmatrix}$$

and $M = LU$:

$$M = \begin{bmatrix} 10.0000 & 1.0000 & 0 & 3.0000 & 0 & 0 & 0 & 5.0000 & 0 & 0 \\ 2.0000 & 9.0000 & 0 & 0.6000 & 0 & 0 & 5.0000 & 1.0000 & 0 & 2.0000 \\ 0 & 0 & 21.0000 & 5.0000 & 7.0000 & 0 & 0 & 0 & 0 & 4.0000 \\ 4.0000 & 0.4000 & 1.0000 & 18.0000 & 8.0000 & 0 & 0 & 2.0000 & 0 & 0.1905 \\ 0 & 0 & 4.0000 & 7.0000 & 25.0000 & 4.0000 & 1.0000 & 0 & 0 & 2.0000 \\ 0 & 0 & 0 & 0 & 3.0000 & 14.0000 & 9.0000 & 0 & 0 & 0.1780 \\ 0 & 1.0000 & 4.0000 & 0.9524 & 2.0000 & 3.0000 & 12.0000 & 1.0000 & 1.0000 & 1.0287 \\ 1.0000 & 0.1000 & 5.0000 & 1.4905 & 1.6667 & 0 & 5.0000 & 10.0000 & 0.5260 & 0.9524 \\ 0 & 0 & 0 & 0 & 0 & 0 & 6.0000 & 0.6312 & 20.0000 & 0 \\ 0 & 2.0000 & 3.0000 & 0.7143 & 4.0000 & 0.5751 & 1.2801 & 0 & 0 & 35.0000 \end{bmatrix}$$

The results of each method are summarized.

Method	No pre-conditioning	Jacobi	SSOR	ILU(0)
iterations	10	7	4	4
error	3.1623	0.2338	0.1319	0.1300
	0.9439	0.0124	0.0190	0.0165
	0.2788	0.0032	0.0022	0.0012
	0.0948	0.0011	0.0002	0.0001
	0.0332	0.0003		
	0.0056	0.0001		
	0.0018	0.0000		
	0.0005			
	0.0003			
	0.0000			

Note that, without preconditioning, the GMRES method requires the full $n = 10$ iterations to converge, but with preconditioning the number of iterations is reduced. The ILU(0) is the best method, as indicated by the smallest convergence error, but only slightly over the SSOR method. In this particular example, the choice of $\omega = 1$ led to the fewest number of SSOR iterations, but this is not always the case. ■

2.8 Problems

1. Show that the number of multiplications and divisions required in the LU factorization of an $n \times n$ square matrix is $n(n^2 - 1)/3$.

2. Consider the system $Ax = b$, where

$$a_{ij} = \frac{1}{i + j - 1} \quad i, j = 1, \ldots, 4$$

and

$$b_i = \frac{1}{3}\sum_{j=1}^{4} a_{ij}$$

Using only four decimal places of accuracy, solve this system using LU factorization with

(a) no pivoting

(b) partial pivoting

Comment on the differences in solutions (if any).

3. Prove that the matrix

$$A = \begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix}$$

does not have an LU factorization.

4. Assuming that an LU factorization of A is available, write an algorithm to solve the equation $x^T A = b^T$.

5. For the following matrix, find $A = LU$ (no pivoting) and $PA = LU$ (with partial pivoting)

(a)

$$A = \begin{bmatrix} 6 & -2 & 2 & 4 \\ 12 & -8 & 4 & 10 \\ 3 & -13 & 3 & 3 \\ -6 & 4 & 2 & -18 \end{bmatrix}$$

(b)

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

6. Write an LU factorization-based algorithm to find the inverse of any nonsingular matrix A.

7. Solve the system of Problem 5(a) with the vector

$$b = \begin{bmatrix} 1 \\ 1 \\ 1 \\ 1 \end{bmatrix}$$

(a) Using LU factorization and forward/backward substitution
(b) Using a Jacobi iteration. How many iterations are required?
(c) Using a Gauss–Seidel iteration. How many iterations are required?
(d) Using the conjugate gradient method. How many iterations are required?
(e) Using the GMRES method. How many iterations are required?

Use a starting vector of

$$x = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}$$

and a convergence error of 10^{-5} for the iterative methods.

8. Apply the Gauss–Seidel iteration to the system

$$A = \begin{bmatrix} 0.96326 & 0.81321 \\ 0.81321 & 0.68654 \end{bmatrix}$$

$$b = \begin{bmatrix} 0.88824 \\ 0.74988 \end{bmatrix}$$

Use $x^0 = [0.33116 \;\; 0.70000]^T$ and explain what happens.

9. Solve the system of equations in Problem 2 using the conjugate gradient method.

10. Solve the system of equations in Problem 2 using the GMRES method.

11. Consider an $n \times n$ tridiagonal matrix of the form

$$T_a = \begin{bmatrix} a & -1 & & & & \\ -1 & a & -1 & & & \\ & -1 & a & -1 & & \\ & & -1 & a & -1 & \\ & & & -1 & a & -1 \\ & & & & -1 & a \end{bmatrix}$$

where a is a real number.

(a) Verify that the eigenvalues of T_a are given by

$$\lambda_j = a - 2\cos(j\theta) \quad j = 1, \ldots, n$$

where

$$\theta = \frac{\pi}{n+1}$$

(b) Let $a = 2$.

i. Will the Jacobi iteration converge for this matrix?

ii. Will the Gauss–Seidel iteration converge for this matrix?

12. An alternative conjugate gradient algorithm for solving $Ax = b$ may be based on the error functional $E_k\left(x^k\right) = \langle x^k - x, x^k - x\rangle$ where $\langle\cdot\rangle$ denotes inner product. The solution is given as

$$x^{k+1} = x^k + \alpha_k \sigma_k$$

Using $\sigma_1 = -A^T r_0$ and $\sigma_{k+1} = -A^T r_k + \beta_k \sigma_k$, derive this conjugate gradient algorithm. The coefficients α_k and β_k can be expressed as

$$\alpha_{k+1} = \frac{\|r_k\|^2}{\|\sigma_{k+1}\|^2}$$

$$\beta_{k+1} = \frac{\|r_{k+1}\|^2}{\|r_k\|^2}$$

Repeat Example 2.8 using this conjugate gradient algorithm.

13. Write a subroutine with two inputs (A, flag) that will generate, for any nonsingular matrix A, the outputs (Q, P) such that if

- flag=0, $A = LU, P = I$
- flag=1, $PA = LU$

where

$$L = \begin{bmatrix} l_{11} & 0 & 0 & \dots & 0 \\ l_{21} & l_{22} & 0 & \dots & 0 \\ l_{31} & l_{32} & l_{33} & \dots & 0 \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ l_{n1} & l_{n2} & l_{n3} & \dots & l_{nn} \end{bmatrix} \quad \text{and} \quad U = \begin{bmatrix} 1 & u_{12} & u_{13} & \dots & u_{1n} \\ 0 & 1 & u_{23} & \dots & u_{2n} \\ 0 & 0 & 1 & \dots & u_{3n} \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ 0 & 0 & 0 & \dots & 1 \end{bmatrix}$$

and

$$Q = L + U - I$$

14. For the following nonsingular matrices, use the subroutine of Problem 13 and obtain matrices P and Q in each of the following cases:

(a)

$$\begin{bmatrix} 0 & 0 & 1 \\ 3 & 1 & 4 \\ 2 & 1 & 0 \end{bmatrix}$$

(b)

$$\begin{bmatrix} 10^{-10} & 0 & 0 & 1 \\ 0 & 0 & 1 & 4 \\ 0 & 2 & 1 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix}$$

15. Write a subroutine with two inputs (A, b) that will generate, for any nonsingular matrix A, the output (x) such that

$$Ax = b$$

using forward and backward substitution. This subroutine should incorporate the subroutine developed in Problem 13.

16. Using the subroutines of Problems 13 and 15, solve the following system of equations:

$$\begin{bmatrix} 2 & 5 & 6 & 11 \\ 4 & 6 & 8 & 2 \\ 4 & 3 & 7 & 0 \\ 1 & 26 & 3 & 4 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix} = \begin{bmatrix} 1 \\ 1 \\ 1 \\ 1 \end{bmatrix}$$

17. For the matrix given below, using the following preconditioners with the GMRES method.

    (a) Jacobi

    (b) SSOR with $\omega = 1.0$

    (c) ILU(0)

    Let $b = [1 \quad 1 \quad \ldots 1]^T$ and x^0 be the zero vector. Compare the number of iterations required in each method. Use a convergence tolerance of 10^{-5}.

$$A = \begin{bmatrix} 10 & 0 & 43 & 0 & 23 & 0 & 0 & 0 & 0 & 0 \\ 60 & 41 & 10 & 0 & 0 & 47 & 0 & 0 & 64 & 0 \\ 48 & 0 & 27 & 96 & 85 & 0 & 0 & 29 & 95 & 36 \\ 0 & 0 & 0 & 25 & 0 & 0 & 0 & 0 & 0 & 67 \\ 0 & 97 & 29 & 0 & 13 & 78 & 61 & 0 & 71 & 0 \\ 0 & 54 & 0 & 29 & 0 & 45 & 0 & 69 & 0 & 85 \\ 0 & 0 & 0 & 0 & 0 & 0 & 3 & 55 & 12 & 84 \\ 0 & 0 & 0 & 0 & 0 & 48 & 25 & 43 & 61 & 26 \\ 32 & 0 & 0 & 7 & 0 & 0 & 92 & 0 & 46 & 62 \\ 0 & 78 & 0 & 0 & 0 & 0 & 0 & 65 & 0 & 59 \end{bmatrix}$$