7.1 Naive Gaussian Elimination

One of the fundamental problems in many scientific and engineering applications is to solve an algebraic linear system $\boldsymbol{Ax} = \boldsymbol{b}$ for the unknown vector $\boldsymbol{x}$ when the coefficient matrix $\boldsymbol{A}$ and right-hand side vector $\boldsymbol{b}$ are known. Such systems arise naturally in various

applications, such as approximating nonlinear equations by linear equations or differential equations by algebraic equations. The cornerstone of many numerical methods for solving a variety of practical computational problems is the efficient and accurate solution of linear systems. The system of linear algebraic equations $\boldsymbol{Ax} = \boldsymbol{b}$ may or may not have a solution, and if it has a solution, it may or may not be unique. Gaussian elimination is the standard method for solving the linear system by using a calculator or a computer. This method is undoubtedly familiar to most readers, since it is the simplest way to solve a linear system by hand. When the system has no solution, other approaches are used, such as linear least squares, which is discussed in Chapter 14. In this chapter and most of the next one, we assume that the coefficient matrix $\boldsymbol{A}$ is $n \times n$ and invertible (nonsingular).

In a pure mathematical approach, the solution to the problem $\boldsymbol{Ax} = \boldsymbol{b}$ is simply $\boldsymbol{x} = \boldsymbol{A}^{-1}\boldsymbol{b}$, where $\boldsymbol{A}^{-1}$ is the inverse matrix. But in most applications, it is advisable to solve the system directly for the unknown vector $\boldsymbol{x}$ rather than explicitly computing the inverse matrix.

In applied mathematics and in many applications, it can be a daunting task for even the largest and fastest computers to solve accurately extremely large systems involving thousands or millions of unknowns. Some of the questions are the following: How do we store such large systems in the computer? How do we know that the computed answers are correct? What is the precision of the computed results? Can the algorithm fail? How long will it take to compute answers? What is the asymptotic operation count of the algorithm? Will the algorithm be unstable for certain systems? Can instability be controlled by pivoting? (Permuting the order of the rows of the matrix is called **pivoting**.) Which strategy of pivoting should be used? How do we know whether the matrix is ill-conditioned and whether the answers are accurate?

Gaussian elimination transforms a linear system into an upper triangular form, which is easier to solve. This process, in turn, is equivalent to finding the factorization $\boldsymbol{A} = \boldsymbol{LU}$, where $\boldsymbol{L}$ is a unit lower triangular matrix and $\boldsymbol{U}$ is an upper triangular matrix. This factorization is especially useful when solving many linear systems involving the same coefficient matrix but different right-hand sides, which occurs in various applications.

When the coefficient matrix $\boldsymbol{A}$ has a special structure such as being symmetric, positive definite, triangular, banded, block, or sparse, the general approach of Gaussian elimination with partial pivoting needs to be modified or rewritten specifically for the system. When the coefficient matrix has predominantly zero entries, the system is sparse and iterative methods can involve much less computer memory than Gaussian elimination. We will address many of these issues in this chapter and the next one.

Our objective in this chapter is to develop a good program for solving a system of n linear equations in n unknowns:

$$
\begin{cases}
a_{11}x_1 + a_{12}x_2 + a_{13}x_3 + \cdots + a_{1n}x_n = b_1 \\
a_{21}x_1 + a_{22}x_2 + a_{23}x_3 + \cdots + a_{2n}x_n = b_2 \\
a_{31}x_1 + a_{32}x_2 + a_{33}x_3 + \cdots + a_{3n}x_n = b_3 \\
\quad\vdots \qquad\quad \vdots \qquad\quad \vdots \qquad\qquad\quad \vdots \qquad \vdots \\
a_{i1}x_1 + a_{i2}x_2 + a_{i3}x_3 + \cdots + a_{in}x_n = b_i \\
\quad\vdots \qquad\quad \vdots \qquad\quad \vdots \qquad\qquad\quad \vdots \qquad \vdots \\
a_{n1}x_1 + a_{n2}x_2 + a_{n3}x_3 + \cdots + a_{nn}x_n = b_n
\end{cases} \tag{1}
$$

In compact form, this system can be written simply as

$$\sum_{j=1}^{n} a_{ij}x_j = b_i \qquad (1 \leqq i \leqq n)$$

In these equations, a_{ij} and b_i are prescribed real numbers (data), and the unknowns x_j are to be determined. Subscripts on the letter a are separated by a comma only if necessary for clarity—for example, in $a_{32,75}$ but not in a_{ij}.

A Larger Numerical Example

In this section, the simplest form of Gaussian elimination is explained. The adjective **naive** applies because this form is not usually suitable for automatic computation unless essential modifications are made, as in Section 7.2. We illustrate naive Gaussian elimination with a specific example that has four equations and four unknowns:

$$\begin{cases} 6x_1 - 2x_2 + 2x_3 + 4x_4 = 16 \\ 12x_1 - 8x_2 + 6x_3 + 10x_4 = 26 \\ 3x_1 - 13x_2 + 9x_3 + 3x_4 = -19 \\ -6x_1 + 4x_2 + x_3 - 18x_4 = -34 \end{cases} \tag{2}$$

In the first step of the elimination procedure, certain multiples of the first equation are subtracted from the second, third, and fourth equations so as to eliminate x_1 from these equations. Thus, we want to create 0's as coefficients for each x_1 below the first (where 12, 3, and -6 now stand). It is clear that we should subtract 2 times the first equation from the second. (This multiplier is simply the quotient $\frac{12}{6}$.) Likewise, we should subtract $\frac{1}{2}$ times the first equation from the third. (Again, this multiplier is just $\frac{3}{6}$.) Finally, we should subtract -1 times the first equation from the fourth. When all of this has been done, the result is

$$\begin{cases} 6x_1 - 2x_2 + 2x_3 + 4x_4 = 16 \\ -4x_2 + 2x_3 + 2x_4 = -6 \\ -12x_2 + 8x_3 + x_4 = -27 \\ 2x_2 + 3x_3 - 14x_4 = -18 \end{cases} \tag{3}$$

Note that the first equation was not altered in this process, although it was used to produce the 0 coefficients in the other equations. In this context, it is called the **pivot equation**.

Notice also that Systems (2) and (3) are *equivalent* in the following technical sense: Any solution of (2) is also a solution of (3), and vice versa. This follows at once from the fact that if equal quantities are added to equal quantities, the resulting quantities are equal. One can get System (2) from System (3) by adding 2 times the first equation to the second, and so on.

In the second step of the process, we mentally ignore the first equation and the first column of coefficients. This leaves a system of three equations with three unknowns. The same process is now repeated using the top equation in the smaller system as the current pivot equation. Thus, we begin by subtracting 3 times the second equation from the third. (The multiplier is just the quotient $\frac{-12}{-4}$.) Then we subtract $-\frac{1}{2}$ times the second equation

from the fourth. After doing the arithmetic, we arrive at

$$\begin{cases} 6x_1 - 2x_2 + 2x_3 + 4x_4 = 16 \\ \quad -4x_2 + 2x_3 + 2x_4 = -6 \\ \qquad\quad 2x_3 - 5x_4 = -9 \\ \qquad\quad 4x_3 - 13x_4 = -21 \end{cases} \tag{4}$$

The final step consists in subtracting 2 times the third equation from the fourth. The result is

$$\begin{cases} 6x_1 - 2x_2 + 2x_3 + 4x_4 = 16 \\ \quad -4x_2 + 2x_3 + 2x_4 = -6 \\ \qquad\quad 2x_3 - 5x_4 = -9 \\ \qquad\qquad -3x_4 = -3 \end{cases} \tag{5}$$

This system is said to be in **upper triangular** form. It is equivalent to System (2).

This completes the first phase (**forward elimination**) in the Gaussian algorithm. The second phase (**back substitution**) will solve System (5) for the unknowns *starting at the bottom*. Thus, from the fourth equation, we obtain the last unknown

$$x_4 = \frac{-3}{-3} = 1$$

Putting $x_4 = 1$ in the third equation gives us

$$2x_3 - 5 = -9$$

and we find the next to last unknown

$$x_3 = \frac{-4}{2} = -2$$

and so on. The solution is

$$x_1 = 3 \qquad x_2 = 1 \qquad x_3 = -2 \qquad x_4 = 1$$

Algorithm

To simplify the discussion, we write System (1) in matrix-vector form. The coefficient elements a_{ij} form an $n \times n$ square array, or matrix. The unknowns x_i and the right-hand side elements b_i form $n \times 1$ arrays, or vectors.* (See Appendix D for linear algebra notation and concepts.) Hence, we have

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} & \cdots & a_{1n} \\ a_{21} & a_{22} & a_{23} & \cdots & a_{2n} \\ a_{31} & a_{32} & a_{33} & \cdots & a_{3n} \\ \vdots & \vdots & \vdots & & \vdots \\ a_{i1} & a_{i2} & a_{i3} & \cdots & a_{in} \\ \vdots & \vdots & \vdots & & \vdots \\ a_{n1} & a_{n2} & a_{n3} & \cdots & a_{nn} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ \vdots \\ x_i \\ \vdots \\ x_n \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ b_3 \\ \vdots \\ b_i \\ \vdots \\ b_n \end{bmatrix} \tag{6}$$

*To save space, we occasionally write a vector as $[x_1, x_2, \ldots, x_n]^T$, where the T stands for the **transpose**. It tells us that this is an $n \times 1$ array or vector and *not* $1 \times n$, as would be indicated without the transpose symbol.

or

$$\boldsymbol{Ax} = \boldsymbol{b}$$

Operations between *equations* correspond to operations between *rows* in this notation. We shall use these two words interchangeably.

Now let us organize the naive Gaussian elimination algorithm for the general system, which contains n equations and n unknowns. In this algorithm, the original data are overwritten with new computed values. In the forward elimination phase of the process, there are $n - 1$ principal steps. The first of these steps uses the first equation to produce $n - 1$ zeros as coefficients for each x_1 in all but the first equation. This is done by subtracting appropriate multiples of the first equation from the others. In this process, we refer to the first equation as the first **pivot equation** and to a_{11} as the first **pivot element**. For each of the remaining equations $(2 \leqq i \leqq n)$, we compute

$$\begin{cases} a_{ij} \leftarrow a_{ij} - \left(\dfrac{a_{i1}}{a_{11}}\right) a_{1j} & (1 \leqq j \leqq n) \\ b_i \leftarrow b_i - \left(\dfrac{a_{i1}}{a_{11}}\right) b_1 \end{cases}$$

The symbol $\leftarrow$ indicates a *replacement*. Thus, the content of the memory location allocated to a_{ij} is replaced by $a_{ij} - (a_{i1}/a_{11})a_{1j}$, and so on. This is accomplished by the following line of pseudocode:

$$a_{ij} \leftarrow a_{ij} - (a_{i1}/a_{11})a_{1j}$$

Note that the quantities (a_{i1}/a_{11}) are the **multipliers**. The new coefficient of x_1 in the ith equation will be 0 because $a_{i1} - (a_{i1}/a_{11})a_{11} = 0$.

After the first step, the system will be of the form

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} & \cdots & a_{1n} \\ 0 & a_{22} & a_{23} & \cdots & a_{2n} \\ 0 & a_{23} & a_{33} & \cdots & a_{3n} \\ \vdots & \vdots & \vdots & & \vdots \\ 0 & a_{i2} & a_{i3} & \cdots & a_{in} \\ \vdots & \vdots & \vdots & & \vdots \\ 0 & a_{n2} & a_{n3} & \cdots & a_{nn} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ \vdots \\ x_i \\ \vdots \\ x_n \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ b_3 \\ \vdots \\ b_i \\ \vdots \\ b_n \end{bmatrix}$$

From here on, we will not alter the first equation, nor will we alter any of the coefficients for x_1 (since a multiplier times 0 subtracted from 0 is still 0). Thus, we can mentally ignore the first row and the first column and repeat the process on the smaller system. With the second equation as the pivot equation, we compute for each remaining equation $(3 \leqq i \leqq n)$

$$\begin{cases} a_{ij} \leftarrow a_{ij} - \left(\dfrac{a_{i2}}{a_{22}}\right) a_{2j} & (2 \leqq j \leqq n) \\ b_i \leftarrow b_i - \left(\dfrac{a_{i2}}{a_{22}}\right) b_2 \end{cases}$$

Just prior to the kth step in the forward elimination, the system will appear as follows:

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} & \cdots & & \cdots & & \cdots & a_{1n} \\ 0 & a_{22} & a_{23} & \cdots & & \cdots & & \cdots & a_{2n} \\ 0 & 0 & a_{33} & \cdots & & \cdots & & \cdots & a_{3n} \\ \vdots & \vdots & \vdots & \ddots & & & & & \vdots \\ 0 & 0 & 0 & \cdots & a_{kk} & \cdots & a_{kj} & \cdots & a_{kn} \\ \vdots & \vdots & \vdots & \vdots & \vdots & & \vdots & & \vdots \\ 0 & 0 & 0 & \cdots & a_{ik} & \cdots & a_{ij} & \cdots & a_{in} \\ \vdots & \vdots & \vdots & \vdots & \vdots & & \vdots & & \vdots \\ 0 & 0 & 0 & \cdots & a_{nk} & \cdots & a_{nj} & \cdots & a_{nn} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ \vdots \\ x_k \\ \vdots \\ x_i \\ \vdots \\ x_n \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ b_3 \\ \vdots \\ b_k \\ \vdots \\ b_i \\ \vdots \\ b_n \end{bmatrix}$$

Here, a wedge of 0 coefficients has been created, and the first k equations have been processed and are now fixed. Using the kth equation as the pivot equation, we select multipliers to create 0's as coefficients for each x_i below the a_{kk} coefficient. Hence, we compute for each remaining equation $(k + 1 \leqq i \leqq n)$

$$\begin{cases} a_{ij} \leftarrow a_{ij} - \left(\dfrac{a_{ik}}{a_{kk}}\right) a_{kj} & (k \leqq j \leqq n) \\ b_i \leftarrow b_i - \left(\dfrac{a_{ik}}{a_{kk}}\right) b_k \end{cases}$$

Obviously, we must assume that all the divisors in this algorithm are nonzero.

Pseudocode

We now consider the pseudocode for forward elimination. The coefficient array is stored as a double-subscripted array (a_{ij}); the right-hand side of the system of equations is stored as a single-subscripted array (b_i); the solution is computed and stored in a single-subscripted array (x_i). It is easy to see that the following lines of pseudocode carry out the forward elimination phase of naive Gaussian elimination:

```
integer i, j, k;   real array (a_ij)_{1:n×1:n}, (b_i)_{1:n}
for k = 1 to n − 1 do
    for i = k + 1 to n do
        for j = k to n do
            a_ij ← a_ij − (a_ik/a_kk)a_kj
        end for
        b_i ← b_i − (a_ik/a_kk)b_k
    end for
end for
```

Since the multiplier a_{ik}/a_{kk} does not depend on j, it should be moved outside the j loop. Notice also that the new values in column k will be 0, at least theoretically, because when

$j = k$, we have

$$a_{ik} \leftarrow a_{ik} - (a_{ik}/a_{kk})a_{kk}$$

Since we expect this to be 0, no purpose is served in computing it. The location where the 0 is being created is a good place to store the multiplier. If these remarks are put into practice, the pseudocode will look like this:

```
integer i, j, k;   real xmult;   real array (a_ij)_{1:n×1:n}, (b_i)_{1:n}
for k = 1 to n − 1 do
    for i = k + 1 to n do
        xmult ← a_ik/a_kk
        a_ik ← xmult
        for j = k + 1 to n do
            a_ij ← a_ij − (xmult)a_kj
        end for
        b_i ← b_i − (xmult)b_k
    end for
end for
```

Here, the multipliers are stored because they are part of the $\boldsymbol{LU}$-factorization that can be useful in some applications. This matter is discussed in Section 8.1.

At the beginning of the back substitution phase, the linear system is of the form

$$\begin{cases} a_{11}x_1 + a_{12}x_2 + a_{13}x_3 + \cdots & \cdots + & a_{1n}x_n = b_1 \\ a_{22}x_2 + a_{23}x_3 + \cdots & \cdots + & a_{2n}x_n = b_2 \\ a_{33}x_3 + \cdots & + & a_{3n}x_n = b_3 \\ \ddots & \vdots & \vdots \\ a_{ii}x_i + a_{i,i+1}x_{i+1} + & \cdots + & a_{in}x_n = b_i \\ \ddots & \vdots & \vdots \\ & a_{n-1,n-1}x_{n-1} + & a_{n-1,n}x_n = b_{n-1} \\ & & a_{nn}x_n = b_n \end{cases}$$

where the a_{ij}'s and b_i's are *not* the original ones from System (6) but instead are the ones that have been altered by the elimination process.

The back substitution starts by solving the nth equation for x_n:

$$x_n = \frac{b_n}{a_{nn}}$$

Then, using the $(n-1)$th equation, we solve for x_{n-1}:

$$x_{n-1} = \frac{1}{a_{n-1,n-1}}\left(b_{n-1} - a_{n-1,n}x_n\right)$$

We continue working upward, recovering each x_i by the formula

$$x_i = \frac{1}{a_{ii}}\left(b_i - \sum_{j=i+1}^{n} a_{ij}x_j\right) \qquad (i = n-1, n-2, \ldots, 1) \tag{7}$$

Here is pseudocode to do this:

```
integer i, j, n;   real sum;   real array (a_ij)_{1:n×1:n}, (x_i)_{1:n}
x_n ← b_n/a_nn
for i = n − 1 to 1 step −1 do
    sum ← b_i
    for j = i + 1 to n do
        sum ← sum − a_ij x_j
    end for
    x_i ← sum/a_ii
end for
```

Now we put these segments of pseudocode together to form a procedure, called *Naive_Gauss*, which is intended to solve a system of n linear equations in n unknowns by the method of naive Gaussian elimination. This pseudocode serves a didactic purpose only; a more robust pseudocode will be developed in the next section.

```
procedure Naive_Gauss(n, (a_ij), (b_i), (x_i))
integer i, j, k, n;   real sum, xmult
real array (a_ij)_{1:n×1:n}, (b_i)_{1:n}, (x_i)_{1:n}
for k = 1 to n − 1 do
    for i = k + 1 to n do
        xmult ← a_ik/a_kk
        a_ik ← xmult
        for j = k + 1 to n do
            a_ij ← a_ij − (xmult)a_kj
        end for
        b_i ← b_i − (xmult)b_k
    end for
end for
x_n ← b_n/a_nn
for i = n − 1 to 1 step −1 do
    sum ← b_i
    for j = i + 1 to n do
        sum ← sum − a_ij x_j
    end for
    x_i ← sum/a_ii
end for
end procedure Naive_Gauss
```

Before giving a test example, let us examine the crucial computation in our pseudocode, namely, a triply nested for-loop containing a replacement operation:

```
for k ..... do
    for i ..... do
        for j ..... do
            a_ij ← a_ij − (a_ik/a_kk)a_kj
        end do
    end do
end do
```

Here, we must expect all quantities to be infected with roundoff error. Such a roundoff error in a_{kj} is multiplied by the factor (a_{ik}/a_{kk}). This factor is large if the pivot element $|a_{kk}|$ is small relative to $|a_{ik}|$. Hence, we conclude, tentatively, that small pivot elements lead to large multipliers and to worse roundoff errors.

Testing the Pseudocode

One good way to test a procedure is to set up an artificial problem whose solution is known beforehand. Sometimes the test problem will include a parameter that can be changed to vary the difficulty. The next example illustrates this.

Fixing a value of n, define the polynomial

$$p(t) = 1 + t + t^2 + \cdots + t^{n-1} = \sum_{j=1}^{n} t^{j-1}$$

The coefficients in this polynomial are all equal to 1. We shall try to recover these known coefficients from n values of the polynomial. We use the values of $p(t)$ at the integers $t = 1+i$ for $i = 1, 2, \ldots, n$. If the coefficients in the polynomial are denoted by $x_1, x_2, \ldots, x_n$, we should have

$$\sum_{j=1}^{n} (1+i)^{j-1} x_j = \frac{1}{i}\left[(1+i)^n - 1\right] \qquad (1 \leqq i \leqq n) \tag{8}$$

Here, we have used the formula for the sum of a geometric series on the right-hand side; that is,

$$p(1+i) = \sum_{j=1}^{n} (1+i)^{j-1} = \frac{(1+i)^n - 1}{(1+i) - 1} = \frac{1}{i}\left[(1+i)^n - 1\right] \tag{9}$$

Letting $a_{ij} = (1+i)^{j-1}$ and $b_i = [(1+i)^n - 1]/i$ in Equation (8), we have a linear system.

EXAMPLE 1 We write a pseudocode for a specific test case that solves the system of Equation (8) for various values of n.

Solution Since the naive Gaussian elimination procedure *Naive_Gauss* can be used, all that is needed is a calling program. We decide to use $n = 4, 5, 6, 7, 8, 9, 10$ for the test. Here is a

suitable pseudocode:

```
program Test_NGE
integer parameter m ← 10
integer i, j, n;   real array, (a_ij)_1:m×1:m, (b_i)_1:m, (x_i)_1:m
for n = 4 to 10 do
    for i = 1 to n do
        for j = 1 to n do
            a_ij ← (i + 1)^(j−1)
        end for
        b_i ← [(i + 1)^n − 1]/i
    end for
    call Naive_Gauss(n, (a_ij), (b_i), (x_i))
    output n, (x_i)_1:n
end for
end program Test_NGE
```

When this pseudocode was run on a machine that carries approximately seven decimal digits of accuracy, the solution was obtained with complete precision until n reached 9, and then the computed solution was worthless because one component exhibited a relative error of 16,120%! (Write and run a computer program to see for yourself!) ■

The coefficient matrix for this linear system is an example of a well-known ill-conditioned matrix called the **Vandermonde matrix**, and this accounts for the fact that the system cannot be solved accurately using naive Gaussian elimination. What is amazing is that the trouble happens so suddenly! When $n \geqq 9$, the roundoff error that is present in computing x_i is propagated and magnified throughout the back substitution phase so that most of the computed values for x_i are worthless. Insert some intermediate print statements in the code to see for yourself what is going on here. (See Gautschi [1990] for more information on the Vandermonde matrix and its ill-conditioned nature.)

Residual and Error Vectors

For a linear system $\boldsymbol{Ax} = \boldsymbol{b}$ having the true solution $\boldsymbol{x}$ and a computed solution $\tilde{x}$, we define

$$\boldsymbol{e} = \tilde{\boldsymbol{x}} - \boldsymbol{x} \qquad \textbf{error vector}$$
$$\boldsymbol{r} = \boldsymbol{A}\tilde{\boldsymbol{x}} - \boldsymbol{b} \qquad \textbf{residual vector}$$

An important relationship between the error vector and the residual vector is

$$\boldsymbol{Ae} = \boldsymbol{r}$$

Suppose that two students using different computer systems solve the same linear system, $\boldsymbol{Ax} = \boldsymbol{b}$. What algorithm and what precision each student used are not known. Each vehemently claims to have the correct answer, but the two computer solutions $\tilde{\boldsymbol{x}}$ and $\hat{\boldsymbol{x}}$ are totally different! How do we determine which, if either, computed solution is correct?

We can *check* the solutions by substituting them into the original system, which is the same as computing the **residual vectors** $\tilde{\boldsymbol{r}} = \boldsymbol{A}\tilde{\boldsymbol{x}} - \boldsymbol{b}$ and $\hat{\boldsymbol{r}} = \boldsymbol{A}\hat{\boldsymbol{x}} - \boldsymbol{b}$. Of course, the

computed solutions are not exact because each must contain some roundoff errors. So we would want to accept the solution with the smaller residual vector. However, if we knew the exact solution $\boldsymbol{x}$, then we would just compare the computed solutions with the exact solution, which is the same as computing the **error vectors** $\widetilde{\boldsymbol{e}} = \widetilde{\boldsymbol{x}} - \boldsymbol{x}$ and $\widehat{\boldsymbol{e}} = \widehat{\boldsymbol{x}} - \boldsymbol{x}$. Now the computed solution that produces the smaller error vector would most assuredly be the better answer.

Since the exact solution is usually not known in applications, one would tend to accept the computed solution that has the smaller residual vector. But this may not be the best computed solution if the original problem is sensitive to roundoff errors—that is, is ill-conditioned. In fact, the question of whether a computed solution to a linear system is a good solution is extremely difficult and beyond the scope of this book. Problem 7.1.5 may give some insight into the difficulty of assessing the accuracy of computed solutions of linear systems.

Summary

(1) The basic **forward elimination** procedure using equation k to operate on equations $k+1, k+2, \ldots, n$ is

$$\begin{cases} a_{ij} \leftarrow a_{ij} - (a_{ik}/a_{kk})a_{kj} & (k \leqq j \leqq n, k < i \leqq n) \\ b_i \leftarrow b_i - (a_{ik}/a_{kk})b_k \end{cases}$$

Here we assume $a_{kk} \neq 0$. The basic **back substitution** procedure is

$$x_i = \frac{1}{a_{ii}}\left(b_i - \sum_{j=i+1}^{n} a_{ij}x_j\right) \qquad (i = n-1, n-2, \ldots, 1)$$

(2) When solving the linear system $\boldsymbol{Ax} = \boldsymbol{b}$, if the true or exact solution is $\boldsymbol{x}$ and the approximate or computed solution is $\widetilde{\boldsymbol{x}}$, then important quantities are

error vectors	$\boldsymbol{e} = \widetilde{\boldsymbol{x}} - \boldsymbol{x}$
residual vectors	$\boldsymbol{r} = \boldsymbol{A}\widetilde{\boldsymbol{x}} - \boldsymbol{b}$