7.2 Gaussian Elimination with Scaled Partial Pivoting

Naive Gaussian Elimination Can Fail

To see why the naive Gaussian elimination algorithm is unsatisfactory, consider the following system:

$$\begin{cases} 0x_1 + x_2 = 1 \\ x_1 + x_2 = 2 \end{cases} \tag{1}$$

The pseudocode that we constructed in Section 7.1 would attempt to subtract some multiple of the first equation from the second to produce 0 as the coefficient for x_1 in the second equation. This, of course, is impossible, so the algorithm fails if $a_{11} = 0$.

If a numerical procedure actually fails for some values of the data, then the procedure is probably untrustworthy for values of the data *near* the failing values. To test this dictum, consider the system

$$\begin{cases} \varepsilon x_1 + x_2 = 1 \\ x_1 + x_2 = 2 \end{cases} \tag{2}$$

in which ε is a small number different from 0. Now the naive algorithm of Section 7.1 works, and after forward elimination it produces the system

$$\begin{cases} \varepsilon x_1 + & x_2 = 1 \\ & (1 - \varepsilon^{-1})x_2 = 2 - \varepsilon^{-1} \end{cases} \tag{3}$$

In the back substitution, the arithmetic is as follows:

$$x_2 = \frac{2 - \varepsilon^{-1}}{1 - \varepsilon^{-1}} \approx 1, \qquad x_1 = \varepsilon^{-1}(1 - x_2) \approx 0$$

Now ε^{-1} will be large, so if this calculation is performed by a computer that has a fixed word length, then for small values of ε, both $(2 - \varepsilon^{-1})$ and $(1 - \varepsilon^{-1})$ would be computed as $-\varepsilon^{-1}$.

For example, in an 8-digit decimal machine with a 16-digit accumulator, when $\varepsilon = 10^{-9}$, it follows that $\varepsilon^{-1} = 10^9$. To subtract, the computer must interpret the numbers as

$$\begin{aligned} \varepsilon^{-1} = 10^9 &= 0.10000\,000 \times 10^{10} = 0.10000\,00000\,00000\,0 \times 10^{10} \\ 2 &= 0.20000\,000 \times 10^{1} \ = 0.00000\,00002\,00000\,0 \times 10^{10} \end{aligned}$$

Thus, $(\varepsilon^{-1} - 2)$ is initially computed as $0.09999\,99998\,00000\,0 \times 10^{10}$ and then rounded to $0.10000\,000 \times 10^{10} = \varepsilon^{-1}$.

We conclude that for values of ε sufficiently close to 0, the computer calculates x_2 as 1 and then x_1 as 0. Since the *correct* solution is

$$x_1 = \frac{1}{1 - \varepsilon} \approx 1, \qquad x_2 = \frac{1 - 2\varepsilon}{1 - \varepsilon} \approx 1$$

the relative error in the computed solution for x_1 is extremely large: 100%.

Actually, the naive Gaussian elimination algorithm works well on Systems (1) and (2) if the equations are first permuted:

$$\begin{cases} x_1 + x_2 = 2 \\ 0x_1 + x_2 = 1 \end{cases}$$

and

$$\begin{cases} x_1 + x_2 = 2 \\ \varepsilon x_1 + x_2 = 1 \end{cases}$$

The first system is easily solved obtaining $x_2 = 1$ and $x_1 = 2 - x_2 = 1$. Moreover, the second of these systems becomes

$$\begin{cases} x_1 + x_2 = 2 \\ (1-\varepsilon)x_2 = 1 - 2\varepsilon \end{cases}$$

after the forward elimination. Then from the back substitution, the solution is computed as

$$x_2 = \frac{1-2\varepsilon}{1-\varepsilon} \approx 1, \qquad x_1 = 2 - x_2 \approx 1$$

Notice that we do not have to rearrange the equations in the system: it is necessary only to select a different pivot row. The difficulty in System (2) is not due simply to ε being small but rather to its being small relative to other coefficients in the same row. To verify this, consider

$$\begin{cases} x_1 + \varepsilon^{-1}x_2 = \varepsilon^{-1} \\ x_1 + x_2 = 2 \end{cases} \tag{4}$$

System (4) is mathematically equivalent to (2). The naive Gaussian elimination algorithm fails here. It produces the triangular system

$$\begin{cases} x_1 + \varepsilon^{-1}x_2 = \varepsilon^{-1} \\ \left(1-\varepsilon^{-1}\right)x_2 = 2 - \varepsilon^{-1} \end{cases}$$

and then, in the back substitution, it produces the erroneous result

$$x_2 = \frac{2-\varepsilon^{-1}}{1-\varepsilon^{-1}} \approx 1, \qquad x_1 = \varepsilon^{-1} - \varepsilon^{-1}x_2 \approx 0$$

This situation can be resolved by interchanging the two equations in (4):

$$\begin{cases} x_1 + x_2 = 2 \\ x_1 + \varepsilon^{-1}x_2 = \varepsilon^{-1} \end{cases}$$

Now the naive Gaussian elimination algorithm can be applied, resulting in the system

$$\begin{cases} x_1 + x_2 = 2 \\ \left(\varepsilon^{-1} - 1\right)x_2 = \varepsilon^{-1} - 2 \end{cases}$$

The solution is

$$x_2 = \frac{\varepsilon^{-1}-2}{\varepsilon^{-1}-1} \approx 1, \qquad x_1 = 2 - x_2 \approx 1$$

which is the correct solution.

Partial Pivoting and Complete Partial Pivoting

Gaussian elimination with **partial pivoting** selects the pivot row to be the one with the maximum pivot entry in absolute value from those in the leading column of the reduced submatrix. Two rows are interchanged to move the designated row into the pivot row position. Gaussian elimination with **complete pivoting** selects the pivot entry as the maximum pivot entry from all entries in the submatrix. (This complicates things because some of the unknowns are rearranged.) Two rows and two columns are interchanged to accomplish this. In practice, partial pivoting is almost as good as full pivoting and involves significantly less work. See Wilkinson [1963] for more details on this matter. Simply picking the largest number in magnitude as is done in partial pivoting may work well, but here row scaling does not play a role—the relative sizes of entries in a row are not considered. Systems with equations having coefficients of disparate sizes may cause difficulties and should be viewed with suspicion. Sometimes a scaling strategy may ameliorate these problems. In this book, we present Gaussian elimination with scaled partial pivoting, and the pseudocode contains an implicit pivoting scheme.

In certain situations, Gaussian elimination with the simple partial pivoting strategy may lead to an incorrect solution. Consider the augmented matrix

$$\left[\begin{array}{cc|c} 2 & 2c & 2c \\ 1 & 1 & 2 \end{array}\right]$$

where c is a parameter that can take on very large numerical values and the variables are x and y. The first row is selected as the pivot row by choosing the larger number in the first column. Since the multiplier is $1/2$, one step in the row reduction process brings us to

$$\left[\begin{array}{cc|c} 2 & 2c & 2c \\ 0 & 1-c & 2-c \end{array}\right]$$

Now suppose that we are working with a computer of limited word length. So in this computer, we obtain $1 - c \approx -c$ and $2 - c \approx -c$. Consequently, the computer contains these numbers:

$$\left[\begin{array}{cc|c} 2 & 2c & 2c \\ 0 & -c & -c \end{array}\right]$$

Thus, as the solution, we obtain $y = 1$ and $x = 0$, whereas the correct solution is $x = y = 1$.

On the other hand, Gaussian elimination with scaled partial pivoting selects the second row as the pivot row. The scaling constants are $(2c, 1)$, and the larger of the two ratios for selecting the pivot row from $\{2/(2c), 1\}$ is the second one. Now the multiplier is 2, and one step in the row reduction process brings us to

$$\left[\begin{array}{cc|c} 0 & 2c-2 & 2c-4 \\ 1 & 1 & 2 \end{array}\right]$$

On our computer of limited word length, we find $2c - 2 \approx 2c$ and $2c - 4 \approx 2c$. Consequently, the computer contains these numbers:

$$\left[\begin{array}{cc|c} 0 & 2c & 2c \\ 1 & 1 & 2 \end{array}\right]$$

Now we obtain the correct solution, $y = 1$ and $x = 1$.

Gaussian Elimination with Scaled Partial Pivoting

These simple examples should make it clear that the *order* in which we treat the equations significantly affects the accuracy of the elimination algorithm in the computer. In the naive Gaussian elimination algorithm, we use the first equation to eliminate x_1 from the equations that follow it. Then we use the second equation to eliminate x_2 from the equations that follow it, and so on. The order in which the equations are used as pivot equations is the **natural** order $\{1, 2, \ldots, n\}$. Note that the last equation (equation number n) is *not* used as an operating equation in the natural ordering: At no time are multiples of it subtracted from other equations in the naive algorithm.

From the previous examples, it is clear that a strategy is needed for selecting new pivots at each stage in Gaussian elimination. Perhaps the best approach is **complete pivoting**, which involves searches over all entries in the submatrices for the largest entry in absolute value and then interchanges rows and columns to move it into the pivot position. This would be quite expensive, since it involves a great amount of searching and data movement. However, searching just the first column in the submatrix at each stage accomplishes most of what is needed (avoiding small or zero pivots). This is **partial pivoting**, and it is the most common approach. It does not involve an examination of the elements in the rows, since it looks only at column entries. We advocate a strategy that simulates a scaling of the row vectors and then selects as a pivot element the relatively largest entry in a column. Also, rather than interchanging rows to move the desired element into the pivot position, we use an indexing array to avoid the data movement. This procedure is not as expensive as complete pivoting, and it goes beyond partial pivoting to include an examination of all elements in the original matrix. Of course, other strategies for selecting pivot elements could be used.

The Gaussian elimination algorithm now to be described uses the equations in an order that is determined by the actual system being solved. For instance, if the algorithm were asked to solve System (1) or (2), the order in which the equations would be used as pivot equations would not be the natural order $\{1, 2\}$ but rather $\{2, 1\}$. This order is automatically determined by the computer program. The order in which the equations are employed is denoted by the row vector $[\ell_1, \ell_2, \ldots, \ell_n]$, where ℓ_n is not actually being used in the forward elimination phase. Here, the ℓ_i are integers from 1 to n in a possibly different order. We call $\boldsymbol{\ell} = [\ell_1, \ell_2, \ldots, \ell_n]$ the **index vector**. The strategy to be described now for determining the index vector is termed **scaled partial pivoting**.

At the beginning, a **scale factor** must be computed for each equation in the system. Referring to the notation in Section 7.1, we define

$$s_i = \max_{1 \leq j \leq n} |a_{ij}| \qquad (1 \leq i \leq n)$$

These n numbers are recorded in the **scale vector** $\boldsymbol{s} = [s_1, s_2, \ldots, s_n]$.

In starting the forward elimination process, we do not arbitrarily use the first equation as the pivot equation. Instead, we use the equation for which the ratio $|a_{i,1}|/s_i$ is greatest. Let ℓ_1 be the first index for which this ratio is greatest. Now appropriate multiples of equation ℓ_1 are subtracted from the other equations to create 0's as coefficients for each x_1 except in the pivot equation.

The best way of keeping track of the indices is as follows: At the beginning, define the index vector $\boldsymbol{\ell}$ to be $[\ell_1, \ell_2, \ldots, \ell_n] = [1, 2, \ldots, n]$. Select j to be the first index associated

with the largest ratio in the set:

$$\left\{ \frac{|a_{\ell_i 1}|}{s_{\ell_i}} : 1 \leqq i \leqq n \right\}$$

Now interchange ℓ_j with ℓ_1 in the index vector $\boldsymbol{\ell}$. Next, use multipliers

$$\frac{a_{\ell_i 1}}{a_{\ell_1 1}}$$

times row ℓ_1, and subtract from equations ℓ_i for $2 \leqq i \leqq n$. It is important to note that only entries in $\boldsymbol{\ell}$ are being interchanged and *not* the equations. This eliminates the time-consuming and unnecessary process of moving the coefficients of equations around in the computer memory!

In the second step, the ratios

$$\left\{ \frac{|a_{\ell_i,2}|}{s_{\ell_i}} : 2 \leqq i \leqq n \right\}$$

are scanned. If j is the first index for the largest ratio, interchange ℓ_j with ℓ_2 in $\boldsymbol{\ell}$. Then multipliers

$$\frac{a_{\ell_i 2}}{a_{\ell_2 2}}$$

times equation ℓ_2 are subtracted from equations ℓ_i for $3 \leqq i \leqq n$.

At step k, select j to be the first index corresponding to the largest of the ratios,

$$\left\{ \frac{|a_{\ell_i k}|}{s_{\ell_i}} : k \leqq i \leqq n \right\}$$

and interchange ℓ_j and ℓ_k in index vector $\boldsymbol{\ell}$. Then multipliers

$$\frac{a_{\ell_i k}}{a_{\ell_k k}}$$

times pivot equation ℓ_k are subtracted from equations ℓ_i for $k+1 \leqq i \leqq n$.

Notice that the scale factors are *not* changed after each pivot step. Intuitively, one might think that after each step in the Gaussian algorithm, the remaining (modified) coefficients should be used to recompute the scale factors instead of using the original scale vector. Of course, this could be done, but it is generally believed that the extra computations involved in this procedure are not worthwhile in the majority of linear systems. The reader is encouraged to explore this question. (See Computer Problem 7.2.16.)

EXAMPLE 1 Solve this system of linear equations:

$$\begin{cases} 0.0001x + y = 1 \\ x + y = 2 \end{cases}$$

using no pivoting, partial pivoting, and scaled partial pivoting. Carry at most five significant digits of precision (rounding) to see how finite precision computations and roundoff errors can affect the calculations.

Solution By direct substitution, it is easy to verify that the true solution is $x = 1.0001$ and $y = 0.99990$ to five significant digits.

For no pivoting, the first equation in the original system is the pivot equation, and the multiplier is $xmult = 1/0.0001 = 10000$. Multiplying the first equation by this multiplier and subtracting the result from the second equation, the necessary calculations are $(10000)(0.0001) - 1 = 0$, $(10000)(1) - 1 = 9999$, and $(10000)(1) - 2 = 9998$. The new system of equations is

$$\begin{cases} 0.0001x + y = 1 \\ \qquad 9999y = 9998 \end{cases}$$

From the second equation, we obtain $y = 9998/9999 \approx 0.99990$. Using this result and the first equation, we find $0.0001x = 1 - y = 1 - 0.999900 = 0.0001$ and $x = 0.0001/0.0001 = 1$. Notice that we have lost the last significant digit in the correct value of x.

We repeat the solution using partial pivoting in the original system. Examining the first column of x coefficients $(0.0001, 1)$, we see that the second is larger, so the second equation is used as the pivot equation. We can interchange the two equations, obtaining

$$\begin{cases} \qquad x + y = 2 \\ 0.0001x + y = 1 \end{cases}$$

The multiplier is $xmult = 0.0001/1 = 0.0001$. This multiple of the first equation is subtracted from the second equation. The calculations are $(-0.0001)(1) + 0.0001 = 0$, $(0.0001)(1) - 1 = 0.99990$, and $(0.0001)(2) - 1 = 0.99980$. The new system of equations is

$$\begin{cases} \quad x + y = 2 \\ 0.99990y = 0.99980 \end{cases}$$

We obtain $y = 0.99980/0.99990 \approx 0.99990$. Now, using the second equation and this value, we find $x = 2 - y = 2 - 0.99990 = 1.0001$. Both computed values of x and y are correct to five significant digits.

We repeat the solution using scaled partial pivoting on the original system. Since the scaling constants are $s = (1, 1)$ and the ratios for determining the pivot equation are $(0.0001/1, 1/1)$, the second equation is now the pivot equation. We do not actually interchange the equations but can work with an index array $\ell = (2, 1)$ that tells us to use the second equation as the first pivot equation. The rest of the calculations are as above for partial pivoting. The computed values of x and y are correct to five significant digits.

We cannot promise that scaled partial pivoting will be better than partial pivoting, but it clearly has some advantages. For example, suppose that someone wants to force the first equation in the original system to be the pivot equation and multiply it by a large number such as 20,000, obtaining

$$\begin{cases} 2x + 20000y = 20000 \\ \qquad x + y = 2 \end{cases}$$

Partial pivoting ignores the fact that the coefficients in the first equation differ by orders of magnitude and selects the first equation as the pivot equation. However, scaled partial pivoting uses the scaling constants $(20000, 1)$, and the ratios for determining the pivot equations are $(2/20000, 1/1)$. Scaled partial pivoting continues to select the second equation as the pivot equation! ■

A Larger Numerical Example

We are not quite ready to write pseudocode, but let us consider what has been described in a concrete example. Consider

$$\begin{bmatrix} 3 & -13 & 9 & 3 \\ -6 & 4 & 1 & -18 \\ 6 & -2 & 2 & 4 \\ 12 & -8 & 6 & 10 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix} = \begin{bmatrix} -19 \\ -34 \\ 16 \\ 26 \end{bmatrix} \tag{5}$$

The index vector is $\boldsymbol{\ell} = [1, 2, 3, 4]$ at the beginning. The scale vector does not change throughout the procedure and is $\boldsymbol{s} = [13, 18, 6, 12]$. To determine the first pivot row, we look at four ratios:

$$\left\{ \frac{|a_{\ell_i,1}|}{s_{\ell_i}} : i = 1, 2, 3, 4 \right\} = \left\{ \frac{3}{13}, \frac{6}{18}, \frac{6}{6}, \frac{12}{12} \right\} \approx \{0.23, 0.33, 1.0, 1.0\}$$

We select the index j as the *first* occurrence of the largest value of these ratios. In this example, the largest of these occurs for the index $j = 3$. So row three is to be the pivot equation in step 1 ($k = 1$) of the elimination process. In the index vector ℓ, entries ℓ_k and ℓ_j are interchanged so that the new index vector is $\boldsymbol{\ell} = [3, 2, 1, 4]$. Thus, the pivot equation is ℓ_k, which is $\ell_1 = 3$. Now appropriate multiples of the third equation are subtracted from the other equations so as to create 0's as coefficients for x_1 in each of those equations. Explicitly, $\frac{1}{2}$ times row three is subtracted from row one, -1 times row three is subtracted from row two, and 2 times row three is subtracted from row four. The result is

$$\begin{bmatrix} 0 & -12 & 8 & 1 \\ 0 & 2 & 3 & -14 \\ 6 & -2 & 2 & 4 \\ 0 & -4 & 2 & 2 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix} = \begin{bmatrix} -27 \\ -18 \\ 16 \\ -6 \end{bmatrix}$$

In the next step ($k = 2$), we use the index vector $\boldsymbol{\ell} = [3, 2, 1, 4]$ and scan the ratios corresponding to rows two, one, and four:

$$\left\{ \frac{|a_{\ell_i,2}|}{s_{\ell_i}} : i = 2, 3, 4 \right\} = \left\{ \frac{2}{18}, \frac{12}{13}, \frac{4}{12} \right\} \approx \{0.11, 0.92, 0.33\}$$

looking for the largest value. We find that the largest is the second ratio, and we therefore set $j = 3$ and interchange ℓ_k with ℓ_j in the index vector. Thus, the index vector becomes $\boldsymbol{\ell} = [3, 1, 2, 4]$. The pivot equation for step 2 in the elimination is now row one, and $\ell_2 = 1$. Next, multiples of the first equation are subtracted from the second equation and the fourth equation. The appropriate multiples are $-\frac{1}{6}$ and $\frac{1}{3}$, respectively. The result is

$$\begin{bmatrix} 0 & -12 & 8 & 1 \\ 0 & 0 & \frac{13}{3} & -\frac{83}{6} \\ 6 & -2 & 2 & 4 \\ 0 & 0 & -\frac{2}{3} & \frac{5}{3} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix} = \begin{bmatrix} -27 \\ -\frac{45}{2} \\ 16 \\ 3 \end{bmatrix}$$

The third and final step ($k = 3$) is to examine the ratios corresponding to rows two and four:

$$\left\{ \frac{|a_{\ell_i,3}|}{s_{\ell_i}} : i = 3, 4 \right\} = \left\{ \frac{13/3}{18}, \frac{2/3}{12} \right\} \approx \{0.24, 0.06\}$$

with the index vector $\boldsymbol{\ell} = [3, 1, 2, 4]$. The larger value is the first, so we set $j = 3$. Since this is step $k = 3$, interchanging ℓ_k with ℓ_j leaves the index vector unchanged, $\boldsymbol{\ell} = [3, 1, 2, 4]$. The pivot equation is row two and $\ell_3 = 2$, and we subtract $-\frac{2}{13}$ times the second equation from the fourth equation. So the forward elimination phase ends with the final system

$$\begin{bmatrix} 0 & -12 & 8 & 1 \\ 0 & 0 & \frac{13}{3} & -\frac{83}{6} \\ 6 & -2 & 2 & 4 \\ 0 & 0 & 0 & -\frac{6}{13} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix} = \begin{bmatrix} -27 \\ -\frac{45}{2} \\ 16 \\ -\frac{6}{13} \end{bmatrix}$$

The order in which the pivot equations were selected is displayed in the final index vector $\boldsymbol{\ell} = [3, 1, 2, 4]$.

Now, reading the entries in the index vector from the last to the first, we have the order in which the back substitution is to be performed. The solution is obtained by using equation $\ell_4 = 4$ to determine x_4, and then equation $\ell_3 = 2$ to find x_3, and so on. Carrying out the calculations, we have

$$\begin{aligned} x_4 &= \frac{1}{-6/13}[-6/13] = 1 \\ x_3 &= \frac{1}{13/3}[(-45/2) + (83/6)(1)] = -2 \\ x_2 &= \frac{1}{-12}[-27 - 8(-2) - 1(1)] = 1 \\ x_1 &= \frac{1}{6}[16 + 2(1) - 2(-2) - 4(1)] = 3 \end{aligned}$$

Hence, the solution is

$$\boldsymbol{x} = \begin{bmatrix} 3 & 1 & -2 & 1 \end{bmatrix}^T$$

Pseudocode

The algorithm as programmed carries out the forward elimination phase on the coefficient array (a_{ij}) only. The right-hand side array (b_i) is treated in the next phase. This method is adopted because it is more efficient if several systems must be solved with the same array (a_{ij}) but differing arrays (b_i). Because we wish to treat (b_i) later, it is necessary to store not only the index array but also the various multipliers that are used. These multipliers are conveniently stored in array (a_{ij}) in the positions where the 0 entries would have been created. These multipliers are useful in constructing the $\boldsymbol{LU}$ factorization of the matrix $\boldsymbol{A}$, as we explain in Section 8.1.

We are now ready to write a procedure for forward elimination with scaled partial pivoting. Our approach is to modify procedure *Naive_Gauss* of Section 7.1 by introducing scaling and indexing arrays. The procedure that carries out Gaussian elimination with scaled partial pivoting on the square array (a_{ij}) is called *Gauss*. Its calling sequence is $(n, (a_{ij}), (\ell_i))$, where (a_{ij}) is the $n \times n$ coefficient array and (ℓ_i) is the index array $\boldsymbol{\ell}$. In the pseudocode, (s_i) is the scale array, $\boldsymbol{s}$.

```
procedure Gauss(n, (a_ij), (ℓ_i))
integer i, j, k, n;   real r, rmax, smax, xmult
real array (a_ij)_{1:n×1:n}, (ℓ_i)_{1:n};   real array allocate (s_i)_{1:n}
for i = 1 to n do
    ℓ_i ← i
    smax ← 0
    for j = 1 to n do
        smax ← max(smax, |a_ij|)
    end for
    s_i ← smax
end for
for k = 1 to n − 1 do
    rmax ← 0
    for i = k to n do
        r ← |a_{ℓ_i,k}/s_{ℓ_i}|
        if (r > rmax) then
            rmax ← r
            j ← i
        end if
    end for
    ℓ_j ↔ ℓ_k
    for i = k + 1 to n do
        xmult ← a_{ℓ_i,k}/a_{ℓ_k,k}
        a_{ℓ_i,k} ← xmult
        for j = k + 1 to n do
            a_{ℓ_i,j} ← a_{ℓ_i,j} − (xmult)a_{ℓ_k,j}
        end for
    end for
end for
deallocate array (s_i)
end procedure Gauss
```

A detailed explanation of the above procedure is now presented. In the first loop, the initial form of the index array is being established, namely, $\ell_i = i$. Then the scale array (s_i) is computed.

The statement **for** $k = 1$ **to** $n - 1$ **do** initiates the principal outer loop. The index k is the subscript of the variable whose coefficients will be made 0 in the array (a_{ij}); that is, k is the index of the column in which new 0's are to be created. Remember that the 0's in the array (a_{ij}) do not actually appear because those storage locations are used for the multipliers. This fact can be seen in the line of the procedure where *xmult* is stored in the array (a_{ij}). (See Section 8.1 on the ***LU*** factorization of $\boldsymbol{A}$ for why this is done.)

Once k has been set, the first task is to select the correct pivot row, which is done by computing $|a_{\ell_i k}|/s_{\ell_i}$ for $i = k, k+1, \ldots, n$. The next set of lines in the pseudocode is calculating this greatest ratio, called *rmax* in the routine, and the index j where it occurs. Next, ℓ_k and ℓ_j are interchanged in the array (ℓ_i).

The arithmetic modifications in the array (a_{ij}) due to subtracting multiples of row ℓ_k from rows $\ell_{k+1}, \ell_{k+2}, \ldots, \ell_n$ all occur in the final lines. First the multiplier is computed and stored; then the subtraction occurs in a loop.

Caution: Values in array (a_{ij}) that result as *output* from procedure *Gauss* are not the same as those in array (a_{ij}) at *input*. If the original array must be retained, one should store a duplicate of it in another array.

In the procedure *Naive_Gauss* for naive Gaussian elimination from Section 7.1, the right-hand side $\boldsymbol{b}$ was modified during the forward elimination phase; however, this was not done in the procedure *Gauss*. Therefore, we need to update $\boldsymbol{b}$ before considering the back substitution phase. For simplicity, we discuss updating $\boldsymbol{b}$ for the naive forward elimination first. Stripping out the pseudocode from *Naive_Gauss* that involves the (b_i) array in the forward elimination phase, we obtain

```
for k = 1 to n − 1 do
    for i = k + 1 to n do
        b_i = b_i − a_ik b_k
    end for
end for
```

This updates the (b_i) array based on the stored multipliers from the (a_{ij}) array. When scaled partial pivoting is done in the forward elimination phase, such as in procedure *Gauss*, the multipliers for each step are not one below another in the (a_{ij}) array but are jumbled around. To unravel this situation, all we have to do is introduce the index array (ℓ_i) into the above pseudocode:

```
for k = 1 to n − 1 do
    for i = k + 1 to n do
        b_ℓi = b_ℓi − a_ℓik b_ℓk
    end for
end for
```

After the array b has been processed in the forward elimination, the back substitution process is carried out. It begins by solving the equation

$$a_{\ell_n,n} x_n = b_{\ell_n} \tag{6}$$

whence

$$x_n = \frac{b_{\ell_n}}{a_{\ell_n,n}}$$

Then the equation

$$a_{\ell_{n-1},n-1} x_{n-1} + a_{\ell_{n-1},n} x_n = b_{\ell_{n-1}}$$

is solved for x_{n-1}:

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

After $x_n, x_{n-1}, \ldots, x_{i+1}$ have been determined, x_i is found from the equation

$$a_{\ell_i,i}x_i + a_{\ell_i,i+1}x_{i+1} + \cdots + a_{\ell_i,n}x_n = b_{\ell_i}$$

whose solution is

$$x_i = \frac{1}{a_{\ell_i,i}}\left(b_{\ell_i} - \sum_{j=i+1}^{n} a_{\ell_i,j}x_j\right) \tag{7}$$

Except for the presence of the index array ℓ_i, this is similar to the back substitution formula (7) in Section 7.1 obtained for naive Gaussian elimination.

The procedure for processing the array b and performing the back substitution phase is given next:

```
procedure Solve(n, (a_ij), (ℓ_i), (b_i), (x_i))
integer i, k, n;      real sum
real array (a_ij)_{1:n×1:n}, (ℓ_i)_{1:n}, (b_i)_{1:n}, (x_i)_{1:n}
for k = 1 to n − 1 do
    for i = k + 1 to n do
        b_{ℓ_i} ← b_{ℓ_i} − a_{ℓ_i,k} b_{ℓ_k}
    end for
end for
x_n ← b_{ℓ_n} / a_{ℓ_n,n}
for i = n − 1 to 1 step −1 do
    sum ← b_{ℓ_i}
    for j = i + 1 to n do
        sum ← sum − a_{ℓ_i,j} x_j
    end for
    x_i ← sum / a_{ℓ_i,i}
end for
end procedure Solve
```

Here, the first loop carries out the forward elimination process on array (b_i), using arrays (a_{ij}) and (ℓ_i) that result from procedure *Gauss*. The next line carries out the solution of Equation (6). The final part carries out Equation (7). The variable *sum* is a temporary variable for accumulating the terms in parentheses.

As with most pseudocode in this book, those in this chapter contain only the basic ingredients for good mathematical software. They are not suitable as *production* code for various reasons. For example, procedures for optimizing code are ignored. Furthermore, the procedures do not give warnings for difficulties that may be encountered, such as division by zero! General-purpose software should be **robust**; that is, it should anticipate every possible situation and deal with each in a prescribed way. (See Computer Problem 7.2.11.)

Long Operation Count

Solving large systems of linear equations can be expensive in computer time. To understand why, let us perform an operation count on the two algorithms whose codes have been given. We count only multiplications and divisions (long operations) because they are more time

consuming than addition. Furthermore, we lump multiplications and divisions together even though division is slower than multiplication. In modern computers, all floating-point operations are done in hardware, so long operations may not be as significant, but this still gives an indication of the operational cost of Gaussian elimination.

Consider first procedure *Gauss*. In step 1, the choice of a pivot element requires the calculation of n ratios—that is, n divisions. Then for rows $\ell_2, \ell_3, \ldots, \ell_n$, we first compute a multiplier and then subtract from row ℓ_i that multiplier times row ℓ_1. The zero that is being created in this process is *not* computed. So the elimination requires $n-1$ multiplications per row. If we include the calculation of the multiplier, there are n long operations (divisions or multiplications) per row. There are $n-1$ rows to be processed for a total of $n(n-1)$ operations. If we add the cost of computing the ratios, a total of n^2 operations is needed for step 1.

The next step is like step 1 except that row ℓ_1 is not affected, nor is the column of multipliers created and stored in step 1. So step 2 will require $(n-1)^2$ multiplications or divisions because it operates on a system without row ℓ_1 and without column 1. Continuing this reasoning, we conclude that the total number of long operations for procedure *Gauss* is

$$n^2 + (n-1)^2 + (n-2)^2 + \cdots + 4^2 + 3^2 + 2^2 = \frac{n}{6}(n+1)(2n+1) - 1 \approx \frac{n^3}{3}$$

(The derivation of this formula is outlined in Problem 7.2.16.) Note that the number of long operations in this procedure grows like $n^3/3$, the dominant term.

Now consider procedure *Solve*. The forward processing of the array (b_i) involves $n-1$ steps. The first step contains $n-1$ multiplications, the second contains $n-2$ multiplications, and so on. The total of the forward processing of array (b_i) is thus

$$(n-1) + (n-2) + \cdots + 3 + 2 + 1 = \frac{n}{2}(n-1)$$

(See Problem 7.2.15.) In the back substitution procedure, one long operation is involved in the first step, two in the second step, and so on. The total is

$$1 + 2 + 3 + \cdots + n = \frac{n}{2}(n+1)$$

Thus, procedure *Solve* involves altogether n^2 long operations. To summarize:

■ THEOREM 1

THEOREM ON LONG OPERATIONS

The forward elimination phase of the Gaussian elimination algorithm with scaled partial pivoting, if applied only to the $n \times n$ coefficient array, involves approximately $n^3/3$ long operations (multiplications or divisions). Solving for x requires an additional n^2 long operations.

An intuitive way to think of this result is that the Gaussian elimination algorithm involves a triply nested for-loop. So an $\mathcal{O}(n^3)$ algorithmic structure is driving the elimination process, and the work is heavily influenced by the cube of the number of equations and unknowns.

Numerical Stability

The **numerical stability** of a numerical algorithm is related to the accuracy of the procedure. An algorithm can have different levels of numerical stability because many computations can be achieved in various ways that are algebraically equivalent but may produce different results. A robust numerical algorithm with a high level of numerical stability is desirable. Gaussian elimination is numerically stable for strictly diagonally dominant matrices or symmetric positive definite matrices. (These are properties we will present in Sections 7.3 and 8.1, respectively.) For matrices with a general dense structure, Gaussian elimination with partial pivoting is usually numerically stable in practice. Nevertheless, there exist unstable pathological examples in which it may fail. For additional details, see Golub and Van Loan [1996] and Highman [1996].

An early version of Gaussian elimination can be found in a Chinese mathematics text dating from 150 B.C.

Scaling

Readers should not confuse *scaling* in Gaussian elimination (which is *not* recommended) with our discussion of *scaled* partial pivoting in Gaussian elimination.

The word **scaling** has more than one meaning. It could mean actually dividing each row by its maximum element in absolute value. We certainly do not advocate that. In other words, we do not recommend scaling of the matrix at all. However, we do compute a scale array and use it in selecting the pivot element in Gaussian elimination with scaled partial pivoting. We do not actually scale the rows; we just keep a vector of the "row infinity norms," that is, the maximum element in absolute value for each row. This and the need for a vector of indices to keep track of the pivot rows make the algorithm somewhat complicated, but that is the price to be paid for some degree of robustness in the procedure.

The simple 2×2 example in Equation (4) shows that scaling does not help in choosing a good pivot row. In this example, scaling is of no use. Scaling of the rows is contemplated in Problem 7.2.23 and Computer Problem 7.2.17. Notice that this procedure requires at least n^2 arithmetic operations. Again, we are not recommending it for a general-purpose code.

Some codes actually move the rows around in storage. Because that should not be done in practice, we do not do it in the code, since it might be misleading. Also, to avoid misleading the casual reader, we called our initial algorithm (in the preceding section) *naive*, hoping that nobody would mistake it for a reliable code.

Summary

(1) In performing Gaussian elimination, partial pivoting is highly recommended to avoid zero pivots and small pivots. In Gaussian elimination with scaled partial pivoting, we use a **scale vector** $\boldsymbol{s} = [s_1, s_2, \ldots, s_n]^T$ in which

$$s_i = \max_{1 \leq j \leq n} |a_{ij}| \qquad (1 \leq i \leq n)$$

and an **index vector** $\boldsymbol{\ell} = [\ell_1, \ell_2, \ldots, \ell_n]^T$, initially set as $\boldsymbol{\ell} = [1, 2, \ldots, n]^T$. The scale vector or array is set once at the beginning of the algorithm. The elements in the index vector or array are interchanged rather than the rows of the matrix $\boldsymbol{A}$, which reduces the

amount of data movement considerably. The key step in the pivoting procedure is to select j to be the first index associated with the largest ratio in the set

$$\left\{ \frac{|a_{\ell_i,k}|}{s_{\ell_i}} : k \leqq i \leqq n \right\}$$

and interchange ℓ_j with ℓ_k in the index array $\boldsymbol{\ell}$. Then use multipliers

$$\frac{a_{\ell_i,k}}{a_{\ell_k,k}}$$

times row ℓ_k and subtract from equations ℓ_i for $k+1 \leqq i \leqq n$. The **forward elimination** from equation ℓ_i for $\ell_{k+1} \leqq \ell_i \leqq \ell_n$ is

$$\begin{cases} a_{\ell_i,j} \leftarrow a_{\ell_i,j} - (a_{\ell_i,k}/a_{kk})a_{kj} & (\ell_k \leqq \ell_j \leqq \ell_n) \\ b_{\ell_i} \leftarrow b_{\ell_i} - (a_{\ell_i,k}/a_{\ell_k k})b_{\ell_k} \end{cases}$$

The steps involving the vector $\boldsymbol{b}$ are usually done separately just before the back substitution phase, which we call *updating* the right-hand side. The **back substitution** is

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

(2) For an $n \times n$ system of linear equations $\boldsymbol{Ax} = \boldsymbol{b}$, the forward elimination phase of the Gaussian elimination with scaled partial pivoting involves approximately $n^3/3$ long operations (multiplications or divisions), whereas the back substitution requires only n^2 long operations.

Problems 7.2

[a] **1.** Show how Gaussian elimination with scaled partial pivoting works on the following matrix $\boldsymbol{A}$:

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

[a] **2.** Solve the following system using Gaussian elimination with scaled partial pivoting:

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

Show intermediate matrices at each step.

[a] **3.** Carry out Gaussian elimination with scaled partial pivoting on the matrix

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

Show intermediate matrices.

does not. To preserve stability, the Gauss-Huard method incorporates a pivoting strategy using column interchanges. An error analysis shows that the Gauss-Huard method is as stable as Gauss-Jordan elimination with an appropriate pivoting strategy. Read about these algorithms in papers by Dekker and Hoffmann [1989], Dekker, Hoffmann, and Potma [1997], Hoffmann [1989], and Huard [1979]. Carry out some numerical experiments by programming and testing the Gauss-Jordan and Gauss-Huard algorithms on some dense linear systems.

25. Solve System (5) using mathematical software routines based on Gaussian elimination such as found in Matlab, Maple, or Mathematica. There are a large number of computer programs and software packages for solving linear systems, each of which may use a slightly different pivoting strategy.

7.3 Tridiagonal and Banded Systems

In many applications, including several that are considered later on, extremely large linear systems that have a **banded** structure are encountered. Banded matrices often occur in solving ordinary and partial differential equations. It is advantageous to develop computer codes specifically designed for such linear systems, since they reduce the amount of storage used.

Of practical importance is the **tridiagonal** system. Here, all the nonzero elements in the coefficient matrix must be on the main diagonal or on the two diagonals just above and below the main diagonal (usually called **superdiagonal** and **subdiagonal**, respectively):

$$\begin{bmatrix} d_1 & c_1 & & & & & & \\ a_1 & d_2 & c_2 & & & & & \\ & a_2 & d_3 & c_3 & & & & \\ & & \ddots & \ddots & \ddots & & & \\ & & & a_{i-1} & d_i & c_i & & \\ & & & & \ddots & \ddots & \ddots & \\ & & & & & a_{n-2} & d_{n-1} & c_{n-1} \\ & & & & & & a_{n-1} & d_n \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ \vdots \\ x_i \\ \vdots \\ x_{n-1} \\ x_n \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ b_3 \\ \vdots \\ b_i \\ \vdots \\ b_{n-1} \\ b_n \end{bmatrix} \tag{1}$$

(All elements not in the displayed diagonals are 0's.) A tridiagonal matrix is characterized by the condition $a_{ij} = 0$ if $|i - j| \geqq 2$. In general, a matrix is said to have a **banded structure** if there is an integer k (less than n) such that $a_{ij} = 0$ whenever $|i - j| \geqq k$.

The storage requirements for a banded matrix are less than those for a general matrix of the same size. Thus, an $n \times n$ diagonal matrix requires only n memory locations in the computer, and a tridiagonal matrix requires only $3n - 2$. This fact is important if banded matrices of very large order are being used.

For banded matrices, the Gaussian elimination algorithm can be made very efficient if it is known beforehand that pivoting is unnecessary. This situation occurs often enough to justify special procedures. Here, we develop a code for the tridiagonal system and give a listing for the *pentadiagonal* system (in which $a_{ij} = 0$ if $|i - j| \geqq 3$).

Tridiagonal Systems

The routine to be described now is called procedure *Tri*. It is designed to solve a system of n linear equations in n unknowns, as shown in Equation (1). Both the forward elimination phase and the back substitution phase are incorporated in the procedure, and *no* pivoting is used; that is, the pivot equations are those given by the natural ordering $\{1, 2, \ldots, n\}$. Thus, naive Gaussian elimination is used.

In step 1, we subtract a_1/d_1 times row 1 from row 2, thus creating a 0 in the a_1 position. Only the entries d_2 and b_2 are altered. Observe that c_2 is *not* altered. In step 2, the process is repeated, using the new row 2 as the pivot row. Here is how the d_i's and b_i's are altered in each step:

$$\begin{cases} d_2 \leftarrow d_2 - \left(\dfrac{a_1}{d_1}\right) c_1 \\ b_2 \leftarrow b_2 - \left(\dfrac{a_1}{d_1}\right) b_1 \end{cases}$$

In general, we obtain

$$\begin{cases} d_i \leftarrow d_i - \left(\dfrac{a_{i-1}}{d_{i-1}}\right) c_{i-1} \\ b_i \leftarrow b_i - \left(\dfrac{a_{i-1}}{d_{i-1}}\right) b_{i-1} \qquad (2 \leqq i \leqq n) \end{cases}$$

At the end of the forward elimination phase, the form of the system is as follows:

$$\begin{bmatrix} d_1 & c_1 & & & & & & & \\ & d_2 & c_2 & & & & & & \\ & & d_3 & c_3 & & & & & \\ & & & \ddots & \ddots & & & & \\ & & & & d_i & c_i & & & \\ & & & & & \ddots & \ddots & & \\ & & & & & & d_{n-1} & c_{n-1} \\ & & & & & & & d_n \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ \vdots \\ x_i \\ \vdots \\ x_{n-1} \\ x_n \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ b_3 \\ \vdots \\ b_i \\ \vdots \\ b_{n-1} \\ b_n \end{bmatrix}$$

Of course, the b_i's and d_i's are not as they were at the beginning of this process, but the c_i's are. The back substitution phase solves for $x_n, x_{n-1}, \ldots, x_1$ as follows:

$$\begin{aligned} x_n &\leftarrow \frac{b_n}{d_n} \\ x_{n-1} &\leftarrow \frac{1}{d_{n-1}} (b_{n-1} - c_{n-1} x_n) \end{aligned}$$

Finally, we obtain

$$x_i \leftarrow \frac{1}{d_i} (b_i - c_i x_{i+1}) \qquad (i = n-1, n-2, \ldots, 1)$$

In procedure *Tri* for a tridiagonal system, we use single-dimensioned arrays (a_i), (d_i), and (c_i) for the diagonals in the coefficient matrix and array (b_i) for the right-hand side, and store the solution in array (x_i).

```
procedure Tri(n, (a_i), (d_i), (c_i), (b_i), (x_i))
integer i, n;   real xmult
real array (a_i)_{1:n}, (d_i)_{1:n}, (c_i)_{1:n}, (b_i)_{1:n}, (x_i)_{1:n}
for i = 2 to n do
    xmult ← a_{i-1}/d_{i-1}
    d_i ← d_i − (xmult)c_{i-1}
    b_i ← b_i − (xmult)b_{i-1}
end for
x_n ← b_n/d_n
for i = n − 1 to 1 step −1 do
    x_i ← (b_i − c_i x_{i+1})/d_i
end for
end procedure Tri
```

Notice that the original data in arrays (d_i) and (b_i) have been changed.

A symmetric tridiagonal system arises in the cubic spline development of Chapter 9 and elsewhere. A general symmetric tridiagonal system has the form

$$\begin{bmatrix} d_1 & c_1 & & & & & & \\ c_1 & d_2 & c_2 & & & & & \\ & c_2 & d_3 & c_3 & & & & \\ & & \ddots & \ddots & \ddots & & & \\ & & & c_{i-1} & d_i & c_i & & \\ & & & & \ddots & \ddots & \ddots & \\ & & & & & c_{n-2} & d_{n-1} & c_{n-1} \\ & & & & & & c_{n-1} & d_n \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ \vdots \\ x_i \\ \vdots \\ x_{n-1} \\ x_n \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ b_3 \\ \vdots \\ b_i \\ \vdots \\ b_{n-1} \\ b_n \end{bmatrix} \tag{2}$$

One could overwrite the right-hand side vector $\boldsymbol{b}$ with the solution vector $\boldsymbol{x}$ as well. Thus, a symmetric linear system can be solved with a procedure call of the form

```
call Tri(n, (c_i), (d_i), (c_i), (b_i), (b_i))
```

which reduces the number of linear arrays from five to three.

Strictly Diagonal Dominance

Since procedure *Tri* does not involve pivoting, it is natural to ask whether it is likely to fail. Simple examples can be given to illustrate failure because of attempted division by zero even though the coefficient matrix in Equation (1) is nonsingular. On the other hand, it is not easy to give the weakest possible conditions on this matrix to guarantee the success of the algorithm. We content ourselves with one property that is easily checked and commonly encountered. If the tridiagonal coefficient matrix is diagonally dominant, then procedure *Tri* will not encounter zero divisors.

Cholesky Factorization

Any symmetric matrix that has an $\boldsymbol{LU}$ factorization in which $\boldsymbol{L}$ is unit lower triangular, has an $\boldsymbol{LDL}^T$ factorization. The Cholesky factorization $\boldsymbol{A} = \boldsymbol{LL}^T$ is a simple consequence of it for the case in which $\boldsymbol{A}$ is symmetric and positive definite.

Suppose in the factorization $\boldsymbol{A} = \boldsymbol{LU}$ the matrix $\boldsymbol{L}$ is lower triangular and the matrix $\boldsymbol{U}$ is upper triangular. When $\boldsymbol{L}$ is *unit* lower triangular, it is called the **Doolittle factorization**. When $\boldsymbol{U}$ is *unit* upper triangular, it goes by the name **Crout factorization**. In the case in which $\boldsymbol{A}$ is symmetric positive definite and $\boldsymbol{U} = \boldsymbol{L}^T$, it is called the **Cholesky factorization**. The mathematician André Louis Cholesky proved the following result.

■ **THEOREM 2**

CHOLESKY THEOREM ON $\boldsymbol{LL}^T$ FACTORIZATION

If $\boldsymbol{A}$ is a real, symmetric, and positive definite matrix, then it has a unique factorization, $\boldsymbol{A} = \boldsymbol{LL}^T$, in which $\boldsymbol{L}$ is lower triangular with a positive diagonal.

Recall that a matrix $\boldsymbol{A}$ is **symmetric and positive definite** if $\boldsymbol{A} = \boldsymbol{A}^T$ and $x^T\boldsymbol{A}x > 0$ for every nonzero vector x. It follows at once that $\boldsymbol{A}$ is nonsingular because $\boldsymbol{A}$ obviously cannot map any nonzero vector into 0. Moreover, by considering special vectors of the form $x = (x_1, x_2, \ldots, x_k, 0, 0, \ldots, 0)^T$, we see that the leading principal minors of $\boldsymbol{A}$ are also positive definite. Theorem 1 implies that $\boldsymbol{A}$ has an $\boldsymbol{LU}$ decomposition. By the symmetry of $\boldsymbol{A}$, we then have, from the previous discussion, $\boldsymbol{A} = \boldsymbol{LDL}^T$. It can be shown that $\boldsymbol{D}$ is positive definite, and thus its elements d_{ii} are positive. Denoting by $\boldsymbol{D}^{1/2}$ the diagonal matrix whose diagonal elements are $\sqrt{d_{ii}}$, we have $\boldsymbol{A} = \widetilde{\boldsymbol{L}}\widetilde{\boldsymbol{L}}^T$ where $\widetilde{\boldsymbol{L}} \equiv \boldsymbol{LD}^{1/2}$, which is the Cholesky factorization. We leave the proof of uniqueness to the reader.

The algorithm for the Cholesky factorization is a special case of the general $\boldsymbol{LU}$ factorization algorithm. If $\boldsymbol{A}$ is real, symmetric, and positive definite, then by Theorem 2, it has a unique factorization of the form $\boldsymbol{A} = \boldsymbol{LL}^T$, in which $\boldsymbol{L}$ is lower triangular and has positive diagonal. Thus, in the equation $\boldsymbol{A} = \boldsymbol{LU}$, $\boldsymbol{U} = \boldsymbol{L}^T$. In the kth step of the general algorithm, the diagonal entry is computed by

$$\ell_{kk} = \left(a_{kk} - \sum_{s=1}^{k-1} \ell_{ks}^2 \right)^{1/2} \tag{22}$$

The algorithm for the **Cholesky factorization** will then be as follows:

integer i, k, n, s; **real array** $(a_{ij})_{1:n\times 1:n}, (\ell_{ij})_{1:n\times 1:n}$
for $k = 1$ **to** n **do**
$$\ell_{kk} \leftarrow \left(a_{kk} - \sum_{s=1}^{k-1} \ell_{ks}^2 \right)^{1/2}$$
for $i = k + 1$ **to** n **do**
$$\ell_{ik} \leftarrow \left(a_{ik} - \sum_{s=1}^{k-1} \ell_{is}\ell_{ks} \right) \Big/ \ell_{kk}$$
end do
end do

Theorem 2 guarantees that $\ell_{kk} > 0$. Observe that Equation (22) gives us the following bound:

$$a_{kk} = \sum_{s=1}^{k} \ell_{ks}^2 \geqq \ell_{kj}^2 \qquad (j \leqq k)$$

from which we conclude that

$$|\ell_{kj}| \leqq \sqrt{a_{kk}} \qquad (1 \leqq j \leqq k)$$

Hence, any element of $\boldsymbol{L}$ is bounded by the square root of a corresponding diagonal element in $\boldsymbol{A}$. This implies that the elements of $\boldsymbol{L}$ do not become large relative to $\boldsymbol{A}$ even without any pivoting. In the Cholesky algorithm (and the Doolittle algorithms), the dot products of vectors should be computed in double precision to avoid a buildup of roundoff errors.

EXAMPLE 2 Determine the Cholesky factorization of the matrix in Example 1.

Solution Using the results from Example 1, we write

$$\boldsymbol{A} = \boldsymbol{L}\boldsymbol{D}\boldsymbol{L}^T = (\boldsymbol{L}\boldsymbol{D}^{1/2})(\boldsymbol{D}^{1/2}\boldsymbol{L}^T) = \widetilde{\boldsymbol{L}}\widetilde{\boldsymbol{L}}^T$$

where

$$\begin{aligned}
\widetilde{\boldsymbol{L}} &= \boldsymbol{L}\boldsymbol{D}^{1/2} \\
&= \begin{bmatrix} 1 & 0 & 0 & 0 \\ \frac{3}{4} & 1 & 0 & 0 \\ \frac{1}{2} & \frac{2}{3} & 1 & 0 \\ \frac{1}{4} & \frac{1}{3} & \frac{1}{2} & 1 \end{bmatrix} \begin{bmatrix} 2 & 0 & 0 & 0 \\ 0 & \frac{1}{2}\sqrt{3} & 0 & 0 \\ 0 & 0 & \sqrt{\frac{2}{3}} & 0 \\ 0 & 0 & 0 & \frac{1}{\sqrt{2}} \end{bmatrix} \\
&= \begin{bmatrix} 2 & 0 & 0 & 0 \\ \frac{3}{2} & \frac{1}{2}\sqrt{3} & 0 & 0 \\ 1 & \frac{1}{3}\sqrt{3} & \sqrt{\frac{2}{3}} & 0 \\ \frac{1}{2} & \frac{1}{6}\sqrt{3} & \frac{1}{2}\sqrt{\frac{2}{3}} & \frac{1}{\sqrt{2}} \end{bmatrix} = \begin{bmatrix} 2.0000 & 0 & 0 & 0 \\ 1.5000 & 0.8660 & 0 & 0 \\ 1.0000 & 0.5774 & 0.8165 & 0 \\ 0.5000 & 0.2887 & 0.4082 & 0.7071 \end{bmatrix}
\end{aligned}$$

Clearly, $\widetilde{\boldsymbol{L}}$ is the lower triangular matrix in the Cholesky factorization $\boldsymbol{A} = \widetilde{\boldsymbol{L}}\widetilde{\boldsymbol{L}}^T$. ■

Multiple Right-Hand Sides

Many software packages for solving linear systems allow the input of multiple right-hand sides. Suppose an $n \times m$ matrix $\boldsymbol{B}$ is

$$\boldsymbol{B} = [\boldsymbol{b}^{(1)}, \boldsymbol{b}^{(2)}, \ldots, \boldsymbol{b}^{(m)}]$$

in which each column corresponds to a right-hand side of the m linear systems

$$\boldsymbol{A}\boldsymbol{x}^{(j)} = \boldsymbol{b}^{(j)}$$

for $1 \leqq j \leqq m$. Thus, we can write

$$\boldsymbol{A}[\boldsymbol{x}^{(1)}, \boldsymbol{x}^{(2)}, \ldots, \boldsymbol{x}^{(m)}] = [\boldsymbol{b}^{(1)}, \boldsymbol{b}^{(2)}, \ldots, \boldsymbol{b}^{(m)}]$$

or

$$\boldsymbol{AX} = \boldsymbol{B}$$

For example, procedure *Gauss* can be used once to produce a factorization of $\boldsymbol{A}$, and procedure *Solve* can be used m times with right-hand side vectors $\boldsymbol{b}^{(j)}$ to find the m solution vectors $\boldsymbol{x}^{(j)}$ for $1 \leqq j \leqq m$. Since the factorization phase can be done in $\frac{1}{3}n^3$ long operations while each of the back substitution phases requires n^2 long operations, this entire process can be done in $\frac{1}{3}n^3 + mn^2$ long operations. This is much less than $m\left(\frac{1}{3}n^3 + n^2\right)$, which is what it would take if each of the m linear systems were solved separately.

Computing A^{-1}

In some applications, such as in statistics, it may be necessary to compute the inverse of a matrix $\boldsymbol{A}$ and explicitly display it as $\boldsymbol{A}^{-1}$. This can be done by using procedures *Gauss* and *Solve*. If an $n \times n$ matrix $\boldsymbol{A}$ has an inverse, it is an $n \times n$ matrix $\boldsymbol{X}$ with the property that

$$\boldsymbol{AX} = \boldsymbol{I} \tag{23}$$

where $\boldsymbol{I}$ is the identity matrix. If $\boldsymbol{x}^{(j)}$ denotes the jth column of $\boldsymbol{X}$ and $\boldsymbol{I}^{(j)}$ denotes the jth column of $\boldsymbol{I}$, then matrix Equation (23) can be written as

$$\boldsymbol{A}[\boldsymbol{x}^{(1)}, \boldsymbol{x}^{(2)}, \ldots, \boldsymbol{x}^{(n)}] = [\boldsymbol{I}^{(1)}, \boldsymbol{I}^{(2)}, \ldots, \boldsymbol{I}^{(n)}]$$

This can be written as n linear systems of equations of the form

$$\boldsymbol{Ax}^{(j)} = \boldsymbol{I}^{(j)} \qquad (1 \leqq j \leqq n)$$

Now use procedure *Gauss* once to produce a factorization of $\boldsymbol{A}$, and use procedure *Solve* n times with the right-hand side vectors $\boldsymbol{I}^{(j)}$ for $1 \leqq j \leqq n$. This is equivalent to solving, one at a time, for the columns of $\boldsymbol{A}^{-1}$, which are $\boldsymbol{x}^{(j)}$. Hence,

$$\boldsymbol{A}^{-1} = [\boldsymbol{x}^{(1)}, \boldsymbol{x}^{(2)}, \ldots, \boldsymbol{x}^{(n)}]$$

A word of caution on computing the inverse of a matrix: In solving a linear system $\boldsymbol{Ax} = \boldsymbol{b}$, it is not advisable to determine $\boldsymbol{A}^{-1}$ and then compute the matrix-vector product $\boldsymbol{x} = \boldsymbol{A}^{-1}\boldsymbol{b}$ because this requires many unnecessary calculations, compared to directly solving $\boldsymbol{Ax} = \boldsymbol{b}$ for $\boldsymbol{x}$.

Example Using Software Packages

A **permutation matrix** is an $n \times n$ matrix $\boldsymbol{P}$ that arises from the identity matrix by permuting its rows. It then turns out that permuting the rows of any $n \times n$ matrix $\boldsymbol{A}$ can be accomplished by multiplying $\boldsymbol{A}$ on the left by $\boldsymbol{P}$. Every permutation matrix is nonsingular, since the rows still form a basis for $\mathbb{R}^n$. When Gaussian elimination with row pivoting is performed on a matrix $\boldsymbol{A}$, the result is expressible as

$$\boldsymbol{PA} = \boldsymbol{LU}$$

where $\boldsymbol{L}$ is lower triangular and $\boldsymbol{U}$ is upper triangular. The matrix $\boldsymbol{PA}$ is $\boldsymbol{A}$ with its rows rearranged. If we have the $\boldsymbol{LU}$ factorization of $\boldsymbol{PA}$, how do we solve the system $\boldsymbol{Ax} = \boldsymbol{b}$?