The advantage of Gaussian elimination and backsubstitution over Gauss-Jordan elimination is simply that the former is faster in raw operations count: The innermost loops of Gauss-Jordan elimination, each containing one subtraction and one multiplication, are executed N^3 and N^2M times (where there are N equations and M unknowns). The corresponding loops in Gaussian elimination are executed only $\frac{1}{3}N^3$ times (only half the matrix is reduced, and the increasing numbers of predictable zeros reduce the count to one-third), and $\frac{1}{2}N^2M$ times, respectively. Each backsubstitution of a right-hand side is $\frac{1}{2}N^2$ executions of a similar loop (one multiplication plus one subtraction). For $M \ll N$ (only a few right-hand sides) Gaussian elimination thus has about a factor three advantage over Gauss-Jordan. (We could reduce this advantage to a factor 1.5 by *not* computing the inverse matrix as part of the Gauss-Jordan scheme.)

For computing the inverse matrix (which we can view as the case of $M = N$ right-hand sides, namely the N unit vectors which are the columns of the identity matrix), Gaussian elimination and backsubstitution at first glance require $\frac{1}{3}N^3$ (matrix reduction) $+\frac{1}{2}N^3$ (right-hand side manipulations) $+\frac{1}{2}N^3$ (N backsubstitutions) $= \frac{4}{3}N^3$ loop executions, which is more than the N^3 for Gauss-Jordan. However, the unit vectors are quite special in containing all zeros except for one element. If this is taken into account, the right-side manipulations can be reduced to only $\frac{1}{6}N^3$ loop executions, and, for matrix inversion, the two methods have identical efficiencies.

Both Gaussian elimination and Gauss-Jordan elimination share the disadvantage that all right-hand sides must be known in advance. The LU decomposition method in the next section does not share that deficiency, and also has an equally small operations count, both for solution with any number of right-hand sides, and for matrix inversion. For this reason we will not implement the method of Gaussian elimination as a routine.

CITED REFERENCES AND FURTHER READING:

Ralston, A., and Rabinowitz, P. 1978, *A First Course in Numerical Analysis*, 2nd ed. (New York: McGraw-Hill), §9.3–1.

Isaacson, E., and Keller, H.B. 1966, *Analysis of Numerical Methods* (New York: Wiley), §2.1.

Johnson, L.W., and Riess, R.D. 1982, *Numerical Analysis*, 2nd ed. (Reading, MA: Addison-Wesley), §2.2.1.

Westlake, J.R. 1968, *A Handbook of Numerical Matrix Inversion and Solution of Linear Equations* (New York: Wiley).

2.3 LU Decomposition and Its Applications

Suppose we are able to write the matrix $\mathbf{A}$ as a product of two matrices,

$$\mathbf{L} \cdot \mathbf{U} = \mathbf{A} \tag{2.3.1}$$

where $\mathbf{L}$ is *lower triangular* (has elements only on the diagonal and below) and $\mathbf{U}$ is *upper triangular* (has elements only on the diagonal and above). For the case of a 4×4 matrix $\mathbf{A}$, for example, equation (2.3.1) would look like this:

$$\begin{bmatrix} \alpha_{11} & 0 & 0 & 0 \\ \alpha_{21} & \alpha_{22} & 0 & 0 \\ \alpha_{31} & \alpha_{32} & \alpha_{33} & 0 \\ \alpha_{41} & \alpha_{42} & \alpha_{43} & \alpha_{44} \end{bmatrix} \cdot \begin{bmatrix} \beta_{11} & \beta_{12} & \beta_{13} & \beta_{14} \\ 0 & \beta_{22} & \beta_{23} & \beta_{24} \\ 0 & 0 & \beta_{33} & \beta_{34} \\ 0 & 0 & 0 & \beta_{44} \end{bmatrix} = \begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{bmatrix} \tag{2.3.2}$$

We can use a decomposition such as (2.3.1) to solve the linear set

$$\mathbf{A} \cdot \mathbf{x} = (\mathbf{L} \cdot \mathbf{U}) \cdot \mathbf{x} = \mathbf{L} \cdot (\mathbf{U} \cdot \mathbf{x}) = \mathbf{b} \tag{2.3.3}$$

by first solving for the vector $\mathbf{y}$ such that

$$\mathbf{L} \cdot \mathbf{y} = \mathbf{b} \tag{2.3.4}$$

and then solving

$$\mathbf{U} \cdot \mathbf{x} = \mathbf{y} \tag{2.3.5}$$

What is the advantage of breaking up one linear set into two successive ones? The advantage is that the solution of a triangular set of equations is quite trivial, as we have already seen in §2.2 (equation 2.2.4). Thus, equation (2.3.4) can be solved by *forward substitution* as follows,

$$\begin{aligned} y_1 &= \frac{b_1}{\alpha_{11}} \\ y_i &= \frac{1}{\alpha_{ii}} \left[b_i - \sum_{j=1}^{i-1} \alpha_{ij} y_j \right] \qquad i = 2, 3, \ldots, N \end{aligned} \tag{2.3.6}$$

while (2.3.5) can then be solved by *backsubstitution* exactly as in equations (2.2.2)–(2.2.4),

$$\begin{aligned} x_N &= \frac{y_N}{\beta_{NN}} \\ x_i &= \frac{1}{\beta_{ii}} \left[y_i - \sum_{j=i+1}^{N} \beta_{ij} x_j \right] \qquad i = N-1, N-2, \ldots, 1 \end{aligned} \tag{2.3.7}$$

Equations (2.3.6) and (2.3.7) total (for each right-hand side $\mathbf{b}$) N^2 executions of an inner loop containing one multiply and one add. If we have N right-hand sides which are the unit column vectors (which is the case when we are inverting a matrix), then taking into account the leading zeros reduces the total execution count of (2.3.6) from $\frac{1}{2}N^3$ to $\frac{1}{6}N^3$, while (2.3.7) is unchanged at $\frac{1}{2}N^3$.

Notice that, once we have the LU decomposition of $\mathbf{A}$, we can solve with as many right-hand sides as we then care to, one at a time. This is a distinct advantage over the methods of §2.1 and §2.2.

Performing the LU Decomposition

How then can we solve for **L** and **U**, given **A**? First, we write out the i, jth component of equation (2.3.1) or (2.3.2). That component always is a sum beginning with

$$\alpha_{i1}\beta_{1j} + \cdots = a_{ij}$$

The number of terms in the sum depends, however, on whether i or j is the smaller number. We have, in fact, the three cases,

$$i < j: \qquad \alpha_{i1}\beta_{1j} + \alpha_{i2}\beta_{2j} + \cdots + \alpha_{ii}\beta_{ij} = a_{ij} \tag{2.3.8}$$

$$i = j: \qquad \alpha_{i1}\beta_{1j} + \alpha_{i2}\beta_{2j} + \cdots + \alpha_{ii}\beta_{jj} = a_{ij} \tag{2.3.9}$$

$$i > j: \qquad \alpha_{i1}\beta_{1j} + \alpha_{i2}\beta_{2j} + \cdots + \alpha_{ij}\beta_{jj} = a_{ij} \tag{2.3.10}$$

Equations (2.3.8)–(2.3.10) total N^2 equations for the $N^2 + N$ unknown α's and β's (the diagonal being represented twice). Since the number of unknowns is greater than the number of equations, we are invited to specify N of the unknowns arbitrarily and then try to solve for the others. In fact, as we shall see, it is always possible to take

$$\alpha_{ii} \equiv 1 \qquad i = 1, \ldots, N \tag{2.3.11}$$

A surprising procedure, now, is *Crout's algorithm*, which quite trivially solves the set of $N^2 + N$ equations (2.3.8)–(2.3.11) for all the α's and β's by just arranging the equations in a certain order! That order is as follows:

- Set $\alpha_{ii} = 1$, $i = 1, \ldots, N$ (equation 2.3.11).
- For each $j = 1, 2, 3, \ldots, N$ do these two procedures: First, for $i = 1, 2, \ldots, j$, use (2.3.8), (2.3.9), and (2.3.11) to solve for β_{ij}, namely

$$\beta_{ij} = a_{ij} - \sum_{k=1}^{i-1} \alpha_{ik}\beta_{kj}. \tag{2.3.12}$$

  (When $i = 1$ in 2.3.12 the summation term is taken to mean zero.) Second, for $i = j+1, j+2, \ldots, N$ use (2.3.10) to solve for α_{ij}, namely

$$\alpha_{ij} = \frac{1}{\beta_{jj}}\left(a_{ij} - \sum_{k=1}^{j-1} \alpha_{ik}\beta_{kj}\right). \tag{2.3.13}$$

  Be sure to do both procedures before going on to the next j.

If you work through a few iterations of the above procedure, you will see that the α's and β's that occur on the right-hand side of equations (2.3.12) and (2.3.13) are already determined by the time they are needed. You will also see that every a_{ij} is used only once and never again. This means that the corresponding α_{ij} or β_{ij} can be stored in the location that the a used to occupy: the decomposition is "in place." [The diagonal unity elements α_{ii} (equation 2.3.11) are not stored at all.] In brief, Crout's method fills in the combined matrix of α's and β's,

$$\begin{bmatrix} \beta_{11} & \beta_{12} & \beta_{13} & \beta_{14} \\ \alpha_{21} & \beta_{22} & \beta_{23} & \beta_{24} \\ \alpha_{31} & \alpha_{32} & \beta_{33} & \beta_{34} \\ \alpha_{41} & \alpha_{42} & \alpha_{43} & \beta_{44} \end{bmatrix} \tag{2.3.14}$$

by columns from left to right, and within each column from top to bottom (see Figure 2.3.1).

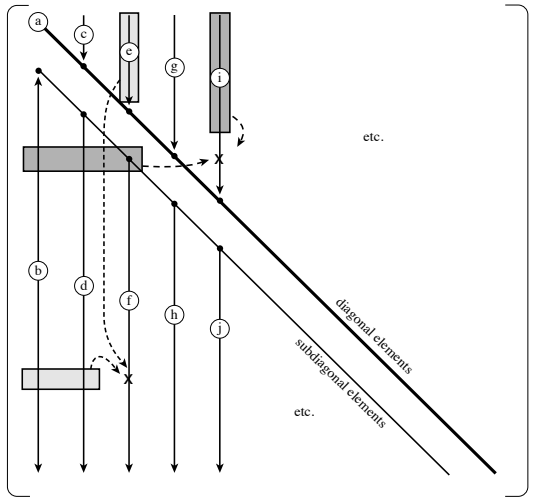

Figure 2.3.1. Crout's algorithm for *LU* decomposition of a matrix. Elements of the original matrix are modified in the order indicated by lower case letters: a, b, c, etc. Shaded boxes show the previously modified elements that are used in modifying two typical elements, each indicated by an "x".

What about pivoting? Pivoting (i.e., selection of a salubrious pivot element for the division in equation 2.3.13) is absolutely essential for the stability of Crout's method. Only partial pivoting (interchange of rows) can be implemented efficiently. However this is enough to make the method stable. This means, incidentally, that we don't actually decompose the matrix **A** into LU form, but rather we decompose a rowwise permutation of **A**. (If we keep track of what that permutation is, this decomposition is just as useful as the original one would have been.)

Pivoting is slightly subtle in Crout's algorithm. The key point to notice is that equation (2.3.12) in the case of $i = j$ (its final application) is *exactly the same* as equation (2.3.13) except for the division in the latter equation; in both cases the upper limit of the sum is $k = j - 1$ $(= i - 1)$. This means that we don't have to commit ourselves as to whether the diagonal element β_{jj} is the one that happens to fall on the diagonal in the first instance, or whether one of the (undivided) α_{ij}'s below it in the column, $i = j+1, \ldots, N$, is to be "promoted" to become the diagonal β. This can be decided after all the candidates in the column are in hand. As you should be able to guess by now, we will choose the largest one as the diagonal β (pivot element), then do all the divisions by that element *en masse*. This is *Crout's*

method with partial pivoting. Our implementation has one additional wrinkle: It initially finds the largest element in each row, and subsequently (when it is looking for the maximal pivot element) scales the comparison *as if* we had initially scaled all the equations to make their maximum coefficient equal to unity; this is the *implicit pivoting* mentioned in §2.1.

```
SUBROUTINE ludcmp(a,n,np,indx,d)
INTEGER n,np,indx(n),NMAX
REAL d,a(np,np),TINY
PARAMETER (NMAX=500,TINY=1.0e-20)  Largest expected n, and a small number.
    Given a matrix a(1:n,1:n), with physical dimension np by np, this routine replaces it by
    the LU decomposition of a rowwise permutation of itself. a and n are input. a is output,
    arranged as in equation (2.3.14) above; indx(1:n) is an output vector that records the
    row permutation effected by the partial pivoting; d is output as ±1 depending on whether
    the number of row interchanges was even or odd, respectively. This routine is used in
    combination with lubksb to solve linear equations or invert a matrix.
INTEGER i,imax,j,k
REAL aamax,dum,sum,vv(NMAX)            vv stores the implicit scaling of each row.
d=1.                                   No row interchanges yet.
do 12 i=1,n                            Loop over rows to get the implicit scaling informa-
    aamax=0.                               tion.
    do 11 j=1,n
        if (abs(a(i,j)).gt.aamax) aamax=abs(a(i,j))
    enddo 11
    if (aamax.eq.0.) pause 'singular matrix in ludcmp' No nonzero largest element.
    vv(i)=1./aamax                     Save the scaling.
enddo 12
do 19 j=1,n                            This is the loop over columns of Crout's method.
    do 14 i=1,j-1                      This is equation (2.3.12) except for i = j.
        sum=a(i,j)
        do 13 k=1,i-1
            sum=sum-a(i,k)*a(k,j)
        enddo 13
        a(i,j)=sum
    enddo 14
    aamax=0.                           Initialize for the search for largest pivot element.
    do 16 i=j,n                        This is i = j of equation (2.3.12) and i = j+1 ... N
        sum=a(i,j)                         of equation (2.3.13).
        do 15 k=1,j-1
            sum=sum-a(i,k)*a(k,j)
        enddo 15
        a(i,j)=sum
        dum=vv(i)*abs(sum)             Figure of merit for the pivot.
        if (dum.ge.aamax) then         Is it better than the best so far?
            imax=i
            aamax=dum
        endif
    enddo 16
    if (j.ne.imax)then                 Do we need to interchange rows?
        do 17 k=1,n                    Yes, do so...
            dum=a(imax,k)
            a(imax,k)=a(j,k)
            a(j,k)=dum
        enddo 17
        d=-d                           ...and change the parity of d.
        vv(imax)=vv(j)                 Also interchange the scale factor.
    endif
    indx(j)=imax
    if(a(j,j).eq.0.)a(j,j)=TINY
      If the pivot element is zero the matrix is singular (at least to the precision of the al-
      gorithm). For some applications on singular matrices, it is desirable to substitute TINY
      for zero.
```



```
    if(j.ne.n)then                     Now, finally, divide by the pivot element.
        dum=1./a(j,j)
        do 18 i=j+1,n
            a(i,j)=a(i,j)*dum
        enddo 18
    endif
enddo 19                               Go back for the next column in the reduction.
return
END
```

Here is the routine for forward substitution and backsubstitution, implementing equations (2.3.6) and (2.3.7).

```
SUBROUTINE lubksb(a,n,np,indx,b)
INTEGER n,np,indx(n)
REAL a(np,np),b(n)
    Solves the set of n linear equations A · X = B. Here a is input, not as the matrix A but
    rather as its LU decomposition, determined by the routine ludcmp. indx is input as the
    permutation vector returned by ludcmp. b(1:n) is input as the right-hand side vector B,
    and returns with the solution vector X. a, n, np, and indx are not modified by this routine
    and can be left in place for successive calls with different right-hand sides b. This routine
    takes into account the possibility that b will begin with many zero elements, so it is efficient
    for use in matrix inversion.
INTEGER i,ii,j,ll
REAL sum
ii=0                                   When ii is set to a positive value, it will become the in-
do 12 i=1,n                                dex of the first nonvanishing element of b. We now do
    ll=indx(i)                             the forward substitution, equation (2.3.6). The only new
    sum=b(ll)                              wrinkle is to unscramble the permutation as we go.
    b(ll)=b(i)
    if (ii.ne.0)then
        do 11 j=ii,i-1
            sum=sum-a(i,j)*b(j)
        enddo 11
    else if (sum.ne.0.) then
        ii=i                           A nonzero element was encountered, so from now on we will
    endif                                  have to do the sums in the loop above.
    b(i)=sum
enddo 12
do 14 i=n,1,-1                         Now we do the backsubstitution, equation (2.3.7).
    sum=b(i)
    do 13 j=i+1,n
        sum=sum-a(i,j)*b(j)
    enddo 13
    b(i)=sum/a(i,i)                    Store a component of the solution vector X.
enddo 14
return                                 All done!
END
```

The LU decomposition in `ludcmp` requires about $\frac{1}{3}N^3$ executions of the inner loops (each with one multiply and one add). This is thus the operation count for solving one (or a few) right-hand sides, and is a factor of 3 better than the Gauss-Jordan routine `gaussj` which was given in §2.1, and a factor of 1.5 better than a Gauss-Jordan routine (not given) that does not compute the inverse matrix. For inverting a matrix, the total count (including the forward and backsubstitution as discussed following equation 2.3.7 above) is $(\frac{1}{3} + \frac{1}{6} + \frac{1}{2})N^3 = N^3$, the same as `gaussj`.

To summarize, this is the preferred way to solve the linear set of equations $\mathbf{A} \cdot \mathbf{x} = \mathbf{b}$:

```
call ludcmp(a,n,np,indx,d)
call lubksb(a,n,np,indx,b)
```

The answer $\mathbf{x}$ will be returned in b. Your original matrix $\mathbf{A}$ will have been destroyed.

If you subsequently want to solve a set of equations with the same $\mathbf{A}$ but a different right-hand side $\mathbf{b}$, you repeat *only*

```
call lubksb(a,n,np,indx,b)
```

not, of course, with the original matrix $\mathbf{A}$, but with a and indx as were already returned from ludcmp.

Inverse of a Matrix

Using the above LU decomposition and backsubstitution routines, it is completely straightforward to find the inverse of a matrix column by column.

```
INTEGER np,indx(np)
REAL a(np,np),y(np,np)
...
do 12 i=1,n                          Set up identity matrix.
    do 11 j=1,n
        y(i,j)=0.
    enddo 11
    y(i,i)=1.
enddo 12
call ludcmp(a,n,np,indx,d)           Decompose the matrix just once.
do 13 j=1,n                          Find inverse by columns.
    call lubksb(a,n,np,indx,y(1,j))
      Note that FORTRAN stores two-dimensional matrices by column, so y(1,j) is the
      address of the jth column of y.
enddo 13
```

The matrix y will now contain the inverse of the original matrix a, which will have been destroyed. Alternatively, there is nothing wrong with using a Gauss-Jordan routine like gaussj (§2.1) to invert a matrix in place, again destroying the original. Both methods have practically the same operations count.

Incidentally, if you ever have the need to compute $\mathbf{A}^{-1} \cdot \mathbf{B}$ from matrices $\mathbf{A}$ and $\mathbf{B}$, you should LU decompose $\mathbf{A}$ and then backsubstitute with the columns of $\mathbf{B}$ instead of with the unit vectors that would give $\mathbf{A}$'s inverse. This saves a whole matrix multiplication, and is also more accurate.

Determinant of a Matrix

The determinant of an LU decomposed matrix is just the product of the diagonal elements,

$$\det = \prod_{j=1}^{N} \beta_{jj} \tag{2.3.15}$$

We don't, recall, compute the decomposition of the original matrix, but rather a decomposition of a rowwise permutation of it. Luckily, we have kept track of whether the number of row interchanges was even or odd, so we just preface the product by the corresponding sign. (You now finally know the purpose of returning d in the routine ludcmp.)

Calculation of a determinant thus requires one call to ludcmp, with *no* subsequent backsubstitutions by lubksb.

```
INTEGER np,indx(np)
REAL a(np,np)
...
call ludcmp(a,n,np,indx,d)           This returns d as ±1.
do 11 j=1,n
    d=d*a(j,j)
enddo 11
```

The variable d now contains the determinant of the original matrix a, which will have been destroyed.

For a matrix of any substantial size, it is quite likely that the determinant will overflow or underflow your computer's floating-point dynamic range. In this case you can modify the loop of the above fragment and (e.g.) divide by powers of ten, to keep track of the scale separately, or (e.g.) accumulate the sum of logarithms of the absolute values of the factors and the sign separately.

Complex Systems of Equations

If your matrix $\mathbf{A}$ is real, but the right-hand side vector is complex, say $\mathbf{b} + i\mathbf{d}$, then (i) LU decompose $\mathbf{A}$ in the usual way, (ii) backsubstitute $\mathbf{b}$ to get the real part of the solution vector, and (iii) backsubstitute $\mathbf{d}$ to get the imaginary part of the solution vector.

If the matrix itself is complex, so that you want to solve the system

$$(\mathbf{A} + i\mathbf{C}) \cdot (\mathbf{x} + i\mathbf{y}) = (\mathbf{b} + i\mathbf{d}) \tag{2.3.16}$$

then there are two possible ways to proceed. The best way is to rewrite ludcmp and lubksb as complex routines. Complex modulus substitutes for absolute value in the construction of the scaling vector vv and in the search for the largest pivot elements. Everything else goes through in the obvious way, with complex arithmetic used as needed.

A quick-and-dirty way to solve complex systems is to take the real and imaginary parts of (2.3.16), giving

$$\begin{aligned} \mathbf{A} \cdot \mathbf{x} - \mathbf{C} \cdot \mathbf{y} &= \mathbf{b} \\ \mathbf{C} \cdot \mathbf{x} + \mathbf{A} \cdot \mathbf{y} &= \mathbf{d} \end{aligned} \tag{2.3.17}$$

which can be written as a $2N \times 2N$ set of *real* equations,

$$\begin{pmatrix} \mathbf{A} & -\mathbf{C} \\ \mathbf{C} & \mathbf{A} \end{pmatrix} \cdot \begin{pmatrix} \mathbf{x} \\ \mathbf{y} \end{pmatrix} = \begin{pmatrix} \mathbf{b} \\ \mathbf{d} \end{pmatrix} \tag{2.3.18}$$

and then solved with `ludcmp` and `lubksb` in their present forms. This scheme is a factor of 2 inefficient in storage, since **A** and **C** are stored twice. It is also a factor of 2 inefficient in time, since the complex multiplies in a complexified version of the routines would each use 4 real multiplies, while the solution of a $2N \times 2N$ problem involves 8 times the work of an $N \times N$ one. If you can tolerate these factor-of-two inefficiencies, then equation (2.3.18) is an easy way to proceed.

CITED REFERENCES AND FURTHER READING:

Golub, G.H., and Van Loan, C.F. 1989, *Matrix Computations*, 2nd ed. (Baltimore: Johns Hopkins University Press), Chapter 4.

Dongarra, J.J., et al. 1979, *LINPACK User's Guide* (Philadelphia: S.I.A.M.).

Forsythe, G.E., Malcolm, M.A., and Moler, C.B. 1977, *Computer Methods for Mathematical Computations* (Englewood Cliffs, NJ: Prentice-Hall), §3.3, and p. 50.

Forsythe, G.E., and Moler, C.B. 1967, *Computer Solution of Linear Algebraic Systems* (Englewood Cliffs, NJ: Prentice-Hall), Chapters 9, 16, and 18.

Westlake, J.R. 1968, *A Handbook of Numerical Matrix Inversion and Solution of Linear Equations* (New York: Wiley).

Stoer, J., and Bulirsch, R. 1980, *Introduction to Numerical Analysis* (New York: Springer-Verlag), §4.2.

Ralston, A., and Rabinowitz, P. 1978, *A First Course in Numerical Analysis*, 2nd ed. (New York: McGraw-Hill), §9.11.

Horn, R.A., and Johnson, C.R. 1985, *Matrix Analysis* (Cambridge: Cambridge University Press).

2.4 Tridiagonal and Band Diagonal Systems of Equations

The special case of a system of linear equations that is *tridiagonal*, that is, has nonzero elements only on the diagonal plus or minus one column, is one that occurs frequently. Also common are systems that are *band diagonal*, with nonzero elements only along a few diagonal lines adjacent to the main diagonal (above and below).

For tridiagonal sets, the procedures of LU decomposition, forward- and back-substitution each take only $O(N)$ operations, and the whole solution can be encoded very concisely. The resulting routine `tridag` is one that we will use in later chapters.

Naturally, one does not reserve storage for the full $N \times N$ matrix, but only for the nonzero components, stored as three vectors. The set of equations to be solved is

$$\begin{bmatrix} b_1 & c_1 & 0 & \cdots & & & \\ a_2 & b_2 & c_2 & \cdots & & & \\ & & & \cdots & & & \\ & & & \cdots & a_{N-1} & b_{N-1} & c_{N-1} \\ & & & \cdots & 0 & a_N & b_N \end{bmatrix} \cdot \begin{bmatrix} u_1 \\ u_2 \\ \cdots \\ u_{N-1} \\ u_N \end{bmatrix} = \begin{bmatrix} r_1 \\ r_2 \\ \cdots \\ r_{N-1} \\ r_N \end{bmatrix} \tag{2.4.1}$$

Notice that a_1 and c_N are undefined and are not referenced by the routine that follows.


```
SUBROUTINE tridag(a,b,c,r,u,n)
INTEGER n,NMAX
REAL a(n),b(n),c(n),r(n),u(n)
PARAMETER (NMAX=500)
    Solves for a vector u(1:n) of length n the tridiagonal linear set given by equation (2.4.1).
    a(1:n), b(1:n), c(1:n), and r(1:n) are input vectors and are not modified.
    Parameter: NMAX is the maximum expected value of n.
INTEGER j
REAL bet,gam(NMAX)                          One vector of workspace, gam is needed.
if(b(1).eq.0.)pause 'tridag: rewrite equations'
  If this happens then you should rewrite your equations as a set of order N − 1, with u2
  trivially eliminated.
bet=b(1)
u(1)=r(1)/bet
do 11 j=2,n                                 Decomposition and forward substitution.
    gam(j)=c(j-1)/bet
    bet=b(j)-a(j)*gam(j)
    if(bet.eq.0.)pause 'tridag failed'      Algorithm fails; see below.
    u(j)=(r(j)-a(j)*u(j-1))/bet
enddo 11
do 12 j=n-1,1,-1                            Backsubstitution.
    u(j)=u(j)-gam(j+1)*u(j+1)
enddo 12
return
END
```

There is no pivoting in `tridag`. It is for this reason that `tridag` can fail (`pause`) even when the underlying matrix is nonsingular: A zero pivot can be encountered even for a nonsingular matrix. In practice, this is not something to lose sleep about. The kinds of problems that lead to tridiagonal linear sets usually have additional properties which guarantee that the algorithm in `tridag` will succeed. For example, if

$$|b_j| > |a_j| + |c_j| \qquad j = 1, \ldots, N \tag{2.4.2}$$

(called *diagonal dominance*) then it can be shown that the algorithm cannot encounter a zero pivot.

It is possible to construct special examples in which the lack of pivoting in the algorithm causes numerical instability. In practice, however, such instability is almost never encountered — unlike the general matrix problem where pivoting is essential.

The tridiagonal algorithm is the rare case of an algorithm that, in practice, is more robust than theory says it should be. Of course, should you ever encounter a problem for which `tridag` fails, you can instead use the more general method for band diagonal systems, now described (routines `bandec` and `banbks`).

Some other matrix forms consisting of tridiagonal with a small number of additional elements (e.g., upper right and lower left corners) also allow rapid solution; see §2.7.

Band Diagonal Systems

Where tridiagonal systems have nonzero elements only on the diagonal plus or minus one, band diagonal systems are slightly more general and have (say) $m_1 \geq 0$ nonzero elements immediately to the left of (below) the diagonal and $m_2 \geq 0$ nonzero elements immediately to its right (above it). Of course, this is only a useful classification if m_1 and m_2 are both $\ll N$. In that case, the solution of the linear system by LU decomposition can be accomplished much faster, and in much less storage, than for the general $N \times N$ case.