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.

```
#include "nrutil.h"

void tridag(float a[], float b[], float c[], float r[], float u[],
    unsigned long n)
Solves for a vector u[1..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.
{
    unsigned long j;
    float bet,*gam;

    gam=vector(1,n);                    One vector of workspace, gam is needed.
    if (b[1] == 0.0) nrerror("Error 1 in tridag");
    If this happens then you should rewrite your equations as a set of order N − 1, with u2
    trivially eliminated.
    u[1]=r[1]/(bet=b[1]);
    for (j=2;j<=n;j++) {                Decomposition and forward substitution.
        gam[j]=c[j-1]/bet;
        bet=b[j]-a[j]*gam[j];
        if (bet == 0.0)    nrerror("Error 2 in tridag");    Algorithm fails; see be-
        u[j]=(r[j]-a[j]*u[j-1])/bet;                            low.
    }
    for (j=(n-1);j>=1;j--)
        u[j] -= gam[j+1]*u[j+1];        Backsubstitution.
    free_vector(gam,1,n);
}
```

There is no pivoting in `tridag`. It is for this reason that `tridag` can fail 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.

The precise definition of a band diagonal matrix with elements a_{ij} is that

$$a_{ij} = 0 \quad \text{when} \quad j > i + m_2 \quad \text{or} \quad i > j + m_1 \tag{2.4.3}$$

Band diagonal matrices are stored and manipulated in a so-called compact form, which results if the matrix is tilted 45° clockwise, so that its nonzero elements lie in a long, narrow matrix with $m_1 + 1 + m_2$ columns and N rows. This is best illustrated by an example: The band diagonal matrix

$$\begin{pmatrix} 3 & 1 & 0 & 0 & 0 & 0 & 0 \\ 4 & 1 & 5 & 0 & 0 & 0 & 0 \\ 9 & 2 & 6 & 5 & 0 & 0 & 0 \\ 0 & 3 & 5 & 8 & 9 & 0 & 0 \\ 0 & 0 & 7 & 9 & 3 & 2 & 0 \\ 0 & 0 & 0 & 3 & 8 & 4 & 6 \\ 0 & 0 & 0 & 0 & 2 & 4 & 4 \end{pmatrix} \tag{2.4.4}$$

which has $N = 7$, $m_1 = 2$, and $m_2 = 1$, is stored compactly as the 7×4 matrix,

$$\begin{pmatrix} x & x & 3 & 1 \\ x & 4 & 1 & 5 \\ 9 & 2 & 6 & 5 \\ 3 & 5 & 8 & 9 \\ 7 & 9 & 3 & 2 \\ 3 & 8 & 4 & 6 \\ 2 & 4 & 4 & x \end{pmatrix} \tag{2.4.5}$$

Here x denotes elements that are wasted space in the compact format; these will not be referenced by any manipulations and can have arbitrary values. Notice that the diagonal of the original matrix appears in column $m_1 + 1$, with subdiagonal elements to its left, superdiagonal elements to its right.

The simplest manipulation of a band diagonal matrix, stored compactly, is to multiply it by a vector to its right. Although this is algorithmically trivial, you might want to study the following routine carefully, as an example of how to pull nonzero elements a_{ij} out of the compact storage format in an orderly fashion.

```
#include "nrutil.h"

void banmul(float **a, unsigned long n, int m1, int m2, float x[], float b[])
```
Matrix multiply $\mathbf{b} = \mathbf{A} \cdot \mathbf{x}$, where $\mathbf{A}$ is band diagonal with `m1` rows below the diagonal and `m2` rows above. The input vector $\mathbf{x}$ and output vector $\mathbf{b}$ are stored as `x[1..n]` and `b[1..n]`, respectively. The array `a[1..n][1..m1+m2+1]` stores $\mathbf{A}$ as follows: The diagonal elements are in `a[1..n][m1+1]`. Subdiagonal elements are in `a[`j`..n][1..m1]` (with $j > 1$ appropriate to the number of elements on each subdiagonal). Superdiagonal elements are in `a[1..`j`][m1+2..m1+m2+1]` with $j <$ `n` appropriate to the number of elements on each superdiagonal.
```
{
    unsigned long i,j,k,tmploop;

    for (i=1;i<=n;i++) {
        k=i-m1-1;
        tmploop=LMIN(m1+m2+1,n-k);
        b[i]=0.0;
        for (j=LMAX(1,1-k);j<=tmploop;j++) b[i] += a[i][j]*x[j+k];
    }
}
```

It is not possible to store the LU decomposition of a band diagonal matrix **A** quite as compactly as the compact form of **A** itself. The decomposition (essentially by Crout's method, see §2.3) produces additional nonzero "fill-ins." One straightforward storage scheme is to return the upper triangular factor (U) in the same space that **A** previously occupied, and to return the lower triangular factor (L) in a separate compact matrix of size $N \times m_1$. The diagonal elements of U (whose product, times d$= \pm 1$, gives the determinant) are returned in the first column of **A**'s storage space.

The following routine, `bandec`, is the band-diagonal analog of `ludcmp` in §2.3:

```
#include <math.h>
#define SWAP(a,b) {dum=(a);(a)=(b);(b)=dum;}
#define TINY 1.0e-20

void bandec(float **a, unsigned long n, int m1, int m2, float **al,
    unsigned long indx[], float *d)
```
Given an `n` × `n` band diagonal matrix **A** with `m1` subdiagonal rows and `m2` superdiagonal rows, compactly stored in the array `a[1..n][1..m1+m2+1]` as described in the comment for routine `banmul`, this routine constructs an LU decomposition of a rowwise permutation of **A**. The upper triangular matrix replaces a, while the lower triangular matrix is returned in `al[1..n][1..m1]`. `indx[1..n]` is an output vector which 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 `banbks` to solve band-diagonal sets of equations.
```
{
    unsigned long i,j,k,l;
    int mm;
    float dum;

    mm=m1+m2+1;
    l=m1;
    for (i=1;i<=m1;i++) {                Rearrange the storage a bit.
        for (j=m1+2-i;j<=mm;j++) a[i][j-l]=a[i][j];
        l--;
        for (j=mm-l;j<=mm;j++) a[i][j]=0.0;
    }
    *d=1.0;
    l=m1;
    for (k=1;k<=n;k++) {                 For each row...
        dum=a[k][1];
        i=k;
        if (l < n) l++;
        for (j=k+1;j<=l;j++) {           Find the pivot element.
            if (fabs(a[j][1]) > fabs(dum)) {
                dum=a[j][1];
                i=j;
            }
        }
        indx[k]=i;
        if (dum == 0.0) a[k][1]=TINY;
        Matrix is algorithmically singular, but proceed anyway with TINY pivot (desirable in
        some applications).
        if (i != k) {                    Interchange rows.
            *d = -(*d);
            for (j=1;j<=mm;j++) SWAP(a[k][j],a[i][j])
        }
        for (i=k+1;i<=l;i++) {           Do the elimination.
            dum=a[i][1]/a[k][1];
            al[k][i-k]=dum;
            for (j=2;j<=mm;j++) a[i][j-1]=a[i][j]-dum*a[k][j];
            a[i][mm]=0.0;
        }
    }
}
```

Some pivoting is possible within the storage limitations of bandec, and the above routine does take advantage of the opportunity. In general, when TINY is returned as a diagonal element of U, then the original matrix (perhaps as modified by roundoff error) is in fact singular. In this regard, bandec is somewhat more robust than tridag above, which can fail algorithmically even for nonsingular matrices; bandec is thus also useful (with $m_1 = m_2 = 1$) for some ill-behaved tridiagonal systems.

Once the matrix **A** has been decomposed, any number of right-hand sides can be solved in turn by repeated calls to banbks, the backsubstitution routine whose analog in §2.3 is lubksb.

```
#define SWAP(a,b) {dum=(a);(a)=(b);(b)=dum;}

void banbks(float **a, unsigned long n, int m1, int m2, float **al,
    unsigned long indx[], float b[])
```

Given the arrays a, al, and indx as returned from bandec, and given a right-hand side vector b[1..n], solves the band diagonal linear equations $\mathbf{A}\cdot\mathbf{x} = \mathbf{b}$. The solution vector **x** overwrites b[1..n]. The other input arrays are not modified, and can be left in place for successive calls with different right-hand sides.

```
{
    unsigned long i,k,l;
    int mm;
    float dum;

    mm=m1+m2+1;
    l=m1;
    for (k=1;k<=n;k++) {                Forward substitution, unscrambling the permuted rows
        i=indx[k];                          as we go.
        if (i != k) SWAP(b[k],b[i])
        if (l < n) l++;
        for (i=k+1;i<=l;i++) b[i] -= al[k][i-k]*b[k];
    }
    l=1;
    for (i=n;i>=1;i--) {                Backsubstitution.
        dum=b[i];
        for (k=2;k<=l;k++) dum -= a[i][k]*b[k+i-1];
        b[i]=dum/a[i][1];
        if (l < mm) l++;
    }
}
```

The routines bandec and banbks are based on the Handbook routines *bandet1* and *bansol1* in [1].

CITED REFERENCES AND FURTHER READING:

Keller, H.B. 1968, *Numerical Methods for Two-Point Boundary-Value Problems* (Waltham, MA: Blaisdell), p. 74.

Dahlquist, G., and Bjorck, A. 1974, *Numerical Methods* (Englewood Cliffs, NJ: Prentice-Hall), Example 5.4.3, p. 166.

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

Wilkinson, J.H., and Reinsch, C. 1971, *Linear Algebra*, vol. II of *Handbook for Automatic Computation* (New York: Springer-Verlag), Chapter I/6. [1]

Golub, G.H., and Van Loan, C.F. 1989, *Matrix Computations*, 2nd ed. (Baltimore: Johns Hopkins University Press), §4.3.

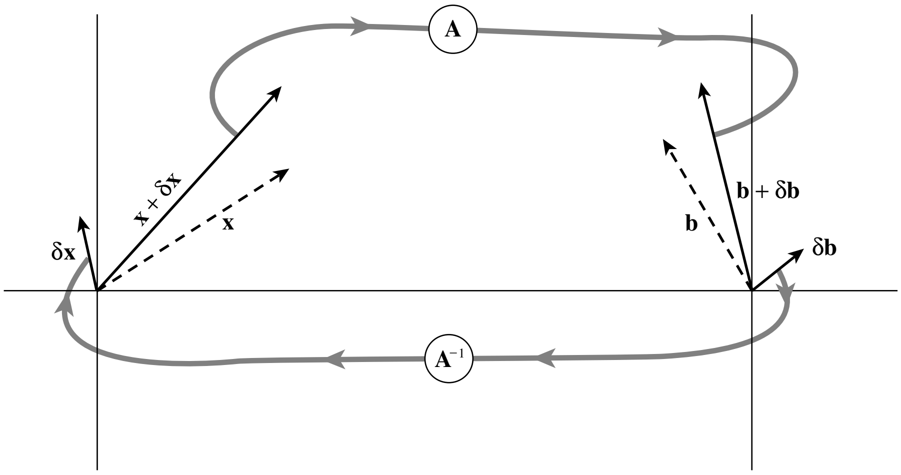

Figure 2.5.1. Iterative improvement of the solution to $\mathbf{A} \cdot \mathbf{x} = \mathbf{b}$. The first guess $\mathbf{x} + \delta\mathbf{x}$ is multiplied by $\mathbf{A}$ to produce $\mathbf{b} + \delta\mathbf{b}$. The known vector $\mathbf{b}$ is subtracted, giving $\delta\mathbf{b}$. The linear set with this right-hand side is inverted, giving $\delta\mathbf{x}$. This is subtracted from the first guess giving an improved solution $\mathbf{x}$.

2.5 Iterative Improvement of a Solution to Linear Equations

Obviously it is not easy to obtain greater precision for the solution of a linear set than the precision of your computer's floating-point word. Unfortunately, for large sets of linear equations, it is not always easy to obtain precision equal to, or even comparable to, the computer's limit. In direct methods of solution, roundoff errors accumulate, and they are magnified to the extent that your matrix is close to singular. You can easily lose two or three significant figures for matrices which (you thought) were *far* from singular.

If this happens to you, there is a neat trick to restore the full machine precision, called *iterative improvement* of the solution. The theory is very straightforward (see Figure 2.5.1): Suppose that a vector $\mathbf{x}$ is the exact solution of the linear set

$$\mathbf{A} \cdot \mathbf{x} = \mathbf{b} \tag{2.5.1}$$

You don't, however, know $\mathbf{x}$. You only know some slightly wrong solution $\mathbf{x} + \delta\mathbf{x}$, where $\delta\mathbf{x}$ is the unknown error. When multiplied by the matrix $\mathbf{A}$, your slightly wrong solution gives a product slightly discrepant from the desired right-hand side $\mathbf{b}$, namely

$$\mathbf{A} \cdot (\mathbf{x} + \delta\mathbf{x}) = \mathbf{b} + \delta\mathbf{b} \tag{2.5.2}$$

Subtracting (2.5.1) from (2.5.2) gives

$$\mathbf{A} \cdot \delta\mathbf{x} = \delta\mathbf{b} \tag{2.5.3}$$