```
    void dsprstx(double sa[], unsigned long ija[], double x[], double b[],
        unsigned long n);
    These are double versions of sprsax and sprstx.

    if (itrnsp) dsprstx(sa,ija,x,r,n);
    else dsprsax(sa,ija,x,r,n);
}
```

```
extern unsigned long ija[];
extern double sa[];              The matrix is stored somewhere.

void asolve(unsigned long n, double b[], double x[], int itrnsp)
{
    unsigned long i;

    for(i=1;i<=n;i++) x[i]=(sa[i] != 0.0 ? b[i]/sa[i] : b[i]);
    The matrix Ã is the diagonal part of A, stored in the first n elements of sa. Since the
    transpose matrix has the same diagonal, the flag itrnsp is not used.
}
```


CITED REFERENCES AND FURTHER READING:

Tewarson, R.P. 1973, *Sparse Matrices* (New York: Academic Press). [1]

Jacobs, D.A.H. (ed.) 1977, *The State of the Art in Numerical Analysis* (London: Academic Press), Chapter I.3 (by J.K. Reid). [2]

George, A., and Liu, J.W.H. 1981, *Computer Solution of Large Sparse Positive Definite Systems* (Englewood Cliffs, NJ: Prentice-Hall). [3]

NAG Fortran Library (Numerical Algorithms Group, 256 Banbury Road, Oxford OX27DE, U.K.). [4]

IMSL Math/Library Users Manual (IMSL Inc., 2500 CityWest Boulevard, Houston TX 77042). [5]

Eisenstat, S.C., Gursky, M.C., Schultz, M.H., and Sherman, A.H. 1977, *Yale Sparse Matrix Package*, Technical Reports 112 and 114 (Yale University Department of Computer Science). [6]

Knuth, D.E. 1968, *Fundamental Algorithms*, vol. 1 of *The Art of Computer Programming* (Reading, MA: Addison-Wesley), §2.2.6. [7]

Kincaid, D.R., Respess, J.R., Young, D.M., and Grimes, R.G. 1982, *ACM Transactions on Mathematical Software*, vol. 8, pp. 302–322. [8]

PCGPAK User's Guide (New Haven: Scientific Computing Associates, Inc.). [9]

Bentley, J. 1986, *Programming Pearls* (Reading, MA: Addison-Wesley), §9. [10]

Golub, G.H., and Van Loan, C.F. 1989, *Matrix Computations*, 2nd ed. (Baltimore: Johns Hopkins University Press), Chapters 4 and 10, particularly §§10.2–10.3. [11]

Stoer, J., and Bulirsch, R. 1980, *Introduction to Numerical Analysis* (New York: Springer-Verlag), Chapter 8. [12]

Baker, L. 1991, *More C Tools for Scientists and Engineers* (New York: McGraw-Hill). [13]

Fletcher, R. 1976, in *Numerical Analysis Dundee 1975*, Lecture Notes in Mathematics, vol. 506, A. Dold and B Eckmann, eds. (Berlin: Springer-Verlag), pp. 73–89. [14]

Saad, Y., and Schulz, M. 1986, *SIAM Journal on Scientific and Statistical Computing*, vol. 7, pp. 856–869. [15]

Bunch, J.R., and Rose, D.J. (eds.) 1976, *Sparse Matrix Computations* (New York: Academic Press).

Duff, I.S., and Stewart, G.W. (eds.) 1979, *Sparse Matrix Proceedings 1978* (Philadelphia: S.I.A.M.).

2.8 Vandermonde Matrices and Toeplitz Matrices

In §2.4 the case of a tridiagonal matrix was treated specially, because that particular type of linear system admits a solution in only of order N operations, rather than of order N^3 for the general linear problem. When such particular types exist, it is important to know about them. Your computational savings, should you ever happen to be working on a problem that involves the right kind of particular type, can be enormous.

This section treats two special types of matrices that can be solved in of order N^2 operations, not as good as tridiagonal, but a lot better than the general case. (Other than the operations count, these two types having nothing in common.) Matrices of the first type, termed *Vandermonde matrices*, occur in some problems having to do with the fitting of polynomials, the reconstruction of distributions from their moments, and also other contexts. In this book, for example, a Vandermonde problem crops up in §3.5. Matrices of the second type, termed *Toeplitz matrices*, tend to occur in problems involving deconvolution and signal processing. In this book, a Toeplitz problem is encountered in §13.7.

These are not the *only* special types of matrices worth knowing about. The *Hilbert matrices*, whose components are of the form $a_{ij} = 1/(i + j - 1),\ i, j = 1, \dots, N$ can be inverted by an exact integer algorithm, and are very *difficult* to invert in any other way, since they are notoriously ill-conditioned (see [1] for details). The Sherman-Morrison and Woodbury formulas, discussed in §2.7, can sometimes be used to convert new special forms into old ones. Reference [2] gives some other special forms. We have not found these additional forms to arise as frequently as the two that we now discuss.

Vandermonde Matrices

A Vandermonde matrix of size $N \times N$ is completely determined by N arbitrary numbers $x_1, x_2, \dots, x_N$, in terms of which its N^2 components are the integer powers $x_i^{j-1},\ i, j = 1, \dots, N$. Evidently there are two possible such forms, depending on whether we view the i's as rows, j's as columns, or vice versa. In the former case, we get a linear system of equations that looks like this,

$$\begin{bmatrix} 1 & x_1 & x_1^2 & \cdots & x_1^{N-1} \\ 1 & x_2 & x_2^2 & \cdots & x_2^{N-1} \\ \vdots & \vdots & \vdots & & \vdots \\ 1 & x_N & x_N^2 & \cdots & x_N^{N-1} \end{bmatrix} \cdot \begin{bmatrix} c_1 \\ c_2 \\ \vdots \\ c_N \end{bmatrix} = \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_N \end{bmatrix} \tag{2.8.1}$$

Performing the matrix multiplication, you will see that this equation solves for the unknown coefficients c_i which fit a polynomial to the N pairs of abscissas and ordinates (x_j, y_j). Precisely this problem will arise in §3.5, and the routine given there will solve (2.8.1) by the method that we are about to describe.

The alternative identification of rows and columns leads to the set of equations

$$\begin{bmatrix} 1 & 1 & \cdots & 1 \\ x_1 & x_2 & \cdots & x_N \\ x_1^2 & x_2^2 & \cdots & x_N^2 \\ & & \cdots & \\ x_1^{N-1} & x_2^{N-1} & \cdots & x_N^{N-1} \end{bmatrix} \cdot \begin{bmatrix} w_1 \\ w_2 \\ w_3 \\ \cdots \\ w_N \end{bmatrix} = \begin{bmatrix} q_1 \\ q_2 \\ q_3 \\ \cdots \\ q_N \end{bmatrix} \tag{2.8.2}$$

Write this out and you will see that it relates to the *problem of moments*: Given the values of N points x_i, find the unknown weights w_i, assigned so as to match the given values q_j of the first N moments. (For more on this problem, consult [3].) The routine given in this section solves (2.8.2).

The method of solution of both (2.8.1) and (2.8.2) is closely related to Lagrange's polynomial interpolation formula, which we will not formally meet until §3.1 below. Notwithstanding, the following derivation should be comprehensible:

Let $P_j(x)$ be the polynomial of degree $N-1$ defined by

$$P_j(x) = \prod_{\substack{n=1 \\ (n\neq j)}}^{N} \frac{x - x_n}{x_j - x_n} = \sum_{k=1}^{N} A_{jk} x^{k-1} \tag{2.8.3}$$

Here the meaning of the last equality is to define the components of the matrix A_{ij} as the coefficients that arise when the product is multiplied out and like terms collected.

The polynomial $P_j(x)$ is a function of x generally. But you will notice that it is specifically designed so that it takes on a value of zero at all x_i with $i \neq j$, and has a value of unity at $x = x_j$. In other words,

$$P_j(x_i) = \delta_{ij} = \sum_{k=1}^{N} A_{jk} x_i^{k-1} \tag{2.8.4}$$

But (2.8.4) says that A_{jk} is exactly the inverse of the matrix of components x_i^{k-1}, which appears in (2.8.2), with the subscript as the column index. Therefore the solution of (2.8.2) is just that matrix inverse times the right-hand side,

$$w_j = \sum_{k=1}^{N} A_{jk} q_k \tag{2.8.5}$$

As for the transpose problem (2.8.1), we can use the fact that the inverse of the transpose is the transpose of the inverse, so

$$c_j = \sum_{k=1}^{N} A_{kj} y_k \tag{2.8.6}$$

The routine in §3.5 implements this.

It remains to find a good way of multiplying out the monomial terms in (2.8.3), in order to get the components of A_{jk}. This is essentially a bookkeeping problem, and we will let you read the routine itself to see how it can be solved. One trick is to define a master $P(x)$ by

$$P(x) \equiv \prod_{n=1}^{N} (x - x_n) \tag{2.8.7}$$

work out its coefficients, and then obtain the numerators and denominators of the specific P_j's via synthetic division by the one supernumerary term. (See §5.3 for more on synthetic division.) Since each such division is only a process of order N, the total procedure is of order N^2.

You should be warned that Vandermonde systems are notoriously ill-conditioned, by their very nature. (As an aside anticipating §5.8, the reason is the same as that which makes Chebyshev fitting so impressively accurate: there exist high-order polynomials that are very good uniform fits to zero. Hence roundoff error can introduce rather substantial coefficients of the leading terms of these polynomials.) It is a good idea always to compute Vandermonde problems in double precision.

The routine for (2.8.2) which follows is due to G.B. Rybicki.

```
#include "nrutil.h"

void vander(double x[], double w[], double q[], int n)
```
Solves the Vandermonde linear system $\sum_{i=1}^{N} x_i^{k-1} w_i = q_k$ $(k = 1, \ldots, N)$. Input consists of the vectors x[1..n] and q[1..n]; the vector w[1..n] is output.
```
{
    int i,j,k;
    double b,s,t,xx;
    double *c;

    c=dvector(1,n);
    if (n == 1) w[1]=q[1];
    else {
        for (i=1;i<=n;i++) c[i]=0.0;        Initialize array.
        c[n] = -x[1];                       Coefficients of the master polynomial are found
        for (i=2;i<=n;i++) {                    by recursion.
            xx = -x[i];
            for (j=(n+1-i);j<=(n-1);j++) c[j] += xx*c[j+1];
            c[n] += xx;
        }
        for (i=1;i<=n;i++) {                Each subfactor in turn
            xx=x[i];
            t=b=1.0;
            s=q[n];
            for (k=n;k>=2;k--) {            is synthetically divided,
                b=c[k]+xx*b;
                s += q[k-1]*b;              matrix-multiplied by the right-hand side,
                t=xx*t+b;
            }
            w[i]=s/t;                       and supplied with a denominator.
        }
    }
    free_dvector(c,1,n);
}
```

Toeplitz Matrices

An $N \times N$ Toeplitz matrix is specified by giving $2N - 1$ numbers R_k, $k = -N + 1, \ldots, -1, 0, 1, \ldots, N - 1$. Those numbers are then emplaced as matrix elements constant along the (upper-left to lower-right) diagonals of the matrix:

$$\begin{bmatrix} R_0 & R_{-1} & R_{-2} & \cdots & R_{-(N-2)} & R_{-(N-1)} \\ R_1 & R_0 & R_{-1} & \cdots & R_{-(N-3)} & R_{-(N-2)} \\ R_2 & R_1 & R_0 & \cdots & R_{-(N-4)} & R_{-(N-3)} \\ \cdots & & & \cdots & & \\ R_{N-2} & R_{N-3} & R_{N-4} & \cdots & R_0 & R_{-1} \\ R_{N-1} & R_{N-2} & R_{N-3} & \cdots & R_1 & R_0 \end{bmatrix} \tag{2.8.8}$$

The linear Toeplitz problem can thus be written as

$$\sum_{j=1}^{N} R_{i-j} x_j = y_i \qquad (i = 1, \ldots, N) \tag{2.8.9}$$

where the x_j's, $j = 1, \ldots, N$, are the unknowns to be solved for.

The Toeplitz matrix is symmetric if $R_k = R_{-k}$ for all k. Levinson [4] developed an algorithm for fast solution of the symmetric Toeplitz problem, by a *bordering method*, that is, a recursive procedure that solves the M-dimensional Toeplitz problem

$$\sum_{j=1}^{M} R_{i-j} x_j^{(M)} = y_i \qquad (i = 1, \ldots, M) \tag{2.8.10}$$

in turn for $M = 1, 2, \ldots$ until $M = N$, the desired result, is finally reached. The vector $x_j^{(M)}$ is the result at the Mth stage, and becomes the desired answer only when N is reached.

Levinson's method is well documented in standard texts (e.g., [5]). The useful fact that the method generalizes to the *nonsymmetric* case seems to be less well known. At some risk of excessive detail, we therefore give a derivation here, due to G.B. Rybicki.

In following a recursion from step M to step $M + 1$ we find that our developing solution $x^{(M)}$ changes in this way:

$$\sum_{j=1}^{M} R_{i-j} x_j^{(M)} = y_i \qquad i = 1, \ldots, M \tag{2.8.11}$$

becomes

$$\sum_{j=1}^{M} R_{i-j} x_j^{(M+1)} + R_{i-(M+1)} x_{M+1}^{(M+1)} = y_i \qquad i = 1, \ldots, M+1 \tag{2.8.12}$$

By eliminating y_i we find

$$\sum_{j=1}^{M} R_{i-j} \left(\frac{x_j^{(M)} - x_j^{(M+1)}}{x_{M+1}^{(M+1)}} \right) = R_{i-(M+1)} \qquad i = 1, \ldots, M \tag{2.8.13}$$

or by letting $i \to M + 1 - i$ and $j \to M + 1 - j$,

$$\sum_{j=1}^{M} R_{j-i} G_j^{(M)} = R_{-i} \tag{2.8.14}$$

where

$$G_j^{(M)} \equiv \frac{x_{M+1-j}^{(M)} - x_{M+1-j}^{(M+1)}}{x_{M+1}^{(M+1)}} \tag{2.8.15}$$

To put this another way,

$$x_{M+1-j}^{(M+1)} = x_{M+1-j}^{(M)} - x_{M+1}^{(M+1)} G_j^{(M)} \qquad j = 1, \ldots, M \tag{2.8.16}$$

Thus, if we can use recursion to find the order M quantities $x^{(M)}$ and $G^{(M)}$ *and* the single order $M + 1$ quantity $x_{M+1}^{(M+1)}$, then all of the other $x_j^{(M+1)}$ will follow. Fortunately, the quantity $x_{M+1}^{(M+1)}$ follows from equation (2.8.12) with $i = M + 1$,

$$\sum_{j=1}^{M} R_{M+1-j} x_j^{(M+1)} + R_0 x_{M+1}^{(M+1)} = y_{M+1} \tag{2.8.17}$$

For the unknown order $M + 1$ quantities $x_j^{(M+1)}$ we can substitute the previous order quantities in G since

$$G_{M+1-j}^{(M)} = \frac{x_j^{(M)} - x_j^{(M+1)}}{x_{M+1}^{(M+1)}} \tag{2.8.18}$$

The result of this operation is

$$x_{M+1}^{(M+1)} = \frac{\sum_{j=1}^{M} R_{M+1-j} x_j^{(M)} - y_{M+1}}{\sum_{j=1}^{M} R_{M+1-j} G_{M+1-j}^{(M)} - R_0} \tag{2.8.19}$$

The only remaining problem is to develop a recursion relation for G. Before we do that, however, we should point out that there are actually two distinct sets of solutions to the original linear problem for a nonsymmetric matrix, namely right-hand solutions (which we have been discussing) and left-hand solutions z_i. The formalism for the left-hand solutions differs only in that we deal with the equations

$$\sum_{j=1}^{M} R_{j-i} z_j^{(M)} = y_i \qquad i = 1, \ldots, M \tag{2.8.20}$$

Then, the same sequence of operations on this set leads to

$$\sum_{j=1}^{M} R_{i-j} H_j^{(M)} = R_i \tag{2.8.21}$$

where

$$H_j^{(M)} \equiv \frac{z_{M+1-j}^{(M)} - z_{M+1-j}^{(M+1)}}{z_{M+1}^{(M+1)}} \tag{2.8.22}$$

(compare with 2.8.14 – 2.8.15). The reason for mentioning the left-hand solutions now is that, by equation (2.8.21), the H_j satisfy exactly the same equation as the x_j except for the substitution $y_i \rightarrow R_i$ on the right-hand side. Therefore we can quickly deduce from equation (2.8.19) that

$$H_{M+1}^{(M+1)} = \frac{\sum_{j=1}^{M} R_{M+1-j} H_j^{(M)} - R_{M+1}}{\sum_{j=1}^{M} R_{M+1-j} G_{M+1-j}^{(M)} - R_0} \tag{2.8.23}$$

By the same token, G satisfies the same equation as z, except for the substitution $y_i \rightarrow R_{-i}$. This gives

$$G_{M+1}^{(M+1)} = \frac{\sum_{j=1}^{M} R_{j-M-1} G_j^{(M)} - R_{-M-1}}{\sum_{j=1}^{M} R_{j-M-1} H_{M+1-j}^{(M)} - R_0} \tag{2.8.24}$$

The same "morphism" also turns equation (2.8.16), and its partner for z, into the final equations

$$\begin{aligned} G_j^{(M+1)} &= G_j^{(M)} - G_{M+1}^{(M+1)} H_{M+1-j}^{(M)} \\ H_j^{(M+1)} &= H_j^{(M)} - H_{M+1}^{(M+1)} G_{M+1-j}^{(M)} \end{aligned} \tag{2.8.25}$$

Now, starting with the initial values

$$x_1^{(1)} = y_1/R_0 \qquad G_1^{(1)} = R_{-1}/R_0 \qquad H_1^{(1)} = R_1/R_0 \tag{2.8.26}$$

we can recurse away. At each stage M we use equations (2.8.23) and (2.8.24) to find $H_{M+1}^{(M+1)}, G_{M+1}^{(M+1)}$, and then equation (2.8.25) to find the other components of $H^{(M+1)}, G^{(M+1)}$. From there the vectors $x^{(M+1)}$ and/or $z^{(M+1)}$ are easily calculated.

The program below does this. It incorporates the second equation in (2.8.25) in the form

$$H_{M+1-j}^{(M+1)} = H_{M+1-j}^{(M)} - H_{M+1}^{(M+1)} G_j^{(M)} \tag{2.8.27}$$

so that the computation can be done "in place."

Notice that the above algorithm fails if $R_0 = 0$. In fact, because the bordering method does not allow pivoting, the algorithm will fail if any of the diagonal principal minors of the original Toeplitz matrix vanish. (Compare with discussion of the tridiagonal algorithm in §2.4.) If the algorithm fails, your matrix is not necessarily singular — you might just have to solve your problem by a slower and more general algorithm such as LU decomposition with pivoting.

The routine that implements equations (2.8.23)–(2.8.27) is also due to Rybicki. Note that the routine's `r[n+j]` is equal to R_j above, so that subscripts on the `r` array vary from 1 to $2N - 1$.

```
#include "nrutil.h"
#define FREERETURN {free_vector(h,1,n);free_vector(g,1,n);return;}

void toeplz(float r[], float x[], float y[], int n)
```
Solves the Toeplitz system $\sum_{j=1}^{N} R_{(N+i-j)}x_j = y_i \;(i = 1, \ldots, N)$. The Toeplitz matrix need not be symmetric. `y[1..n]` and `r[1..2*n-1]` are input arrays; `x[1..n]` is the output array.
```
{
    int j,k,m,m1,m2;
    float pp,pt1,pt2,qq,qt1,qt2,sd,sgd,sgn,shn,sxn;
    float *g,*h;

    if (r[n] == 0.0) nrerror("toeplz-1 singular principal minor");
    g=vector(1,n);
    h=vector(1,n);
    x[1]=y[1]/r[n];                 Initialize for the recursion.
    if (n == 1) FREERETURN
    g[1]=r[n-1]/r[n];
    h[1]=r[n+1]/r[n];
    for (m=1;m<=n;m++) {            Main loop over the recursion.
        m1=m+1;
        sxn = -y[m1];               Compute numerator and denominator for x,
        sd = -r[n];
        for (j=1;j<=m;j++) {
            sxn += r[n+m1-j]*x[j];
            sd += r[n+m1-j]*g[m-j+1];
        }
        if (sd == 0.0) nrerror("toeplz-2 singular principal minor");
        x[m1]=sxn/sd;               whence x.
        for (j=1;j<=m;j++) x[j] -= x[m1]*g[m-j+1];
        if (m1 == n) FREERETURN
        sgn = -r[n-m1];             Compute numerator and denominator for G and H,
        shn = -r[n+m1];
        sgd = -r[n];
        for (j=1;j<=m;j++) {
            sgn += r[n+j-m1]*g[j];
            shn += r[n+m1-j]*h[j];
            sgd += r[n+j-m1]*h[m-j+1];
        }
        if (sd == 0.0 || sgd == 0.0) nrerror("toeplz-3 singular principal minor");
        g[m1]=sgn/sgd;              whence G and H.
        h[m1]=shn/sd;
        k=m;
        m2=(m+1) >> 1;
        pp=g[m1];
        qq=h[m1];
        for (j=1;j<=m2;j++) {
            pt1=g[j];
            pt2=g[k];
            qt1=h[j];
            qt2=h[k];
            g[j]=pt1-pp*qt2;
            g[k]=pt2-pp*qt1;
            h[j]=qt1-qq*pt2;
            h[k--]=qt2-qq*pt1;
        }
    }                               Back for another recurrence.
    nrerror("toeplz - should not arrive here!");
}
```

If you are in the business of solving *very* large Toeplitz systems, you should find out about so-called "new, fast" algorithms, which require only on the order of $N(\log N)^2$ operations, compared to N^2 for Levinson's method. These methods are too complicated to include here.

Papers by Bunch [6] and de Hoog [7] will give entry to the literature.

CITED REFERENCES AND FURTHER READING:

Golub, G.H., and Van Loan, C.F. 1989, *Matrix Computations*, 2nd ed. (Baltimore: Johns Hopkins University Press), Chapter 5 [also treats some other special forms].

Forsythe, G.E., and Moler, C.B. 1967, *Computer Solution of Linear Algebraic Systems* (Englewood Cliffs, NJ: Prentice-Hall), §19. [1]

Westlake, J.R. 1968, *A Handbook of Numerical Matrix Inversion and Solution of Linear Equations* (New York: Wiley). [2]

von Mises, R. 1964, *Mathematical Theory of Probability and Statistics* (New York: Academic Press), pp. 394ff. [3]

Levinson, N., Appendix B of N. Wiener, 1949, *Extrapolation, Interpolation and Smoothing of Stationary Time Series* (New York: Wiley). [4]

Robinson, E.A., and Treitel, S. 1980, *Geophysical Signal Analysis* (Englewood Cliffs, NJ: Prentice-Hall), pp. 163ff. [5]

Bunch, J.R. 1985, *SIAM Journal on Scientific and Statistical Computing*, vol. 6, pp. 349–364. [6]

de Hoog, F. 1987, *Linear Algebra and Its Applications*, vol. 88/89, pp. 123–138. [7]

2.9 Cholesky Decomposition

If a square matrix $\mathbf{A}$ happens to be symmetric and positive definite, then it has a special, more efficient, triangular decomposition. *Symmetric* means that $a_{ij} = a_{ji}$ for $i, j = 1, \dots, N$, while *positive definite* means that

$$\mathbf{v} \cdot \mathbf{A} \cdot \mathbf{v} > 0 \quad \text{for all vectors } \mathbf{v} \tag{2.9.1}$$

(In Chapter 11 we will see that positive definite has the equivalent interpretation that $\mathbf{A}$ has all positive eigenvalues.) While symmetric, positive definite matrices are rather special, they occur quite frequently in some applications, so their special factorization, called *Cholesky decomposition*, is good to know about. When you can use it, Cholesky decomposition is about a factor of two faster than alternative methods for solving linear equations.

Instead of seeking arbitrary lower and upper triangular factors $\mathbf{L}$ and $\mathbf{U}$, Cholesky decomposition constructs a lower triangular matrix $\mathbf{L}$ whose transpose $\mathbf{L}^T$ can itself serve as the upper triangular part. In other words we replace equation (2.3.1) by

$$\mathbf{L} \cdot \mathbf{L}^T = \mathbf{A} \tag{2.9.2}$$

This factorization is sometimes referred to as "taking the square root" of the matrix $\mathbf{A}$. The components of $\mathbf{L}^T$ are of course related to those of $\mathbf{L}$ by

$$L^T_{ij} = L_{ji} \tag{2.9.3}$$

Writing out equation (2.9.2) in components, one readily obtains the analogs of equations (2.3.12)–(2.3.13),

$$L_{ii} = \left(a_{ii} - \sum_{k=1}^{i-1} L_{ik}^2 \right)^{1/2} \tag{2.9.4}$$

and

$$L_{ji} = \frac{1}{L_{ii}} \left(a_{ij} - \sum_{k=1}^{i-1} L_{ik} L_{jk} \right) \qquad j = i+1, i+2, \dots, N \tag{2.9.5}$$

If you apply equations (2.9.4) and (2.9.5) in the order $i = 1, 2, \dots, N$, you will see that the L's that occur on the right-hand side are already determined by the time they are needed. Also, only components a_{ij} with $j \geq i$ are referenced. (Since **A** is symmetric, these have complete information.) It is convenient, then, to have the factor **L** overwrite the subdiagonal (lower triangular but not including the diagonal) part of **A**, preserving the input upper triangular values of **A**. Only one extra vector of length N is needed to store the diagonal part of **L**. The operations count is $N^3/6$ executions of the inner loop (consisting of one multiply and one subtract), with also N square roots. As already mentioned, this is about a factor 2 better than LU decomposition of **A** (where its symmetry would be ignored).

A straightforward implementation is

```
#include <math.h>

void choldc(float **a, int n, float p[])
```
Given a positive-definite symmetric matrix a[1..n][1..n], this routine constructs its Cholesky decomposition, $\mathbf{A} = \mathbf{L} \cdot \mathbf{L}^T$. On input, only the upper triangle of a need be given; it is not modified. The Cholesky factor **L** is returned in the lower triangle of a, except for its diagonal elements which are returned in p[1..n].
```
{
    void nrerror(char error_text[]);
    int i,j,k;
    float sum;

    for (i=1;i<=n;i++) {
        for (j=i;j<=n;j++) {
            for (sum=a[i][j],k=i-1;k>=1;k--) sum -= a[i][k]*a[j][k];
            if (i == j) {
                if (sum <= 0.0)          a, with rounding errors, is not positive definite.
                    nrerror("choldc failed");
                p[i]=sqrt(sum);
            } else a[j][i]=sum/p[i];
        }
    }
}
```

You might at this point wonder about pivoting. The pleasant answer is that Cholesky decomposition is extremely stable numerically, without any pivoting at all. Failure of choldc simply indicates that the matrix **A** (or, with roundoff error, another very nearby matrix) is not positive definite. In fact, choldc is an efficient way to test *whether* a symmetric matrix is positive definite. (In this application, you will want to replace the call to nrerror with some less drastic signaling method.)

Once your matrix is decomposed, the triangular factor can be used to solve a linear equation by backsubstitution. The straightforward implementation of this is

```
void cholsl(float **a, int n, float p[], float b[], float x[])
```
Solves the set of n linear equations $\mathbf{A} \cdot \mathbf{x} = \mathbf{b}$, where a is a positive-definite symmetric matrix. a[1..n][1..n] and p[1..n] are input as the output of the routine choldc. Only the lower triangle of a is accessed. b[1..n] is input as the right-hand side vector. The solution vector is returned in x[1..n]. a, n, and p are not modified and can be left in place for successive calls with different right-hand sides b. b is not modified unless you identify b and x in the calling sequence, which is allowed.
```
{
    int i,k;
    float sum;

    for (i=1;i<=n;i++) {          Solve L · y = b, storing y in x.
        for (sum=b[i],k=i-1;k>=1;k--) sum -= a[i][k]*x[k];
        x[i]=sum/p[i];
    }
    for (i=n;i>=1;i--) {          Solve L^T · x = y.
        for (sum=x[i],k=i+1;k<=n;k++) sum -= a[k][i]*x[k];
```

```
        x[i]=sum/p[i];
    }
}
```

A typical use of `choldc` and `cholsl` is in the inversion of covariance matrices describing the fit of data to a model; see, e.g., §15.6. In this, and many other applications, one often needs $\mathbf{L}^{-1}$. The lower triangle of this matrix can be efficiently found from the output of `choldc`:

```
    for (i=1;i<=n;i++) {
        a[i][i]=1.0/p[i];
        for (j=i+1;j<=n;j++) {
            sum=0.0;
            for (k=i;k<j;k++) sum -= a[j][k]*a[k][i];
            a[j][i]=sum/p[j];
        }
    }
```

CITED REFERENCES AND FURTHER READING:

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

Gill, P.E., Murray, W., and Wright, M.H. 1991, *Numerical Linear Algebra and Optimization*, vol. 1 (Redwood City, CA: Addison-Wesley), §4.9.2.

Dahlquist, G., and Bjorck, A. 1974, *Numerical Methods* (Englewood Cliffs, NJ: Prentice-Hall), §5.3.5.

Golub, G.H., and Van Loan, C.F. 1989, *Matrix Computations*, 2nd ed. (Baltimore: Johns Hopkins University Press), §4.2.

2.10 QR Decomposition

There is another matrix factorization that is sometimes very useful, the so-called QR *decomposition*,

$$\mathbf{A} = \mathbf{Q} \cdot \mathbf{R} \tag{2.10.1}$$

Here $\mathbf{R}$ is upper triangular, while $\mathbf{Q}$ is orthogonal, that is,

$$\mathbf{Q}^T \cdot \mathbf{Q} = \mathbf{1} \tag{2.10.2}$$

where $\mathbf{Q}^T$ is the transpose matrix of $\mathbf{Q}$. Although the decomposition exists for a general rectangular matrix, we shall restrict our treatment to the case when all the matrices are square, with dimensions $N \times N$.

Like the other matrix factorizations we have met (LU, SVD, Cholesky), QR decomposition can be used to solve systems of linear equations. To solve

$$\mathbf{A} \cdot \mathbf{x} = \mathbf{b} \tag{2.10.3}$$

first form $\mathbf{Q}^T \cdot \mathbf{b}$ and then solve

$$\mathbf{R} \cdot \mathbf{x} = \mathbf{Q}^T \cdot \mathbf{b} \tag{2.10.4}$$

by backsubstitution. Since QR decomposition involves about twice as many operations as LU decomposition, it is not used for typical systems of linear equations. However, we will meet special cases where QR is the method of choice.