Matrix multiply $\mathbf{A} \cdot \mathbf{B}^T$ where $\mathbf{A}$ and $\mathbf{B}$ are two sparse matrices in row-index storage mode, and $\mathbf{B}^T$ is the transpose of $\mathbf{B}$. Here, sa and ija store the matrix $\mathbf{A}$; sb and ijb store the matrix $\mathbf{B}$. This routine computes all components of the matrix product (which may be non-sparse!), but stores only those whose magnitude exceeds thresh. On output, the arrays sc and ijc (whose maximum size is input as nmax) give the product matrix in row-index storage mode. For sparse matrix multiplication, this routine will often be preceded by a call to sprstp, so as to construct the transpose of a known matrix into sb, ijb.

```
{
    void nrerror(char error_text[]);
    unsigned long i,ijma,ijmb,j,k,ma,mb,mbb;
    float sum;

    if (ija[1] != ijb[1]) nrerror("sprstm: sizes do not match");
    ijc[1]=k=ija[1];
    for (i=1;i<=ija[1]-2;i++) {                    Loop over rows of A,
        for (j=1;j<=ijb[1]-2;j++) {                and rows of B.
            if (i == j) sum=sa[i]*sb[j]; else sum=0.0e0;
            mb=ijb[j];
            for (ma=ija[i];ma<=ija[i+1]-1;ma++) {
            Loop through elements in A's row. Convoluted logic, following, accounts for the
            various combinations of diagonal and off-diagonal elements.
                ijma=ija[ma];
                if (ijma == j) sum += sa[ma]*sb[j];
                else {
                    while (mb < ijb[j+1]) {
                        ijmb=ijb[mb];
                        if (ijmb == i) {
                            sum += sa[i]*sb[mb++];
                            continue;
                        } else if (ijmb < ijma) {
                            mb++;
                            continue;
                        } else if (ijmb == ijma) {
                            sum += sa[ma]*sb[mb++];
                            continue;
                        }
                        break;
                    }
                }
            }
            for (mbb=mb;mbb<=ijb[j+1]-1;mbb++) {      Exhaust the remainder of B's row.
                if (ijb[mbb] == i) sum += sa[i]*sb[mbb];
            }
            if (i == j) sc[i]=sum;                    Where to put the answer...
            else if (fabs(sum) > thresh) {
                if (k > nmax) nrerror("sprstm: nmax too small");
                sc[k]=sum;
                ijc[k++]=j;
            }
        }
        ijc[i+1]=k;
    }
}
```

Conjugate Gradient Method for a Sparse System

So-called *conjugate gradient methods* provide a quite general means for solving the $N \times N$ linear system

$$\mathbf{A} \cdot \mathbf{x} = \mathbf{b} \tag{2.7.29}$$

The attractiveness of these methods for large sparse systems is that they reference $\mathbf{A}$ only through its multiplication of a vector, or the multiplication of its transpose and a vector. As we have seen, these operations can be very efficient for a properly stored sparse matrix. You, the "owner" of the matrix **A**, can be asked to provide functions that perform these sparse matrix multiplications as efficiently as possible. We, the "grand strategists" supply the general routine, `linbcg` below, that solves the set of linear equations, (2.7.29), using your functions.

The simplest, "ordinary" conjugate gradient algorithm [11-13] solves (2.7.29) only in the case that **A** is symmetric and positive definite. It is based on the idea of minimizing the function

$$f(\mathbf{x}) = \frac{1}{2}\,\mathbf{x}\cdot\mathbf{A}\cdot\mathbf{x} - \mathbf{b}\cdot\mathbf{x} \tag{2.7.30}$$

This function is minimized when its gradient

$$\nabla f = \mathbf{A}\cdot\mathbf{x} - \mathbf{b} \tag{2.7.31}$$

is zero, which is equivalent to (2.7.29). The minimization is carried out by generating a succession of search directions $\mathbf{p}_k$ and improved minimizers $\mathbf{x}_k$. At each stage a quantity α_k is found that minimizes $f(\mathbf{x}_k + \alpha_k\mathbf{p}_k)$, and $\mathbf{x}_{k+1}$ is set equal to the new point $\mathbf{x}_k + \alpha_k\mathbf{p}_k$. The $\mathbf{p}_k$ and $\mathbf{x}_k$ are built up in such a way that $\mathbf{x}_{k+1}$ is also the minimizer of f over the whole vector space of directions already taken, $\{\mathbf{p}_1, \mathbf{p}_2, \dots, \mathbf{p}_k\}$. After N iterations you arrive at the minimizer over the entire vector space, i.e., the solution to (2.7.29).

Later, in §10.6, we will generalize this "ordinary" conjugate gradient algorithm to the minimization of arbitrary nonlinear functions. Here, where our interest is in solving linear, but not necessarily positive definite or symmetric, equations, a different generalization is important, the *biconjugate gradient method*. This method does not, in general, have a simple connection with function minimization. It constructs four sequences of vectors, $\mathbf{r}_k$, $\overline{\mathbf{r}}_k$, $\mathbf{p}_k$, $\overline{\mathbf{p}}_k$, $k = 1, 2, \dots$. You supply the initial vectors $\mathbf{r}_1$ and $\overline{\mathbf{r}}_1$, and set $\mathbf{p}_1 = \mathbf{r}_1$, $\overline{\mathbf{p}}_1 = \overline{\mathbf{r}}_1$. Then you carry out the following recurrence:

$$\begin{aligned}
\alpha_k &= \frac{\overline{\mathbf{r}}_k\cdot\mathbf{r}_k}{\overline{\mathbf{p}}_k\cdot\mathbf{A}\cdot\mathbf{p}_k}\\
\mathbf{r}_{k+1} &= \mathbf{r}_k - \alpha_k\mathbf{A}\cdot\mathbf{p}_k\\
\overline{\mathbf{r}}_{k+1} &= \overline{\mathbf{r}}_k - \alpha_k\mathbf{A}^T\cdot\overline{\mathbf{p}}_k\\
\beta_k &= \frac{\overline{\mathbf{r}}_{k+1}\cdot\mathbf{r}_{k+1}}{\overline{\mathbf{r}}_k\cdot\mathbf{r}_k}\\
\mathbf{p}_{k+1} &= \mathbf{r}_{k+1} + \beta_k\mathbf{p}_k\\
\overline{\mathbf{p}}_{k+1} &= \overline{\mathbf{r}}_{k+1} + \beta_k\overline{\mathbf{p}}_k
\end{aligned} \tag{2.7.32}$$

This sequence of vectors satisfies the *biorthogonality* condition

$$\overline{\mathbf{r}}_i\cdot\mathbf{r}_j = \mathbf{r}_i\cdot\overline{\mathbf{r}}_j = 0, \qquad j < i \tag{2.7.33}$$

and the *biconjugacy* condition

$$\overline{\mathbf{p}}_i\cdot\mathbf{A}\cdot\mathbf{p}_j = \mathbf{p}_i\cdot\mathbf{A}^T\cdot\overline{\mathbf{p}}_j = 0, \qquad j < i \tag{2.7.34}$$

There is also a mutual orthogonality,

$$\overline{\mathbf{r}}_i\cdot\mathbf{p}_j = \mathbf{r}_i\cdot\overline{\mathbf{p}}_j = 0, \qquad j < i \tag{2.7.35}$$

The proof of these properties proceeds by straightforward induction [14]. As long as the recurrence does not break down earlier because one of the denominators is zero, it must terminate after $m \le N$ steps with $\mathbf{r}_{m+1} = \overline{\mathbf{r}}_{m+1} = 0$. This is basically because after at most N steps you run out of new orthogonal directions to the vectors you've already constructed.

To use the algorithm to solve the system (2.7.29), make an initial guess $\mathbf{x}_1$ for the solution. Choose $\mathbf{r}_1$ to be the *residual*

$$\mathbf{r}_1 = \mathbf{b} - \mathbf{A}\cdot\mathbf{x}_1 \tag{2.7.36}$$

and choose $\overline{\mathbf{r}}_1 = \mathbf{r}_1$. Then form the sequence of improved estimates

$$\mathbf{x}_{k+1} = \mathbf{x}_k + \alpha_k\mathbf{p}_k \tag{2.7.37}$$

while carrying out the recurrence (2.7.32). Equation (2.7.37) guarantees that $\mathbf{r}_{k+1}$ from the recurrence is in fact the residual $\mathbf{b} - \mathbf{A} \cdot \mathbf{x}_{k+1}$ corresponding to $\mathbf{x}_{k+1}$. Since $\mathbf{r}_{m+1} = 0$, $\mathbf{x}_{m+1}$ is the solution to equation (2.7.29).

While there is no guarantee that this whole procedure will not break down or become unstable for general $\mathbf{A}$, in practice this is rare. More importantly, the exact termination in at most N iterations occurs only with exact arithmetic. Roundoff error means that you should regard the process as a genuinely iterative procedure, to be halted when some appropriate error criterion is met.

The ordinary conjugate gradient algorithm is the special case of the biconjugate gradient algorithm when $\mathbf{A}$ is symmetric, and we choose $\overline{\mathbf{r}}_1 = \mathbf{r}_1$. Then $\overline{\mathbf{r}}_k = \mathbf{r}_k$ and $\overline{\mathbf{p}}_k = \mathbf{p}_k$ for all k; you can omit computing them and halve the work of the algorithm. This conjugate gradient version has the interpretation of minimizing equation (2.7.30). If $\mathbf{A}$ is positive definite as well as symmetric, the algorithm cannot break down (in theory!). The routine `linbcg` below indeed reduces to the ordinary conjugate gradient method if you input a symmetric $\mathbf{A}$, but it does all the redundant computations.

Another variant of the general algorithm corresponds to a symmetric but non-positive definite $\mathbf{A}$, with the choice $\overline{\mathbf{r}}_1 = \mathbf{A} \cdot \mathbf{r}_1$ instead of $\overline{\mathbf{r}}_1 = \mathbf{r}_1$. In this case $\overline{\mathbf{r}}_k = \mathbf{A} \cdot \mathbf{r}_k$ and $\overline{\mathbf{p}}_k = \mathbf{A} \cdot \mathbf{p}_k$ for all k. This algorithm is thus equivalent to the ordinary conjugate gradient algorithm, but with all dot products $\mathbf{a} \cdot \mathbf{b}$ replaced by $\mathbf{a} \cdot \mathbf{A} \cdot \mathbf{b}$. It is called the *minimum residual* algorithm, because it corresponds to successive minimizations of the function

$$\Phi(\mathbf{x}) = \frac{1}{2}\,\mathbf{r} \cdot \mathbf{r} = \frac{1}{2}\,|\mathbf{A} \cdot \mathbf{x} - \mathbf{b}|^2 \qquad (2.7.38)$$

where the successive iterates $\mathbf{x}_k$ minimize Φ over the same set of search directions $\mathbf{p}_k$ generated in the conjugate gradient method. This algorithm has been generalized in various ways for unsymmetric matrices. The *generalized minimum residual* method (GMRES; see [9,15]) is probably the most robust of these methods.

Note that equation (2.7.38) gives

$$\nabla\Phi(\mathbf{x}) = \mathbf{A}^T \cdot (\mathbf{A} \cdot \mathbf{x} - \mathbf{b}) \qquad (2.7.39)$$

For any nonsingular matrix $\mathbf{A}$, $\mathbf{A}^T \cdot \mathbf{A}$ is symmetric and positive definite. You might therefore be tempted to solve equation (2.7.29) by applying the ordinary conjugate gradient algorithm to the problem

$$(\mathbf{A}^T \cdot \mathbf{A}) \cdot \mathbf{x} = \mathbf{A}^T \cdot \mathbf{b} \qquad (2.7.40)$$

Don't! The condition number of the matrix $\mathbf{A}^T \cdot \mathbf{A}$ is the square of the condition number of $\mathbf{A}$ (see §2.6 for definition of condition number). A large condition number both increases the number of iterations required, and limits the accuracy to which a solution can be obtained. It is almost always better to apply the biconjugate gradient method to the original matrix $\mathbf{A}$.

So far we have said nothing about the *rate* of convergence of these methods. The ordinary conjugate gradient method works well for matrices that are well-conditioned, i.e., "close" to the identity matrix. This suggests applying these methods to the *preconditioned* form of equation (2.7.29),

$$(\widetilde{\mathbf{A}}^{-1} \cdot \mathbf{A}) \cdot \mathbf{x} = \widetilde{\mathbf{A}}^{-1} \cdot \mathbf{b} \qquad (2.7.41)$$

The idea is that you might already be able to solve your linear system easily for some $\widetilde{\mathbf{A}}$ close to $\mathbf{A}$, in which case $\widetilde{\mathbf{A}}^{-1} \cdot \mathbf{A} \approx \mathbf{1}$, allowing the algorithm to converge in fewer steps. The matrix $\widetilde{\mathbf{A}}$ is called a *preconditioner* [11], and the overall scheme given here is known as the *preconditioned biconjugate gradient method* or *PBCG*.

For efficient implementation, the PBCG algorithm introduces an additional set of vectors $\mathbf{z}_k$ and $\overline{\mathbf{z}}_k$ defined by

$$\widetilde{\mathbf{A}} \cdot \mathbf{z}_k = \mathbf{r}_k \qquad \text{and} \qquad \widetilde{\mathbf{A}}^T \cdot \overline{\mathbf{z}}_k = \overline{\mathbf{r}}_k \qquad (2.7.42)$$

and modifies the definitions of α_k, β_k, $\mathbf{p}_k$, and $\overline{\mathbf{p}}_k$ in equation (2.7.32):

$$\begin{aligned}
\alpha_k &= \frac{\overline{\mathbf{r}}_k \cdot \mathbf{z}_k}{\overline{\mathbf{p}}_k \cdot \mathbf{A} \cdot \mathbf{p}_k} \\
\beta_k &= \frac{\overline{\mathbf{r}}_{k+1} \cdot \mathbf{z}_{k+1}}{\overline{\mathbf{r}}_k \cdot \mathbf{z}_k} \\
\mathbf{p}_{k+1} &= \mathbf{z}_{k+1} + \beta_k \mathbf{p}_k \\
\overline{\mathbf{p}}_{k+1} &= \overline{\mathbf{z}}_{k+1} + \beta_k \overline{\mathbf{p}}_k
\end{aligned} \tag{2.7.43}$$

For linbcg, below, we will ask you to supply routines that solve the auxiliary linear systems (2.7.42). If you have no idea what to use for the preconditioner $\widetilde{\mathbf{A}}$, then use the diagonal part of $\mathbf{A}$, or even the identity matrix, in which case the burden of convergence will be entirely on the biconjugate gradient method itself.

The routine linbcg, below, is based on a program originally written by Anne Greenbaum. (See [13] for a different, less sophisticated, implementation.) There are a few wrinkles you should know about.

What constitutes "good" convergence is rather application dependent. The routine linbcg therefore provides for four possibilities, selected by setting the flag itol on input. If itol=1, iteration stops when the quantity $|\mathbf{A} \cdot \mathbf{x} - \mathbf{b}|/|\mathbf{b}|$ is less than the input quantity tol. If itol=2, the required criterion is

$$|\widetilde{\mathbf{A}}^{-1} \cdot (\mathbf{A} \cdot \mathbf{x} - \mathbf{b})|/|\widetilde{\mathbf{A}}^{-1} \cdot \mathbf{b}| < \texttt{tol} \tag{2.7.44}$$

If itol=3, the routine uses its own estimate of the error in $\mathbf{x}$, and requires its magnitude, divided by the magnitude of $\mathbf{x}$, to be less than tol. The setting itol=4 is the same as itol=3, except that the largest (in absolute value) component of the error and largest component of $\mathbf{x}$ are used instead of the vector magnitude (that is, the L_∞ norm instead of the L_2 norm). You may need to experiment to find which of these convergence criteria is best for your problem.

On output, err is the tolerance actually achieved. If the returned count iter does not indicate that the maximum number of allowed iterations itmax was exceeded, then err should be less than tol. If you want to do further iterations, leave all returned quantities as they are and call the routine again. The routine loses its memory of the spanned conjugate gradient subspace between calls, however, so you should not force it to return more often than about every N iterations.

Finally, note that linbcg is furnished in double precision, since it will be usually be used when N is quite large.

```
#include <stdio.h>
#include <math.h>
#include "nrutil.h"
#define EPS 1.0e-14

void linbcg(unsigned long n, double b[], double x[], int itol, double tol,
    int itmax, int *iter, double *err)
```

Solves $\mathbf{A} \cdot \mathbf{x} = \mathbf{b}$ for x[1..n], given b[1..n], by the iterative biconjugate gradient method. On input x[1..n] should be set to an initial guess of the solution (or all zeros); itol is 1,2,3, or 4, specifying which convergence test is applied (see text); itmax is the maximum number of allowed iterations; and tol is the desired convergence tolerance. On output, x[1..n] is reset to the improved solution, iter is the number of iterations actually taken, and err is the estimated error. The matrix $\mathbf{A}$ is referenced only through the user-supplied routines atimes, which computes the product of either $\mathbf{A}$ or its transpose on a vector; and asolve, which solves $\widetilde{\mathbf{A}} \cdot \mathbf{x} = \mathbf{b}$ or $\widetilde{\mathbf{A}}^T \cdot \mathbf{x} = \mathbf{b}$ for some preconditioner matrix $\widetilde{\mathbf{A}}$ (possibly the trivial diagonal part of $\mathbf{A}$).

```
{
    void asolve(unsigned long n, double b[], double x[], int itrnsp);
    void atimes(unsigned long n, double x[], double r[], int itrnsp);
    double snrm(unsigned long n, double sx[], int itol);
    unsigned long j;
    double ak,akden,bk,bkden,bknum,bnrm,dxnrm,xnrm,zm1nrm,znrm;
    double *p,*pp,*r,*rr,*z,*zz;        Double precision is a good idea in this routine.
    p=dvector(1,n);
    pp=dvector(1,n);
    r=dvector(1,n);
    rr=dvector(1,n);
    z=dvector(1,n);
    zz=dvector(1,n);

    Calculate initial residual.
    *iter=0;
    atimes(n,x,r,0);                  Input to atimes is x[1..n], output is r[1..n];
    for (j=1;j<=n;j++) {                  the final 0 indicates that the matrix (not its
        r[j]=b[j]-r[j];                   transpose) is to be used.
        rr[j]=r[j];
    }
/*    atimes(n,r,rr,0); */            Uncomment this line to get the "minimum resid-
    if (itol == 1) {                      ual" variant of the algorithm.
        bnrm=snrm(n,b,itol);
        asolve(n,r,z,0);              Input to asolve is r[1..n], output is z[1..n];
    }                                     the final 0 indicates that the matrix Ã (not
    else if (itol == 2) {                 its transpose) is to be used.
        asolve(n,b,z,0);
        bnrm=snrm(n,z,itol);
        asolve(n,r,z,0);
    }
    else if (itol == 3 || itol == 4) {
        asolve(n,b,z,0);
        bnrm=snrm(n,z,itol);
        asolve(n,r,z,0);
        znrm=snrm(n,z,itol);
    } else nrerror("illegal itol in linbcg");
    while (*iter <= itmax) {          Main loop.
        ++(*iter);
        asolve(n,rr,zz,1);            Final 1 indicates use of transpose matrix Ã^T.
        for (bknum=0.0,j=1;j<=n;j++) bknum += z[j]*rr[j];
        Calculate coefficient bk and direction vectors p and pp.
        if (*iter == 1) {
            for (j=1;j<=n;j++) {
                p[j]=z[j];
                pp[j]=zz[j];
            }
        }
        else {
            bk=bknum/bkden;
            for (j=1;j<=n;j++) {
                p[j]=bk*p[j]+z[j];
                pp[j]=bk*pp[j]+zz[j];
            }
        }
        bkden=bknum;                  Calculate coefficient ak, new iterate x, and new
        atimes(n,p,z,0);                  residuals r and rr.
        for (akden=0.0,j=1;j<=n;j++) akden += z[j]*pp[j];
        ak=bknum/akden;
        atimes(n,pp,zz,1);
        for (j=1;j<=n;j++) {
            x[j] += ak*p[j];
            r[j] -= ak*z[j];
            rr[j] -= ak*zz[j];
        }
        asolve(n,r,z,0);              Solve Ã · z = r and check stopping criterion.
        if (itol == 1)
            *err=snrm(n,r,itol)/bnrm;
        else if (itol == 2)
            *err=snrm(n,z,itol)/bnrm;
        else if (itol == 3 || itol == 4) {
            zm1nrm=znrm;
            znrm=snrm(n,z,itol);
            if (fabs(zm1nrm-znrm) > EPS*znrm) {
                dxnrm=fabs(ak)*snrm(n,p,itol);
                *err=znrm/fabs(zm1nrm-znrm)*dxnrm;
            } else {
                *err=znrm/bnrm;          Error may not be accurate, so loop again.
                continue;
            }
            xnrm=snrm(n,x,itol);
            if (*err <= 0.5*xnrm) *err /= xnrm;
            else {
                *err=znrm/bnrm;          Error may not be accurate, so loop again.
                continue;
            }
        }
        printf("iter=%4d err=%12.6f\n",*iter,*err);
    if (*err <= tol) break;
    }

    free_dvector(p,1,n);
    free_dvector(pp,1,n);
    free_dvector(r,1,n);
    free_dvector(rr,1,n);
    free_dvector(z,1,n);
    free_dvector(zz,1,n);
}
```

The routine linbcg uses this short utility for computing vector norms:

```
#include <math.h>

double snrm(unsigned long n, double sx[], int itol)
Compute one of two norms for a vector sx[1..n], as signaled by itol. Used by linbcg.
{
    unsigned long i,isamax;
    double ans;

    if (itol <= 3) {
        ans = 0.0;
        for (i=1;i<=n;i++) ans += sx[i]*sx[i];      Vector magnitude norm.
        return sqrt(ans);
    } else {
        isamax=1;
        for (i=1;i<=n;i++) {                        Largest component norm.
            if (fabs(sx[i]) > fabs(sx[isamax])) isamax=i;
        }
        return fabs(sx[isamax]);
    }
}
```

So that the specifications for the routines atimes and asolve are clear, we list here simple versions that assume a matrix **A** stored somewhere in row-index sparse format.

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

void atimes(unsigned long n, double x[], double r[], int itrnsp)
{
    void dsprsax(double sa[], unsigned long ija[], double x[], double b[],
        unsigned long n);
    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 $\widetilde{\mathbf{A}}$ is the diagonal part of $\mathbf{A}$, 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.).