2.7 Sparse Linear Systems

A system of linear equations is called *sparse* if only a relatively small number of its matrix elements a_{ij} are nonzero. It is wasteful to use general methods of linear algebra on such problems, because most of the $O(N^3)$ arithmetic operations devoted to solving the set of equations or inverting the matrix involve zero operands. Furthermore, you might wish to work problems so large as to tax your available memory space, and it is wasteful to reserve storage for unfruitful zero elements. Note that there are two distinct (and not always compatible) goals for any sparse matrix method: saving time and/or saving space.

We have already considered one archetypal sparse form in §2.4, the band diagonal matrix. In the tridiagonal case, e.g., we saw that it was possible to save both time (order N instead of N^3) and space (order N instead of N^2). The method of solution was not different in principle from the general method of LU decomposition; it was just applied cleverly, and with due attention to the bookkeeping of zero elements. Many practical schemes for dealing with sparse problems have this same character. They are fundamentally decomposition schemes, or else elimination schemes akin to Gauss-Jordan, but carefully optimized so as to minimize the number of so-called *fill-ins*, initially zero elements which must become nonzero during the solution process, and for which storage must be reserved.

Direct methods for solving sparse equations, then, depend crucially on the precise pattern of sparsity of the matrix. Patterns that occur frequently, or that are useful as way-stations in the reduction of more general forms, already have special names and special methods of solution. We do not have space here for any detailed review of these. References listed at the end of this section will furnish you with an "in" to the specialized literature, and the following list of buzz words (and Figure 2.7.1) will at least let you hold your own at cocktail parties:

- tridiagonal
- band diagonal (or banded) with bandwidth M
- band triangular
- block diagonal
- block tridiagonal
- block triangular
- cyclic banded
- singly (or doubly) bordered block diagonal
- singly (or doubly) bordered block triangular
- singly (or doubly) bordered band diagonal
- singly (or doubly) bordered band triangular
- other (!)

You should also be aware of some of the special sparse forms that occur in the solution of partial differential equations in two or more dimensions. See Chapter 19.

If your particular pattern of sparsity is not a simple one, then you may wish to try an *analyze/factorize/operate* package, which automates the procedure of figuring out how fill-ins are to be minimized. The *analyze* stage is done once only for each pattern of sparsity. The *factorize* stage is done once for each particular matrix that fits the pattern. The *operate* stage is performed once for each right-hand side to

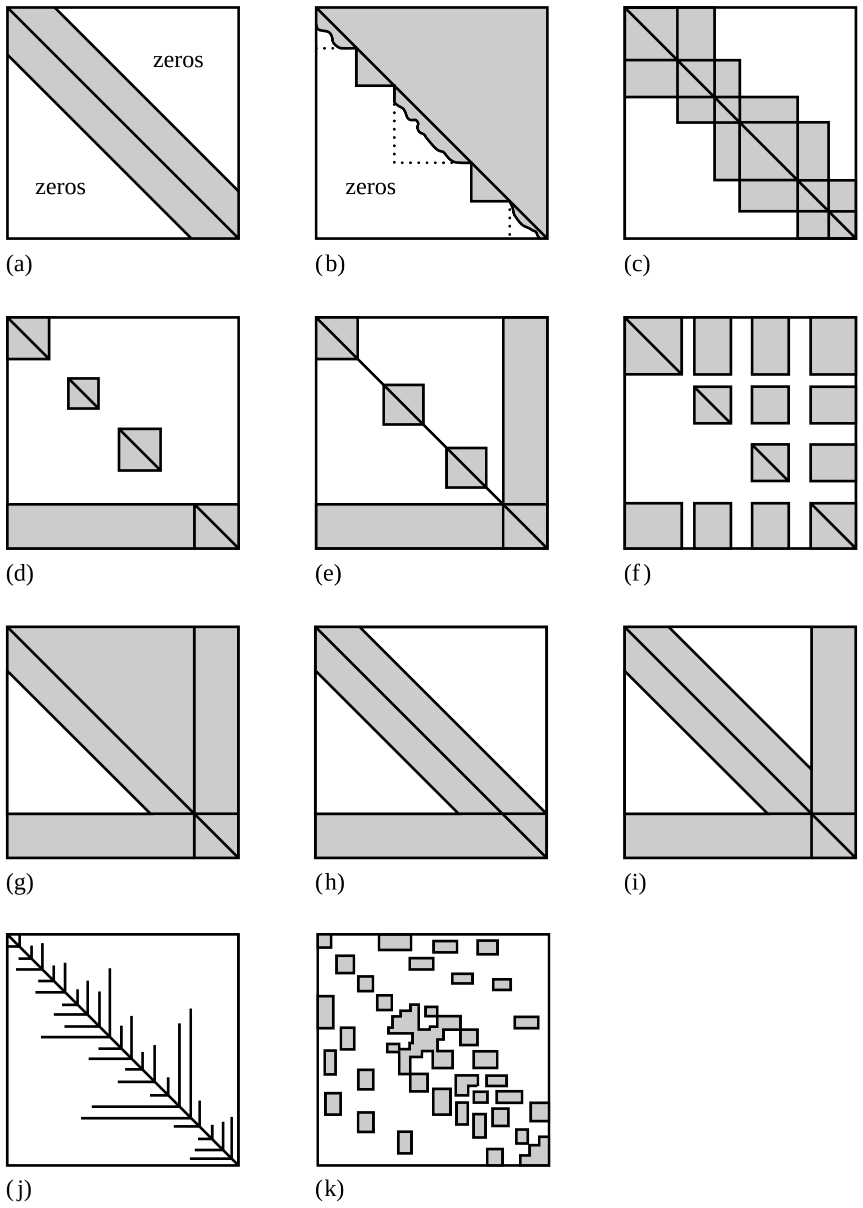

Figure 2.7.1. Some standard forms for sparse matrices. (a) Band diagonal; (b) block triangular; (c) block tridiagonal; (d) singly bordered block diagonal; (e) doubly bordered block diagonal; (f) singly bordered block triangular; (g) bordered band-triangular; (h) and (i) singly and doubly bordered band diagonal; (j) and (k) other! (after Tewarson) [1].

be used with the particular matrix. Consult [2,3] for references on this. The NAG library [4] has an analyze/factorize/operate capability. A substantial collection of routines for sparse matrix calculation is also available from IMSL [5] as the *Yale Sparse Matrix Package* [6].

You should be aware that the special order of interchanges and eliminations, prescribed by a sparse matrix method so as to minimize fill-ins and arithmetic operations, generally acts to decrease the method's numerical stability as compared to, e.g., regular LU decomposition with pivoting. Scaling your problem so as to make its nonzero matrix elements have comparable magnitudes (if you can do it) will sometimes ameliorate this problem.

In the remainder of this section, we present some concepts which are applicable to some general classes of sparse matrices, and which do not necessarily depend on details of the pattern of sparsity.

Sherman-Morrison Formula

Suppose that you have already obtained, by herculean effort, the inverse matrix $\mathbf{A}^{-1}$ of a square matrix $\mathbf{A}$. Now you want to make a "small" change in $\mathbf{A}$, for example change one element a_{ij}, or a few elements, or one row, or one column. Is there any way of calculating the corresponding change in $\mathbf{A}^{-1}$ without repeating your difficult labors? Yes, if your change is of the form

$$\mathbf{A} \;\rightarrow\; (\mathbf{A} + \mathbf{u} \otimes \mathbf{v}) \tag{2.7.1}$$

for some vectors $\mathbf{u}$ and $\mathbf{v}$. If $\mathbf{u}$ is a unit vector $\mathbf{e}_i$, then (2.7.1) adds the components of $\mathbf{v}$ to the ith row. (Recall that $\mathbf{u} \otimes \mathbf{v}$ is a matrix whose i, jth element is the product of the ith component of $\mathbf{u}$ and the jth component of $\mathbf{v}$.) If $\mathbf{v}$ is a unit vector $\mathbf{e}_j$, then (2.7.1) adds the components of $\mathbf{u}$ to the jth column. If both $\mathbf{u}$ and $\mathbf{v}$ are proportional to unit vectors $\mathbf{e}_i$ and $\mathbf{e}_j$ respectively, then a term is added only to the element a_{ij}.

The *Sherman-Morrison* formula gives the inverse $(\mathbf{A} + \mathbf{u} \otimes \mathbf{v})^{-1}$, and is derived briefly as follows:

$$\begin{aligned}
(\mathbf{A} + \mathbf{u} \otimes \mathbf{v})^{-1} &= (\mathbf{1} + \mathbf{A}^{-1} \cdot \mathbf{u} \otimes \mathbf{v})^{-1} \cdot \mathbf{A}^{-1} \\
&= (\mathbf{1} - \mathbf{A}^{-1} \cdot \mathbf{u} \otimes \mathbf{v} + \mathbf{A}^{-1} \cdot \mathbf{u} \otimes \mathbf{v} \cdot \mathbf{A}^{-1} \cdot \mathbf{u} \otimes \mathbf{v} - \ldots) \cdot \mathbf{A}^{-1} \\
&= \mathbf{A}^{-1} - \mathbf{A}^{-1} \cdot \mathbf{u} \otimes \mathbf{v} \cdot \mathbf{A}^{-1} \,(1 - \lambda + \lambda^2 - \ldots) \\
&= \mathbf{A}^{-1} - \frac{(\mathbf{A}^{-1} \cdot \mathbf{u}) \otimes (\mathbf{v} \cdot \mathbf{A}^{-1})}{1 + \lambda}
\end{aligned} \tag{2.7.2}$$

where

$$\lambda \equiv \mathbf{v} \cdot \mathbf{A}^{-1} \cdot \mathbf{u} \tag{2.7.3}$$

The second line of (2.7.2) is a formal power series expansion. In the third line, the associativity of outer and inner products is used to factor out the scalars λ.

The use of (2.7.2) is this: Given $\mathbf{A}^{-1}$ and the vectors $\mathbf{u}$ and $\mathbf{v}$, we need only perform two matrix multiplications and a vector dot product,

$$\mathbf{z} \equiv \mathbf{A}^{-1} \cdot \mathbf{u} \qquad \mathbf{w} \equiv (\mathbf{A}^{-1})^T \cdot \mathbf{v} \qquad \lambda = \mathbf{v} \cdot \mathbf{z} \tag{2.7.4}$$

to get the desired change in the inverse

$$\mathbf{A}^{-1} \quad \rightarrow \quad \mathbf{A}^{-1} - \frac{\mathbf{z} \otimes \mathbf{w}}{1 + \lambda} \tag{2.7.5}$$

The whole procedure requires only $3N^2$ multiplies and a like number of adds (an even smaller number if **u** or **v** is a unit vector).

The Sherman-Morrison formula can be directly applied to a class of sparse problems. If you already have a fast way of calculating the inverse of **A** (e.g., a tridiagonal matrix, or some other standard sparse form), then (2.7.4)–(2.7.5) allow you to build up to your related but more complicated form, adding for example a row or column at a time. Notice that you can apply the Sherman-Morrison formula more than once successively, using at each stage the most recent update of $\mathbf{A}^{-1}$ (equation 2.7.5). Of course, if you have to modify *every* row, then you are back to an N^3 method. The constant in front of the N^3 is only a few times worse than the better direct methods, but you have deprived yourself of the stabilizing advantages of pivoting — so be careful.

For some other sparse problems, the Sherman-Morrison formula cannot be directly applied for the simple reason that storage of the whole inverse matrix $\mathbf{A}^{-1}$ is not feasible. If you want to add only a single correction of the form $\mathbf{u} \otimes \mathbf{v}$, and solve the linear system

$$(\mathbf{A} + \mathbf{u} \otimes \mathbf{v}) \cdot \mathbf{x} = \mathbf{b} \tag{2.7.6}$$

then you proceed as follows. Using the fast method that is presumed available for the matrix **A**, solve the two auxiliary problems

$$\mathbf{A} \cdot \mathbf{y} = \mathbf{b} \qquad \mathbf{A} \cdot \mathbf{z} = \mathbf{u} \tag{2.7.7}$$

for the vectors **y** and **z**. In terms of these,

$$\mathbf{x} = \mathbf{y} - \left[\frac{\mathbf{v} \cdot \mathbf{y}}{1 + (\mathbf{v} \cdot \mathbf{z})} \right] \mathbf{z} \tag{2.7.8}$$

as we see by multiplying (2.7.2) on the right by **b**.

Cyclic Tridiagonal Systems

So-called *cyclic tridiagonal systems* occur quite frequently, and are a good example of how to use the Sherman-Morrison formula in the manner just described. The equations have the form

$$\begin{bmatrix} b_1 & c_1 & 0 & \cdots & & & \beta \\ a_2 & b_2 & c_2 & \cdots & & & \\ & & & \cdots & & & \\ & & & \cdots & a_{N-1} & b_{N-1} & c_{N-1} \\ \alpha & & & \cdots & 0 & a_N & b_N \end{bmatrix} \cdot \begin{bmatrix} x_1 \\ x_2 \\ \cdots \\ x_{N-1} \\ x_N \end{bmatrix} = \begin{bmatrix} r_1 \\ r_2 \\ \cdots \\ r_{N-1} \\ r_N \end{bmatrix} \tag{2.7.9}$$

This is a tridiagonal system, except for the matrix elements α and β in the corners. Forms like this are typically generated by finite-differencing differential equations with periodic boundary conditions (§19.4).

We use the Sherman-Morrison formula, treating the system as tridiagonal plus a correction. In the notation of equation (2.7.6), define vectors **u** and **v** to be

$$\mathbf{u} = \begin{bmatrix} \gamma \\ 0 \\ \vdots \\ 0 \\ \alpha \end{bmatrix} \qquad \mathbf{v} = \begin{bmatrix} 1 \\ 0 \\ \vdots \\ 0 \\ \beta/\gamma \end{bmatrix} \tag{2.7.10}$$

Here γ is arbitrary for the moment. Then the matrix **A** is the tridiagonal part of the matrix in (2.7.9), with two terms modified:

$$b'_1 = b_1 - \gamma, \qquad b'_N = b_N - \alpha\beta/\gamma \tag{2.7.11}$$

We now solve equations (2.7.7) with the standard tridiagonal algorithm, and then get the solution from equation (2.7.8).

The routine `cyclic` below implements this algorithm. We choose the arbitrary parameter $\gamma = -b_1$ to avoid loss of precision by subtraction in the first of equations (2.7.11). In the unlikely event that this causes loss of precision in the second of these equations, you can make a different choice.

```
#include "nrutil.h"

void cyclic(float a[], float b[], float c[], float alpha, float beta,
    float r[], float x[], unsigned long n)
```
Solves for a vector `x[1..n]` the "cyclic" set of linear equations given by equation (2.7.9). `a`, `b`, `c`, and `r` are input vectors, all dimensioned as `[1..n]`, while `alpha` and `beta` are the corner entries in the matrix. The input is not modified.
```
{
    void tridag(float a[], float b[], float c[], float r[], float u[],
        unsigned long n);
    unsigned long i;
    float fact,gamma,*bb,*u,*z;

    if (n <= 2) nrerror("n too small in cyclic");
    bb=vector(1,n);
    u=vector(1,n);
    z=vector(1,n);
    gamma = -b[1];                              Avoid subtraction error in forming bb[1].
    bb[1]=b[1]-gamma;                           Set up the diagonal of the modified tridi-
    bb[n]=b[n]-alpha*beta/gamma;                    agonal system.
    for (i=2;i<n;i++) bb[i]=b[i];
    tridag(a,bb,c,r,x,n);                       Solve A · x = r.
    u[1]=gamma;                                 Set up the vector u.
    u[n]=alpha;
    for (i=2;i<n;i++) u[i]=0.0;
    tridag(a,bb,c,u,z,n);                       Solve A · z = u.
    fact=(x[1]+beta*x[n]/gamma)/                Form v · x/(1 + v · z).
        (1.0+z[1]+beta*z[n]/gamma);
    for (i=1;i<=n;i++) x[i] -= fact*z[i];       Now get the solution vector x.
    free_vector(z,1,n);
    free_vector(u,1,n);
    free_vector(bb,1,n);
}
```

Woodbury Formula

If you want to add more than a single correction term, then you cannot use (2.7.8) repeatedly, since without storing a new $\mathbf{A}^{-1}$ you will not be able to solve the auxiliary problems (2.7.7) efficiently after the first step. Instead, you need the *Woodbury formula*, which is the block-matrix version of the Sherman-Morrison formula,

$$\begin{aligned}(\mathbf{A} + \mathbf{U}\cdot\mathbf{V}^T)^{-1} \\ = \mathbf{A}^{-1} - \left[\mathbf{A}^{-1}\cdot\mathbf{U}\cdot(\mathbf{1}+\mathbf{V}^T\cdot\mathbf{A}^{-1}\cdot\mathbf{U})^{-1}\cdot\mathbf{V}^T\cdot\mathbf{A}^{-1}\right]\end{aligned} \tag{2.7.12}$$

Here A is, as usual, an $N \times N$ matrix, while **U** and **V** are $N \times P$ matrices with $P < N$ and usually $P \ll N$. The inner piece of the correction term may become clearer if written as the tableau,

$$\left[\begin{array}{c} \\ \\ \mathbf{U} \\ \\ \\ \end{array}\right] \cdot \left[\mathbf{1} + \mathbf{V}^T \cdot \mathbf{A}^{-1} \cdot \mathbf{U}\right]^{-1} \cdot \left[\begin{array}{ccc} & \mathbf{V}^T & \end{array}\right] \tag{2.7.13}$$

where you can see that the matrix whose inverse is needed is only $P \times P$ rather than $N \times N$.

The relation between the Woodbury formula and successive applications of the Sherman-Morrison formula is now clarified by noting that, if **U** is the matrix formed by columns out of the P vectors $\mathbf{u}_1, \dots, \mathbf{u}_P$, and **V** is the matrix formed by columns out of the P vectors $\mathbf{v}_1, \dots, \mathbf{v}_P$,

$$\mathbf{U} \equiv \left[\begin{array}{c} \\ \mathbf{u}_1 \\ \\ \end{array}\right] \cdots \left[\begin{array}{c} \\ \mathbf{u}_P \\ \\ \end{array}\right] \qquad \mathbf{V} \equiv \left[\begin{array}{c} \\ \mathbf{v}_1 \\ \\ \end{array}\right] \cdots \left[\begin{array}{c} \\ \mathbf{v}_P \\ \\ \end{array}\right] \tag{2.7.14}$$

then two ways of expressing the same correction to **A** are

$$\left(\mathbf{A} + \sum_{k=1}^{P} \mathbf{u}_k \otimes \mathbf{v}_k\right) = (\mathbf{A} + \mathbf{U} \cdot \mathbf{V}^T) \tag{2.7.15}$$

(Note that the subscripts on **u** and **v** do *not* denote components, but rather distinguish the different column vectors.)

Equation (2.7.15) reveals that, if you have $\mathbf{A}^{-1}$ in storage, then you can either make the P corrections in one fell swoop by using (2.7.12), inverting a $P \times P$ matrix, or else make them by applying (2.7.5) P successive times.

If you don't have storage for $\mathbf{A}^{-1}$, then you *must* use (2.7.12) in the following way: To solve the linear equation

$$\left(\mathbf{A} + \sum_{k=1}^{P} \mathbf{u}_k \otimes \mathbf{v}_k\right) \cdot \mathbf{x} = \mathbf{b} \tag{2.7.16}$$

first solve the P auxiliary problems

$$\begin{aligned} \mathbf{A} \cdot \mathbf{z}_1 &= \mathbf{u}_1 \\ \mathbf{A} \cdot \mathbf{z}_2 &= \mathbf{u}_2 \\ &\cdots \\ \mathbf{A} \cdot \mathbf{z}_P &= \mathbf{u}_P \end{aligned} \tag{2.7.17}$$

and construct the matrix **Z** by columns from the **z**'s obtained,

$$\mathbf{Z} \equiv \left[\begin{array}{c} \\ \mathbf{z}_1 \\ \\ \end{array}\right] \cdots \left[\begin{array}{c} \\ \mathbf{z}_P \\ \\ \end{array}\right] \tag{2.7.18}$$

Next, do the $P \times P$ matrix inversion

$$\mathbf{H} \equiv (\mathbf{1} + \mathbf{V}^T \cdot \mathbf{Z})^{-1} \tag{2.7.19}$$

Finally, solve the one further auxiliary problem

$$\mathbf{A} \cdot \mathbf{y} = \mathbf{b} \tag{2.7.20}$$

In terms of these quantities, the solution is given by

$$\mathbf{x} = \mathbf{y} - \mathbf{Z} \cdot \left[\mathbf{H} \cdot (\mathbf{V}^T \cdot \mathbf{y}) \right] \tag{2.7.21}$$

Inversion by Partitioning

Once in a while, you will encounter a matrix (not even necessarily sparse) that can be inverted efficiently by partitioning. Suppose that the $N \times N$ matrix **A** is partitioned into

$$\mathbf{A} = \begin{bmatrix} \mathbf{P} & \mathbf{Q} \\ \mathbf{R} & \mathbf{S} \end{bmatrix} \tag{2.7.22}$$

where **P** and **S** are square matrices of size $p \times p$ and $s \times s$ respectively ($p + s = N$). The matrices **Q** and **R** are not necessarily square, and have sizes $p \times s$ and $s \times p$, respectively.

If the inverse of **A** is partitioned in the same manner,

$$\mathbf{A}^{-1} = \begin{bmatrix} \widetilde{\mathbf{P}} & \widetilde{\mathbf{Q}} \\ \widetilde{\mathbf{R}} & \widetilde{\mathbf{S}} \end{bmatrix} \tag{2.7.23}$$

then $\widetilde{\mathbf{P}}$, $\widetilde{\mathbf{Q}}$, $\widetilde{\mathbf{R}}$, $\widetilde{\mathbf{S}}$, which have the same sizes as **P**, **Q**, **R**, **S**, respectively, can be found by either the formulas

$$\begin{aligned} \widetilde{\mathbf{P}} &= (\mathbf{P} - \mathbf{Q} \cdot \mathbf{S}^{-1} \cdot \mathbf{R})^{-1} \\ \widetilde{\mathbf{Q}} &= -(\mathbf{P} - \mathbf{Q} \cdot \mathbf{S}^{-1} \cdot \mathbf{R})^{-1} \cdot (\mathbf{Q} \cdot \mathbf{S}^{-1}) \\ \widetilde{\mathbf{R}} &= -(\mathbf{S}^{-1} \cdot \mathbf{R}) \cdot (\mathbf{P} - \mathbf{Q} \cdot \mathbf{S}^{-1} \cdot \mathbf{R})^{-1} \\ \widetilde{\mathbf{S}} &= \mathbf{S}^{-1} + (\mathbf{S}^{-1} \cdot \mathbf{R}) \cdot (\mathbf{P} - \mathbf{Q} \cdot \mathbf{S}^{-1} \cdot \mathbf{R})^{-1} \cdot (\mathbf{Q} \cdot \mathbf{S}^{-1}) \end{aligned} \tag{2.7.24}$$

or else by the equivalent formulas

$$\begin{aligned} \widetilde{\mathbf{P}} &= \mathbf{P}^{-1} + (\mathbf{P}^{-1} \cdot \mathbf{Q}) \cdot (\mathbf{S} - \mathbf{R} \cdot \mathbf{P}^{-1} \cdot \mathbf{Q})^{-1} \cdot (\mathbf{R} \cdot \mathbf{P}^{-1}) \\ \widetilde{\mathbf{Q}} &= -(\mathbf{P}^{-1} \cdot \mathbf{Q}) \cdot (\mathbf{S} - \mathbf{R} \cdot \mathbf{P}^{-1} \cdot \mathbf{Q})^{-1} \\ \widetilde{\mathbf{R}} &= -(\mathbf{S} - \mathbf{R} \cdot \mathbf{P}^{-1} \cdot \mathbf{Q})^{-1} \cdot (\mathbf{R} \cdot \mathbf{P}^{-1}) \\ \widetilde{\mathbf{S}} &= (\mathbf{S} - \mathbf{R} \cdot \mathbf{P}^{-1} \cdot \mathbf{Q})^{-1} \end{aligned} \tag{2.7.25}$$

The parentheses in equations (2.7.24) and (2.7.25) highlight repeated factors that you may wish to compute only once. (Of course, by associativity, you can instead do the matrix multiplications in any order you like.) The choice between using equation (2.7.24) and (2.7.25) depends on whether you want $\widetilde{\mathbf{P}}$ or $\widetilde{\mathbf{S}}$ to have the simpler formula; or on whether the repeated expression $(\mathbf{S} - \mathbf{R} \cdot \mathbf{P}^{-1} \cdot \mathbf{Q})^{-1}$ is easier to calculate than the expression $(\mathbf{P} - \mathbf{Q} \cdot \mathbf{S}^{-1} \cdot \mathbf{R})^{-1}$; or on the relative sizes of **P** and **S**; or on whether $\mathbf{P}^{-1}$ or $\mathbf{S}^{-1}$ is already known.

Another sometimes useful formula is for the determinant of the partitioned matrix,

$$\det \mathbf{A} = \det \mathbf{P} \det(\mathbf{S} - \mathbf{R} \cdot \mathbf{P}^{-1} \cdot \mathbf{Q}) = \det \mathbf{S} \det(\mathbf{P} - \mathbf{Q} \cdot \mathbf{S}^{-1} \cdot \mathbf{R}) \tag{2.7.26}$$

Indexed Storage of Sparse Matrices

We have already seen (§2.4) that tri- or band-diagonal matrices can be stored in a compact format that allocates storage only to elements which can be nonzero, plus perhaps a few wasted locations to make the bookkeeping easier. What about more general sparse matrices? When a sparse matrix of dimension $N \times N$ contains only a few times N nonzero elements (a typical case), it is surely inefficient — and often physically impossible — to allocate storage for all N^2 elements. Even if one did allocate such storage, it would be inefficient or prohibitive in machine time to loop over all of it in search of nonzero elements.

Obviously some kind of indexed storage scheme is required, one that stores only nonzero matrix elements, along with sufficient auxiliary information to determine where an element logically belongs and how the various elements can be looped over in common matrix operations. Unfortunately, there is no one standard scheme in general use. Knuth [7] describes one method. The Yale Sparse Matrix Package [6] and ITPACK [8] describe several other methods. For most applications, we favor the storage scheme used by PCGPACK [9], which is almost the same as that described by Bentley [10], and also similar to one of the Yale Sparse Matrix Package methods. The advantage of this scheme, which can be called *row-indexed sparse storage mode*, is that it requires storage of only about two times the number of nonzero matrix elements. (Other methods can require as much as three or five times.) For simplicity, we will treat only the case of square matrices, which occurs most frequently in practice.

To represent a matrix **A** of dimension $N \times N$, the row-indexed scheme sets up two one-dimensional arrays, call them `sa` and `ija`. The first of these stores matrix element values in single or double precision as desired; the second stores integer values. The storage rules are:

- The first N locations of `sa` store **A**'s diagonal matrix elements, in order. (Note that diagonal elements are stored even if they are zero; this is at most a slight storage inefficiency, since diagonal elements are nonzero in most realistic applications.)
- Each of the first N locations of `ija` stores the index of the array `sa` that contains the first *off-diagonal* element of the corresponding row of the matrix. (If there are no off-diagonal elements for that row, it is one greater than the index in `sa` of the most recently stored element of a previous row.)
- Location 1 of `ija` is always equal to $N + 2$. (It can be read to determine N.)
- Location $N + 1$ of `ija` is one greater than the index in `sa` of the last off-diagonal element of the last row. (It can be read to determine the number of nonzero elements in the matrix, or the number of elements in the arrays `sa` and `ija`.) Location $N + 1$ of `sa` is not used and can be set arbitrarily.
- Entries in `sa` at locations $\geq N + 2$ contain **A**'s off-diagonal values, ordered by rows and, within each row, ordered by columns.
- Entries in `ija` at locations $\geq N+2$ contain the column number of the corresponding element in `sa`.

While these rules seem arbitrary at first sight, they result in a rather elegant storage scheme. As an example, consider the matrix

$$\begin{bmatrix} 3. & 0. & 1. & 0. & 0. \\ 0. & 4. & 0. & 0. & 0. \\ 0. & 7. & 5. & 9. & 0. \\ 0. & 0. & 0. & 0. & 2. \\ 0. & 0. & 0. & 6. & 5. \end{bmatrix} \tag{2.7.27}$$

In row-indexed compact storage, matrix (2.7.27) is represented by the two arrays of length 11, as follows

index k	1	2	3	4	5	6	7	8	9	10	11
ija[k]	7	8	8	10	11	12	3	2	4	5	4
sa[k]	3.	4.	5.	0.	5.	x	1.	7.	9.	2.	6.

(2.7.28)

Here x is an arbitrary value. Notice that, according to the storage rules, the value of N (namely 5) is `ija[1]-2`, and the length of each array is `ija[ija[1]-1]-1`, namely 11. The diagonal element in row `i` is `sa[i]`, and the off-diagonal elements in that row are in `sa[k]` where `k` loops from `ija[i]` to `ija[i+1]-1`, if the upper limit is greater or equal to the lower one (as in C's `for` loops).

Here is a routine, `sprsin`, that converts a matrix from full storage mode into row-indexed sparse storage mode, throwing away any elements that are less than a specified threshold. Of course, the principal use of sparse storage mode is for matrices whose full storage mode won't fit into your machine at all; then you have to generate them directly into sparse format. Nevertheless `sprsin` is useful as a precise algorithmic definition of the storage scheme, for subscale testing of large problems, and for the case where execution time, rather than storage, furnishes the impetus to sparse storage.

```
#include <math.h>

void sprsin(float **a, int n, float thresh, unsigned long nmax, float sa[],
    unsigned long ija[])
```
Converts a square matrix `a[1..n][1..n]` into row-indexed sparse storage mode. Only elements of `a` with magnitude ≥`thresh` are retained. Output is in two linear arrays with dimension `nmax` (an input parameter): `sa[1..]` contains array values, indexed by `ija[1..]`. The number of elements filled of `sa` and `ija` on output are both `ija[ija[1]-1]-1` (see text).
```
{
    void nrerror(char error_text[]);
    int i,j;
    unsigned long k;

    for (j=1;j<=n;j++) sa[j]=a[j][j];       Store diagonal elements.
    ija[1]=n+2;                             Index to 1st row off-diagonal element, if any.
    k=n+1;
    for (i=1;i<=n;i++) {                    Loop over rows.
        for (j=1;j<=n;j++) {                Loop over columns.
            if (fabs(a[i][j]) >= thresh && i != j) {
                if (++k > nmax) nrerror("sprsin: nmax too small");
                sa[k]=a[i][j];              Store off-diagonal elements and their columns.
                ija[k]=j;
            }
        }
        ija[i+1]=k+1;                       As each row is completed, store index to
    }                                           next.
}
```

The single most important use of a matrix in row-indexed sparse storage mode is to multiply a vector to its right. In fact, the storage mode is optimized for just this purpose. The following routine is thus very simple.

```
void sprsax(float sa[], unsigned long ija[], float x[], float b[],
    unsigned long n)
```
Multiply a matrix in row-index sparse storage arrays `sa` and `ija` by a vector `x[1..n]`, giving a vector `b[1..n]`.
```
{
    void nrerror(char error_text[]);
    unsigned long i,k;

    if (ija[1] != n+2) nrerror("sprsax: mismatched vector and matrix");
    for (i=1;i<=n;i++) {
        b[i]=sa[i]*x[i];                          Start with diagonal term.
        for (k=ija[i];k<=ija[i+1]-1;k++)          Loop over off-diagonal terms.
            b[i] += sa[k]*x[ija[k]];

    }
}
```

It is also simple to multiply the *transpose* of a matrix by a vector to its right. (We will use this operation later in this section.) Note that the transpose matrix is not actually constructed.

```
void sprstx(float sa[], unsigned long ija[], float x[], float b[],
    unsigned long n)
Multiply the transpose of a matrix in row-index sparse storage arrays sa and ija by a vector
x[1..n], giving a vector b[1..n].
{
    void nrerror(char error_text[]);
    unsigned long i,j,k;

    if (ija[1] != n+2) nrerror("mismatched vector and matrix in sprstx");
    for (i=1;i<=n;i++) b[i]=sa[i]*x[i];           Start with diagonal terms.
    for (i=1;i<=n;i++) {                          Loop over off-diagonal terms.
        for (k=ija[i];k<=ija[i+1]-1;k++) {
            j=ija[k];
            b[j] += sa[k]*x[i];
        }
    }
}
```

(Double precision versions of sprsax and sprstx, named dsprsax and dsprstx, are used by the routine atimes later in this section. You can easily make the conversion, or else get the converted routines from the *Numerical Recipes* diskettes.)

In fact, because the choice of row-indexed storage treats rows and columns quite differently, it is quite an involved operation to construct the transpose of a matrix, given the matrix itself in row-indexed sparse storage mode. When the operation cannot be avoided, it is done as follows: An index of all off-diagonal elements by their columns is constructed (see §8.4). The elements are then written to the output array in column order. As each element is written, its row is determined and stored. Finally, the elements in each column are sorted by row.

```
void sprstp(float sa[], unsigned long ija[], float sb[], unsigned long ijb[])
Construct the transpose of a sparse square matrix, from row-index sparse storage arrays sa and
ija into arrays sb and ijb.
{
    void iindexx(unsigned long n, long arr[], unsigned long indx[]);
    Version of indexx with all float variables changed to long.
    unsigned long j,jl,jm,jp,ju,k,m,n2,noff,inc,iv;
    float v;

    n2=ija[1];                                    Linear size of matrix plus 2.
    for (j=1;j<=n2-2;j++) sb[j]=sa[j];            Diagonal elements.
    iindexx(ija[n2-1]-ija[1],(long *)&ija[n2-1],&ijb[n2-1]);
    Index all off-diagonal elements by their columns.
    jp=0;
    for (k=ija[1];k<=ija[n2-1]-1;k++) {           Loop over output off-diagonal elements.
        m=ijb[k]+n2-1;                            Use index table to store by (former) columns.
        sb[k]=sa[m];
        for (j=jp+1;j<=ija[m];j++) ijb[j]=k;         Fill in the index to any omitted rows.
        jp=ija[m];                              Use bisection to find which row element
        jl=1;                                       m is in and put that into ijb[k].
        ju=n2-1;
        while (ju-jl > 1) {
            jm=(ju+jl)/2;
            if (ija[jm] > m) ju=jm; else jl=jm;
        }
        ijb[k]=jl;
    }
    for (j=jp+1;j<n2;j++) ijb[j]=ija[n2-1];
    for (j=1;j<=n2-2;j++) {                     Make a final pass to sort each row by
        jl=ijb[j+1]-ijb[j];                         Shell sort algorithm.
        noff=ijb[j]-1;
        inc=1;
        do {
            inc *= 3;
            inc++;
        } while (inc <= jl);
        do {
            inc /= 3;
            for (k=noff+inc+1;k<=noff+jl;k++) {
                iv=ijb[k];
                v=sb[k];
                m=k;
                while (ijb[m-inc] > iv) {
                    ijb[m]=ijb[m-inc];
                    sb[m]=sb[m-inc];
                    m -= inc;
                    if (m-noff <= inc) break;
                }
                ijb[m]=iv;
                sb[m]=v;
            }
        } while (inc > 1);
    }
}
```

The above routine embeds internally a sorting algorithm from §8.1, but calls the external routine `iindexx` to construct the initial column index. This routine is identical to `indexx`, as listed in §8.4, except that the latter's two `float` declarations should be changed to `long`. (The *Numerical Recipes* diskettes include both `indexx` and `iindexx`.) In fact, you can often use `indexx` *without* making these changes, since many computers have the property that numerical values will sort correctly independently of whether they are interpreted as floating or integer values.

As final examples of the manipulation of sparse matrices, we give two routines for the multiplication of two sparse matrices. These are useful for techniques to be described in §13.10.

In general, the product of two sparse matrices is not itself sparse. One therefore wants to limit the size of the product matrix in one of two ways: either compute only those elements of the product that are specified in advance by a known pattern of sparsity, or else compute all nonzero elements, but store only those whose magnitude exceeds some threshold value. The former technique, when it can be used, is quite efficient. The pattern of sparsity is specified by furnishing an index array in row-index sparse storage format (e.g., `ija`). The program then constructs a corresponding value array (e.g., `sa`). The latter technique runs the danger of excessive compute times and unknown output sizes, so it must be used cautiously.

With row-index storage, it is much more natural to multiply a matrix (on the left) by the *transpose* of a matrix (on the right), so that one is crunching rows on rows, rather than rows on columns. Our routines therefore calculate $\mathbf{A} \cdot \mathbf{B}^T$, rather than $\mathbf{A} \cdot \mathbf{B}$. This means that you have to run your right-hand matrix through the transpose routine `sprstp` before sending it to the matrix multiply routine.

The two implementing routines, sprspm for "pattern multiply" and sprstm for "threshold multiply" are quite similar in structure. Both are complicated by the logic of the various combinations of diagonal or off-diagonal elements for the two input streams and output stream.

```
void sprspm(float sa[], unsigned long ija[], float sb[], unsigned long ijb[],
    float sc[], unsigned long ijc[])
```

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 only those components of the matrix product that are *pre-specified* by the input index array ijc, which is not modified. On output, the arrays sc and ijc 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,m,ma,mb,mbb,mn;
    float sum;

    if (ija[1] != ijb[1] || ija[1] != ijc[1])
        nrerror("sprspm: sizes do not match");
    for (i=1;i<=ijc[1]-2;i++) {          Loop over rows.
        j=m=i;                           Set up so that first pass through loop does the
        mn=ijc[i];                           diagonal component.
        sum=sa[i]*sb[i];
        for (;;) {                       Main loop over each component to be output.
            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];
            }
            sc[m]=sum;
            sum=0.0;                     Reset indices for next pass through loop.
            if (mn >= ijc[i+1]) break;
            j=ijc[m=mn++];
        }
    }
}
```

```
#include <math.h>

void sprstm(float sa[], unsigned long ija[], float sb[], unsigned long ijb[],
    float thresh, unsigned long nmax, float sc[], unsigned long ijc[])
```

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 **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 **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 **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 **A**, but it does all the redundant computations.

Another variant of the general algorithm corresponds to a symmetric but non-positive definite **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

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

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

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

For any nonsingular matrix **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

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

Don't! The condition number of the matrix $\mathbf{A}^T \cdot \mathbf{A}$ is the square of the condition number of **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 **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),

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

The idea is that you might already be able to solve your linear system easily for some $\widetilde{\mathbf{A}}$ close to **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

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

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 $\widetilde{\mathbf{A}}$ (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 $\widetilde{\mathbf{A}}^T$.
        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 $\widetilde{\mathbf{A}}\cdot\mathbf{z}=\mathbf{r}$ 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 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 $a_{ij} = 1/(i + j - 1),\ i, j = 1, \ldots, N$ 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 $x_1, x_2, \ldots, x_N$, in terms of which its N^2 components are the integer powers $x_i^{j-1},\ i, j = 1, \ldots, N$. 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.