REAL absa, absb
absa = abs(a)
absb = abs(b)
if(absa.gt.absb)then
  pythag = absa*sqrt(1.+(absb/absa)**2)
else
  if(absb.eq.0.)then
    pythag = 0.
  else
    pythag = absb*sqrt(1.+(absa/absb)**2)
  endif
endif
return
END

(Double precision versions of svdcmp, svbksb, and pythag, named dsvdcmp, dsvbksb, and dpythag, are used by the routine ratlsq in §5.13. You can easily make the conversions, or else get the converted routines from the Numerical Recipes diskette.)

CITED REFERENCES AND FURTHER READING:

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 archetypical sparse form in §2.4, the banded 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 $M$
- 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 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 \( A^{-1} \) of a square matrix \( A \). Now you want to make a “small” change in \( A \), 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 \( A^{-1} \) without repeating
your difficult labors? Yes, if your change is of the form

\[ A \rightarrow (A + u \otimes v) \]  

(2.7.1)

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

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

\[
(A + u \otimes v)^{-1} = (1 + A^{-1} \cdot u \otimes v)^{-1} \cdot A^{-1} \\
= (1 - A^{-1} \cdot u \otimes A^{-1} \cdot u \otimes v \cdot A^{-1} \cdot u \otimes v - \ldots) \cdot A^{-1} \\
= A^{-1} - A^{-1} \cdot u \otimes v \cdot A^{-1} (1 - \lambda + \lambda^2 - \ldots) \\
= A^{-1} - \frac{(A^{-1} \cdot u) \otimes (v \cdot A^{-1})}{1 + \lambda}
\]

(2.7.2)

where

\[
\lambda \equiv v \cdot A^{-1} \cdot u
\]

(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 \( \lambda \).

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

\[
z \equiv A^{-1} \cdot u \quad w \equiv (A^{-1})^T \cdot v \quad \lambda = v \cdot z
\]

(2.7.4)

to get the desired change in the inverse

\[
A^{-1} \rightarrow A^{-1} - \frac{z \otimes w}{1 + \lambda}
\]

(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 \( A^{-1} \) (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 \( A^{-1} \)
is not feasible. If you want to add only a single correction of the form \( u \otimes v \), and solve the linear system

\[
(A + u \otimes v) \cdot x = b
\]  

(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

\[
A \cdot y = b \quad A \cdot z = u
\]  

(2.7.7)

for the vectors \( y \) and \( z \). In terms of these,

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

(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

\[
\begin{bmatrix}
 b_1 & c_1 & 0 & \cdots \\
 a_2 & b_2 & c_2 & \cdots \\
  & \ddots & \ddots & \ddots \\
 \alpha & \cdots & a_{N-1} & b_{N-1} & c_{N-1} & 0 & a_{N} & b_{N}
\end{bmatrix}
\begin{bmatrix}
 x_1 \\
 x_2 \\
 \vdots \\
 x_{N-1} \\
 x_N
\end{bmatrix}
= 
\begin{bmatrix}
 r_1 \\
 r_2 \\
 \vdots \\
 r_{N-1} \\
 r_N
\end{bmatrix}
\]  

(2.7.9)

This is a tridiagonal system, except for the matrix elements \( \alpha \) and \( \beta \) 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

\[
u = \begin{bmatrix}
 \gamma \\
 0 \\
 \vdots \\
 0 \\
 \alpha
\end{bmatrix}, \quad v = \begin{bmatrix}
 1 \\
 0 \\
 \vdots \\
 0 \\
 \beta/\gamma
\end{bmatrix}
\]  

(2.7.10)

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

\[
b'_1 = b_1 - \gamma, \quad b'_N = b_N - \alpha \beta / \gamma
\]  

(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 \texttt{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.
SUBROUTINE cyclic(a,b,c,alpha,beta,r,x,n)
INTEGER n,NMAX
REAL alpha,beta,a(n),b(n),c(n),r(n),x(n)
PARAMETER (NMAX=500)
C USES tridag
Solves for a vector \( x(1:n) \) the "cyclic" set of linear equations given by equation (2.7.9).
\( a, b, \) and \( r \) are input vectors, while \( \alpha \) and \( \beta \) are the corner entries in the matrix.

The input is not modified.

INTEGER i
REAL fact,gamma,bb(NMAX),u(NMAX),z(NMAX)
if(n.le.2)pause 'n too small in cyclic'
if(n.gt.NMAX)pause 'NMAX too small in cyclic'

\[ \gamma = -b(1) \]
Avoid subtraction error in forming \( bb(1) \).

\[ bb(1) = b(1) - \gamma \]
Set up the diagonal of the modified tridiagonal system.

\[ bb(n) = b(n) - \alpha \beta / \gamma \]
do i=2,n-1
\[ bb(i) = b(i) \]
enddo

call tridag(a,bb,c,r,x,n)
Solve \( A \cdot x = r \).

\[ u(1) = \gamma \]
Set up the vector \( u \).

\[ u(n) = \alpha \]
do i=2,n-1
\[ u(i) = 0. \]
enddo

call tridag(a,bb,c,u,z,n)
Solve \( A \cdot z = u \).

\[ \text{fact} = (x(1) + \beta \times x(n) / \gamma) / (1 + z(1) + \beta \times z(n) / \gamma) \]
Form \( \text{v} \cdot x / (1 + \text{v} \cdot z) \).
do i=1,n
\[ x(i) = x(i) - \text{fact} \times z(i) \]
enddo
return
END

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 \( A^{-1} \) 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,

\[
(A + U \cdot V^T)^{-1} = A^{-1} - A^{-1} \cdot U \cdot (1 + V^T \cdot A^{-1} \cdot U)^{-1} \cdot V^T \cdot A^{-1}
\] (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,

\[
\begin{bmatrix}
U & \\
1 + V^T \cdot A^{-1} \cdot U & V^T
\end{bmatrix}^{-1}
\] (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 $u_1, \ldots, u_P$, and $V$ is the matrix formed by columns out of the $P$ vectors $v_1, \ldots, v_P$, then two ways of expressing the same correction to $A$ are

$$
(A + \sum_{k=1}^{P} u_k \otimes v_k) = (A + U \cdot V^T)
$$

(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 $A^{-1}$ 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 $A^{-1}$, then you must use (2.7.12) in the following way: To solve the linear equation

$$
(A + \sum_{k=1}^{P} u_k \otimes v_k) \cdot x = b
$$

first solve the $P$ auxiliary problems

\begin{align*}
A \cdot z_1 &= u_1 \\
A \cdot z_2 &= u_2 \\
&\quad \ldots \\
A \cdot z_P &= u_P
\end{align*}

(2.7.17)

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

$$
Z \equiv \begin{bmatrix} z_1 \\ \vdots \\ z_P \end{bmatrix}
$$

(2.7.18)

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

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

(2.7.19)

Finally, solve the one further auxiliary problem

$$
A \cdot y = b
$$

(2.7.20)

In terms of these quantities, the solution is given by

$$
x = y - Z \cdot [H \cdot (V^T \cdot y)]
$$

(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

\[
A = \begin{bmatrix} P & Q \\ R & S \end{bmatrix}
\] (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,

\[
A^{-1} = \begin{bmatrix} \tilde{P} & \tilde{Q} \\ \tilde{R} & \tilde{S} \end{bmatrix}
\] (2.7.23)

then \( \tilde{P}, \tilde{Q}, \tilde{R}, \tilde{S} \), which have the same sizes as \( P, Q, R, S \), respectively, can be found by either the formulas

\[
\tilde{P} = (P - Q \cdot S^{-1} \cdot R)^{-1}
\]

\[
\tilde{Q} = -(P - Q \cdot S^{-1} \cdot R)^{-1} \cdot (Q \cdot S^{-1})
\]

\[
\tilde{R} = -(S^{-1} \cdot R) \cdot (P - Q \cdot S^{-1} \cdot R)^{-1}
\]

\[
\tilde{S} = S^{-1} + (S^{-1} \cdot R) \cdot (P - Q \cdot S^{-1} \cdot R)^{-1} \cdot (Q \cdot S^{-1})
\] (2.7.24)

or else by the equivalent formulas

\[
\tilde{P} = P^{-1} + (P^{-1} \cdot Q) \cdot (S - R \cdot P^{-1} \cdot Q)^{-1} \cdot (R \cdot P^{-1})
\]

\[
\tilde{Q} = -(P^{-1} \cdot Q) \cdot (S - R \cdot P^{-1} \cdot Q)^{-1}
\]

\[
\tilde{R} = -(S - R \cdot P^{-1} \cdot Q)^{-1} \cdot (R \cdot P^{-1})
\]

\[
\tilde{S} = (S - R \cdot P^{-1} \cdot Q)^{-1}
\] (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 \( \tilde{P} \) or \( \tilde{S} \) to have the simpler formula; or on whether the repeated expression \( (S - R \cdot P^{-1} \cdot Q)^{-1} \) is easier to calculate than the expression \( (P - Q \cdot S^{-1} \cdot R)^{-1} \); or on the relative sizes of \( P \) and \( S \); or on whether \( P^{-1} \) or \( S^{-1} \) is already known.

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

\[
\det A = \det P \det (S - R \cdot P^{-1} \cdot Q) = \det S \det (P - Q \cdot S^{-1} \cdot R)
\] (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 logical size \( N \times N \) 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 logical size \( N \times N \), the row-indexed scheme sets up two one-dimensional arrays, call them \( aa \) 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 \( aa \) 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 \( aa \) 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 \( aa \) 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 \( aa \) 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 logical length of the arrays \( aa \) and \( ija \).) Location \( N + 1 \) of \( aa \) is not used and can be set arbitrarily.
- Entries in \( aa \) 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 \( aa \).

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

\[
\begin{bmatrix}
3 & 0 & 1 & 0 & 0 \\
0 & 4 & 0 & 0 & 0 \\
0 & 7 & 5 & 9 & 0 \\
0 & 0 & 0 & 2 & 0 \\
0 & 0 & 0 & 6 & 5 \\
\end{bmatrix}
\]

(2.7.27)

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

<table>
<thead>
<tr>
<th>index ( k )</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
</tr>
</thead>
<tbody>
<tr>
<td>( ija(k) )</td>
<td>7</td>
<td>8</td>
<td>8</td>
<td>10</td>
<td>11</td>
<td>12</td>
<td>3</td>
<td>2</td>
<td>4</td>
<td>5</td>
<td>4</td>
</tr>
<tr>
<td>( aa(k) )</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>0</td>
<td>5</td>
<td>( x )</td>
<td>1</td>
<td>7</td>
<td>9</td>
<td>2</td>
<td>6</td>
</tr>
</tbody>
</table>

(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 FORTRAN do 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.

```fortran
INTEGER n, np, nmax, sa(nmax), ija(nmax)
REAL thresh, a(np, np), sa(np, np)

subroutine sprsin(a, n, np, thresh, nmax, sa, ija)
    
    ! Converts a square matrix $a(1:n,1:n)$ with physical dimension $np$ into row-indexed sparse storage mode. Only elements of $a$ with magnitude $\geq$thresh are retained. Output is in two linear arrays with physical dimension nmax (an input parameter): $sa(1:\star)$ contains array values, indexed by $ija(1:\star)$. The logical sizes of $sa$ and $ija$ on output are both $ija(ija(1)-1)-1$ (see text).

    INTEGER i, j, k
    do 11 j = 1, n
        ! Store diagonal elements.
        sa(j) = a(j, j)
    enddo 11
    ija(1) = n + 2
    ! Index to 1st row off-diagonal element, if any.
    k = n + 1
    do 13 i = 1, n
        ! Loop over rows.
        do 12 j = 1, n
            ! Loop over columns.
            if (abs(a(i, j)).ge.thresh) then
                if (i .ne. j) then
                    ! Store off-diagonal elements and their columns.
                    k = k + 1
                    if (k.gt.nmax) pause 'nmax too small in sprsin'
                    sa(k) = a(i, j)
                    ija(k) = j
                endif
            endif
        enddo 12
        ija(i + 1) = k + 1
        ! As each row is completed, store index to next.
    enddo 13
    return
end
```

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.

```fortran
INTEGER i, j, k
if (ija(1).ne.n+2) pause 'mismatched vector and matrix in sprsax'

doi 12 i = 1, n
    ! Start with diagonal term.
    b(i) = sa(i) * x(i)
    do = k = ija(i), ija(i+1)-1
        ! Loop over off-diagonal terms.
        b(i) = b(i) + sa(k) * x(ija(k))
    enddo = k
enddo 12
return
end
```

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 FORTRAN do 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.

```fortran
INTEGER n, np, nmax, sa(nmax), ija(nmax)
REAL thresh, a(np, np), sa(np, np)

subroutine sprsin(a, n, np, thresh, nmax, sa, ija)
    
    ! Converts a square matrix $a(1:n,1:n)$ with physical dimension $np$ into row-indexed sparse storage mode. Only elements of $a$ with magnitude $\geq$thresh are retained. Output is in two linear arrays with physical dimension nmax (an input parameter): $sa(1:\star)$ contains array values, indexed by $ija(1:\star)$. The logical sizes of $sa$ and $ija$ on output are both $ija(ija(1)-1)-1$ (see text).

    INTEGER i, j, k
    do 11 j = 1, n
        ! Store diagonal elements.
        sa(j) = a(j, j)
    enddo 11
    ija(1) = n + 2
    ! Index to 1st row off-diagonal element, if any.
    k = n + 1
    do 13 i = 1, n
        ! Loop over rows.
        do 12 j = 1, n
            ! Loop over columns.
            if (abs(a(i, j)).ge.thresh) then
                if (i .ne. j) then
                    ! Store off-diagonal elements and their columns.
                    k = k + 1
                    if (k.gt.nmax) pause 'nmax too small in sprsin'
                    sa(k) = a(i, j)
                    ija(k) = j
                endif
            endif
        enddo 12
        ija(i + 1) = k + 1
        ! As each row is completed, store index to next.
    enddo 13
    return
end
```

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.

```fortran
INTEGER i, j, k
if (ija(1).ne.n+2) pause 'mismatched vector and matrix in sprsax'

doi 12 i = 1, n
    ! Start with diagonal term.
    b(i) = sa(i) * x(i)
    do = k = ija(i), ija(i+1)-1
        ! Loop over off-diagonal terms.
        b(i) = b(i) + sa(k) * x(ija(k))
    enddo = k
enddo 12
return
end
```
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.

```fortran
SUBROUTINE sprstx(sa,ija,x,b,n)
INTEGER n,ija(*)
REAL b(n),sa(*),x(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).
INTEGER i,j,k
if (ija(1).ne.n+2) pause 'mismatched vector and matrix in sprstx'
do i=1,n Start with diagonal terms.
   b(i)=sa(i)*x(i)
endo
do i=1,n Loop over off-diagonal terms.
do k=ija(i),ija(i+1)-1
   j=ija(k)
   b(j)=b(j)+sa(k)*x(i)
endo
enddo
END
```

(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.

```fortran
SUBROUTINE sprstp(sa,ija,sb,ijb)
INTEGER ija(*),ijb(*)
REAL sa(*),sb(*)
C USES iindexx
Version of iindexx with all REAL variables changed to INTEGER.
Construct the transpose of a sparse square matrix, from row-index sparse storage arrays sa and ija into arrays sb and ijb.
INTEGER j,jl,jm,jp,ju,k,m,n2,noff,inc,iv
REAL v
n2=ija(1) Linear size of matrix plus 2.
do j=1,n2-2 Diagonal elements.
   sb(j)=sa(j)
endo
call iindexx(ija(n2-1)-ija(1),ija(n2),ijb(n2))
Index all off-diagonal elements by their columns.
jp=0
do k=ija(1),ija(n2-1)-1 Loop over output off-diagonal elements.
m=ija(k)+n2-1 Use index table to store by (former) columns.
   sb(k)=sa(m)
do j=jp+1,ija(m) Fill in the index to any omitted rows.
      ijb(j)=k
endo
jp=ija(m) Use bisection to find which row element m is in and put that into ijb(k).
jl=1
ju=n2-1
5 if (ju-jl.gt.1) then
   jm=(ju+jl)/2
   if(ija(jm).gt.m)then
      ju=jm
   else
      jl=jl+1
5 enddo
```

Permission granted to make one paper copy for your personal use and supply all or part of the material to other people for educational or personal use. Other than that, it is expressly forbidden to copy, republish, or reproduce this or any other copyrighted machine-readable file (including this one) to any server/computer, is strictly prohibited. To order Numerical Recipes books, diskettes, or CDROMs visit our website http://www.nr.com or call 1-800-872-7423 (North America only) or send email to trade@cup.cam.ac.uk (outside North America).
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`, listed in §8.4, except that the latter’s two `REAL` declarations should be changed to `integer`. (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 $A \cdot B^T$, rather than $A \cdot B$. 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
2.7 Sparse Linear Systems

SUBROUTINE sprspm(sa,ija, sb,ijb,sc,ijc)
INTEGER ija(*),ijb(*),ijc(*)
REAL sa(*),sb(*),sc(*)

Matrix multiply $A \cdot B^T$ where $A$ and $B$ are two sparse matrices in row-index storage mode, and $B^T$ is the transpose of $B$. Here, $sa$ and $ija$ store the matrix $A$; $sb$ and $ijb$ store the matrix $B$. 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 $sb$, $ijb$.

INTEGER i,ijma,ijmb,j,m,ma,mb,mbb,mn
REAL sum

if (ija(1).ne.ijb(1).or.ija(1).ne.ijc(1))
* pause 'sprspm sizes do not match'
do 1 i=1,ijc(1)-2
j=i
 Loop over rows.
m=1
 Set up so that first pass through loop does the diagonal component.
mn=ijc(i)
 sum=sa(i)*sb(i)

1 continue
 mb=ijb(j)
 Loop through elements in A's row. Convoluted logic, following, accounts for the various combinations of diagonal and off-diagonal elements.
do 2 ma=ija(ma),ija(ma+1)-1
 if(ijma.eq.j)then
 sum=sum+sa(ma)*sb(j)
 else
 if(mb.lt.ijb(mbb))then
 ijmb=ijb(mbb)
 if(ijmb.eq.i)then
 sum=sum+sa(i)*sb(mb)
 mb=mb+1
 goto 2
 else if(ijmb.gt.ijma)then
 mb=mb+1
 goto 2
 else if(ijmb.eq.ijma)then
 sum=sum+sa(ma)*sb(mb)
 mb=mb+1
 goto 2
 endif
 endif
 endif

m=mn
mn=mn+1
j=ijc(m)
goto 1
continue
2
endo:
do 3 mbb=mb,ijb(j+1)-1
 if(ijbb(mbb).eq.i)then
 sum=sum+sa(i)*sb(mbb)
 endif
 enddo:
do 3 mbb=mb,ijb(j+1)-1
 if(ijbb(mbb).eq.i)then
 sum=sum+sa(i)*sb(mbb)
 endif
 enddo:
do 1
 sc(m)=sum
 sum=0.e0

if(mn.ge.ijc(i+1))goto 3
 m=mn
 mn=mn+1
j=ijc(m)
goto 1
continue
3
endo:
return
END
SUBROUTINE sprstm(sa,ija,sb,ijb,thresh,nmax,sc,ijc)
INTEGER nmax,ija(*),ijb(*),ijc(nmax)
REAL thresh,sa(*),sb(*),sc(nmax)
Matrix multiply $A \cdot B^T$ where $A$ and $B$ are two sparse matrices in row-index storage mode, and $B^T$ is the transpose of $B$. Here, $sa$ and $ija$ store the matrix $A$; $sb$ and $ijb$ store the matrix $B$. 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 $sb$, $ijb$.

INTEGER i,ijma,ijmb,j,k,ma,mb,mbb
REAL sum
if (ija(1).ne.ijb(1)) pause 'sprstm sizes do not match'
k=i=ija(1)
ijc(1)=k
do i=1,ija(1)-2
  Loop over rows of $A$.
do
j=1,ijb(1)-2
  and rows of $B$.
do
if(i.eq.j)then
  Loop through elements in $A$'s row. Convoluted logic, following, accounts for the various combinations of diagonal and off-diagonal elements.
else
  sum=sa(i)*sb(j)
endif
mb=ijb(j)
do ma=ija(ma),ija(ma+1)-1
  Loop through elements in $A$'s row.
do
if(ijma.eq.j)then
  sum=sum+sa(ma)*sb(j)
else
  if(mb.lt.ijb(j+1))then
    ijm=ijb(mb)
    if(ijm.eq.i)then
      sum=sum+sa(i)*sb(mb)
      goto 2
    else if(ijm.lt.ija(ma))then
      mb=mb+1
      goto 2
    else if(ijm.eq.ija(ma))then
      sum=sum+sa(ma)*sb(mb)
      mb=mb+1
      goto 2
    endif
  else
    sum=sum+sa(ma)*sb(mb)
  endif
endif
endo
if(ijb(mbb).eq.i)then
  sum=sum+sa(i)*sb(mbb)
endif
endo
if(i.eq.j)then
  Where to put the answer...
else if(abs(sum).gt.thresh)then
  k=k+1
endif
endo
ijc(i+1)=k
endo
return
END
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

\[
A \cdot x = b
\]  
(2.7.29)

The attractiveness of these methods for large sparse systems is that they reference \( A \) 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 efficiently performed on a properly stored sparse matrix. You, the “owner” of the matrix \( A \), can be asked to provide subroutines that perform these sparse matrix multiplications as efficiently as possible. We, the “grand strategists” supply the general routine, \( \text{linbcg} \), below, that solves the set of linear equations, (2.7.29), using your subroutines.

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

\[
f(x) = \frac{1}{2} x \cdot A \cdot x - b \cdot x
\]  
(2.7.30)

This function is minimized when its gradient

\[
\nabla f = A \cdot x - b
\]  
(2.7.31)

is zero, which is equivalent to (2.7.29). The minimization is carried out by generating a succession of search directions \( p_k \) and improved minimizers \( x_k \). At each stage a quantity \( \alpha_k \) is found that minimizes \( f(x_k + \alpha_k p_k) \), and \( x_{k+1} \) is set equal to the new point \( x_k + \alpha_k p_k \). The \( p_k \) and \( x_k \) are built up in such a way that \( x_{k+1} \) is also the minimizer of \( f \) over the whole vector space of directions already taken, \( \{ p_1, p_2, \ldots, p_k \} \). 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 \textit{biconjugate gradient method}. This method does not, in general, have a simple connection with function minimization. It constructs four sequences of vectors, \( r_k, T_k, p_k, \overline{p}_k, k = 1, 2, \ldots \). You supply the initial vectors \( r_1 \) and \( T_1 \), and set \( p_1 = r_1, \overline{p}_1 = T_1 \). Then you carry out the following recurrence:

\[
\begin{align*}
\alpha_k &= \frac{r_k \cdot r_k}{p_k \cdot A \cdot p_k} \\
r_{k+1} &= r_k - \alpha_k A \cdot p_k \\
T_{k+1} &= T_k - \alpha_k A^T \cdot \overline{p}_k \\
\beta_k &= \frac{T_{k+1} \cdot r_{k+1}}{T_k \cdot r_k} \\
p_{k+1} &= r_{k+1} + \beta_k p_k \\
\overline{p}_{k+1} &= T_{k+1} + \beta_k \overline{p}_k
\end{align*}
\]  
(2.7.32)

This sequence of vectors satisfies the \textit{biorthogonality} condition

\[
T_i \cdot r_j = \overline{p}_i \cdot T_j = 0, \quad j < i
\]  
(2.7.33)

and the \textit{biconjugacy} condition

\[
T_i \cdot A \cdot p_j = p_i \cdot A^T \cdot \overline{p}_j = 0, \quad j < i
\]  
(2.7.34)

There is also a mutual orthogonality,

\[
T_i \cdot \overline{p}_j = r_i \cdot T_j = 0, \quad j < i
\]  
(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 \( m \leq N \) steps with \( \mathbf{r}_{m+1} = \mathbf{r}_{m+1} = 0 \). 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

\[
\mathbf{r}_1 = \mathbf{b} - \mathbf{A} \cdot \mathbf{x}_1
\]  

(2.7.36)

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

\[
\mathbf{x}_{k+1} = \mathbf{x}_k + \alpha_k \mathbf{p}_k
\]

(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 \( \mathbf{p}_1 = \mathbf{r}_1 \). Then \( \mathbf{r}_k = \mathbf{r}_k \) and \( \mathbf{p}_k = \mathbf{p}_k \) 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 \texttt{linbcon} 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 \( \mathbf{r}_1 = \mathbf{A} \cdot \mathbf{r}_1 \) instead of \( \mathbf{r}_1 = \mathbf{r}_1 \). In this case \( \mathbf{r}_k = \mathbf{A} \cdot \mathbf{r}_k \) and \( \mathbf{p}_k = \mathbf{A} \cdot \mathbf{p}_k \) 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

\[
\Phi(x) = \frac{1}{2} \mathbf{r} \cdot \mathbf{r} = \frac{1}{2} |\mathbf{A} \cdot \mathbf{x} - \mathbf{b}|^2
\]  

(2.7.38)

where the successive iterates \( \mathbf{x}_k \) minimize \( \Phi \) 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

\[
\nabla \Phi(x) = \mathbf{A}^T \cdot (\mathbf{A} \cdot \mathbf{x} - \mathbf{b})
\]  

(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

\[
(\mathbf{A}^T \cdot \mathbf{A}) \cdot \mathbf{x} = \mathbf{A}^T \cdot \mathbf{b}
\]  

(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),

\[
(\mathbf{A}^{-1} \cdot \mathbf{A}) \cdot \mathbf{x} = \widetilde{\mathbf{A}}^{-1} \cdot \mathbf{b}
\]

(2.7.41)

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

For efficient implementation, the PBCG algorithm introduces an additional set of vectors $z_k$ and $\tilde{z}_k$ defined by
\begin{equation}
\tilde{A} \cdot z_k = r_k \quad \text{and} \quad \tilde{A}^T \cdot z_k = \tilde{r}_k
\end{equation}
and modifies the definitions of $\alpha_k$, $\beta_k$, $p_k$, and $\tilde{p}_k$ in equation (2.7.32):
\begin{align}
\alpha_k &= \frac{z_k \cdot r_k}{p_k \cdot A \cdot p_k} \\
\beta_k &= \frac{r_{k+1} \cdot z_{k+1}}{r_k \cdot z_k}
\end{align}
\begin{align}
p_{k+1} &= z_{k+1} + \beta_k p_k \\
\tilde{p}_{k+1} &= \tilde{z}_{k+1} + \beta_k \tilde{p}_k
\end{align}
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 $A$, then use the diagonal part of $A$, 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 \cite{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 $|A \cdot x - b|/|b|$ is less than the input quantity $tol$. If $itol=2$, the required criterion is
\begin{equation}
|\tilde{A}^{-1} \cdot (A \cdot x - b)|/|\tilde{A}^{-1} \cdot b| < tol
\end{equation}
If $itol=3$, the routine uses its own estimate of the error in $x$, and requires its magnitude, divided by the magnitude of $x$, 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 $x$ are used instead of the vector magnitude (that is, the $L_\infty$ 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.

SUBROUTINE linbcg(n,b,x,itol,itmax,iter,err)
INTEGER iter, itmax, itol, n, NMAX
DOUBLE PRECISION err, to1, b(*), x(*), EPS
PARAMETER (NMAX=1024, EPS=1.d-14)
USES atimes, asolve, norm
Solves $A \cdot x = b$ 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 $A$ is referenced only through the user-supplied routines atimes, which computes the product of either $A$ or its transpose on a vector; and asolve, which solves $\tilde{A} \cdot x = b$ or $\tilde{A}^T \cdot x = b$ for some preconditioner matrix $\tilde{A}$ (possibly the trivial diagonal part of $A$).

INTEGER j

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

SUBROUTINE linbcg(n,b,x,itol,itmax,iter,err)
INTEGER iter, itmax, itol, n, NMAX
DOUBLE PRECISION err, to1, b(*), x(*), EPS
PARAMETER (NMAX=1024, EPS=1.d-14)
USES atimes, asolve, norm
Solves $A \cdot x = b$ 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 $A$ is referenced only through the user-supplied routines atimes, which computes the product of either $A$ or its transpose on a vector; and asolve, which solves $\tilde{A} \cdot x = b$ or $\tilde{A}^T \cdot x = b$ for some preconditioner matrix $\tilde{A}$ (possibly the trivial diagonal part of $A$).

INTEGER j
DOUBLE PRECISION ak, akden, bk, bkden, bknum, bnrm, dxnrm,
* xnrm, zm1nrm, znrm, p(NMAX), pp(NMAX), r(NMAX), rr(NMAX),
* z(NMAX), zz(NMAX), snrm

iter=0

Calculate initial residual.
Input to atimes is (1:n), output is r(1:n);
the final 0 indicates that the matrix (not
its transpose) is to be used.

iter=0

call atimes(n, x, r, 0)
do 11 j=1, n
   r(j)=b(j)-r(j)
   rr(j)=r(j)
enddo 11

C call atimes(n, r, rr, 0)
if(itol.eq.1) then
   bnrm=snrm(n, b, itol)
call asolve(n, r, z, 0)
else if(itol.eq.2) then
   call asolve(n, b, z, 0)
   bnrm=snrm(n, z, itol)
call asolve(n, r, z, 0)
else if(itol.eq.3.or.itol.eq.4) then
   call asolve(n, b, z, 0)
   bnrm=snrm(n, z, itol)
call asolve(n, r, z, 0)
else
   pause 'illegal itol in linbcg'
endif

Uncomment this line to get the "minimum
residual" variant of the algorithm.
Input to asolve is (1:n), output is z(1:n);
the final 0 indicates that the matrix \( A \) (not
its transpose) is to be used.

100 if (iter.le.itmax) then
   iter=iter+1
   call asolve(n, rr, zz, 1)

   Final 1 indicates use of transpose matrix \( A^T \).

   Calculate coefficient bx and direction vectors
   p and pp.

   bknum=0.d0
   do 12 j=1, n
      bknum=bknum+z(j)*rr(j)
   enddo 12

   if(itol.eq.1) then
      do 13 j=1, n
         p(j)=z(j)
         pp(j)=zz(j)
      enddo 13
   else
      bk=bknum/bkden
      do 14 j=1, n
         p(j)=bk*p(j)+z(j)
         pp(j)=bk*pp(j)+zz(j)
      enddo 14
   endif

   Calculate coefficient ak, new iterate x,
   and new residuals r and rr.

   akden=0.d0
   do 15 j=1, n
      akden=akden+z(j)*pp(j)
   enddo 15
   ak=aknum/akden

   call atimes(n, p, z, 0)
   akden=0.d0
   do 16 j=1, n
      akden=akden+z(j)*pp(j)
   enddo 16
   ak=aknum/akden
   call atimes(n, pp, zz, 1)
   do 16 j=1, n
      x(j)=x(j)+ak*p(j)
      r(j)=r(j)-ak*z(j)
      rr(j)=rr(j)-ak*zz(j)
   enddo 16

   Solve \( A^T \cdot z = r \) and check stopping criterion.

   if(itol.eq.1) then
      err=snrm(n, r, itol)/bnrm
   else if(itol.eq.2) then
      err=snrm(n, z, itol)/bnrm
   else if(itol.eq.3.or.itol.eq.4) then
      zm1nrm=znrm
   endif
znrm=snrm(n,z,itol)
if(abs(zm1nrm-znrm).gt.EPS*znrm) then
  dxnrm=abs(ak)*snrm(n,p,itol)
  err=znrm/abs(zm1nrm-znrm)*dxnrm
else
  err=znrm/bnrm
endif
xnrm=snrm(n,x,itol)
if(err.le.0.5d0*xnrm) then
  err=err/xnrm
else
  err=znrm/bnrm
endif
write (*,*) ' iter=',iter,' err=',err
if(err.gt.tol) goto 100
endif
return
END

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

FUNCTION snrm(n,sx,itol)
INTEGER n,itol,i,isamax
DOUBLE PRECISION sx(n),snrm
Compute one of two norms for a vector sx(1:n), as signaled by itol. Used by linbcg.
if (itol.le.3)then
  do: : i=1,n
     snrm=snrm+sx(i)**2
  enddo
  snrm=sqrt(snrm)
else
  isamax=1
  do: : i=1,n
     if(abs(sx(i)).gt.abs(sx(isamax))) isamax=i
  enddo
  snrm=abs(sx(isamax))
endif
return
END

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.

SUBROUTINE atimes(n,x,r,itrnsp)
INTEGER n,itrnsp,ija,NMAX
DOUBLE PRECISION x(n),r(n),sa
PARAMETER (NMAX=1000)
COMMON /mat/ sa(NMAX),ija(NMAX)
USES dsprsax,dsprstx
C USES sprsax, sprstx
if (itrnsp.eq.0) then
  call dsprsax(sa,ija,x,r,n)
else
  call dsprstx(sa,ija,x,r,n)
endif
return
END

The matrix is stored somewhere.

DOUBLE PRECISION versions of sprsax and sprstx.
SUBROUTINE asolve(n,b,x,itrnsp)
INTEGER n,itrnsp,i,ija,NMAX
DOUBLE PRECISION x(n),b(n),sa
PARAMETER (NMAX=1000)
COMMON /mat/ sa(NMAX),ija(NMAX)
do i=1,n
   x(i)=b(i)/sa(i)
enddo
return
END

CITED REFERENCES AND FURTHER READING:
Press), Chapter 1.3 (by J.K. Reid). [2]
[4]
Eisenstat, S.C., Gursky, M.C., Schultz, M.H., and Sherman, A.H. 1977, Yale Sparse Matrix Pack-
age, 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]
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]
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]
pp. 856–869. [15]
Bunch, J.R., and Rose, D.J. (eds.) 1976, Sparse Matrix Computations (New York: Academic
Press).
SIAM).

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