GENERAL SPARSE ELIMINATION REQUIRES NO PERMANENT INTEGER STORAGE *

RANDOLPH E. BANK† AND R.KENT SMITH‡

Abstract. General sparse elimination is designed to take maximum advantage of the sparsity of an $N \times N$ matrix A. Only the nonzeros of A are stored, along with some extra integer overhead to identify the nonzero matrix elements. This extra integer storage may be avoided for the triangular factors generated by an LDU decomposition, generally without increasing the order of complexity. In addition to permanent storage for the nonzero elements of the factors, our procedure requires at most $5N$ temporary integer storage.

Key words. sparse Gaussian elimination, sparse matrix

AMS subject classifications. 65F05, 65N20

1. Introduction.

We consider the solution of

$$Ax = b \tag{1.1}$$

where A is large sparse $N \times N$ nonsingular matrix with no special structure other than a symmetric sparsity pattern. Such problems can be treated by general sparse Gaussian elimination methods [1] - [6]. In such a scheme, one finds a permutation matrix P and computes the decomposition

$$PAP^t = LDU$$

where L is unit lower triangular, U is unit upper triangular, and D is diagonal. If $A^t = A$ then $L^t = U$. The solution of the linear system is then computed via

$$\begin{aligned} Lw &= Pb \\ Dy &= w \\ Uz &= y \\ x &= P^t z \end{aligned}$$

As is common in such investigations, we assume the factorization exists for any permutation matrix P. This will be true, for example, if the symmetric part of A is positive definite.

Traditionally Gaussian elimination algorithms have been partitioned into four distinct phases:

1. Ordering (compute P)
2. Symbolic factorization
3. Numerical factorization (compute LDU)
4. Forward and backward substitution (compute x)

*Received by the editord May 5, 1986; accepted for publication (in revised form September 22, 1986

†Department of Mathematics, University of California at San Diego, La Jolla, California 92093. The work of this author was partly supported by the Office of Naval Research under contract N00014-82K-0197.

‡AT&T Bell Laboratories, Murray Hill, New Jersey 07974.

Several efficient hueristic algorithms are available for ordering general sparse matrices, for example, the minimum degree algorithm [4, 5] Since our procedures are independent of the choice of P, we assume for convenience that $P = I$, or equivalently, that A has been reordered to reflect the appropriate choice of P.

To take maximum advantage of the sparsity, the matrix A is represented computationally by a linear array A containing only nonzero elements. In order to access this data, various pointers and row/column indices are stored to establish the correspondence between array entries and matrix entries.

In sparse Gaussian elimination schemes, the matrices LDU are also stored in a linear array, U, containing only the nonzero elements. Generally, this array is much larger than A, due to the "fillin" which occurs during the elimination process. The purpose of symbolic factorization is to compute an auxiliary integer data structure for U, similar to that used for A, which allows the numeric factorization and backsolves to be carried out statically. For the simplest schemes, the amount of integer storage required is of the same order as the size of U, ($|U|$) although compression schemes can result in significant reductions [2, 3, 4].

This paper presents an algorithm where the numeric factorization and solution steps can be carried out without an additional permanent integer data structure. The relevant information is computed as needed, thus eliminating the symbolic factorization step. In addition to the storage required for A, U, x, b etc, the numeric factorization algorithm uses 5 integer vectors of total length $5N$, and the solution phase requires 3 integer vectors of total length $3N$. Using the ideas of Rose, Tarjan, and Lueker [6] the fillin is computed in an optimal $O(N + |U|)$ time, which is usually far less than the cost of the numerical factorization [4, 5]. The overall order of complexity of the numerical operations in this procedure is the same as for other general sparse schemes. While some aspects of the non-numerical complexity remain unclear, it appears to be of the same order as the numerical complexity in the practical problems we have solved.

The remainder of the paper is organized as follows. The data structure used to store A is described in Section 2. This structure is a variation of the data structure described in [1] and is based on the data structure used in the symmetric codes of the Yale Sparse Matrix Package [2, 3]. In Section 3, we briefly review the relevant graph theoretic results upon which our procedure is based. This is not self-contained and assumes a basic background of the graph theoretic model of Gaussian elimination [4, 5]. We then present our procedures and analyze their correctness and complexity. The appendix contains a prototype FORTRAN implementation of our procedures.

2. A Sparse Matrix Data Structure. Let A be an $N \times N$ matrix with elements a_{ij}. We assume A is sparse with a symmetric sparsity structure; that is, both a_{ij} and a_{ji} are to be treated as non-zero elements (i.e. stored and processed) if $|a_{ij}| + |a_{ji}| > 0$. We further assume that the diagonal entries a_{ii} are non-zero.

In our scheme the nonzero entries of A are stored in a linear array A, and accessed through an integer array JA. Let η_i be the number of non-zeroes in the strict upper triangular part of column i, and set $\eta = \sum_{i=1}^{N} \eta_i$. The array JA is of length $N+1+\eta$ and the array A is of length $N + 1 + \eta$ if $A^t = A$. If $A^t \neq A$, then the array A is of length $N+1+2\eta$. The entries of $JA(i)$ $1 \leq i \leq N+1$ are pointers defined as follows:

$$JA(1) = N + 2$$
$$JA(i+1) = JA(i) + \eta_i, \quad 1 \leq i \leq N$$

The locations $JA(i)$ to $JA(i+1) - 1$ contain the η_i row indices corresponding to the

column i in the strictly upper triangular matrix.

In a similar manner, the array A is defined as follows:

$$\begin{aligned} A(i) &= a_{ii} \\ A(N+1) &\quad \text{is arbitrary} \\ A(k) &= a_{ji}, \quad j = JA(k), \quad JA(i) \le k \le JA(i+1) - 1 \end{aligned}$$

If $A^t \neq A$, then

$$A(k+\eta) = a_{ij}, \quad j = JA(k), \quad JA(i) \le k \le JA(i+1) - 1$$

In words, the A array stores the diagonal first, followed by the strict upper triangle stored column-wise. If $A^t \neq A$, then this is followed by the strict lower triangle stored row-wise. Since A is structurally symmetric, the row indexes for the upper triangle are identical to the column indexes for the lower triangle, and hence need not be duplicated in storage.

As an example, let

$$A = \begin{pmatrix} a_{11} & a_{12} & a_{13} & 0 & 0 \\ a_{21} & a_{22} & 0 & a_{24} & 0 \\ a_{31} & 0 & a_{33} & a_{34} & a_{35} \\ 0 & a_{42} & a_{43} & a_{44} & 0 \\ 0 & 0 & a_{53} & 0 & a_{55} \end{pmatrix}$$

Then

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
JA	7	7	8	9	11	12	1	1	2	3	3					
A	a_{11}	a_{22}	a_{33}	a_{44}	a_{55}		a_{12}	a_{13}	a_{24}	a_{34}	a_{35}	a_{21}	a_{31}	a_{42}	a_{43}	a_{53}
	Diagonal						Upper Triangle					Lower Triangle				

This data structure is similar to the data structure described in [1], except that the roles of the rows and columns have been reversed. The data structure has a number of interesting features which we briefly summarize below.

1. For symmetric matrices this data structure saves N integer storage locations compared to the data structure used by the symmetric Yale Sparse Matrix codes [2, 3], which store all entries for a given row in consecutive locations in A. For nonsymmetric matrices, about half of the integer storage is saved.
2. For many matrix calculations a single indirect address computation in JA can be used to process both a lower and a upper triangular element in A. This loop unrolling technique can save overhead costs on vector machines such at the Cray. For example, the following procedure[1] computes $y = Ax$:

```
      procedure MULT(N, A, x, y)
1.    for i ← 1 to N
2.          y(i) ← A(i)x(i)
3.    lshift ← JA(N + 1) − JA(1)
4.    ushift ← 0
5.    for i ← 1 to N
```

[1] In this paper, we adopt the convention that zero trip loops are not executed.

```
6.          for k ← JA(i) to JA(i+1) − 1
7.              j ← JA(k)
8.              y(i) ← y(i) + A(k + lshift)x(j)
9.              y(j) ← y(j) + A(k + ushift)x(i)
        end MULT
```

For symmetric matrices, set $lshift \leftarrow 0$ in line 3. Also, $y = A^t\ x$ may be readily computed by setting, $lshift \leftarrow 0, ushift \leftarrow JA(N+1) - JA(1)$.

3. When solving 1.1 by iterative methods like SSOR, independent access to the diagonal, lower triangular and upper triangular parts of A is required. These components are easily accessible in our data structure but may be somewhat more complicated and expensive with other sparse storage schemes.
4. It is often convenient to view A as a bordered matrix of the form

$$A = \begin{pmatrix} \bar{A} & c \\ r^t & d \end{pmatrix}$$

where $\bar{A}$ is an $(N-1) \times (N-1)$ matrix, c and r are $N-1$ vectors and d is a scalar. In some applications $\bar{A}$ is "fixed" while c, r and d may change, requiring an update of the numeric factorization. Since the nonzeros of c and r are stored in consecutive locations in U, the factorization is easy to update.

Our data structure for storing the factored matrix U, is analogous to the real array A. The diagonal elements of D are stored first, followed by the strictly upper triangular part of U, stored column-wise. For nonsymmetric problems, this is followed by the strictly lower triangular part of L, stored row-wise. The diagonal elements of U and L are unity and therefore not stored. As we will see in the next section, an analogue for the complete integer data structure JA is not required.

3. A Sparse Gaussian Elimination Procedure. Let $G = (V, E)$ denote the ordered, undirected graph corresponding to the matrix A. V is the vertex set $\{v_i\}_{i=1}^N$, and E is the edge set with $e_{ij} (= e_{ji}) \in E$ if and only if $|a_{ij}| + |a_{ji}| \neq 0$, $i \neq j$. For each vertex $v_i \in V$, $adj_G(v_i) = \{k | e_{ik} \in E\}$ is the index set of vertices adjacent to v_i in G.

Gaussian elimination on A corresponds to a sequence of elimination graphs G_i, $0 \leq i \leq N-1$, defined inductively as follows (with $G_0 \equiv G$): to create G_i from G_{i-1}, remove v_i and all its incident edges from G_{i-1}, and add new edges as required to pairwise connect all vertices in $adj_{G_{i-1}}(v_i)$ (see [4, 5]).

Let F denote the set of edges added during the elimination process, and let $G' = (V, E \cup F)$. Gaussian elimination applied to G' produces no additional fillin edges. Thus the object of symbolic factorization in this context is to compute the set $E \cup F$.

Following Rose, Tarjan and Lueker [6], we define the function $m(i)$, by

$$m(i) = \min\{k > i | k \in adj_{G'}(v_i)\}, \quad 1 \leq i \leq N-1 \tag{3.1}$$

Note that $m(i)$ can also be defined as $\min\{k \in adj_{G_{i-1}}(v_i)\}$. The following characterization of the set $E \cup F$, although not explicitly given in [6], is implicit in that work (see lemma 5 of [6] and also [3, 4]).

THEOREM 3.1. *Let $G = (V, E)$ be the ordered, undirected graph corresponding to A, and let F denote the set of fillin edges generated by Gaussian elimination. Then $e_{ij} \in E \cup F$, $i < j$, if and only if*

(i) $e_{ij} \in E$ or

(ii) there exists a sequence $(k_1, k_2 ... k_p)$ *such that*

(a) $k_1 = \ell$, $k_p = j$, $e_{\ell j} \in E$

(b) $i = k_q$ *for some* $2 \le q \le p-1$

(c) $k_q = m(k_{q-1})$, $2 \le q \le p$

Theorem 3.1 and its variations form the basis of the optimal algorithms for computing $E \cup F$; see [3, 4, 6] for several such algorithms.

Another optimal order algorithm for computing $|U|$, consistent with the storage scheme described in Section 2, is presented below. The algorithm uses 3 integer arrays – M of length $N-1$, $LIST$ of length N, and JU of length $N+1$. The array JU is not necessary for computing $|E \cup F|$, but plays an important role in the factorization and solution steps based upon

```
          procedure FILLIN(N, JA, JU, M, LIST)
1.        M(i) ← 0, 1 ≤ i ≤ N − 1
2.        LIST(i) ← 0, 1 ≤ i ≤ N
3.        JU(1) ← N + 1
4.        for i ← 1 to N
5.              length ← 0
6.              LIST(i) ← i
7.              for j ← JA(i) to JA(i + 1) − 1
8.                    k ← JA(j)
9.                    while LIST(k) = 0 do
10.                         LIST(k) ← LIST(i)
12.                         LIST(i) ← k
12.                         length ← length + 1
13.                         If M(k) = 0, then M(k) ← i
14.                         k ← M(k)
15.                   JU(i + 1) ← JU(i) + length
16.                   k ← i
17.                   for j ← 1 to length + 1
18.                         ksave ← k
19.                         k ← LIST(k)
20.                         LIST(ksave) ← 0
      end FILLIN
```

THEOREM 3.2. *Procedure FILLIN correctly computes* $|E \cup F|$ *in* $O(N + |E \cup F|)$ *time.*

Proof. We show by induction that $FILLIN$ correctly computes the set $S_i = \{k | m(k) = i\}$; thus $m(k)$ is computed correctly, and the correctness of the overall procedure then follows from Theorem 3.1. The case $i = 1$ is trivial, we assume the sets S_k, $1 \le k \le i-1$ have been correctly computed.

In processing column i, we start with the seed index, $k \leftarrow JA(j)$, line 8, corresponding to edges in E, and simply begin to apply Theorem 3.1 generate those edges in F for this column. During this process we may encounter indices for which $m(k)$ has been previously defined. If an undefined value of $m(k)$ is found, line 13, then we set $m(k) = i$; if $m(k) < i$, it would have been previously defined, and if $m(k) > i$, it would violate the definition of m (equation (3.1)). Thus S_i is correctly defined.

Each edge found is added to the linked list $LIST$, so edges are counted exactly once. By setting $LIST(i) \leftarrow i$ on line 6, no edges e_{ki} with $k > i$ will be found. Thus $LIST$ contains the row indices of the nonzero elements in column i of U (row i in L) on completion of the loop beginning at line 7. The "cleanup" loop (lines 16-20)

merely reinitializes the $LIST$ array. The array JU contains pointers to U which are analogous to the first $N+1$ entries in JA.

As for the complexity, the two inner loops (lines 7-14 and 17-20) require $O(1)$ work for each found edge, and each edge in $E \cup F$ is found exactly once. The remaining computations are obviously $O(N)$. □

4. Applications. Procedure $FILLIN$ forms the basis for the numerical factorization and solution steps in Gaussian elimination. The need for a separate symbolic factorization routine is eliminated by embedding floating point matrix operations involving U in the procedure. In this section, we consider several variations on the procedure $FILLIN$.

4.1. Solution Step. First consider the solution phase from a given LDU factorization. The strategy is to construct the relevant integer arrays while solving $Ly = b$. These arrays may then be used in the solution of the triangular system, $Ux = z$. Since the order of the elements in the triangular systems is arbitrary, the elements in U are processed in the order in which they were created in procedure $FILLIN$. To solve $Ly = b$ for symmetric matrices, the cleanup loop (lines 16-20 in $FILLIN$) could be replaced by:

```
16.   k ← i
17.   sum ← 0
18.   for j ← JU(i) to JU(i+1) − 1
19.         ksave ← k
20.         k ← LIST(k)
21.         LIST(ksave) ← 0
22.         sum ← sum + U(j)y(k)
23.   y(i) ← b(i) − sum
24.   LIST(k) ← 0
```

The solution of $Ux = z$ follows the same general pattern except that the columns are processed in the reverse order. This presents no difficulty since the arrays M and JU have been previously computed. Finally, note that by adding the lshift and ushift parameters as in procedure $MULT$, the same code can treat symmetric problems, and nonsymmetric problems for A and A^t.

4.2. Numeric Factorization. We now consider the prototype elimination step in the numerical factorization procedure. Let $\bar{A}$ denote the upper left $k \times k$ principal submatrix of A and $\bar{L}\bar{D}\bar{U}$ denote its factorization. At the $k+1$-st step we inductively factor the upper left $(k+1) \times (k+1)$ submatrix of A as

$$\begin{pmatrix} \bar{A} & c \\ r^t & d \end{pmatrix} = \begin{pmatrix} \bar{L} & 0 \\ \bar{r}^t & 1 \end{pmatrix} \begin{pmatrix} \bar{D} & 0 \\ 0 & \bar{d} \end{pmatrix} \begin{pmatrix} \bar{U} & \bar{c} \\ 0 & 1 \end{pmatrix}$$

where

$$\begin{aligned} \bar{L}\bar{D}\bar{c} &= c, \\ \bar{U}^t\bar{D}\bar{r} &= r, \\ \bar{d} &= d - \bar{r}^t\bar{D}\bar{c} \end{aligned}$$

Since $\bar{L}\bar{D}\bar{U}$ have previously been computed, this step requires the solution of at most two lower triangular systems of equations. The vectors $\bar{c}$ and $\bar{r}$ are sparse, say with p nonzeros. Note that the nonzeros of c and r are a subset of those in $\bar{c}$ and $\bar{r}$. Thus the linear systems can be reduced to $p \times p$ lower triangular systems by deleting rows

and columns corresponding to zero elements in $\bar{c}$ and $\bar{r}$. These $p \times p$ subsystems are solved by forward substitution, taking advantage of any structural zeros that remain. The inner product used to compute $\bar{d}$ is similarly reduced to length p, and the multiplication by $\bar{D}$ is easy to avoid using intermediate values generated in the computation of $\bar{c}$ and $\bar{r}$.

To take advantage of the sparsity of these systems, the column indices associated with $\bar{r}$ and $\bar{c}$ are first generated by $FILLIN$. The resulting list of indices, $ORDER$, must be ordered in increasing k for the solution of the triangular systems. The sorting procedure is quite simple since the lists of indices corresponding to each seed index, $k \leftarrow JA(i)$, are already sorted. The problem is thus one of merging a few ordered lists. Furthermore, once an element common to both lists is encountered, the remainder of both lists is identical (by Theorem 3.1) and the merging process can be terminated. If $ORDER$ contains p entries, the cost of this ordering is certainly bounded by $O(p \log p)$, the cost of sorting a set of p completely unordered indices. This cost is usually much less than the cost of the numerical computations. Since $LIST$ is unordered with respect to the nonzeros in U, an additional vector, $INDEX$, which points to the location in U of each element is required.

If the seed indices in JA are ordered by decreasing size, then the sorting at this stage can be avoided entirely. To do this, one should process the seed indices in JA in reverse order (i.e., by increasing size) and order the indices as they are generated. This will not in general order all the indices by increasing size, but it will implicitly produce a permutation under whose application the lower triangular matrix remains lower triangular, and the forward substitution can proceed in the usual fashion.

The second step in the factorization procedure is to construct a linked list , $LIST$, of indices for each $k < i$ on the list $ORDER$. In this process, only indices which lie in the intersection of $ORDER$ are needed. At the moment, we have no simple way to avoid the computation of the entire linked list for each column k, and intersecting it with that for column i. The intersection procedure itself is very efficient and can be done in $O(1)$ work per entry of $LIST$. On typical finite element pde problems, between 50-60% of the row indices computed in this step are discarded. Since for this class of problems, the percentage seems to remain bounded independent of N, the order of complexity of the numerical factorization is unimpaired. However, since this computation contributes to the highest order term, it has significant impact on the overall performance of the algorithm. At present, we have no meaningful theoretical bound on the non-numerical complexity of this step, and this remains the weakest aspect of our procedure.

The numerical factorization requires 5 integer arrays – M, $LIST$ and JU as in $FILLIN$, plus $ORDER$ and $INDEX$ as described above. Both these arrays are of length N, so that a total of $5N$ temporary integer storage is used.

4.3. Incomplete Factorization. The procedure $FILLIN$ can be adapted in a simple way to compute storage for incomplete LDU factorizations to be used as preconditioned iterative methods for solving (1.1). In particular, an incomplete LDU factorization can be computed simply by limiting the length of the sequences generated by the seed indices. For example, if the length is bounded by one, only the seed indices themselves are allowed, and the sparsity pattern is the same as for A. If the length is bounded by two, k and $m(k)$ are allowed for each seed index k. This corresponds to allowing one extra level of fillin in the incomplete factorization. In general, if the length is bounded by p, then a maximum of $p-1$ extra levels of fillin are allowed. If there is a constraint on total storage, this procedure could be used in an iterative

fashion to determine the maximum fillin level, p, that could be accommodated. All the results of this section carry over transparently to the case of incomplete factorizations.

4.4. Symbolic Factorization. The algorithms presented in the proceeding sections are designed to minimize the storage requirements of sparse Gaussian elimination methods. However, it may be desirable to trade space for time by allowing some permanent integer storage. One must be cautious in this approach, since the advantage of our scheme may be diluted if this storage is comparable to standard compressed storage schemes [3].

One possibility is to define an integer data structure JL, of length $N+1+\eta$, (the same length as JA) as follows:

$JL(i) = m(i)$, $1 \le i \le N-1$,
$JL(k-1) =$ starting index in U of seed $JA(k)$, $N+2 \le k \le N+1+\eta$.
$JL(N+1+\eta) = 1+$ ending index in U of seed $JA(N+1+\eta)$.

Once generated, JL eliminates the need for the M, $LIST$, and JU arrays in subsequent calculations, and its creation can be regarded as a symbolic factorization. In many problems, there is little or no storage penalty except that now the storage is permanent rather than temporary. For 5 point finite difference matrices, JL is about $3N$ in length, as is the combined storage of M, $LIST$ and JU. In the numerical factorization, only one additional array of length N, $INDEX$, is required. The use of the JL array is illustrated in the algorithm for solving $Ly = b$

```
          procedure LSOLVE(N, JA, JL, U, y, b)
1.        for i ← 1 to N
2.              sum ← 0
3.              for j ← JA(i) to JA(i+1) − 1
4.                    k ← JA(j)
5.                    for l ← JL(j−1) to JL(j) − 1
6.                          sum ← sum + U(l)y(k)
7.                          k ← JL(k)
8.                    y(i) ← b(i) − sum
          end LSOLVE
```

The order of complexity of $LSOLVE$ is obviously the same as in the previous example, but the non-numerical overhead is reduced. At present, the compromise between storage and computational efficiency is still an open question and deserves further study.

Appendix: Sample FORTRAN program. This appendix contains three sample FORTRAN routines for sparse Gaussian elimination. To illustrate the algorithms presented in this paper, we assume the arrays JA and A have been reordered to reflect an appropriate choice of the permutation matrix P. Furthermore, the row indices for each column in JA are given in descending order, to avoid sorting in the numerical factorization step. Subroutine $FILLIN$ computes the JL array as described in section 4.4. Subroutine FACTOR computes the numerical factorization. Subroutine SOLVE performs the forward and backward substitution.

```
      subroutine FILLIN (n, ja, jl, list)
         integer ja(*), jl(*), list(*), uptr
c
c     compute jl
c
c     input: n, ja
```

```
c       output: jl
c       workspace: list(n)
c
c       remarks:
c
c           1. for convenience, jl(n) is the size of the strict
c              upper triangular part of the factored matrix
c
c           2. the row indices for each column in ja are assumed
c              to be in decreasing order
c
c       initialize
c
        do 10 i = 1, n
           jl(i) = 0
   10      list(i) = 0
c
c       the main loop
c
        jl(n+1) = n + 2
        do 50 i = 1, n
           if (ja(i) .ge. ja(i+1))  go to 50
c
c       loop over seed indices in decreasing order
c
           list(i) = i
           length = 1
           do 30 iseed = ja(i), ja(i+1)-1
              k = ja(iseed)
              uptr = jl(iseed-1)
   20         list(k) = list(i)
              list(i) = k
              uptr = uptr + 1
              length = length + 1
              if (jl(k) .eq. 0) jl(k) = i
              k = jl(k)
              if (list(k) .eq. 0) go to 20
   30         jl(iseed) = uptr
c
c       clean up loop for list array
c
           k = i
           do 40 j = 1, length
              ksave = k
              k = list(k)
   40         list(ksave) = 0
   50   continue
c
c       compute size of upper triangle
```

```
c
        lenjl = n + 1 + ja(n+1) - ja(1)
        jl(n) = jl(lenjl) - jl(n+1)
        return
        end

        subroutine FACTOR (n, isym, ja, a, jl, u, index, ierr)
           integer ja(*), jl(*), index(*), ashift, ushift
           real a(*), u(*), dsum, usum, lsum
c
c       compute u
c
c       input: n, isym, ja, a, jl
c       output: u, ierr
c       workspace: index(n)
c
c       remarks:
c
c           1. isym = 0 means nonsymmetric storage for a and u
c              isym = 1 means symmetric storage for a and u
c
c           2. ierr = 0 is a normal return
c              ierr = i means the i-th diagonal element of the
c                       factored matrix was zero
c
c           3. for symmetric matrices, the computation of lsum in the
c              do 60 loop, and the computation of u(indexk+ushift) in
c              the do 80 loop are redundant
c
        ierr = 0
c
c       compute offsets for a and u
c
        if (isym .eq. 1) then
           ashift = 0
           ushift = 0
           lenu = n + 1 + jl(n)
        else
           ashift = ja(n+1) - ja(1)
           ushift = jl(n)
           lenu = n + 1 + 2 * jl(n)
        end if
c
c       initialize
c
        do 10 i = 1, lenu
   10      u(i) = 0.0e0
        do 20 i = 1, n
   20      index(i) = 0
```

```
c
c       the main loop
c
        do 100 i = 1, n
           dsum = 0.0e0
           if (ja(i) .ge. ja(i+1))  go to 90
c
c       loop over seed indices for column i in increasing order
c
           do 60 iseed = ja(i+1)-1, ja(i), -1
              k = ja(iseed)
c
c       move off diagonal entries from a to u
c
              u(jl(iseed-1)) = a(iseed)
              u(jl(iseed-1)+ushift) = a(iseed+ashift)
c
              do 50 indexk = jl(iseed-1), jl(iseed)-1
                 index(k) = indexk
                 if (ja(k) .ge. ja(k+1))  go to 50
                 lsum = u(indexk+ushift)
                 usum = u(indexk)
c
c       loop over seed indices for column k in decreasing order
c
                 do 40 kseed = ja(k), ja(k+1)-1
                    j = ja(kseed)
                    do 30 indexj = jl(kseed-1), jl(kseed)-1
c
c       test for intersection between columns i and k
c
                       if (index(j) .gt. 0)  then
                          lsum = lsum - u(indexj) * u(index(j)+ushift)
                          usum = usum - u(indexj+ushift) * u(index(j))
                       end if
c
   30                  j = jl(j)
   40              continue
                   u(indexk+ushift) = lsum
                   u(indexk) = usum
   50              k = jl(k)
   60         continue
c
c        clean up loop for index array, compute diagonal
c
           do 80 iseed = ja(i), ja(i+1)-1
              k = ja(iseed)
              do 70 indexk = jl(iseed-1), jl(iseed)-1
c
```

```
                  usave = u(indexk+ushift)
                  u(indexk) = u(indexk) * u(k)
                  u(indexk+ushift) = usave * u(k)
                  dsum = dsum + u(indexk) * usave
                  index(k) = 0
c
   70             k = jl(k)
   80         continue
c
   90         u(i) = a(i) - dsum
              if (u(i) .eq. 0.0e0)  go to 200
              u(i) = 1.0e0 / u(i)
  100     continue
          return
c
c         error return
c
  200     ierr = i
          return
          end

          subroutine SOLVE (n, isym, ja, jl, u, x, b)
              integer ja(*), jl(*), ushift, lshift
              real u(*), x(*), b(*)
c
c         compute the solution x of a * x = b
c
c         input: n, isym, ja, jl, u, b
c         output: x
c
c         remark: isym = 1 means symmetric storage is used for u
c                 isym = 0 means nonsymmetric storage is used for u
c                 isym = -1 means nonsymmetric storage is used, and the
c                          problem is to be solved with the transpose of a
c
c         compute offsets for u
c
          if (isym .eq. 0) then
              lshift = jl(n)
          else
              lshift = 0
          end if
          if (isym .eq. -1) then
              ushift = jl(n)
          else
              ushift = 0
          end if
c
c         lower triangular system
```

```
c
      do 30 i = 1, n
         sum = 0.0e0
         if (ja(i) .ge. ja(i+1)) go to 30
         do 20 iseed = ja(i), ja(i+1)-1
            k = ja(iseed)
            do 10 j = jl(iseed-1), jl(iseed)-1
               sum = sum + u(j+lshift) * x(k)
10             k = jl(k)
20       continue
30       x(i) = b(i) - sum
c
c     diagonal system
c
      do 40 i = 1, n
40       x(i) = u(i) * x(i)
c
c     upper triangular system
c
      do 70 i = n, 1, -1
         if (ja(i) .ge. ja(i+1)) go to 70
         do 60 iseed = ja(i), ja(i+1)-1
            k = ja(iseed)
            do 50 j = jl(iseed-1), jl(iseed)-1
               x(k) = x(k) - u(j+ushift) * x(i)
50             k = jl(k)
60       continue
70    continue
      return
      end
```

REFERENCES

[1] R. E. Bank, W. M. Coughran, W. Fichtner, E. H. Grosse, and R. K. Smith, *Transient simulation of silicon devices and circuits*, IEEE Trans. on Electron Devices, ED-32 (1985), pp. 1992–2005.

[2] S. C. Eisenstat, M. C. Gursky, M. H. Schultz, and A. H. Sherman, *Yale sparse matrix package I: The symmetric codes*, Internat. J. Numer. Meth. Engrg., 18 (1982), pp. 1145–1151.

[3] S. C. Eisenstat, M. H. Schultz, and A. H. Sherman, *Algorithms and data structures for sparse symmetric Gaussian elimination*, SIAM J. Sci. Stat. Comput., 2 (1982), pp. 225–237.

[4] A. George and J. Liu, *Computer Solution of Large Sparse Positive Definite Systems*, Prentice Hall, Englewood Cliffs, NJ, 1981.

[5] D. J. Rose, *A graph theoretic study of the numeric solution of sparse positive definite systems*, in Graph Theory and Computing, Academic Press, New York, 1972.

[6] D. J. Rose, R. E. Tarjan, and G. S. Lueker, *Algorithmic aspects of vertex elimination on graphs*, SIAM J. Comput., 5 (1976), pp. 226–283.