SIAM J. SCI. STAT. COMPUT.
Vol. 2, No. 2, June 1981

0196-5204/81/0202-0010 $01.00/0

ALGORITHMS AND DATA STRUCTURES FOR SPARSE SYMMETRIC GAUSSIAN ELIMINATION*

STANLEY C. EISENSTAT†, MARTIN H. SCHULTZ† AND ANDREW H. SHERMAN‡

Abstract. In this paper we present algorithms and data structures that may be used in the efficient implementation of symmetric Gaussian elimination for sparse systems of linear equations with positive definite coefficient matrices. The techniques described here serve as the basis for the symmetric codes in the Yale Sparse Matrix Package.

Key words. sparse matrices, sparse Gaussian elimination, data structures

1. Introduction.

A central task in the numerical solution of important scientific and engineering problems is quite often the solution of a system of linear equations

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

where A is an $N \times N$ sparse symmetric positive definite matrix. For instance, this is frequently the situation when finite difference or finite element methods are used to discretize linear partial differential equations arising from mathematical models in such areas as structural analysis. As another example, the modeling of nonlinear phenomena may lead to a large sparse system of nonlinear equations that can often be solved using a variant of Newton's method in which, at each step, a system of linear equations like (1.1) must be solved.

In the past several years, much attention has been focused on the use of Gaussian elimination for the solution of (1.1). With a variety of theoretical and practical tools, great progress has been made towards the joint goals of numerical accuracy in the solution and economy in terms of computing time and memory space. Several packages of Fortran subprograms [5], [10], [12] for the solution of (1.1) have been developed and widely distributed as a part of research in this area, and these have been gaining increasing acceptance in the scientific community.

The intent of this paper is to present and analyze several algorithms that are used in the Yale Sparse Matrix Package [5], one of the Fortran packages mentioned above. Publications elsewhere [5]–[7], [16] describe the software package and its use and discuss the rationale behind some of the design decisions. In this paper, however, we provide a complete detailed analysis. The conclusions of the analysis are borne out by a variety of experimental results reported, for example, in [4], [13], [17].

It is well known that the matrix A of (1.1) can always be factored in the form

$$A = U^T D U \tag{1.2}$$

where U is unit upper triangular and D is diagonal. Symmetric Gaussian elimination

* Received by the editors March 31, 1980, and in revised form, February 5, 1981. This work was supported in part by the U.S. Office of Naval Research under grant N00014-76-C-0277 and by the U.S. Air Force Office of Scientific Research under contract F49620-77-C-0037.

† Department of Computer Science, Yale University, New Haven, Connecticut 06520.

‡ Department of Computer Sciences, University of Texas at Austin, Austin, Texas 78712. Part of the work of this author was completed while on leave at the Department of Computer Science, University of Toronto, Toronto, Ontario M5S 1A7.

for (1.1) is equivalent to first factoring A in this way and then successively solving the systems:

$$U^T y = b, \quad Dz = y, \quad Ux = z \tag{1.3}$$

to obtain x. For reasons of economy, we wish to factor A and compute x without storing or operating on zeros in A and U. This requires a certain amount of storage and operational overhead; that is, extra storage for pointers to the locations of the nonzeros in addition to that needed for the numerical values of the matrix, and extra nonnumeric "bookkeeping" operations in addition to the required arithmetic operations. Our goal here is to describe a set of algorithms that can be implemented with little overhead.

We use a particularly robust algorithm designed by Chang [2] and previously used by Gustavson [11]. The computation is broken up into three distinct steps: symbolic factorization (SYMFAC), numeric factorization (NUMFAC), and forward- and back-solution (SOLVE). The SYMFAC step computes the zero structure of U (i.e., the positions of the nonzeros in U) from that of A, disregarding the actual numerical entries of A. The NUMFAC step then uses the structural information generated by SYMFAC to compute the numerical values of U and D. Finally, the SOLVE step uses the information produced by both SYMFAC and NUMFAC to solve the resulting triangular and diagonal systems for x.

The main advantage of splitting up the computation is flexibility. If several linear systems have identical coefficient matrices but different right-hand sides, only one SYMFAC and one NUMFAC step are needed; the different right-hand sides require only separate SOLVE steps. Similarly, a sequence of linear systems whose coefficient matrices have identical zero structures but different numerical entries can be solved by using just one SYMFAC step combined with separate NUMFAC and SOLVE steps for each system.

The algorithms for the SYMFAC, NUMFAC, and SOLVE steps are clearly interdependent and, to some extent, depend also on the data structures used to store A and U. Since experience shows that the NUMFAC step requires substantially more computational effort than the other steps, the development in this paper is driven by the requirements of the NUMFAC algorithm; that is, we first design an efficient NUMFAC algorithm and then tailor the SYMFAC and SOLVE algorithms and data structures accordingly. Thus, in § 2, we construct efficient NUMFAC and SOLVE algorithms; in § 3, we describe data structures for efficiently storing A and U; and in § 4, we develop an efficient SYMFAC algorithm.

As an aside, we note that this paper does not consider the possibility of reordering the equations and unknowns of (1.1) in order to reduce the number of nonzeros in U and the number of arithmetic operations required to factor A and solve for x. In practice, it is quite important to do this (see, for example, [17]), but the effect on the topics we discuss here is minimal, since for positive definite systems, the reordering may be completed as a preprocessing step. In fact, the Yale Sparse Matrix Package (and, for that matter, most software packages available for solving (1.1) with sparse Gaussian elimination) accept as input a permutation corresponding to a reordering of the equations and unknowns and provide at least one subprogram for computing a good permutation.

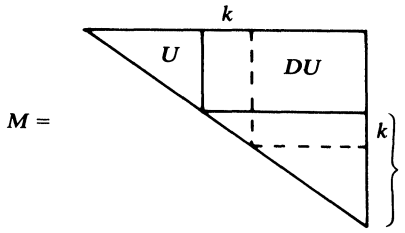
2. Numerical factorization and solution. We begin by examining a factorization algorithm for dense matrices. Such algorithms are well known (see Forsythe and Moler [8]), and Algorithm 2.1 is a row-oriented version. In the algorithm the diagonal entries d_{kk} are stored in a vector D of length N in which $D(k) = d_{kk}$. To avoid notational

confusion, we have presented the algorithm as if the matrix computations were performed on an upper triangular matrix M, although in a standard implementation, all the computations would be performed on the upper triangular portions of A and U.

ALGORITHM 2.1

1. **For** $k = 1$ **to** N **do**
2. $\quad[D(k) = a_{kk};$
3. $\quad$**For** $j = k+1$ **to** N **do**
4. $\quad\quad[m_{kj} = a_{kj}];$
5. $\quad$**For** $i = 1$ **to** $k-1$ **do**
6. $\quad\quad[t = m_{ik};$
7. $\quad\quad m_{ik} = m_{ik}/D(i);$
8. $\quad\quad D(k) = D(k) - t \cdot m_{ik};$
9. $\quad\quad$**For** $j = k+1$ **to** N **do**
10. $\quad\quad\quad[m_{kj} = m_{kj} - m_{ik} \cdot m_{ij}]]];$

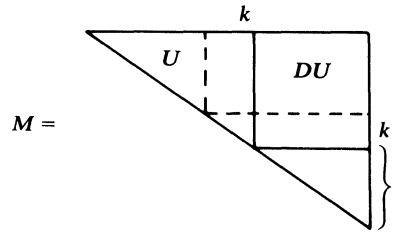
At any time during the execution of Algorithm 2.1, part of M contains entries of U, part contains entries of DU (the matrix product of D and U), and part is unspecified. Figure 2.1a shows the contents of M just prior to the start of the kth step of the factorization. During the kth step, the algorithm computes d_{kk} (line 8), the kth column of U in the kth column of M (line 7), and the kth row of DU in the kth row of M (lines 9–10). Figure 2.1b shows the contents of M at the conclusion of the kth step. At the end of the factorization, M contains exactly the entries of U.



$M =$ Unspecified

$$m_{ij} = \begin{cases} 0 & j < i \\ 1 & j = i < k \\ u_{ij} & i < j < k \\ (DU)_{ij} & i < k;\ j \geqq k \\ \text{Unspecified} & k \leqq i \leqq j \end{cases}$$

(a) *Prior to kth factorization step*



$M =$ Unspecified

$$m_{ij} = \begin{cases} 0 & j < i \\ 1 & j = i \leqq k \\ u_{ij} & i < j \leqq k \\ (DU)_{ij} & i \leqq k;\ j > k \\ \text{Unspecified} & k < i \leqq j \end{cases}$$

(b) *After kth factorization step*

FIG. 2.1.

When A is dense, Algorithm 2.1 can be implemented efficiently, since it stores and operates on just the diagonal of D and the upper triangles of A and U. However, when A is sparse, the algorithm fails to exploit the zeros in A and U to reduce the storage and work. The bulk of this section is devoted to the development of an efficient numerical factorization algorithm for this case.

Conceptually, at least, it is possible to avoid arithmetic operations on zeros by explicitly testing the operands prior to using them; unfortunately, there are two serious problems with this simple approach. First, all the entries in the upper triangles of A

and M would have to be stored, since any of them could be tested or used as m_{kj} in line 10 of Algorithm 2.1. Second, there would be more test operations than arithmetic operations, so that the running time of such an algorithm would be asymptotically proportional to the amount of testing rather than to the amount of arithmetic op(A)[1].

To overcome these difficulties, assume that at the beginning of the kth step of the algorithm, we know:

(i) the set R_k^A of columns $j>k$ for which $a_{kj}\neq 0$;
(ii) the set R_k^U of columns $j>k$ for which $u_{kj}\neq 0$;
(iii) the set C_k of rows $i<k$ for which $u_{ik}\neq 0$;
(iv) for each $i<k$, the set $R_{i,k}^U$ of columns $j>k$ for which $u_{ij}\neq 0$.

Then we can modify Algorithm 2.1 to obtain Algorithm 2.2 in which the only entries of M used are those corresponding to nonzeros in A or U.

ALGORITHM 2.2

```
1.  For k=1 to N do
2.      [D(k)=a_kk;
3.      For j∈R_k^U do
4.          [m_kj=0];
5.      For j∈R_k^A do
6.          [m_kj=a_kj];
7.      For i∈C_k do
8.          [t=m_ik;
9.          m_ik=m_ik/D(i);
10.         D(k)=D(k)-t·m_ik;
11.         For j∈R_{i,k}^U do
12.             [m_kj=m_kj-m_ik·m_ij]]];
```

We now turn to the implementation of Algorithm 2.2. This requires that we deal with the problem of "fillin", that is, with the creation of nonzeros in matrix positions of U corresponding to zeros in A. A straightforward means of handling fillin involves list-processing. At the kth step of the algorithm, all of the fillin occurs in the kth row, so, by keeping that row as a linked list, it is possible to easily insert new nonzeros in their proper places. After the kth step no additional fillin will occur in the kth row, so it can be stored in a more efficient manner if desired. Unfortunately, the list-processing approach is not particularly efficient in terms of computation time, since potentially it requires a sweep through the entire linked list for each execution of the innermost loop, even though $|R_{i,k}^U|$ might be quite small.[2]

To avoid list-processing, we use a technique known as "row expansion" that was first suggested by Gustavson [11]. A modification of Algorithm 2.2 using row expansion is given as Algorithm 2.3. During the kth step of Algorithm 2.3, row expansion consists of three phases:

(i) The kth row of A is expanded into a vector V of length N so that $V(j)=a_{kj}$ for $j\in R_k^U\cup\{k\}$. We assume that $R_k^A\subseteq R_k^U$; that is, we do not allow for accidental cancellation in which a nonzero in A becomes zero in U.

(ii) For each $i\in C_k$, a multiple of row i of U is subtracted from V. No special attention need be given to fillin since, if the jth column fills in, then $V(j)$ was set to zero in (i) before the row operations began.

[1] We use op(A) to denote the number of multiplications and divisions required to factor the sparse matrix A, including only operations involving two nonzero operands.

[2] We use $|S|$ to denote the size of the set S.

(iii) The kth row of DU is obtained by storing $V(j)$ in $(DU)_{kj}$ for $j \in R_k^U$. Since $(DU)_{kj} \neq 0$ if and only if $j = k$ or $u_{kj} \neq 0$ (i.e., $j \in R_k^U$), this storage operation stores only nonzeros. Moreover, since all fillin in the kth row occurs during the kth step of the algorithm, this storage operation will implicitly take care of the insertion of fillin entries in the kth row of U.

At this point we have an implementation that is quite time-efficient but apparently requires a great deal of storage for auxiliary information about A and U. Certainly, the sets R_k^A and R_k^U must be stored because they describe the zero structures of A and U, respectively. However, the sets C_k and $R_{i,k}^U$ do not really contain new information about U; instead, they simply present the same structure information in a somewhat different way. To save storage, we compute these sets from the sets R_k^U.

Observe that C_k and $R_{i,k}^U$, $1 \leqq i \leqq k$, are required only during the kth step of Algorithm 2.3. We can exploit this by a scheme in which the computation of these sets need not be completed until the end of the $(k-1)$st step. During previous steps we allow these sets to be only partially computed.

ALGORITHM 2.3

```
1.  For k = 1 to N do
2.      [D(k) = a_kk;
3.      For j ∈ R_k^U do
4.          [V(j) = 0];
5.      For j ∈ R_k^A do
6.          [V(j) = a_kj];
7.      For i ∈ C_k do
8.          [t = m_ik;
9.          m_ik = m_ik/D(i);
10.         D(k) = D(k) − t · m_ik;
11.         For j ∈ R_{i,k}^U do
12.             [V(j) = V(j) − m_ik · m_ij]]];
```

The key to this scheme is to keep the sets R_k^U ordered by increasing column number. Then for $i < k$, $R_{i,k}^U$ contains simply the segment of R_i^U beginning with the first entry larger than k and including all succeeding entries. If each set R_k^U is stored as a sequence of integers in consecutive locations in an array JU, then we can use a vector IL of length N to obtain the sets $R_{i,k}^U$ as required. We just make certain that, at the beginning of the kth step, $IL(i)$ points to the location of the first entry in $R_{i,k}^U$. At the conclusion of the kth step, we must update $IL(i)$ for exactly those $i \in C_k \cup \{k\}$, since $R_{i,k}^U = R_{i,k+1}^U$ for all other i. For $i \in C_k$ we set $IL(i) = IL(i) + 1$, while for $i = k$, we set $IL(k)$ to point to the location of the first entry of R_k^U.

To compute the sets C_k, we employ sets P_k that are constructed in such a way that, during the kth step,

(i) $P_k = C_k$, and

(ii) for $j > k$, P_j contains those row indices $i < k$ such that $i \in C_j$ but $i \notin C_m$, $k \leqq m < j$.

During the kth step we use P_k in place of C_k, and, at the conclusion of the kth step, we update the sets P_j, $j > k$, so that conditions (i) and (ii) are satisfied during the $(k+1)$st step. This update is accomplished by moving the entries of P_k into the appropriate sets P_j, $j > k$. For each $i \in P_k$, if $j > k$ is the smallest entry of $R_{i,k+1}^U$, then i is placed into the set P_j. If $R_{i,k+1}^U$ is empty, i is not placed into any set.

To store the sets P_k we use a special form of linked list in which the data and pointers coincide. All of the information about these sets is stored in a single array JL

of length N. At the kth step each entry $JL(j)$, $k \leqq j \leqq N$, continues the row index that is the first entry of P_j, if any, or zero otherwise. For each nonzero $JL(i)$, $JL(JL(i))$ is either the next row index in the same set as $JL(i)$ or zero if there are no more entries in that set. This scheme works because, at the kth step, the sets P_j, $k \leqq j \leqq N$, are disjoint and contain only row indices $i < k$. As an illustration, Figure 2.2a shows a storage configuration that might ensue during the factorization algorithm.

$$C_k = \{2, 4, 5\} \quad C_{k+1} = \{2, 3\} \quad C_{k+2} = \{1, 2, 4\}$$

$$R_1^U = \{2, \cdots, k-1, k+2, \cdots\}$$

$$R_2^U = \{3, \cdots, k, k+1, k+2, \cdots\}$$

$$R_3^U = \{4, \cdots, k-1, k+1, \cdots\}$$

$$R_4^U = \{5, \cdots, k, k+2, \cdots\}$$

$$R_5^U = \{6, \cdots, k\}$$

(a) At step k:
$$\begin{cases} P_k = \{2, 4, 5\} \\ P_{k+1} = \{3\} \\ P_{k+2} = \{1\} \end{cases}$$

(b) At step $k+1$:
$$\begin{cases} P_k = \emptyset \\ P_{k+1} = \{3, 2\} \\ P_{k+2} = \{1, 4\} \end{cases}$$

FIG. 2.2

To actually update JL to reflect a move of row index i from P_k to P_j, we save temporarily the value of $JL(i)$ (since it is the next row index in P_k), set $JL(i) = JL(j)$, and set $JL(j) = i$. We then go on to the next entry of P_k. Figure 2.2b shows the storage configuration that would result following the update of JL at the kth step, beginning from the configuration shown in Figure 2.2a.

Algorithm 2.4 is a modification of Algorithm 2.3 that incorporates the refinements introduced in this section. In addition, it uses the vector D for both D and V, based on the observation that, at the kth step, the algorithm requires only the first $k-1$ entries of D and the last $N-k+1$ entries of V. From our previous discussion it is clear that the algorithm is efficient in terms of storage, since only about $2N$ storage locations are required in addition to the space for the numerical entries of A, D, and U and the structure descriptions of A and U.

To determine the operational overhead of the algorithm, we examine separately the times required for arithmetic operations and set-updating operations. There are $O(\text{op}(A))$ arithmetic operations (since we avoid operations on zeros), and each one requires constant time, due to the use of row expansion. Thus the NUMFAC algorithm requires a total of $O(\text{op}(A))$ time for arithmetic operations. At the kth step, one set-updating operation is required for each entry of $C_k \cup \{k\}$, so that, overall, $O(nz(U)+N)$ updating operations[3] are performed. Since each set-updating operation costs only constant time, together they require a total of only $O(nz(U)+N)$ time, and the entire algorithm runs in $O(\text{op}(A))$ time since $nz(U)+N \leqq \text{op}(A)$.

We conclude this section by presenting the SOLVE algorithm for the forward- and back-solution of sparse symmetric systems. Algorithm 2.5 successively solves the

[3] We use $nz(M)$ to denote the number of nonzero entries of the array M that must be stored in any sparse representation of M. For symmetric matrices, this includes only nonzero entries in the upper triangle.

ALGORITHM 2.4

```
1.  For k = 1 to N do
2.      [P_k = ∅];
3.  For k = 1 to N do
4.      [D(k) = a_kk;
5.      For j ∈ R_k^U do
6.          [D(j) = 0];
7.      For j ∈ R_k^A do
8.          [D(j) = a_kj];
        For i ∈ P_k do
10.         [t = m_ik;
11.         m_ik = m_ik/D(i);
12.         D(k) = D(k) − t · m_ik;
13.         For j ∈ R_{i,k}^U do
14.             [D(j) = D(j) − m_ik · m_ij];
15.         R_{i,k}^U = R_{i,k}^U − {k + 1};
16.         If R_{i,k+1}^U ≠ ∅ then do
17.           [j = min (R_{i,k+1}^U);
18.           P_j = P_j ∪ {i}]];
19.     If R_k^U ≠ ∅ then do
20.       [j = min (R_k^U);
21.       P_j = P_j ∪ {k}]];
```

ALGORITHM 2.5

```
1.  For k = 1 to N do
2.      [y_k = b_k];
3.  For k = 1 to N − 1 do
4.      [For j ∈ R_k^U do
5.          [y_j = y_j − u_kj · y_k]];
6.  For k = 1 to N do
7.      [z_k = y_k/D(k)];
8.  For k = N to 1 by −1 do
9.      [x_k = z_k;
10.     For j ∈ R_k^U do
11.         [x_k = x_k − u_kj · x_j]];
```

systems (1.3), and it fits in well with the NUMFAC algorithm because it uses only the nonzero entries of A, D, and U and the structure information for A and U. It is quite easy to see that the algorithm requires only $O(nz(U)+N)$ time, since it operates exactly once on each nonzero in U in solving each of the two triangular systems. The cost of solving the diagonal system, of course, is only $O(N)$.

3. Storage of sparse matrices. In the last section we developed an efficient algorithm for the numerical U^TDU factorization of A. That algorithm used as data the nonzeros of A and U and the sets R_k^A and R_k^U, $1 \leqq k \leqq N$, that describe the structures of the rows of A and U, respectively. In this section we describe data structures that can be used to efficiently store all of this information.

All of the storage schemes we consider are row-oriented; that is, information about the matrix is organized row-by-row. This makes it easy to access information about the rows but quite difficult to obtain information about the columns. In general, this might

be a serious drawback in a storage scheme; however, as we have seen, information about the columns is not required in this application.

The first storage scheme is the "uncompressed storage scheme" that is employed in various forms by Gustavson [11], Curtis and Reid [3], Munksgaard [12], and others. The nonzero matrix entries are stored by rows in order of increasing column index. To identify the entries of any row, it is necessary to know where the row begins, how many entries it contains, and in what columns the entries lie. This extra information about the structure of the matrix is the storage overhead mentioned in § 1.

The uncompressed storage scheme for the matrix A uses three arrays (IA, JA, and A) as illustrated in Fig. 3.1. The array A contains the nonzero entries of the upper

$$A = \begin{bmatrix} a_{11} & a_{12} & 0 & 0 & 0 & 0 \\ a_{21} & a_{22} & a_{23} & a_{24} & a_{25} & 0 \\ 0 & a_{32} & a_{33} & 0 & 0 & 0 \\ 0 & a_{42} & 0 & a_{44} & 0 & a_{46} \\ 0 & a_{52} & 0 & 0 & a_{55} & 0 \\ 0 & 0 & 0 & a_{64} & 0 & a_{66} \end{bmatrix}$$

	row 1		row 2				row 3	row 4		row 5	row 6
A:	a_{11}	a_{12}	a_{22}	a_{23}	a_{24}	a_{25}	a_{33}	a_{44}	a_{46}	a_{55}	a_{66}
k	1	2	3	4	5	6	7	8	9	10	11
JA	1	2	2	3	4	5	3	4	6	5	6
IA	1	3	7	8	10	11	12				

FIG. 3.1

triangle stored row-by-row.[4] The array JA contains the column indices that correspond to the nonzeros in the array A: if $A(k)$ contains a_{ij}, then $JA(k) = j$. Finally, the array IA contains $N+1$ pointers that delimit the rows of nonzeros in A and indices in JA; i.e., $A(IA(i))$ is the first entry stored for the ith row, and $A(IA(i+1)-1)$ is the last. Thus the length of the ith row is given by $IA(i+1)-IA(i)$. Since R_i^A includes all but the first of the column indices stored in JA for row i, we see that the uncompressed storage scheme can be used quite naturally with Algorithms 2.4 and 2.5.

The storage overhead incurred when using the uncompressed storage scheme for A is the storage for the arrays IA and JA. Since IA has $N+1$ entries and JA has one entry per nonzero in the upper triangle of A, the storage overhead is approximately equal to $nz(A)$.

It is possible to use the uncompressed storage scheme for U as well as A, as illustrated in Fig. 3.2a. However, this ignores certain features of U that allow a potentially significant reduction in the space required for column indices. Since storage is often at a premium when solving sparse linear systems, we have chosen to use a more complex "compressed storage scheme" that usually leads to a substantial reduction in storage overhead at the cost of at most a small increase in operational overhead (see [7]).

Figure 3.2a shows the data structures required to store a particular matrix U in the uncompressed storage scheme. It is immediately evident that the diagonal entries

[4] A is symmetric, so only the upper triangle is stored.

$$U = \begin{bmatrix} 1 & u_{12} & 0 & 0 & 0 & 0 \\ 0 & 1 & u_{23} & u_{24} & u_{25} & 0 \\ 0 & 0 & 1 & u_{34} & u_{35} & 0 \\ 0 & 0 & 0 & 1 & u_{45} & u_{46} \\ 0 & 0 & 0 & 0 & 1 & u_{56} \\ 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix}$$

(a)

U:	1	u_{12}	1	u_{23}	u_{24}	u_{25}	1	u_{34}	u_{35}	1	u_{45}	u_{46}	1	u_{56}	1
k:	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
JU:	1	2	2	3	4	5	3	4	5	4	5	6	5	6	6

IU:	1	3	7	10	13	15	16

(b)

U:	u_{12}	u_{23}	u_{24}	u_{25}	u_{34}	u_{35}	u_{45}	u_{46}	u_{56}
k:	1	2	3	4	5	6	7	8	9
JU:	2	3	4	5	4	5	5	6	6

IU:	1	2	5	7	9	10	10

(c)

U:	u_{12}	u_{23}	u_{24}	u_{25}	u_{34}	u_{35}	u_{45}	u_{46}	u_{56}
k:	1	2	3	4	5	6	7	8	9
JU:	2	3	4	5	6				

IU:	1	2	5	7	9	10	10

JU:	1	2	3	4	5	5

FIG. 3.2

need not be stored, since they are always equal to one and occur as the first stored entry of each row. Figure 3.2b shows the data structures required when the diagonal entries are omitted.

We now consider ways to "compress" JU. Assume that we are creating the data structure for U by adding one row at a time, and consider the situation as we add row k. Let row j be the highest-numbered row whose indices are stored terminating in the last position in JU used to store column indices for the first $k-1$ rows. There are two main cases in which compression can take place.

First, the indices for row k are the same as the last several indices for some preceding row i. In this case we can use the indices stored for row i to avoid storing a separate set of indices for row k; all that is needed is a pointer to locate the start of the indices for row k within those for row i.[5] In Figure 3.2b the indices for row 3 are 4, 5; while those for row 2 are 3, 4, 5. Instead of storing the indices for row 3 separately, we simply use the last two indices already stored for row 2.

[5] The number of indices for row k can be determined from IU.

Second, the first several indices for row k are the same as the last several for row j. In this case we can overlap the indices for rows j and k. Again all that is needed is a pointer to locate the beginning of the indices for row k. In Figure 3.2b the indices for row 4 are 5, 6; while those for row 3 are 4, 5. Overlapping these two sets of indices allows us to avoid duplicating the index 5 in JU.

In general, the compressed indices in JU do not correspond directly to the nonzeros stored in the array U, so an extra array of pointers (IJU) is required to locate the start of the indices for each row (see Figure 3.2c). Thus the array U contains the nonzero entries of the strict upper triangle of U stored row-by-row; the $N+1$ entries of IU delimit the rows in U as before; JU is the compressed array of column indices; and the N entries of IJU point to the first column index in JU for each row. The nonzeros of the ith row of the strict upper triangle of U are stored in $U(IU(i))$ through $U(IU(i+1)-1)$, and the corresponding column indices are stored in $JU(IJU(i))$ through $JU(IJU(i)+IU(i+1)-IU(i)-1)$. Note that, since the entries of each row are ordered by increasing column number, the column indices stored in JU for row i form precisely the set R_i^U required by Algorithms 2.4 and 2.5.

The storage overhead for the compressed storage of U is the number of locations required for IU, JU, and IJU. Although, for small problems, this overhead can be slightly larger than that for the uncompressed scheme, it is usually substantially smaller. For certain systems arising in the solution of elliptic boundary value problems in a square, for instance, the overhead is actually $O(N)$ storage locations as opposed to the $O(N \log N)$ nonzeros in U (see [16]).

4. Symbolic factorization. In § 2 we presented NUMFAC and SOLVE algorithms that required as input data the sets R_k^A and R_k^U, $1 \leqq k \leqq N$. As we saw in the last section, the sets R_k^A are available directly from the uncompressed storage scheme used to store A. In this section we develop a SYMFAC algorithm to efficiently compute the ordered sets R_k^U. To simplify the presentation, we will assume that the R_k^U are to be computed as unordered sets; at the end of the section, we will discuss how to obtain ordered sets.

At the kth step we will compute R_k^U from R_k^A and the sets R_i^U, $i<k$. An examination of Algorithm 2.3 shows that $u_{kj} \neq 0$, $j>k$, if and only if either

(i) $a_{kj} \neq 0$, or

(ii) $u_{ij} \neq 0$ for some $i \in C_k$.

Thus $j \in R_k^U$ if and only if either

(i) $j \in R_k^A$ or

(ii) $j \in R_{i,k}^U$ for some $i \in C_k$.

This observation leads to Algorithm 4.1, which could be implemented efficiently by replacing references to $R_{i,k}^U$ and C_k as in Algorithm 2.4.

ALGORITHM 4.1

1. **For** $k=1$ **to** N **do**
2. $\quad [R_k^U = \emptyset]$;
3. **For** $k=1$ **to** $N-1$ **do**
4. $\quad$ [**For** $j \in R_k^A$ **do**
5. $\quad\quad [R_k^U = R_k^U \cup \{j\}]$;
6. $\quad$ **For** $i \in C_k$ **do**
7. $\quad\quad$ [**For** $j \in R_{i,k}^U$ **do**
8. $\quad\quad\quad [R_k^U = R_k^U \cup \{j\}]]]$;

However, we can do better. Consider the example in Figure 4.1 in which fillin occurs in u_{24} and u_{34}. In computing R_3^U we use both $R_{1,3}^U$ and $R_{2,3}^U$, even though $R_{1,3}^U$ is redundant in the sense that $R_{1,3}^U \subseteq R_{1,2}^U \subseteq R_2^U$ and hence $R_{1,3}^U \subseteq R_{2,3}^U$.

$$A = \begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & 0 \\ a_{31} & a_{32} & a_{33} & 0 \\ a_{41} & 0 & 0 & a_{44} \end{bmatrix}$$

$$U = \begin{bmatrix} 1 & u_{12} & u_{13} & u_{14} \\ 0 & 1 & u_{23} & u_{24} \\ 0 & 0 & 1 & u_{34} \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$R^U_{1,3} = \{4\}$ $\quad R^U_2 = \{3, 4\}$

$R^U_{1,2} = \{3, 4\}$ $\quad R^U_{2,3} = \{4\}$

FIG. 4.1

In fact, this observation can be generalized. Let l_k be the set of rows $i \in C_k$ for which k is the minimum column index in R^U_i. The following result expresses a type of "transitivity condition" for the fillin in symmetric Gaussian elimination and implies that, in forming R^U_k, it suffices to examine $R^U_{i,k}$ for $i \in l_k$. (Similar results have been developed by others, e.g., [15].)

THEOREM 4.1 (Sherman [16]). *Let $i \in C_k$. Then either $i \in l_k$ or there is an m, $i < m < k$, such that $m \in l_k$ and $R^U_{i,k} \subseteq R^U_{m,k}$.*

Using the theorem we see that $j \in R^U_k$ if and only if either

(i) $j \in R^A_k$, or

(ii) $j \in R^U_{i,k}$ for some $i \in l_k$.

Algorithm 4.2 is a symbolic factorization algorithm based on this observation. The sets l_k are formed by adding each row i to the proper set l_m as soon as R^U_i is computed. In terms of implementation, the sets l_k are similar to the sets P_k introduced in § 2. In particular, only l_k and the partially computed sets l_m, $k < m \leqq N$, are required at the kth step. Since these are disjoint sets containing only row indices less than k, we can store them using a single array of length N, just as we did for the sets P_k.

ALGORITHM 4.2

```
1.  For k = 1 to N do
2.      [R_k^U = ∅;
3.      l_k = ∅];
4.  For k = 1 to N − 1 do
5.      [For j ∈ R_k^A do
6.          [R_k^U = R_k^U ∪ {j}];
7.      For i ∈ l_k do
8.          [For j ∈ R_{i,k}^U do
9.              [If j ∉ R_k^U then do
10.                 [R_k^U = R_k^U ∪ {j}]]];
11.     If R_k^U ≠ ∅ then do
12.       [i = min (R_k^U);
13.       l_i = l_i ∪ {k}]];
```

The cost of Algorithm 4.2 is determined by the costs of its innermost loop (lines 7–10) and of lines 5–6 and 11–13, since the remainder of the algorithm requires only $O(N)$ operations. At the kth step, for each $i \in l_k$ and $j \in R^U_{i,k}$, lines 9–10 are executed once. Since each row i is a member of exactly one set l_k, lines 9–10 are executed at most once for each entry of the combined sets R^U_i, $l \leqq i \leqq N$ (i.e., once for each nonzero entry of U). If the characteristic vector of R^U_k (cf., Aho, Hopcroft, and Ullman [1], p.

49]) is computed along with R_k^U, then the test in line 9 requires constant time, since it can be done by examining one entry of the characteristic vector. Furthermore, the union operation in line 10 also requires only constant time since j is simply appended to the end of R_k^U, and the jth entry of the characteristic vector is set to one. Hence a total of $O(nz(U))$ time is spent in the innermost loop of the algorithm.

At the kth step, line 6 is executed once for each entry in R_k^A. Thus overall, it is executed just once for each entry of the sets R_k^A, $1 \leqq k \leqq N$, combined (i.e., at most once for each nonzero entry of A). Again the union operation requires only constant time, and, since $nz(A) \leqq nz(U)$, the algorithm spends at most $O(nz(U))$ time in lines 5–6.

Finally, note that, for each k, lines 11–13 require $O(|R_k^U|)$ time, since the minimum operation of line 11 requires that much time, while the union operation of line 11 requires constant time. Thus overall, lines 11–13 and the entire algorithm both require only $O(nz(U))$ time.

As discussed earlier, we need each set R_k^U in increasing order. One solution to this difficulty is to modify Algorithm 4.2 to compute ordered sets directly by changing the set union operations to list insertion operations so that each new entry is inserted into its proper place. However, the time required for a list insertion operation depends on the length of the list into which the insertion is made, and, except in special cases, the modified algorithm may require more than $O(nz(U))$ time asymptotically. In practice, the entire loop in lines 8–10 would be replaced with a list merging operation to merge the ordered list R_i^U into R_k^U. However, the same conclusion would follow, since the merging operation could require $O(|R_k^U| + |R_i^U|)$ time.

An alternative to the use of list operations is to first compute the unordered sets and then sort them. Viewing the set $R = \bigcup_{k=1}^{N} R_k^U$ as a set of ordered pairs (k, j) for $j \in R_k^U$, we wish to sort R in lexicographic order; that is, so that (k, j) precedes (k', j') in R if and only if either

(i) $k < k'$ or

(ii) $k = k'$ and $j < j'$.

Since all the entries of the sets are integers between 1 and N, this sorting process can be accomplished efficiently by using a bucket sort (cf. Aho, Hopcroft, and Ullman [1, pp. 77–84]). This requires time proportional to the number of entries in the combined sets, that is, $O(nz(U))$ time, and, combining the sort with Algorithm 4.2, we obtain an algorithm for computing the ordered sets in $O(nz(U))$ time (cf., Rose and Whitten [15], Rose, Tarjan, and Lueker [14], George and Liu [9], and Sherman [16]).

From this discussion it seems clear that asymptotically one should use the second method for obtaining ordered sets. Surprisingly, however, experiments indicate that it is usually faster to compute the ordered sets directly with list operations. This is especially true for linear systems arising in the use of finite difference or finite element methods for the solution of partial differential equations, since it can be proved that computing the ordered sets directly requires only $O(nz(U))$ time for such systems. For this reason, the software based on Algorithm 4.2 (see [5]) uses the list merging version described above.

REFERENCES

[1] A. AHO, J. E. HOPCROFT AND J. D. ULLMAN, *The Design and Analysis of Computer Algorithms*, Addison-Wesley, Reading, MA, 1974.

[2] A. CHANG, *Application of sparse matrix methods in electric power system analysis*, in *Sparse Matrix Proceedings*, R. A. Willoughby, ed., Rep. RAI, IBM Research, Yorktown Heights, NY, 1968, pp. 113–122.

[3] A. R. CURTIS AND J. K. REID, *Fortran subroutines for the solution of sparse sets of linear equations*, AERE rep. R.6844, HMSO, London, 1971.

[4] S. C. EISENSTAT, J. A. GEORGE, R. GRIMES, D. R. KINCAID AND A. H. SHERMAN, *Some comparisons of software packages for large sparse linear systems*, in *Advances in Computer Methods for Partial Differential Equations—III*, R. Vichnevetsky and R. S. Stepleman, eds., IMACS, New Brunswick, NJ, 1979, pp. 98–106.

[5] S. C. EISENSTAT, M. C. GURSKY, M. H. SCHULTZ AND A. H. SHERMAN, *The Yale sparse matrix package I: Symmetric matrices*, Rep. 112, Dept. of Computer Science, Yale University, New Haven, CT, 1977.

[6] S. C. EISENSTAT, M. H. SCHULTZ AND A. H. SHERMAN, *Application of sparse matrix methods to partial differential equations*, in *Advances in Computer Methods for Partial Differential Equations—I*, R. Vichnevetsky, ed., AICA, New Brunswick, NJ, 1975, pp. 40–45.

[7] S. C. EISENSTAT, M. H. SCHULTZ AND A. H. SHERMAN, *Considerations in the design of software for sparse Gaussian elimination*, in *Sparse Matrix Computations*, J. R. Bunch and D. J. Rose, eds., Academic Press, New York, 1976, pp. 263–274.

[8] G. E. FORSYTHE AND C. B. MOLER, *Computer Solution of Linear Algebraic Systems*, Prentice-Hall, Englewood Cliffs, NJ, 1967.

[9] J. A. GEORGE AND J. W.-H. LIU, *An optimal algorithm for symbolic factorization of symmetric matrices*, Res. rep. CS-78-11, Department of Computer Science, University of Waterloo, Ontario, 1978.

[10] J. A. GEORGE AND J. W.-H. LIU, *User guide for SPARSPAK: Waterloo sparse linear equations package*, Res. rep. CS-78-30, Dept. of Computer Science, University of Waterloo, Ontario, 1978.

[11] F. G. GUSTAVSON, *Some basic techniques for solving sparse systems of linear equations*, in *Sparse Matrices and Their Applications*, D. J. Rose and R. A. Willoughby, eds., Plenum Press, New York, 1972, pp. 41–52.

[12] N. MUNSGAARD, *New factorization codes for sparse, symmetric and positive definite matrices*, BIT 19 (1979), pp. 43–52.

[13] D. J. ROSE, A. H. SHERMAN, R. E. TARJAN AND G. F. WHITTEN, *Algorithms and software for in-core factorization of sparse symmetric positive definite matrices*. Comput. & Structures 11 (1980), pp. 597–608.

[14] D. J. ROSE, R. E. TARJAN AND G. S. LUEKER, *Algorithmic aspects of vertex elimination on graphs*, SIAM J. Comput. 5 (1976), pp. 266–283.

[15] D. J. ROSE AND G. F. WHITTEN, private communication.

[16] A. H. SHERMAN, *On the efficient solution of sparse systems of linear and nonlinear equations*, doctoral dissertation, Department of Computer Science, Yale University, New Haven, CT, 1975.

[17] P. T. WOO, S. C. EISENSTAT, M. H. SCHULTZ AND A. H. SHERMAN, *Application of sparse matrix techniques to reservoir simulation*, in *Sparse Matrix Computations*, J. R. Bunch and D. J. Rose, eds., Academic Press, New York, 1976, pp. 427–438.