Efficient Implementation of Sparse Symmetric Gaussian Elimination*

S.C. Eisenstat, M.H. Schultz,
and A.H. Sherman†

Research Report #48

---

* This paper has appeared in the Proceedings of the AICA International Symposium on Computer Methods for Partial Differential Equations, Bethlehem, Pa., June, 1975.

† Department of Computer Science, Yale University, New Haven, Connecticut 06520. This research supported in part by NSF Grant GJ-43157 and ONR Grant N0014-67-A-0097-0016.

1. Introduction

Consider the system of linear equations

$$A \underset{\sim}{x} = \underset{\sim}{b} \tag{1.1}$$

where A is an $N \times N$ symmetric, positive definite matrix and $\underset{\sim}{x}$ and $\underset{\sim}{b}$ are vectors of length N. Such systems arise frequently in scientific computation, particularly in the numerical solution of partial differential equations, and it is very important to be able to solve them efficiently.

When A is dense (i.e. most $a_{ij} \neq 0$), then it is standard to solve (1.1) with dense symmetric Gaussian elimination† (cf. Dahlquist and Björck [3], pp. 162-5). A is factored into the product U^tDU, where U is a unit upper triangular matrix (i.e. $u_{ii} = 1$, $u_{ij} = 0$, $i > j$) and D is a positive diagonal matrix. The solution $\underset{\sim}{x}$ is then obtained by solving

$$U^t \underset{\sim}{y} = \underset{\sim}{b}, \quad D \underset{\sim}{z} = \underset{\sim}{y}, \quad \text{and} \quad U \underset{\sim}{x} = \underset{\sim}{z}.$$

Since A is symmetric and positive definite, this procedure is numerically stable (cf. Dahlquist and Björck [3], p. 164).

Unfortunately, in many numerical computations N is quite large (several hundred or more), and A is sparse (i.e. most $a_{ij} = 0$). In that case dense symmetric Gaussian elimination is not very efficient, since it does not exploit the large number of zeroes in A and U

---

† In this paper we consider symmetric Gaussian elimination to be identical to a U^tDU factorization followed by backsolution, and we use the term in that sense.

in order to decrease both the storage and work requirements. Historically, iterative methods such as SOR (cf. Young [9]) have been quite popular for the solution of large, sparse, symmetric, positive definite systems of linear equations. However, iterative methods have a number of difficulties not present in Gaussian elimination: estimating iteration parameters, selecting good starting guesses, and determining when to stop the iteration. Because of these difficulties, it is desirable to have a form of Gaussian elimination which is efficient for large, sparse linear systems.

The basic idea of sparse symmetric Gaussian elimination is to factor A and compute $\underset{\sim}{x}$ without storing or operating on the zeroes in A and U. To do this requires a certain amount of overhead: i.e. extra storage for pointers in addition to that needed for nonzeroes in A and U, and extra non-numeric operations in addition to the required arithmetic operations. However, when N is large, the overhead should not increase the solution cost by more than a small constant factor. In this paper we describe one efficient implementation of sparse symmetric Gaussian elimination.

We begin in Section 2 by presenting an algorithm for dense symmetric Gaussian elimination which can be easily adapted to avoid operating on zeroes in A and U, and we discuss the information required to do this. In Section 3 we describe an efficient implementation of sparse symmetric Gaussian elimination, emphasizing an intuitive understanding rather than algorithmic details. Finally, in Section 4 we describe the data structures required to store A and U efficiently. We do not

discuss the problem of reordering A to reduce the storage or the work for sparse symmetric Gaussian elimination, nor do we present any numerical examples. Both of these topics are treated in another paper by the authors in these proceedings (cf. Eisenstat, Schultz, and Sherman [4]).

2. Symmetric Gaussian Elimination

In this section we present an algorithm for dense symmetric Gaussian elimination. We then give a straightforward, but inefficient, modification of the dense algorithm which takes advantage of the sparseness in A and U. Finally, we determine what information about A and U is required to make sparse symmetric Gaussian elimination efficient, and we present the algorithm which is implemented in Section 3.

Algorithm 2.1 is a row-oriented algorithm for dense symmetric Gaussian elimination. The diagonal entries d_{kk} are stored in an array D in which $D(k) = d_{kk}$, and in practice, all of the computations would be performed on the upper triangular matrix U. However, to avoid notational confusion we present the algorithm using the upper triangular array M instead. At any time during the factorization portion 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 k-th step of the factorization. During the k-th step, the algorithm computes the k-th column of U in the k-th column

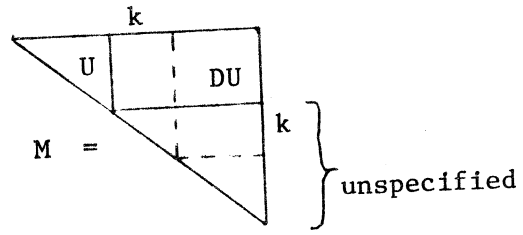
```
line
      Comment: U^tDU Factorization
 1    For k ← 1 to N do
 2        [For j ← k+1 to N do
 3            [m_kj ← a_kj];
 4         m_kk ← 1;
 5         d_kk ← a_kk;
 6         For i ← 1 to k-1 do
 7             [d_kk ← d_kk - m_ik^2/d_ii;
 8              m_ik ← m_ik/d_ii;
 9              For j ← k+1 to N do
10                  [m_kj ← m_kj - m_ik*m_ij]]];

      Comment: Now M = U
      Comment: Backsolution to obtain x
11    For k ← 1 to N do
12        [y_k ← b_k];
13    For k ← 1 to N-1 do
14        [For j ← k+1 to N do
15            [y_j ← y_j - m_kj*y_k]];
16    For k ← 1 to N do
17        [z_k ← y_k/d_kk];
18    For k ← N to 1 do
19        [x_k ← z_k;
20         For j ← k+1 to N do
21             [x_k ← x_k - m_kj*x_j]];
```

ALGORITHM 2.1

---

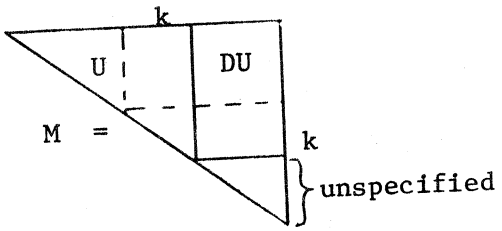
of M (line 8), the k-th row of DU in the k-th row of M (lines 9-10) and d_{kk} (line 7). At the conclusion of the k-th step, the contents of M have been modified as shown in Figure 2.1b. At the end of the factorization, M contains exactly the entries of U. Lines 11-21 of the algorithm perform the backsolution to obtain $\underset{\sim}{x}$.

When A is dense, Algorithm 2.1 may be implemented efficiently,



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

(a)
Prior to the k-th factorization step



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

(b)
After k-th factorization step

FIGURE 2.1

since it stores and operates on just the upper triangle of A, D, and U. However, when A is sparse, Algorithm 2.1 fails to exploit the zeroes in A and U to reduce the storage and work.

Conceptually, at least, it is possible to avoid arithmetic operations on zeroes by testing the operands prior to using them (see Algorithm 2.2). Unfortunately, there are two serious problems with this approach. First, all of the entries of the upper triangles of A and M must be stored, since any of them could be tested in lines 6, 9, 14, or 20, or used as m_{kj} in line 10. And second, there are more test operations than arithmetic operations, so that the running time of Algorithm 2.2 would be proportional to the amount of testing rather than the amount of arithmetic.

```
line
      Comment: Sparse U^tDU Factorization
 1    For k ← 1 to N do
 2        [For j ∈ {n: u_kn ≠ 0} do
 3            [m_kj ← a_kj];
 4         m_kk ← 1;
 5         d_kk ← a_kk;
 6         For i ∈ {n < k: u_nk ≠ 0} do
 7             [d_kk ← d_kk - m_ik^2/d_ii;
 8              m_ik ← m_ik/d_ii;
 9              For j ∈ {n > k: u_in ≠ 0} do
10                  [m_kj ← m_kj - m_ik*m_ij]]];
      Comment: Now M = U
      Comment: Backsolution to obtain x
11    For k ← 1 to N do
12        [y_k ← b_k];
13    For k ← 1 to N-1 do
14        [For j ∈ {n > k: u_kn ≠ 0} do
15            [y_j ← y_j - m_kj*y_k]];
16    For k ← 1 to N do
17        [z_k ← y_k/d_kk];
18    For k ← N to 1 do
19        [x_k ← z_k;
20         For j ∈ {n > k: u_kn ≠ 0} do
21             [x_k ← x_k - m_kj*x_j]];
```

ALGORITHM 2.2

To see how to solve these problems, let us assume that for each k, $1 \le k \le N$, we have

(i) the set ra_k of columns $j \ge k$ for which $a_{kj} \neq 0$;

(ii) the ordered set ru_k of columns $j > k$ for which $u_{kj} \neq 0$;

(iii) the set cu_k of rows $i < k$ for which $u_{ik} \neq 0$.

```
line
     Comment: Sparse U^t D U factorization
 1   For k ← 1 to N do
 2       [For j ∈ ru_k do
 3           [m_kj ← 0];
 4        For j ∈ {n ∈ ra_k; n > k} do
 5           [m_kj ← a_kj];
 6        m_kk ← 1;
 7        d_kk ← a_kk;
 8        For i ∈ cu_k do
 9           [d_kk ← d_kk - m_ik^2/d_ii;
10            m_ik ← m_ik/d_ii;
11            For j ∈ {n ∈ ru_i: n > k} do
12               [m_kj ← m_kj - m_ik*m_ij]]];
     Comment: Now M = U
     Comment: Backsolution to obtain x
13   For k ← 1 to N do
14       [y_k ← b_k];
15   For k ← 1 to N-1 do
16       [For j ∈ ru_k do
17           [y_j ← y_j - m_kj*y_k]];
18   For k ← 1 to N do
19       [z_k ← y_k/d_kk];
20   For k ← N to 1 do
21       [x_k ← z_k;
22        For j ∈ ru_k do
23           [x_k ← x_k - m_kj*x_j]];
```

ALGORITHM 2.3

We can then modify Algorithm 2.2 to obtain Algorithm 2.3 in which the only entries of M which are used are those corresponding to non-zeroes in A or U.

Ignoring the usually small amount of storage for $\underset{\sim}{x}$, $\underset{\sim}{y}$, $\underset{\sim}{b}$, and

D, the storage required for Algorithm 2.3 is just the storage needed for the entries of M which are actually used plus the storage needed for the sets $\{ra_k\}$, $\{ru_k\}$, and $\{cu_k\}$. The total amount can be shown to be proportional to the number of nonzeroes in U (cf. Sherman [8], Eisenstat and Sherman [5]). Furthermore, the algorithm includes no test operations, since the sets $\{ra_k\}$, $\{ru_k\}$, and $\{cu_k\}$ contain all of the information required to avoid operating on zeroes in M. Hence the running time of the algorithm is proportional to the number of arithmetic operations performed on nonzero operands.

The most important thing to note about the sets $\{ra_k\}$, $\{ru_k\}$, and $\{cu_k\}$ is that they describe the structure of A and U in terms of the positions of the nonzero entries. They do not depend at all on the values of the nonzero entries, and that fact makes it possible to obtain them quickly and without any testing.

3. Implementation

In this section we outline an actual implementation of sparse symmetric Gaussian elimination (cf. Eisenstat and Sherman [5] for further details). Following Chang [2] and Gustavson [6], we break the computation into three parts: symbolic factorization, numeric factorization, and backsolution. The symbolic factorization computes the structure of the rows of U from the structure of the rows of A. The numeric factorization uses the structure information to efficiently compute the U^tDU factorization of A. And the backsolution computes the solution

```
line
 1    For  k ← 1 to N  do
 2        [ℓ_k ← ∅;
 3         ru_k ← ∅];
 4    For  k ← 1 to N-1  do
      Comment: Form ru_k by set unions
 5        [For  j ∈ {n ∈ ra_k:  n > k}  do
 6            [ru_k ← ru_k ∪ {j} ];
 7         For  i ∈ ℓ_k  do
 8            [For  j ∈ {n ∈ ru_i:  n > k}  do
 9                [If  j ∉ ru_k  then
10                    [ru_k ← ru_k ∪ {j} ]]];
      Comment: ru_k is now complete, but unordered
11         m ← min {j: j ∈ ru_k ∪ {N+1}};
12         If  m < N+1  then
13            [ℓ_m ← ℓ_m ∪ {k} ]];
      Comment: Sort the sets ru_k with a lexicographic
               sort to obtain ordered sets
```

ALGORITHM 3.1

$\underset{\sim}{x}$ from the factorization and the right hand side $\underset{\sim}{b}$.

We split up the computation to gain flexibility. If several linear systems have identical coefficient matrices but different right hand sides, then only one symbolic factorization and one numeric factorization are needed; the different right hand sides require separate backsolutions. Similarly, only one symbolic factorization with separate numeric factorizations and backsolutions is needed to solve several systems in which the coefficient matrices have identical zero structures but different nonzero entries.

```
line
 1    For  k ← 1 to N  do
 2        [p_k ← ∅];
 3    For  k ← 1 to N  do
 4        [For  j ∈ ru_k  do
 5            [m_kj ← 0];
 6         For  j ∈ {n ∈ ra_k :  n > k}  do
 7            [m_kj ← a_kj];
 8         d_kk ← a_kk;
 9         For  i ∈ p_k  do
10            [d_kk ← d_kk - m_ij^2/d_ii;
11             m_ik ← m_ik/d_ii;
12             For  j ∈ {n ∈ ru_i :  n > k}  do
13                [m_kj ← m_kj - m_ik·m_ij];
14             p_k ← p_k - {i};
15             m ← min{{n ∈ ru_i :  n > k} ∪ {N+1}};
16             If  m < N+1  then
17                [p_m ← p_m ∪ {i} ]];
18         m ← min {ru_k ∪ {N+1}};
19         If  m < N+1  then
20            [p_m ← p_m ∪ {k} ]];
       Comment:  Now  M = U.
```

ALGORITHM 3.2

As in Section 2, ra_k denotes the set of column indices $j \geq k$ for which $a_{kj} \neq 0$, ru_k denotes the ordered set of column indices $j > k$ for which $u_{kj} \neq 0$, and cu_k denotes the set of row indices $i < k$ for which $u_{ik} \neq 0$. The sets $\{ra_k\}$ describe the structure of A and are input parameters. The symbolic factorization algorithm computes the sets $\{ru_k\}$ from the sets $\{ra_k\}$. The sets $\{cu_k\}$, which are needed only for the numeric factorization, are computed as

required.

Algorithm 3.1 performs the symbolic factorization of A. At the k-th step of the algorithm, an unordered form of ru_k is computed from ra_k and the sets $\{ru_i\}$ for $i < k$. An examination of Algorithm 2.3 shows that for $j > k$, $u_{kj} \neq 0$ if and only if either

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

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

Thus for $j > k$, $j \in ru_k$ if and only if either

(i) $j \in ra_k$ or

(ii) $j \in ru_i$ for some $i \in cu_k$.

However, it can be shown that not all rows $i \in cu_k$ need to be examined (cf. Rose and Tarjan [7], Sherman [8]). It suffices to examine just those rows $i \in cu_k$ for which k is the smallest column index in ru_i. Let ℓ_k be the set of rows $i \in cu_k$ which satisfy this condition. Then for $j > k$, $j \in ru_k$ if and only if either

(i) $j \in ra_k$ or

(ii) $j \in ru_i$ for some $i \in \ell_k$.

The sets $\{\ell_k\}$ are formed during the algorithm by adding each row to the proper set as soon as the structure of that row is computed. Since each row appears in at most one set, the total storage required for the sets $\{\ell_k\}$ is $O(N)$ locations.

The cost of the symbolic factorization algorithm is determined by the number of times that lines 7-10 must be executed and by the cost of the final sorting operation. The remaining portion of the algorithm can contribute only a small cost proportional to the number of nonzeroes in A (from lines 5-6). But each row i can be a member of at most one set ℓ_k, and lines 8-10 are executed just once for each $j \in ru_i$. Hence during the entire execution of Algorithm 3.1, lines 8-10 can be executed only once for each entry of the sets $\{ru_k\}$ combined (i.e. once for each nonzero in U). Furthermore, it is known (cf. Aho, Hopcroft, and Ullman [1], pp. 77-84) that sorting the sets $\{ru_k\}$ with a lexicographic sort also requires time proportional to the number of entries in the combined sets. Thus the total running time of Algorithm 3.1 is proportional to the number of nonzeroes in U (cf. Rose and Tarjan [7] or Sherman [8] for detailed proofs of this result).

The numeric U^tDU factorization of the matrix A is implemented in Algorithm 3.2. Basically, the algorithm is a straightforward implementation of the factorization portion of Algorithm 2.3. After the symbolic factorization, the sets $\{ra_k\}$ and $\{ru_k\}$ are available. The only difficulty is the computation of the sets $\{cu_k\}$, but these can be generated by examining the sets $\{ru_k\}$.

The sets $\{cu_k\}$ are generated progressively using sets $\{p_k\}$. During the k-th step we require that $p_k = cu_k$ and that for $j > k$, p_j contains those row indices $i \in cu_j$ for which j is the minimum column index in ru_i satisfying $j \geq k$. At any time, each row i is in at most one set p_k, so that the total storage required for the sets $\{p_k\}$

cu_1: -; cu_2: 1; cu_3: 2; cu_4: 1,2,3,; cu_5: 1,4

ru_1:	2,4,5	p_1:	-
ru_2:	3,4	p_2:	-
ru_3:	4	p_3:	-
ru_4:	5	p_4:	-
ru_5:	-	p_5:	-

(a)

ru_1:	(2,) 4,5	p_2:	1
ru_2:	3,4	p_3:	-
ru_3:	4	p_4:	-
ru_4:	5	p_5:	-
ru_5:	-		

(b)

---

cu_k: 2,4,5; cu_{k+1}: 2,3; cu_{k+2}: 1,2,4

ru_1:	2,...,k-1, (k+2,) ...	p_k:	2,4,5
ru_2:	3,..., (k,) k+1,k+2,...	p_{k+1}:	3
ru_3:	4,...,k-1, (k+1,) ...	p_{k+2}:	1
ru_4:	5,..., (k,) k+2,...		
ru_5:	6,..., (k)		

(c)

ru_1:	2,...,k-1, (k+2,) ...	p_{k+1}:	3,2
ru_2:	3,...,k, (k+1,) k+2,...	p_{k+2}:	1,4
ru_3:	4,...,k-1, (k+1,) ...		
ru_4:	5,...,k, (k+2,) ...		
ru_5:	6,...,k		

(d)

FIGURE 3.1

is $O(N)$ locations.

The updating of the sets $\{p_k\}$ is fairly complex, and it may best be illustrated inductively (see Figure 3.1). As shown in Figure 3.1a, all the sets $\{p_k\}$ are initially empty. Since cu_1 is also empty, $p_1 = cu_1$ during the first step. At the end of the first step, row 1 is added to p_j, where j is the minimum column index in ru_1 (see Figure 3.1b). Now if $cu_2 = 1$, then $2 \in ru_1$ and $p_2 = 1$. Otherwise both cu_2 and p_2 are empty. In either case, however, $p_2 = cu_2$, and the second step of the algorithm can be performed.

The more general situation during the k-th step of Algorithm 3.2 is illustrated in Figures 3.1c and 3.1d. At the beginning of the step (Figure 3.1c), $p_k = cu_k$, and for $j > k$, p_j contains those row indices $i \in cu_j$ for which j is the minimum column index in ru_i satisfying $j \geq k$. Hence $p_k = 2,4,5$, $p_{k+1} = 3$, and $p_{k+2} = 1$. In particular, p_{k+1} includes all the indices in cu_{k+1}, except for those which are also in cu_k. At the end of the k-th step, the sets $\{p_j\}$ for $j > k$ must be updated for the k+1-st step. This is done by moving each entry of p_k from p_k into the proper set p_j. For each $i \in p_k$, row i is added to p_j, where j is the minimum index in ru_i which satisfies $j > k$. If ru_i contains no such index, then row i is not added to any set (e.g. row 5 in Figure 3.1d). Finally, row k is added to the set p_j, where j is the smallest column index in ru_k. Figure 3.1d shows the sets $\{p_k\}$ which remain after the k-th step is completed. Now $p_{k+1} = cu_{k+1}$, so that the k+1-st step can be performed.

```
line
 1    For  k ← 1 to N  do
 2         [y_k ← b_k];
 3    For  k ← 1 to N-1  do
 4         [For  j ∈ ru_k  do
 5              [y_j ← y_j - u_kj·y_j]];
 6    For  k ← 1 to N  do
 7         [z_k ← y_k/d_kk];
 8    For  k ← N to 1  do
 9         [x_k ← z_k;
10          For  j ∈ ru_k  do
11              [x_k ← x_k - u_kj·x_j]];
```

ALGORITHM 3.3

As we pointed out in Section 2, Algorithm 2.3 runs in time proportional to the number of arithmetic operations which it performs. Algorithm 3.2 will also run in approximately the same amount of time, provided that the updating of the sets $\{p_k\}$ is a low order cost. By carefully examining the way in which this updating is done, we see that there is at most one updating operation for each entry of the combined sets $\{ru_k\}$. Therefore, updating the sets $\{p_k\}$ requires a total time proportional to the number of nonzeroes in U. Thus the algorithm, as a whole, runs in time proportional to the number of arithmetic operations performed, since that is usally much larger than the number of nonzeroes in U.

The last part of the implementation is the backsolution algorithm, Algorithm 3.3. This algorithm is exactly the same as the backsolution portion of Algorithm 2.3, and we will not discuss it at any length here.

We simply note that it performs one arithmetic operation for each entry of D and four arithmetic operations for each entry of U.

4. Storage of Sparse Matrices

In this section we describe the data structures used to efficiently store the matrices A and U (or equivalently, M).* The storage schemes which we present are designed for use with sparse Gaussian elimination, and they take advantage of the row-by-row nature of our algorithm.

In general, our algorithms operate on entire rows of A and M at once. Hence it is important to be able to easily access rows of the matrix rather than individual entries. For this reason, the storage schemes described here are row-oriented: i.e. all of the nonzero entries of a matrix row are stored consecutively. In order to identify the entries of a row, we need to know where the row starts, how long it is, and in what column each entry of the row lies. The extra storage required for this information is the overhead mentioned in the Introduction, and it is important to reduce it as much as possible.

Since A is symmetric, we need only store its upper triangle. We do this with the row-by-row storage scheme used in previous implementations of sparse symmetric Gaussian elimination (e.g. see Gustavson [6] and the references therein). As shown in Figure 4.1, the scheme requires three one-dimensional arrays: IA, JA, and A. The nonzero entries of

---

* The diagonal matrix D is stored in a one dimensional array D so that $D(k) = d_{kk}$.

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

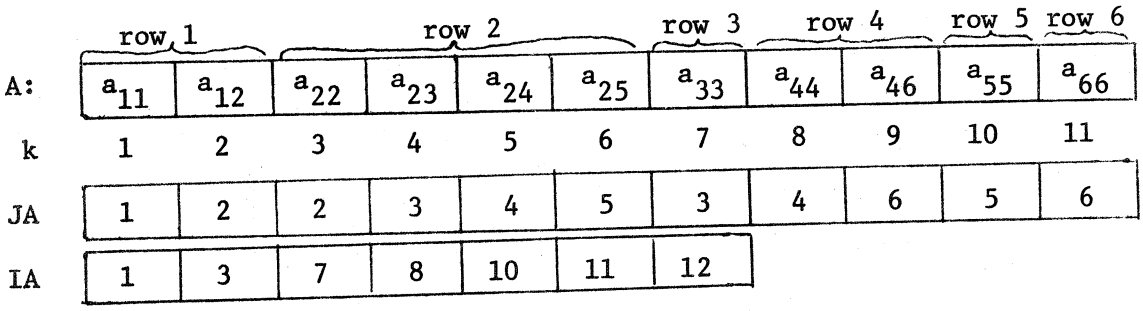


FIGURE 4.1

---

the upper triangle of A are stored row-by-row in the array A. The array JA contains the column indices which correspond to the entries of A: i.e. if $A(k) = a_{IJ}$, then JA(k) = J. Finally, IA contains N+1 entries which delimit the rows in A and JA. If a_{IJ} is the first (i.e. leftmost) entry of the I-th row, and if $A(k) = a_{IJ}$, then IA(I) = k. IA(N+1) is defined so that the nonzero entries of the I-th row are stored consecutively in A(IA(I)) - A(IA(I+1) - 1) and so that the column indices corresponding to the I-th row are stored consecutively in JA(IA(I)) - JA(IA(I+1) - 1). The set of column indices stored in JA for the I-th row of A is the set ra_I discussed in the last section.

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

(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	5	6			
IU:	1	2	5	7	9	10	10		
ISU:	1	2	3	5	6	6			

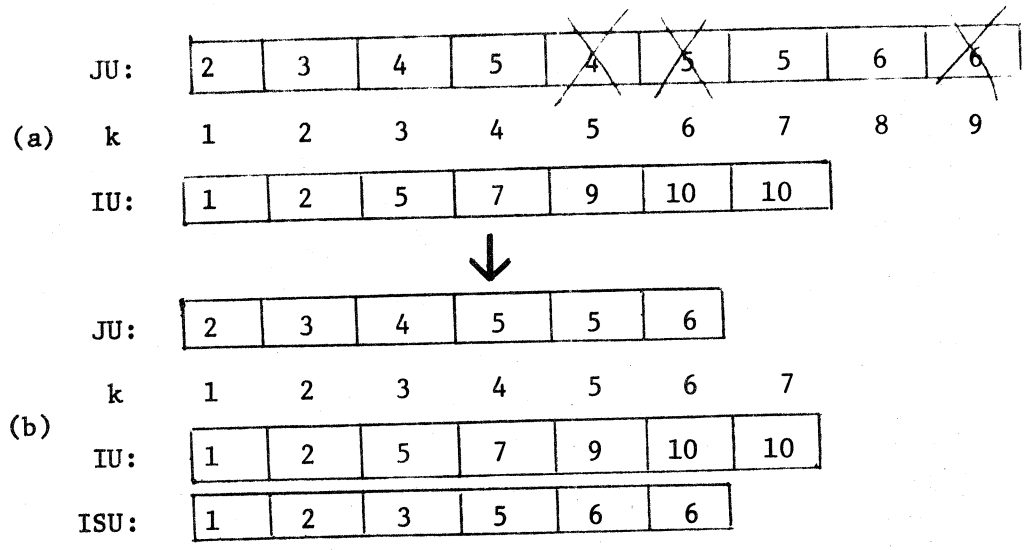
FIGURE 4.2

The overhead in the row-by-row storage scheme for A is the storage required for IA and JA. But since IA has N+1 entries and JA has one entry for each nonzero stored in A, the total storage is approximately twice the number of nonzeroes in the upper triangle of A.

The matrix U can also be stored with the row-by-row storage scheme just described (cf. Gustavson [6]), since only its upper triangle needs to be stored. This storage scheme requires the arrays IU, JU, and U corresponding to IA, JA, and A (see Figure 4.2a). However, using the row-by-row storage scheme ignores certain features of U which can be exploited to reduce the storage. Most obvious is the fact that all the diagonal entries of U are known to be 1, so that they need not be stored. Taking advantage of this leads to the storage scheme shown in Figure 4.2b in which the strict upper triangle of U is stored in row-by-row form.

An examination of Figure 4.2b reveals the fact that the column indices for certain rows of U in JU are actually consecutive subsets of the column indices for previous rows. For example, the column indices for row 3 are 4,5, while those for row 2 are 3,4,5. There is really no need to store the column indices for row 3 separately from those of row 2, provided that we know where to find them as a subset of the column indices for row 2.

Using this observation, we can compact the column indices in JU by deleting the column indices for row I if they appear as a consecutive subset of the column indices for some row J, with $J < I$. This



Locations of Column Indices

Row	Before Compaction (a)	After Compaction (b)
1	JU(1)	JU(1)
2	JU(2) - JU(4)	JU(2) - JU(4)
3	JU(5) - JU(6)	JU(3) - JU(4)
4	JU(7) - JU(8)	JU(5) - JU(6)
5	JU(9)	JU(6)
6	-	-

FIGURE 4.3

technique is illustrated in Figure 4.3. In order to find the column indices for any row in the compacted JU array, we must know where to look and how many column indices there are. We use the array ISU to locate the first column index for each row, and we determine the number of indices by using IU to compute the number of nonzeroes in the row. Thus the column indices for the I-th row are stored in

JU(ISU(I)) - JU(ISU(I) + IU(I+1) - IU(I) - 1), since there are IU(I+1) - IU(I) nonzeroes in the I-th row.

We can obtain the compacted row-by-row storage scheme for U by compacting the JU array used in the row-by-row storage scheme of the strict upper triangle of U. The scheme requires the four arrays IU, JU, ISU, and U, as shown in Figure 4.2c. U contains the nonzero entries of the strict upper triangle of U, stored row-by-row, and the N+1 entries of IU delimit the rows in U. JU is the compacted array of column indices, and the entries of ISU locate the first column index in JU for each row. The nonzeroes of the I-th row of U are stored in U(IU(I)) - U(IU(I+1) - 1), and the corresponding column indices are stored in JU(ISU(I)) - JU(ISU(I) + IU(I+1) - IU(I) - 1). The set of indices stored in JU for the I-th row of U is the ordered set ru_I defined in Section 2.

For the compacted row-by-row storage scheme, the overhead is the storage required for IU, JU, and ISU. It is not usually possible to determine a priori the amount of overhead, but it is often the case that it is less than the overhead would be with the ordinary row-by-row storage scheme. In fact, there are examples arising in the numerical solution of partial differential equations where it can be proved that U contains O(N log N) nonzero entries, while JU contains only O(N) column indices (cf. Sherman [8]).

5. Conclusion

In this paper we have presented an efficient implementation for sparse symmetric Gaussian elimination. The algorithms described here, along with the corresponding algorithms for nonsymmetric sparse Gaussian elimination, have been incorporated into a package of FORTRAN subroutines.* The subroutines in this package were used to perform some numerical experiments reported on in another paper by the authors in these proceedings (cf. Eisenstat, Schultz, and Sherman [4]). The results of the experiments support the view that sparse Gaussian elimination may be an effective technique for solving the linear systems which arise in numerical computation.

---

* Copies of the package may be obtained from the authors.

References

[1] A.V. Aho, J.E. Hopcroft, and J.D. Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley Publishing Co., Reading, Mass., 1974.

[2] A. Chang. Application of Sparse Matrix Methods in Electric Power System Analysis. In Willoughby, ed., *Sparse Matrix Proceedings*. IBM Research Report RA1, Yorktown Heights, New York, 1968, pp. 113-122.

[3] G. Dahlquist and A. Björck. *Numerical Methods*. Prentice-Hall, Englewood Cliffs, N.J., 1974.

[4] S.C. Eisenstat, M.H. Schultz, and A.H. Sherman. Application of Sparse Matrix Methods to Partial Differential Equations. In the Proceedings of the AICA International Symposium on Computer Methods for Partial Differential Equations, Bethlehem, Penn., 1975.

[5] S.C. Eisenstat and A.H. Sherman. Subroutines for the Efficient Implementation of Sparse Gaussian Elimination. To appear.

[6] F.G. Gustavson. Basic Techniques for Solving Sparse Systems of Linear Equations. In Rose and Willoughby, eds., *Sparse Matrices and Their Applications*. Plenum Press, New York, 1972, pp. 41-52.

[7] D.J. Rose and R.E. Tarjan. Algorithmic Aspects of Vertex Elimination on Graphs. Submitted to *SIAM Journal on Computing*, 1975.

[8] A. Sherman. Ph.D. dissertation. Department of Computer Science, Yale University, New Haven, Conn., 1975.

[9] D.M. Young. *Iterative Solution of Large Linear Systems*. Academic Press, New York, 1971.