Symbolic Factorization for Sparse Gaussian Elimination with Partial Pivoting

Alan George
Esmond Ng

Department of Computer Science
University of Waterloo
Waterloo, Ontario, CANADA

CS-84-43
November 1984

Symbolic Factorization for Sparse Gaussian Elimination with Partial Pivoting[†]

Alan George

Esmond Ng

Department of Computer Science
University of Waterloo
Waterloo, Ontario, CANADA

CS-84-43
November 1984

† Research was sponsored by the Canadian Natural Sciences and Engineering Research Council under Grant A8111, and was also sponsored by the Applied Mathematical Sciences Research Program, Office of Energy Research, U.S. Department of Energy under contract DE-AC05-84OR21400 with the Martin Marietta Energy Systems Inc., and by the U.S. Air Force Office of Scientific Research under contract AFOSR-ISSA-84-00056.

ABSTRACT

Let $Ax=b$ be a large sparse nonsingular system of linear equations to be solved using Gaussian elimination with partial pivoting. The factorization obtained can be expressed in the form $A=P_1M_1P_2M_2 \cdots P_{n-1}M_{n-1}U$, where P_k is an elementary permutation matrix reflecting the row interchange that occurs at step k during the factorization, M_k is a unit lower triangular matrix whose k-th column contains the multipliers, and U is an upper triangular matrix.

Consider the k-th step of the elimination. Suppose we replace the structure of row k of the partially reduced matrix by the union of the structures of those rows which are candidates for the pivot row and then perform symbolically Gaussian elimination without partial pivoting. Assume that this is done at each step k, and let $\bar{L}$ and $\bar{U}$ denote the resulting lower and upper triangular matrices respectively. Then the structures of $\bar{L}$ and $\bar{U}$ respectively contain the structures of $\sum_{k=1}^{n-1} M_k$ and U. This paper describes an algorithm which determines the structure of $\bar{L}$ and $\bar{U}$, and sets up an efficient data structure for them. Since the algorithm depends only on the structure of A, the data structure can be created in advance of the actual numerical computation, which can then be performed very efficiently using the fixed storage scheme. Although the data structure is more generous than it needs to be for any *specific* sequence $P_1, P_2, \cdots, P_{n-1}$, experiments indicate that the approach is competitive or superior to conventional methods. Another important point is that the storage scheme is large enough to accommodate the QR factorization of A, so it is also useful in the context of computing a sparse orthogonal decomposition of A. The algorithm is shown to execute in time bounded by $|\bar{L}|+|\bar{U}|$, where $|M|$ denotes the number of nonzeros in the matrix M.

1. Introduction

Let A be an $n \times n$ nonsingular matrix and b be an n-vector, and consider the problem of finding an n-vector x such that

$$Ax = b \quad .$$

A standard approach for solving the problem involves reducing A to upper triangular form using elementary row eliminations (i.e., Gaussian elimination). In order to maintain numerical stability, one may have to interchange rows at each step of the elimination process (i.e., use partial pivoting). Thus, we may express the elimination as follows:

$$A = P_1 M_1 P_2 M_2 \cdots P_{n-1} M_{n-1} U \quad ,$$

where P_k is an $n \times n$ elementary permutation matrix corresponding to the row interchange at step k, M_k is an $n \times n$ unit lower triangular matrix whose k-th column contains the multipliers used at the k-th step, and U is an $n \times n$ upper triangular matrix. When A is sparse, it is well known that fill-in occurs during the triangular decomposition. That is, there may be more nonzeros in $\sum_{k=1}^{n-1} M_k$ and U than in A. Our objective is to develop a scheme that can create from the ***structure*** of A and ***prior to*** the numerical decomposition a data structure which can accommodate all the nonzeros in $\sum_{k=1}^{n-1} M_k$ and U. This process is known as *symbolic factorization*. Note that for sparse Gaussian elimination with partial pivoting, the positions of nonzeros in the triangular factors M_k and U depend not only on the structure of the matrix A, but also depend on the row interchanges. Since the row interchanges are not known until the numerical decomposition is performed, the symbolic factorization process must be able to generate a data structure that is large enough to accommodate the nonzeros for ***any*** possible (and valid) sequence of elementary row permutations $P_1, P_2, \cdots, P_{n-1}$.

There are important reasons why it is desirable to perform such a symbolic factorization. First, since a symbolic factorization produces a data structure that exploits the sparsity of the triangular factors, the numerical decomposition can be performed using a ***static*** storage scheme. There is no need to perform storage allocations for the fill-in during the numerical computation. This reduces both storage and execution time overhead for the numerical phase. Second, we obtain from the symbolic factorization a bound on the amount of space we need in order to solve the linear system. This immediately tells us if the numerical computation is feasible. (Of course, this is important only if the symbolic factorization can be performed cheaply both in terms of storage and execution time and if the bound on space is reasonably tight.)

In [6] we have proposed an implementation of Gaussian elimination with partial pivoting for sparse matrices using a static storage scheme. The approach is based on the following results. Assume the diagonal elements of A are nonzero. Then it can be shown that the ***structure*** of the Cholesky factor L_C of A^TA contains the structure of U^T. That is, if U_{ij} is (structurally) nonzero, then $(L_C)_{ji}$ will also be (structurally) nonzero. Furthermore, the structure of the k-th column of L_C contains the structure of the k-th column of M_k. These results are ***independent*** of the choice of the pivot row at each step. Thus we can use the structures of L_C and L_C^T to ***bound*** the structures of $\sum_{k=1}^{n-1} M_k$ and U respectively. The advantage of this approach is that if A^TA is sparse (and this is the case in some classes of problems), the structure of the Cholesky factor L_C of A^TA can be determined from that of A^TA efficiently [4], and hence a data structure which exploits the sparsity of L_C and L_C^T can be set up. Thus this provides an efficient symbolic factorization scheme for sparse Gaussian elimination with partial pivoting. Notice that permuting the columns of A corresponds to ordering the rows and columns of A^TA symmetrically. It is well known that for sparse A^TA, the sparsity of the Cholesky factor L_C depends very much on the choice of this symmetric ordering. Thus we may choose a column ordering for A so as to reduce the amount of fill-in in L_C. Numerical experiments have shown that an implementation using the static storage scheme described above, together with a good column ordering for A, is competitive with existing methods for solving certain classes of sparse systems of linear equations [6].

However, in the approach described above, it is often true that the structure of L_C substantially *overestimates* the structure of $\sum_{k=1}^{n-1} M_k$ or U^T. The objective of this paper is to propose a better approach for predicting where nonzeros will appear in sparse Gaussian elimination with partial pivoting, and to present a symbolic factorization algorithm that will generate a tighter data structure which exploits the sparsity of the triangular factors. The basic idea is to allocate space for nonzeros that would be introduced by all possible pivotal sequences that could occur when Gaussian elimination with partial pivoting is applied to a matrix having the structure of A.

An outline of the paper is as follows. In Section 2, the new approach for predicting fill-in and a symbolic factorization algorithm are described. The complexity of the algorithm is also considered. Some enhancements to the algorithm are presented in Section 3, and in Section 4, some numerical experiments and comparisons are provided. Finally, some concluding remarks and open problems are given Section 5.

2. A symbolic factorization algorithm

In the following discussion, it is convenient to assume that the diagonal elements of the $n \times n$ matrix A are nonzero (in which case the matrix A is said to have a *zero-free* diagonal). If A is nonsingular and does not have a zero-free diagonal, it is always possible to permute the rows (or columns) of A so that the permuted matrix has a zero-free diagonal [1]. We will use the notation $Nonz(B)$ to denote the structure of a matrix B; that is,

$$Nonz(B) = \{ (i,j) \mid B_{ij} \neq 0 \} \quad .$$

When M is a matrix, $|M|$ denotes the number of nonzeros in M, and when M is a set, $|M|$ denotes the number of elements in M.

Now partition A into

$$A = \begin{pmatrix} \hat{\alpha} & \hat{u}^T \\ \hat{v} & \hat{B} \end{pmatrix} \quad ,$$

and consider applying one step of Gaussian elimination to A with partial pivoting.

$$\begin{aligned} P_1^T A &= P_1^T \begin{pmatrix} \hat{\alpha} & \hat{u}^T \\ \hat{v} & \hat{B} \end{pmatrix} = \begin{pmatrix} \alpha & u^T \\ v & B \end{pmatrix} \\ &= \begin{pmatrix} 1 & 0 \\ v/\alpha & I \end{pmatrix} \begin{pmatrix} \alpha & u^T \\ 0 & B - vu^T/\alpha \end{pmatrix} \\ &= \begin{pmatrix} 1 & 0 \\ l & I \end{pmatrix} \begin{pmatrix} \alpha & u^T \\ 0 & A_1 \end{pmatrix} \end{aligned}$$

Our goal is to be able to allocate space for the nonzeros in l, u and A_1, *regardless of the choice of the pivot row* (that is, the choice of P_1). Our approach is based on the observation that only the first row of A and rows j of A, where $\hat{v}_j \neq 0$, are involved in the elimination. For convenience, row 1 and rows j of A, where $\hat{v}_j \neq 0$, are referred to as *candidate pivot rows.* Thus only the structures of the candidate pivot rows may be affected after the first step of Gaussian elimination with partial pivoting. Moreover, the new structures of these rows depend only on their original structures. In fact, assuming structural and numerical cancellations do not occur, the new structure of each of these rows ***must*** be contained in the ***union*** of the structures of all the candidate pivot rows.

The discussion above also explains why we prefer $\hat{\alpha}$ to be nonzero. Suppose $\hat{\alpha}$ is zero. Then we know that the structure of $\hat{u}^T$ is not affected during the elimination except that it is moved to another row in the matrix. However, in order to have sufficient space to store the nonzeros, we have to treat the row $(\hat{\alpha} \;\; \hat{u}^T)$ as a candidate pivot row as well. Thus the prediction of fill-in may be too generous. This problem is eliminated if we arrange the rows of A so that $\hat{\alpha}$ is nonzero.

Assuming structural and numerical cancellations do not occur, the union of the structures of the candidate pivot rows is identical to the structure of

$$\bar{u}^T = \hat{u}^T + \hat{v}^T\hat{B} \quad .$$

Hence the discussion above can be summarized as follows.

$$\begin{aligned} Nonz(l) &= Nonz(\hat{v}) \\ Nonz(u^T) &\subseteq Nonz(\hat{u}^T + \hat{v}^T\hat{B}) \\ Nonz(A_1) &\subseteq Nonz(\hat{B} - \hat{v}(\hat{u}^T + \hat{v}^T\hat{B})) = Nonz(\hat{B} - \hat{v}(\hat{u}^T + \hat{v}^T\hat{B})/\hat{\alpha}) \end{aligned}$$

It is convenient to denote $\hat{v}$ by $\bar{l}$, and $\hat{B}-\hat{v}(\hat{u}^T+\hat{v}^T\hat{B})/\hat{\alpha}$ by $\bar{A}_1$. We should emphasize that the right hand sides, which can be computed easily from the structure of A, are *independent* of the choice of the actual pivot row (i.e., the choice of P_1) even though the left hand sides are. Hence we can use the structures of $\bar{l}$, $\bar{u}$ and $\bar{A}_1$ to *bound* those of l, u and A_1 respectively, *regardless of the choice of* P_1.

It is interesting to note that the decomposition

$$\begin{pmatrix} 1 & 0 \\ \bar{l} & I \end{pmatrix} \begin{pmatrix} \hat{\alpha} & \bar{u}^T \\ 0 & \bar{A}_1 \end{pmatrix}$$

can be obtained by applying one step of Gaussian elimination without row interchanges to the following $n \times n$ matrix $\bar{A}$.

$$\bar{A} = \begin{pmatrix} \hat{\alpha} & \hat{u}^T+\hat{v}^T\hat{B} \\ \hat{v} & \hat{B} \end{pmatrix}$$

Just in terms of structures, what we have done effectively is to replace the first row of A by the union of the structures of the candidate pivot rows and then apply one step of Gaussian elimination without partial pivoting.

Since $\hat{B}$ has a zero-free diagonal, $\bar{A}_1=\hat{B}-\hat{v}(\hat{u}^T+\hat{v}^T\hat{B})/\hat{\alpha}$ must also have a zero-free diagonal under the assumption that exact cancellation does not occur. Thus the argument and idea above can be applied recursively to A_1 and $\bar{A}_1$, yielding a procedure for generating a sequence of vectors whose structures can be used to bound the structures of the triangular factors (M_k and U). As we are going to see later, this approach often generates a storage scheme which is much tighter than that provided by the method in [6].

An important advantage of this new approach is that at each step of the symbolic factorization, all we need is the ability to identify the candidate pivot rows and their structures, and the ability to update the structures of these candidate pivot rows by the union of their original structures. Notice that we do not have to be concerned with the actual numerical values or the actual row interchanges that may occur during the numerical factorization. Thus the symbolic factorization procedure can be used to set up a static data structure in advance of the numerical computation, which can then be performed using the fixed storage scheme.

The success of the symbolic factorization procedure described above relies heavily on the efficiency of identifying the candidate pivot rows and updating their structures at each step. As we are going to show below, these can be done very easily and efficiently. The symbolic factorization procedure can be implemented without even modifying the structure of A! Consider the first step of the symbolic factorization. The structure of row j $(j>1)$ of $\bar{A}_1$ is either unchanged (if the row is not a candidate pivot row and is not involved in the elimination process), or the same as that of $\bar{u}^T$ (if the row is a candidate pivot row and is involved in the elimination process). Thus, all we need is an indicator $mask(j)$ for each row j that tells us whether row j is involved in step 1. In subsequent steps, if $mask(j)$ is set, then it means that the structure of row j has been used and its new structure should be obtained from (in this case) $\bar{u}^T$.

The other information we have to maintain is when the new structure of the modified rows will be used again and where the new structure can be found. Suppose $\bar{l}_{\sigma(1)}$, $\bar{l}_{\sigma(2)}$, $\cdots$, $\bar{l}_{\sigma(p)}$ are nonzero, and $\bar{u}_{\rho(1)}$, $\bar{u}_{\rho(2)}$, $\cdots$, $\bar{u}_{\rho(q)}$ are nonzero. Since the diagonal elements of A (and hence of $\bar{A}$) are assumed to be nonzero, $A_{\sigma(1),\sigma(1)}\neq 0$, $A_{\rho(1),\rho(1)}\neq 0$, and $\rho(1)\leq\sigma(1)$. Note that A_{ij} must be zero for $2\leq j<\rho(1)$, $i=\sigma(1)$, $\sigma(2)$, $\cdots$, $\sigma(p)$. See Figure 2.1 for an illustration. Thus the new structures of the modified rows will be used again (at the earliest) at step $\rho(1)$. We should emphasize that the new structures are identical to that of $\bar{u}^T$. Also, at step $\rho(1)$, the candidate pivot rows will include those (whose row indices are greater than $\rho(1)$) of step 1, but the candidate pivot rows of step 1 could be conveniently found by examining the structure of $\bar{l}$. To summarize, the only other information we have to record is the fact that at step $\rho(1)$, we have to look at $\bar{l}$ and $\bar{u}$ to obtain the correct structure of the modified matrix. In the algorithm described below, we have used a set $S_{\rho(1)}$ for this purpose. After the first step, we have $S_{\rho(1)}=\{1\}$.

At this point, it is appropriate to present the entire symbolic factorization algorithm which is a generalization of what we have just described. The structure of A is represented by two sets.

(1) C_j contains the structure of column j of A.

$$C_j = \{ i \mid A_{ij}\neq 0 \}$$

(2) R_i contains the structure of row i of A.

$$R_i = \{ j \mid A_{ij}\neq 0 \}$$

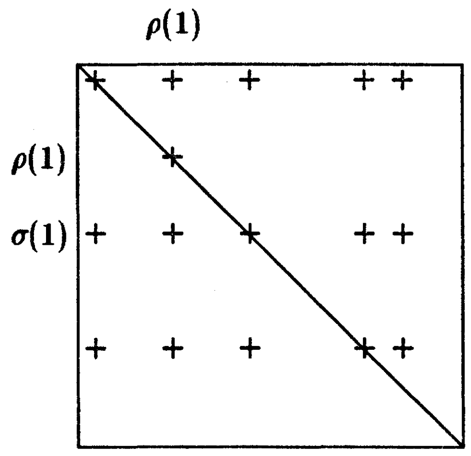

Figure 2.1: Position of nonzeros after the first step of symbolic factorization.

Similarly, the structures of the sequence of vectors obtained from the symbolic factorization algorithm are represented by sets.

(1) $\bar{L}_k$ contains the structure of a column vector that bounds the structure of column k of M_k.

(2) $\bar{U}_k$ contains the structure of row vector that bounds the structure of row k of U.

The algorithm

(1) Global initialization

(1.1) Set $mask(k)=0$, for $k=1,2,\cdots,n$ to denote that no rows have been considered yet.

(1.2) Set $S_k=\varnothing$, for $k=1,2,\cdots,n$ to denote that step k has not yet been influenced by any previous steps.

(2) For $k=1,2,\cdots,n$, do the following:

(2.1) Local initialization: Set $\bar{L}_k=\varnothing$ and $\bar{U}_k=\varnothing$.

(2.2) Obtain structures of those unprocessed rows:
For $i \in C_k$, if $mask(i)=0$, then set $\bar{L}_k=\bar{L}_k \cup \{i\}$ and $\bar{U}_k=\bar{U}_k \cup R_i$, and set $mask(i)=1$ (to indicate that row i has been looked at already).

(2.3) If $S_k=\varnothing$, then go to Step (2.5). (Step k is not influenced by any previous steps.)

(2.4) If $S_k \neq \varnothing$, then $\bar{L}_k$ and $\bar{U}_k$ should include $\bar{L}_i$ and $\bar{U}_i$ respectively, for $i \in S_k$. Hence, for $i \in S_k$, collect influence caused by step i:

$$\bar{L}_k = \bar{L}_k \cup \bar{L}_i - \{1, \cdots, k-1\}$$

$$\bar{U}_k = \bar{U}_k \cup \bar{U}_i - \{1, \cdots, k-1\}$$

(2.5) Update S:
Let $\sigma = \min\{t \mid t \in \bar{L}_k - \{k\}\}$ and $\rho = \min\{t \mid t \in \bar{U}_k - \{k\}\}$. Since $\rho \leq \sigma$, the structure of $\bar{L}_k$ and $\bar{U}_k$ will influence the determination of the structures at step ρ (if $|\bar{L}_k| > 1$). Hence set $S_\rho = S_\rho \cup \{k\}$ if $|\bar{L}_k| > 1$.

The complexity of the algorithm is linear in the number of elements in $\bigcup_i \bar{L}_i$ and $\bigcup_i \bar{U}_i$ as the following discussion shows. Since each i, $i = 1, 2 \cdots, n$, will be included in at most one S_ρ, $S_\rho \cap S_{\rho'} = \varnothing$, for $\rho \neq \rho'$. Thus we access the elements of $\bar{L}_i$ and $\bar{U}_i$ at most once for each i (see Step 2.4). Furthermore, we access the elements of C_i and R_i at most once in this algorithm (see Step 2.2). Hence the complexity of the algorithm is

$$O(|A|, \sum_{k=1}^{n-1} |\bar{L}_k| + |\bar{U}_k|) \quad .$$

The overhead in storage is very small in this algorithm too. There are n elements in *mask* for marking the rows. Since at step k, k is included in at most one set S_ρ, $\sum_i |S_i| \leq n$.

For convenience, we define a lower triangular matrix $\bar{L}$ so that $\bar{L}_{ij} \neq 0$ if $i \in \bar{L}_j$ and $\bar{L}_{ij} = 0$ otherwise. Similarly, let $\bar{U}$ be an upper triangular matrix such that $\bar{U}_{ij} \neq 0$ if $j \in \bar{U}_i$ and $\bar{U}_{ij} = 0$ otherwise. Hence,

$$Nonz(U) \subseteq Nonz(\bar{U})$$

and

$$Nonz(\sum_{k=1}^{n-1} M_k) \subseteq Nonz(\bar{L}) \quad .$$

In other words, a static storage scheme that exploits the sparsity of $\bar{L}$ and $\bar{U}$ is always large enough to accommodate the fill-in in M_k and U, regardless of the choice pivot rows.

It is interesting to note that the static data structure obtained from the symbolic factorization algorithm not only has enough space to accommodate the fill-in created in sparse Gaussian elimination with partial pivoting, but also has enough space to accommodate fill-in produced in computing an orthogonal decomposition of A using

Householder transformations. Partition A into

$$A = \begin{pmatrix} \hat{\alpha} & \hat{u}^T \\ \hat{v} & \hat{B} \end{pmatrix} .$$

Suppose H is an $n \times n$ Householder transformation that reduces $\begin{pmatrix} \hat{\alpha} \\ \hat{v} \end{pmatrix}$ to $\begin{pmatrix} \overline{\alpha} \\ 0 \end{pmatrix}$. As was pointed out in [5], H has the form

$$H = I - \frac{1}{\pi} \begin{pmatrix} \beta \\ v \end{pmatrix} \begin{pmatrix} \beta & v^T \end{pmatrix} ,$$

where $Nonz(v) = Nonz(\hat{v})$. Thus,

$$\begin{aligned} HA &= \left(I - \frac{1}{\pi} \begin{pmatrix} \beta \\ v \end{pmatrix} \begin{pmatrix} \beta & v^T \end{pmatrix} \right) \begin{pmatrix} \hat{\alpha} & \hat{u}^T \\ \hat{v} & \hat{B} \end{pmatrix} \\ &= \begin{pmatrix} \overline{\alpha} & \hat{u}^T - \frac{1}{\pi}(\beta^2 \hat{u}^T + \beta v^T \hat{B}) \\ 0 & \hat{B} - \frac{1}{\pi}(\beta v \hat{u}^T + v v^T \hat{B}) \end{pmatrix} . \end{aligned}$$

Hence

$$Nonz(v) = Nonz(\hat{v}) = Nonz(\bar{l}) ,$$

$$Nonz(\hat{u}^T - \frac{1}{\pi}(\beta^2 \hat{u}^T + \beta v^T \hat{B})) = Nonz(\hat{u}^T + \hat{v}^T \hat{B}) = Nonz(\overline{u}^T) ,$$

and

$$Nonz(\hat{B} - \frac{1}{\pi}(\beta v \hat{u}^T + v v^T \hat{B})) = Nonz(\hat{B} - \hat{v}(\hat{u}^T + \hat{v}^T \hat{B})/\hat{\alpha}) = Nonz(\overline{A}_1) .$$

The argument can again be applied recursively to $\overline{A}_1$ and to $\hat{B} - \frac{1}{\pi}(\beta v \hat{u}^T + v v^T \hat{B})$. Thus, the symbolic factorization algorithm described above can be used to generate a static storage scheme for sparse orthogonal decomposition using Householder transformations. Note that both the upper triangular matrix and the Householder transformations can be saved in the data structure.

Although the symbolic factorization algorithm has been described in terms of $n \times n$ matrices, with some slight modifications it can also be applied to rectangular matrices. The lower trapezoidal portion of $\overline{L}$ and the upper trapezoidal portion of $\overline{U}$ are stored respectively by columns and rows. If the matrix A is $m \times n$, then the ***mask*** vector will be of size m and the number of elements in the sets S_i will be $\min\{m,n\}$.

3. Enhancements

3.1. Subscription compression

From the sets $\{\bar{L}_i\}$ and $\{\bar{U}_i\}$, we can generate a storage scheme for storing the nonzeros of $\bar{L}$ and $\bar{U}$ (and hence the nonzeros of M_k and U). For example, consider the upper triangular matrix $\bar{U}$. One way to represent $\bar{U}$ is to store the nonzeros row by row in a floating-point array UNZ, since the structure of $\bar{U}$ is generated row by row. Thus we need an integer array XUNZ to store the beginning positions in UNZ of the nonzeros in the rows of $\bar{U}$. More specifically, the nonzeros of row i of $\bar{U}$ are found in UNZ(p), for p=XUNZ(i), XUNZ(i)+1, $\cdots$, XUNZ(i+1)−1. (It is assumed that XUNZ($n+1$)=$1+\sum_i |\bar{U}_i|$.) It is also necessary to store the column subscripts of the numbers in UNZ. Hence we need an additional integer array SUBU where SUBU(p) contains the column index of the nonzero stored in UNZ(p). The lower triangular matrix $\bar{L}$ is represented column by column using a similar storage scheme. We refer to this storage scheme as the ***uncompressed storage scheme.*** Thus, if we use this uncompressed storage scheme, we need approximately $\sum_i(|\bar{L}_i| + |\bar{U}_i|)$ floating-point locations for the numerical values, $\sum_i(|\bar{L}_i| + |\bar{U}_i|)$ integer locations for the subscripts, and $2n+2$ integer locations for the pointers. If an integer location and a floating-point location have the same number of bits, then the total number of locations required by the uncompressed storage scheme is approximately

$$2\sum_i(|\bar{L}_i| + |\bar{U}_i|) \quad .$$

It has been observed from experiments that very often the set of subscripts of a row (say row i) of $\bar{U}$ is a subsequence of the subscripts of a previous row (in particular, row $i-1$). This is also true for the columns of the lower triangular matrix $\bar{L}$. Thus, it makes sense to remove these redundancies using a process we call ***subscript compression,*** so that the total number of subscripts that have to be stored is smaller than $\sum_i(|\bar{L}_i| + |\bar{U}_i|)$. The use of subscript compression is common in storage schemes for solving sparse symmetric positive definite systems [8].

On the other hand, if the subscripts are compressed, there is no longer a one-to-one correspondence between the numerical values in, say UNZ, and the subscripts in SUBU. We need additional information in order to be able to retrieve the subscripts. For example, consider the upper triangular matrix $\bar{U}$ again. We now need an extra integer array XSUBU of size n with XSUBU(i) indicating where we can find the beginning of the subscript sequence for the nonzeros in row i. That is, the nonzeros of row i are stored in UNZ(p), for p=XUNZ(i), XUNZ(i)+1, $\cdots$, XUNZ(i+1)−1. The corresponding column subscripts are stored in SUBU(q), for q=XSUBU(i), XSUBU(i)+1, $\cdots$. Of course, the length of the subscript sequence is simply given by XUNZ($i+1$)−XUNZ(i). We refer to this storage scheme as the *compressed storage scheme.* As we will see in the numerical experiments presented in the next section, the additional storage required by XSUBL and XSUBU is usually more than offset by the saving in storage for the subscripts. However, since additional indexing is needed in order to retrieve the nonzeros of a row of $\bar{U}$ or a column of $\bar{L}$, the numerical computation phase using the compressed storage scheme is expected to be slower than that using the uncompressed storage scheme.

There are two techniques employed in subscript compression. First, after we have determined the structure of the k-th row of $\bar{U}$ (or k-th column of $\bar{L}$), we can compare it with that of the $(k-1)$-st row of $\bar{U}$ (or $(k-1)$-st column of $\bar{L}$) to find out if the "tail" of the latter sequence is the same as the "head" of the former one. If they are the same, then we do not have to store the "head" of the subscript sequence for the current row (or column).

Another observation not only allows us to compress the subscripts, but also improves the performance of the symbolic factorization algorithm in terms of the execution time. Very often, some of the sets S_i have only one element and there are no unprocessed rows at step i. (In particular, after the symbolic factorization algorithm has gone through a number of steps, say $k<n$, when all the n rows of A have been processed, the situation above occurs very frequently.) Suppose $S_i=\{l\}$. This simply means that the structures of $\bar{L}_i$ and $\bar{U}_i$ are determined entirely by $\bar{L}_l-\{i\}$ and $\bar{U}_l-\{i\}$ respectively. Hence we do not have to access the elements of $\bar{L}_l$ and $\bar{U}_l$ at all. This allows us to compress the subscripts and it would also reduce the execution time of the symbolic factorization algorithm.

The techniques described above have been incorporated into our implementation, and numerical experiments have indicated that there is indeed a significant reduction in both the speed of the symbolic factorization algorithm and the total number of subscripts that have to be stored.

3.2. Column ordering

At each step of the symbolic factorization algorithm, we replace the structures of the candidate pivot rows by the union of their original structures. This operation is independent of the initial row ordering. That is, the number of nonzeros ($|\bar{L}|$ and $|\bar{U}|$) determined by the symbolic factorization algorithm is constant, regardless of the choice of the initial row ordering, as long as this row ordering preserves the zero-free diagonal. (However, the choice of row ordering may have an effect on the extent to which subscript compression can be achieved.)

Column orderings, on the hand, may affect the sparsity of the triangular matrices $\bar{L}$ and $\bar{U}$ which is indirectly related to the sparsity of the triangular factors $\sum_{k=1}^{n-1} M_k$ and

U. (Again, the column ordering has to preserve the zero-free diagonal.) Consider the two matrices A and B below, where B is obtained by interchanging the first and the last columns of A.

$$A = \begin{bmatrix} \times & \times & \times & \times & \times \\ & \times & & & \times \\ & & \times & & \times \\ & & & \times & \times \\ \times & & & & \times \end{bmatrix} \qquad B = \begin{bmatrix} \times & \times & \times & \times & \times \\ \times & \times & & & \\ \times & & \times & & \\ \times & & & \times & \\ \times & & & & \times \end{bmatrix}$$

The union of the structures of the matrices $\bar{L}$ and $\bar{U}$ in each case is displayed below.

$$\bar{L}_A + \bar{U}_A \; : \; \begin{bmatrix} \times & \times & \times & \times & \times \\ & \times & \times & \times & \times \\ & & \times & \times & \times \\ & & & \times & \times \\ \times & \times & \times & \times & \times \end{bmatrix} \qquad \bar{L}_B + \bar{U}_B \; : \; \begin{bmatrix} \times & \times & \times & \times & \times \\ \times & \times & \times & \times & \times \\ \times & \times & \times & \times & \times \\ \times & \times & \times & \times & \times \\ \times & \times & \times & \times & \times \end{bmatrix}$$

Hence it is important to choose a column ordering for A so that the triangular matrices $\bar{L}$ and $\bar{U}$ are sparse.

In our experiments, since we want to compare the storage scheme generated by the new symbolic factorization algorithm with that created using the approach described in [6], we have decided to use a column ordering that is in general effective for the method in [6]. Recall that in [6], one uses the structures of the Cholesky factors of A^TA to bound the structures of the triangular factors $\sum_{k=1}^{n-1} M_k$ and U of A. Hence it is the sparsity of the Cholesky factors of A^TA that has to be preserved. It is well known that, when A^TA is sparse, a minimum degree ordering is an effective ordering in reducing fill-in in its Cholesky factors [7]. Hence we use that ordering to permute the columns of A before we apply our symbolic factorization algorithm to A.

A natural way to order the columns of A is as follows. Consider the first step of the symbolic factorization algorithm. Since we have to replace the structures of the candidate pivot rows by the union of their original structures, a $p \times q$ dense submatrix is introduced into the matrix, where p is the number of candidate pivot rows and q is the number of nonzeros in the union of these candidate pivot rows. By rearranging the columns of A, p and q will be changed. Thus we could permute the columns (and rows) before we carry out this step of symbolic factorization so that p, q or even the product pq is small (or minimized). Efficient implementation of these ideas is under investigation.

3.3. Dense rows and dense columns

In some applications, the resulting matrix may have a few rows or columns that are relatively dense. In the method proposed in [6], since we are working with A^TA, dense rows of A form large dense submatrices in A^TA and this is not desirable. In our present approach, the effect of having some dense rows in A may not be very severe, as long as when one of these dense rows becomes a candidate pivot row, the number of candidate pivot rows at that time is small. However, since we order the rows and columns of A^TA to obtain a column ordering for A, these dense rows may affect the quality of the column ordering because the sparsity of the remaining rows of A may be lost when we form A^TA. One technique which we have found useful is to remove these dense rows from A when we order the columns of A and then to put them back into A when we carry out symbolic factorization. Some of the numerical experiments presented in the next section will illustrate the improvement that may be achieved.

The effect of dense columns is not as clear as that of dense rows. Suppose the last column of A is dense. It should be clear that such a dense column should have no effect on the sparsity of $\bar{L}$ and $\bar{U}$. On the other hand, suppose the first column of A is dense. Then the number of candidate pivot rows at the first step of symbolic factorization is large and the union of these candidate pivot rows may be dense. Even if the union is sparse, more nonzeros will likely be added to this union as the symbolic factorization algorithm proceeds. In other words, it is desirable to have any dense columns appear last.

Instead of working with A^TA and A, we may work with AA^T and A^T, since a triangular factorization of A is as useful as a triangular factorization of A^T. Dense rows and dense columns in A now become respectively dense columns and dense rows in A^T. In terms of applying the symbolic factorization algorithm to A and A^T, the effect of having dense rows and dense columns in each case is in general different. Thus, depending on how severe the fill-in is, we may prefer to work with A^T rather than A. As an example, suppose all the rows of A are sparse, but the first column is dense. Then applying the symbolic factorization algorithm to A may result in a lot of fill-in in $\bar{L}$ and $\bar{U}$. However, if we work with A^T, then the first row of A^T is now dense and A^T has no dense columns. We can apply the symbolic factorization algorithm to A^T. Together with the row withholding strategy in determining a good column ordering, this latter approach may generate a sparse $\bar{L}$ and $\bar{U}$.

Clearly, the key problem is to decide when a row or column should be considered dense. This is an interesting topic for future research. A related question is how to decide whether to work with A or A^T. The answer is probably problem dependent, and needs further study.

4. Numerical experiments

In this section, we present some numerical experiments to illustrate the efficiency of our symbolic factorization algorithm and to compare the performance with the method proposed in [6]. The experiments involved the solution of sparse systems of linear equations

$$P^T(QA)PP^Tx = P^T(Qb) \quad ,$$

where A and b are respectively the coefficient matrix and the right hand side vector, Q is a permutation matrix chosen so that QA has a zero-free diagonal, and P is a permutation matrix corresponding to a minimum degree ordering of A^TA. (We premultiply QAP by P^T since we want to preserve the zero-free diagonal.) Symbolic factorization is applied to the matrix $B=P^TQAP$ to create a data structure for the numerical decomposition of B. The triangular factors M_k and U, which are computed using Gaussian elimination with partial pivoting, are stored in the fixed storage scheme.

All experiments were carried out on a DEC VAX 11/780 computer. The programs were written in ANSI standard FORTRAN using single precision floating-point arithmetic, and were compiled using the Berkeley FORTRAN 77 compiler. In the discussion below, storage requirements are expressed in terms of number of storage locations required and execution times are in seconds. (Note that one floating-point location and on integer location have the same number of bits.)

There are three sets of test problems in our experiments. These problems arise from typical scientific and engineering applications. The first set consists of nine finite element problems which are described in [3]. The numerical values were generated using a uniform random number generator. The second set is a collection of problems arising from chemical engineering calculations. The third set is a set of problems arising from various applications, such as surveying and linear programming. The last two sets of problems were kindly provided by Iain Duff, Roger Grimes, John Lewis and Bill Poole [2]. Numerical values were provided for most of the problems in the last two sets. For problems that do not have numerical values, we generated these numerical values using a uniform random number generator. The right hand side vector for each problem was chosen so that the solution vector consisted of all ones. The characteristics of the problems are summarized in Table 4.1.

In presenting the results, we have employed the following notation.

(1) "store" denotes the number of storage locations required.

(2) "time" denotes the execution time (in seconds) required.

(3) "lower" and "upper" refer to the lower and upper triangular portions of a matrix respectively.

problem	order	number of nonzeros	number of nonzeros per row		number of nonzeros per column		comment
			minimum	maximum	minimum	maximum	
1	936	6264	4	7	4	7	small hole square (finite element mesh)
2	1009	6865	4	7	4	7	graded L (finite element mesh)
3	1089	7361	3	7	3	7	plain square (finite element mesh)
4	1440	9504	3	7	3	7	large hole square (finite element mesh)
5	1180	7750	3	7	3	7	+ shaped domain (finite element mesh)
6	1377	8993	3	7	3	7	H shaped domain (finite element mesh)
7	1138	7450	4	7	4	7	3 hole problem (finite element mesh)
8	1141	7465	4	7	4	7	6 hole problem (finite element mesh)
9	1349	9101	4	7	4	7	pinched hole problem (finite element mesh)
10	156	371	1	7	1	6	simple chemical plant model (chemical engineering problem)
11	167	507	1	9	1	19	rigorous model of a chemical stage (chemical engineering problem)
12	381	2157	1	25	1	50	multiply fed column, 24 components (chemical engineering problem)
13	132	414	1	9	1	19	rigorous flash unit (chemical engineering problem)
14	67	294	1	6	2	10	cavett problem with 5 components (chemical engineering problem)
15	655	2854	1	12	1	35	16 stage column section, some stages simplified (chemical engineering problem)
16	479	1910	1	12	1	35	8 stage column section, all sections rigorous (chemical engineering problem)
17	497	1727	1	28	1	55	rigorous flash unit with recycling (chemical engineering problem)
18	989	3537	1	12	1	26	7 stage column section, all sections rigorous (chemical engineering problem)
19	113	655	1	20	1	27	unsymmetric pattern supplied by Morven Gentleman
20	199	701	1	6	2	9	unsymmetric pattern of order 199 given by Willoughby
21	130	1282	1	124	1	124	unsymmetric matrix from laser problem given by A.R. Curtis
22	663	1712	1	426	1	4	unsymmetric basis from LP problem (Shell)
23	363	3279	1	33	1	34	unsymmetric basis from LP problem (Stair)
24	822	4841	1	304	1	21	unsymmetric basis from LP problem (BP)
25	541	4285	1	11	5	541	unsymmetric facsimile convergence matrix

Table 4.1: Characteristics of test problems.

(4) "factor time" and "solve time" are respectively the execution times (in seconds) required for the numerical factorization and numerical triangular solutions.

(5) "percentage utilization" is the ratio of the actual number of off-diagonal nonzeros obtained in the numerical factorization to the number of off-diagonal nonzeros accommodated by the static data structure which is obtained from a symbolic factorization algorithm.

Also for convenience, the approach described in [6] is referred to as "SF-A^TA", and the new symbolic factorization algorithm described in Section 2 of this paper is referred to as "SF-A". The new symbolic factorization algorithm with subscript compression as discussed in Section 3.1 is abbreviated as "SFC-A".

For comparison purpose, we first summarize in Table 4.2 the results of using SF-A^TA.

Problem	Symbolic factorization					Numerical solution			Percentage utilization	
	store	time	number of subscripts	number of nonzeros lower	number of nonzeros upper	store	factor time	solve time	lower	upper
1	30620	1.017	7713	37554	37554	91247	55.817	1.600	50.36	68.82
2	34159	1.150	8914	46945	46945	111887	78.483	1.983	50.13	67.29
3	36244	1.200	9227	47484	47484	113998	82.650	2.100	50.50	66.65
4	45900	1.600	11317	49273	49273	122825	63.717	2.167	50.01	65.52
5	36751	1.200	8588	26570	26570	72350	20.833	1.233	49.28	64.03
6	42304	1.333	9699	28556	28556	79206	21.117	1.367	49.24	64.61
7	36507	1.200	9460	35291	35291	90286	40.500	1.533	50.43	65.14
8	37169	1.200	10040	38748	38748	97807	51.633	1.700	50.42	66.82
9	43842	1.433	10465	53161	53161	128930	74.567	2.333	49.34	63.55
10	2348	0.050	417	566	566	2955	0.133	0.033	29.68	38.52
11	3276	0.100	559	875	875	3814	0.267	0.067	30.51	40.11
12	23172	1.133	6993	18334	18334	47092	17.917	0.983	32.47	36.27
13	2645	0.083	466	742	742	3140	0.183	0.033	27.49	40.57
14	1746	0.050	385	843	843	2676	0.367	0.050	34.99	42.35
15	21058	0.767	5143	12906	12906	36852	8.467	0.667	24.14	39.12
16	13617	0.517	3118	7383	7383	22197	4.800	0.417	31.79	38.48
17	15416	0.583	2223	6363	6363	19424	1.917	0.367	13.30	24.96
18	23907	0.750	4746	8143	8143	29935	2.883	0.467	23.04	37.34
19	3702	0.117	557	1322	1322	4220	0.500	0.050	22.54	64.45
20	4364	0.133	1143	2194	2194	7324	1.067	0.117	36.46	56.61
21	16829	2.083	260	7763	7763	16958	10.250	0.350	67.51	48.78
22	188207	62.300	964	91080	91080	189093	14.850	4.050	0.55	1.54
23	15420	0.683	4007	7772	7772	22820	5.167	0.433	24.14	49.83
24	118904	22.833	6809	57817	57817	129843	38.883	2.600	11.79	15.02
25	24429	0.883	6634	16600	16600	44705	13.650	0.800	40.37	46.77

Table 4.2: Symbolic factorization using A^TA (SF-A^TA).

The first experiment we did involved implementing the symbolic factorization algorithm described in Section 2, and used the algorithm to generate a static data structure for sparse Gaussian elimination with partial pivoting. Numerical factorization and triangular solutions were performed using that a fixed storage scheme. The results are presented in Table 4.3. By comparing the results in Tables 4.2 and 4.3, we have the following observations. First, the number of nonzeros in the lower triangular matrix $|\bar{L}|$ determined by the SF-A symbolic factorization algorithm is smaller than that in the Cholesky factor L_C of A^TA predicted in SF-A^TA. This is true in all test problems, and the reductions are large in some cases. As a result, the percentage utilization for the lower triangular matrix $\bar{L}$ is significantly improved using the SF-A approach. That is, in terms of accommodating the fill-in in $\sum_{k=1}^{n-1} M_k$, the structure of $\bar{L}$ is much tighter than the structure of the Choleksy factor L_C of A^TA. It is interesting to note that the number of nonzeros in the upper triangular matrix $\bar{U}$ determined by the SF-A symbolic factorization algorithm is either identical or close to that in L_C^T in SF-A^TA. This suggests that the structures of the upper triangular matrices $\bar{U}$ in SF-A^TA and L_C^T in SF-A may be close to each other. This is supported by the fact that the values for the percentage utilization in both cases are almost the same.

Second, the SF-A symbolic factorization algorithm described in Section 2 is in general slower than the SF-A^TA symbolic factorization algorithm. (There are instances in which the SF-A algorithm runs faster than the SF-A^TA algorithm. See Problems 22 and 24.) In most cases, the execution time of the SF-A symbolic factorization algorithm is about twice to three times that of the SF-A^TA symbolic factorization algorithm. In the SF-A^TA algorithm, the structures of the Cholesky factors L_C and L_C^T of A^TA are used to bound the structures of the triangular factors $\sum_{k=1}^{n-1} M_k$ and U. Only one set of subscripts is needed to describe the structures of L_C and L_C^T. However, in the SF-A symbolic factorization algorithm, the structure of $\bar{L}$ is in general not the same as the structure of $\bar{U}^T$. Thus two sets of subscripts are needed, one set for describing the structures of the lower triangular matrix and another for describing the structure of the upper triangular matrix. Consequently, extra work is done in the SF-A algorithm to generate these two sets of subscripts, so it not surprising that the SF-A algorithm runs slower than the SF-A^TA algorithm. The discussion above also explains why the storage requirements for the symbolic factorization and numerical solution in Table 4.3 are larger than those in Table 4.2, since in SF-A, there is exactly one subscript for every nonzero stored. This is not the case in SF-A^TA because subscript compression is employed in the symbolic factorization algorithm.

Third, the execution times for the numerical factorization and triangular solution in SF-A are smaller than those in SF-A^TA. One reason is that there are fewer nonzeros in $\bar{L}+\bar{U}$ in SF-A than in $L_C+L_C^T$ in SF-A^TA. In addition, the uncompressed storage scheme created by the SF-A symbolic factorization algorithm is simpler than the compressed storage scheme generated by the SF-A^TA symbolic factorization algorithm. We will return to this later.

The percentage utilization for Problem 22 is extremely small for both SF-A^TA and SF-A. This is due to the existence of a few dense rows in the coefficient matrix A, as one can see from Table 4.1. These dense rows effectively destroy the sparsity of some of the other rows when we determine a column ordering from A^TA. As we are going to see later in this section, there will be substantial improvements when we withhold these dense rows from A in determining a column ordering.

Problem	Symbolic factorization						Numerical solution			Percentage utilization	
	store	time	number of subscripts		number of nonzeros		store	factor	solve	lower	upper
			lower	upper	lower	upper		time	time		
1	79755	3.500	20310	37554	20310	37554	124154	40.667	1.233	93.13	68.82
2	96533	4.133	25765	46945	25765	46945	154503	59.783	1.683	91.33	67.29
3	98974	4.233	25875	47484	25875	47484	156521	64.967	1.550	92.67	66.65
4	109146	4.900	26462	49273	26462	49273	164432	45.317	1.617	93.11	65.52
5	68025	3.200	14152	26570	14152	26570	92066	15.000	0.933	92.52	64.03
6	75615	3.617	15300	28556	15300	28556	100107	17.067	1.100	91.91	64.61
7	80603	3.967	19029	35291	19029	35291	118884	30.250	1.183	93.53	65.14
8	86013	3.783	20922	38748	20922	38748	129611	36.983	1.283	93.38	66.82
9	113251	4.900	28395	53161	28395	53161	175255	51.900	1.733	92.38	63.55
10	3105	0.150	256	544	256	544	3006	0.133	0.050	65.63	40.07
11	3866	0.217	349	830	349	830	3863	0.183	0.033	76.50	42.29
12	34446	1.717	8780	17539	8780	17539	56069	11.417	0.583	67.80	37.91
13	3109	0.150	268	690	268	690	3106	0.100	0.033	76.12	43.62
14	2530	0.150	426	843	426	843	3143	0.200	0.033	69.25	42.35
15	29413	1.433	5358	11794	5358	11794	40201	5.500	0.450	58.16	42.81
16	19514	0.917	3698	7203	3698	7203	26115	3.233	0.267	63.47	39.44
17	15622	0.783	1447	5748	1447	5748	18865	0.933	0.200	58.47	27.63
18	27303	1.383	2771	7565	2771	7565	29575	1.683	0.317	67.70	40.20
19	4078	0.200	313	1322	313	1322	4289	0.317	0.050	95.21	64.45
20	6549	0.350	960	2194	960	2194	8101	0.783	0.100	83.33	56.61
21	19032	1.050	7402	7763	7402	7763	31502	9.217	0.317	70.81	48.78
22	112290	5.350	11153	91080	11153	91080	210435	6.567	2.333	4.52	1.54
23	19725	1.083	2562	6972	2562	6972	22337	2.433	0.267	73.22	55.55
24	91038	4.083	16574	56559	16574	56559	153666	20.100	1.617	41.14	15.35
25	38193	1.933	7610	16600	7610	16600	53291	9.467	0.567	88.07	46.77

Table 4.3: Symbolic factorization algorithm of Section 2 (SF-A).
(Without subscript compression)

The analysis in Section 2 showed that the complexity of the SF-A symbolic factorization algorithm is $O(|A|,\sum_k|\bar{L}_k|+|\bar{U}_k|)$. That is, for large sparse problems, the complexity of the SF-A symbolic factorization algorithm is essentially proportional to the number of subscripts generated by the algorithm. To verify this, we apply the SF-A algorithm to a sequence of finite element problems defined on graded-L meshes. The results, which are given in Table 4.4, indeed suggest that the complexity of the new symbolic factorization algorithm is $O(|A|,\sum_k|\bar{L}_k|,|\bar{U}_k|)$.

In terms of the number of nonzeros accommodated by the data structure, the results presented so far have shown that the symbolic factorization algorithm in SF-A is much better than that in SF-A^TA. However, in terms of storage and time required, the SF-A algorithm is not as efficient as the SF-A^TA algorithm. The high storage requirement is due to the fact that there is one subscript for every nonzero determined by the SF-A algorithm. If the number of nonzeros determined by the SF-A algorithm

order	number of nonzeros	time t	number of subscripts			ratio t/nz
			lower	upper	total (nz)	
265	1753	0.783	3942	7088	11030	0.0000710
406	2716	1.683	7377	13104	20481	0.0000822
577	3889	2.200	11520	21058	32578	0.0000675
778	5272	3.050	18697	33433	52130	0.0000585
1009	6865	4.167	25765	46945	72710	0.0000573
1270	8668	5.483	34376	62783	97159	0.0000564
1561	10681	7.800	47797	86720	134517	0.0000580
1882	12904	9.317	57384	105836	163220	0.0000571
2233	15337	11.583	73112	134773	207885	0.0000557
2614	17980	15.217	91393	167508	258901	0.0000588
3025	20833	18.333	111237	203579	314816	0.0000582
3466	22896	20.483	132385	242870	375255	0.0000546

Table 4.4: Applying SF-A to a sequence of graded-L finite element problems

is η, then we require at least η locations to execute the SF-A symbolic factorization algorithm, since we need η locations to store the subscripts. Consequently, we need at least 2η locations to perform the numerical solution. In Section 3, subscript compression was proposed as a means to reduce the storage and execution time required by the SF-A symbolic factorization algorithm. Our next experiment, which involved incorporating subscript compression in the implementation of the new symbolic factorization algorithm (SFC-A), is intended to illustrate the improvement that may be achieved. The results are presented in Table 4.5. They show that the new symbolic factorization algorithm with subscript compression performs extremely well. The number of subscripts that have to be stored is much smaller than that required in SF-A. Also note that even though the storage scheme in SFC-A now becomes more complicated, the reduction in the number of subscripts is large enough to offset the additional storage required for the more sophisticated storage scheme. As a result, the storage requirements for both the symbolic factorization phase and the numerical solution phase in SFC-A are much smaller than those required in SF-A. In fact, in most cases, the storage required is even less than that required in SF-A^TA.

It is interesting to point out that the SFC-A symbolic factorization algorithm runs faster than the SF-A symbolic factorization algorithm. This is because fewer subscripts have to be generated in the former case. As noted earlier, extra indexing is needed in using the storage scheme created by the SFC-A algorithm. Thus, we should expect the execution times for the numerical solution in SFC-A to be larger than those in SF-A. The results in Table 4.5 show that this is indeed the case; however, the increase is not very significant. Furthermore, even though the execution times for the numerical solution in SFC-A have been increased slightly, they are still less than those in SF-A^TA.

Problem	Symbolic factorization						Numerical solution			Percentage utilization	
	store	time	number of subscripts		number of nonzeros		store	factor time	solve time	lower	upper
			lower	upper	upper	lower					
1	35459	1.883	3981	7713	20310	37554	79858	45.467	1.283	93.13	68.82
2	39202	1.833	4445	8914	25765	46945	97172	66.850	1.650	91.33	67.29
3	42143	2.033	5121	9227	25875	47484	99690	66.600	1.667	92.67	66.65
4	53535	2.633	5925	11317	26462	49273	108821	51.883	1.750	93.11	65.52
5	42636	2.133	4383	8588	14152	26570	66677	17.800	1.050	92.52	64.03
6	49045	2.317	4831	9699	15300	28556	73537	18.083	1.183	91.91	64.61
7	42623	2.017	4602	9460	19029	35291	80904	32.517	1.250	93.53	65.14
8	43461	2.050	4794	10040	20922	38748	87059	41.983	1.350	93.38	66.82
9	50791	2.483	5931	10465	28395	53161	112795	57.400	1.800	92.38	63.55
10	3176	0.167	152	405	256	544	3077	0.100	0.050	65.63	40.07
11	3720	0.167	173	524	349	830	3717	0.167	0.050	76.50	42.29
12	17897	0.983	2462	6544	8780	17539	39520	12.100	0.567	67.80	37.91
13	2992	0.167	140	435	268	690	2989	0.133	0.033	76.12	43.62
14	1953	0.100	171	385	426	843	2566	0.267	0.033	69.25	42.35
15	20140	1.067	1579	4988	5358	11794	30928	6.167	0.417	58.16	42.81
16	13771	0.733	1158	3040	3698	7203	20372	3.733	0.267	63.47	39.44
17	12082	0.683	558	2101	1447	5748	15325	1.067	0.217	58.47	27.63
18	25026	1.283	1523	4556	2771	7565	27298	1.800	0.333	67.70	40.20
19	3351	0.183	111	569	313	1322	3562	0.333	0.067	95.21	64.45
20	5393	0.250	440	1158	960	2194	6945	0.983	0.083	83.33	56.61
21	4780	0.267	391	260	7402	7763	17250	10.283	0.333	70.81	48.78
22	14988	3.850	2638	965	11153	91080	113133	7.250	2.167	4.52	1.54
23	15337	0.800	800	3618	2562	6972	17949	2.700	0.233	73.22	55.55
24	29887	2.617	4195	6141	16574	56559	92515	23.183	1.600	41.14	15.35
25	23868	1.167	2167	6634	7610	16600	38966	10.867	0.550	88.07	46.77

Table 4.5: Symbolic factorization algorithm with subscript compression (SFC-A).

For the SF-A symbolic factorization algorithm, our analysis in Section 2 showed that its run time is proportional to the number of subscripts generated. We do not know if the SFC-A symbolic factorization algorithm behaves in a similar manner. The analysis of the complexity of the SFC-A algorithm is under investigation. However, preliminary experiments using the graded-L problems show that the execution time of the SFC-A algorithm appears to be proportional to the number of subscripts generated as well. See Table 4.6 for details.

Recall that the column ordering we used for a matrix A in the experiments was an ordering obtained by applying a variant of the minimum degree algorithm to the matrix A^TA. As we have pointed out at the end of Section 3, dense rows in the matrix A may affect the quality of the column ordering since they form dense submatrices in A^TA, and consequently the sparsity of some sparse rows may be destroyed, and this may cause unnecessary fill-in in $\bar{L}$ and $\bar{U}$. (Problem 22 is an example.) We have suggested a solution to this problem. These dense rows can be withheld from A when the ordering is determined from A^TA. Then the column ordering is applied to the

order	number of nonzeros	time	number of subscripts			ratio
		t	lower	upper	total (nz)	t/nz
265	1753	0.483	1029	2087	3116	0.000155
406	2716	0.733	1667	3346	5013	0.000146
577	3889	1.033	2459	5102	7561	0.000137
778	5272	1.400	3365	7003	10368	0.000135
1009	6865	1.800	4445	8914	13359	0.000135
1270	8668	2.400	6070	11639	17709	0.000136
1561	10681	3.050	7202	14543	21745	0.000140
1882	12904	3.700	9178	17118	26296	0.000141
2233	15337	4.383	11276	20864	32140	0.000136
2614	17980	5.250	12713	24933	37646	0.000139
3025	20833	6.033	15779	28270	44049	0.000137
3466	22896	7.117	18746	32280	51026	0.000139

Table 4.6: Applying SFC-A to a sequence of graded-L finite element problems

original matrix A when we apply the symbolic factorization algorithm (with subscript compression). This strategy has been implemented and applied to those problems in Table 4.1 that have rows with more than 50 nonzeros. The results are given in Table 4.7. If we compare these results with the corresponding results in Table 4.5, we observe that there are indeed significant improvements when dense rows are withheld in determining the column orderings. In each case, there is a large reduction in the number of nonzeros determined by the symbolic factorization algorithm. Consequently, there is also a large reduction in the storage required by the numerical solution phase. Furthermore, the factorization and solution times are much smaller than those in Table 4.5. In some cases, the run time for the symbolic factorization algorithm and the number of (compressed) subscripts generated are also smaller.

Problem	Symbolic factorization						Numerical solution			Percentage utilization	
	store	time	number of subscripts		number of nonzeros		store	factor time	solve time	lower	upper
			lower	upper	lower	upper					
21	4561	0.600	131	301	1213	7804	10883	1.133	0.217	54.33	11.38
22	13168	0.950	429	1354	617	2216	11913	0.317	0.133	86.87	29.78
24	28855	1.683	2164	7140	7501	18756	44607	8.150	0.633	56.14	37.09

Table 4.7: Symbolic factorization algorithm with subscript compression (SFC-A). (With withholding of dense rows in determining a column ordering.)

Another strategy to possibly improve the performance of the symbolic factorization algorithm is to apply the symbolic factorization algorithm to the matrix A^T and compute (numerically) a triangular factorization of it. This decomposition is equally useful in solving $Ax=b$. In Table 4.8, we have provided the results of applying SFC-A to the transpose of some of the matrices described in Table 4.1. Rows with more than 50 nonzeros are withheld when a column ordering for A^T is determined. (Row withholding occurs in Problems 17, 21 and 25.) Comparing the results in Table 4.8 with those in Tables 4.5 and 4.7, we see that it may be desirable to use the transpose in some instances. For example, Problem 22 has a few very dense rows but no dense columns. Applying SFC-A to the matrix itself results in low percentage utilization, high storage requirement and large execution time. Another interesting example is Problem 17 which has a few dense columns but no dense rows. Applying SFC-A to the transpose with withholding dense rows in determining the column ordering seems to be better than just applying SFC-A to the matrix itself.

However, the opposite can also hold. As an example, the results (including storage requirements and execution times) of applying SFC-A (with withholding dense rows in determining the column ordering) to Problem 21 are better than the corresponding results in Table 4.8. At this point, we do not know an effective scheme for deciding whether to use A or A^T, other than simply applying the SFC-A symbolic factorization algorithm to both A and A^T, and choosing the one that requires the least storage. Since the SFC-A symbolic factorization algorithm is very efficient, this is not an unreasonable thing to do.

5. Concluding remarks

In this paper, we have described a new symbolic factorization algorithm for Gaussian elimination with partial pivoting. Several enhancements have been proposed to improve the performance of the symbolic factorization algorithm. Preliminary numerical experiments have shown that the data structure created by the new symbolic factorization algorithm (together with the enhancements) for sparse Gaussian elimination with partial pivoting is much tighter than that generated by the approach proposed in [6]. Storage required for the numerical solution phase is smaller in the new approach than in [6]. There are also substantial improvements in execution times for the numerical solution phase.

Several problems are still outstanding. First, preliminary results indicate that the run time of the new symbolic factorization algorithm with subscript compression is proportional to the number of (compressed) subscripts generated. Thus one of the problems to look at is the analysis of the complexity of the new symbolic factorization algorithm with subscript compression. Second, the column ordering that is used for A in the experiments presented in this paper is obtained by applying the minimum degree ordering algorithm to the matrix A^TA. As we have pointed out in Section 3.2, there

Problem	Symbolic factorization						Numerical solution			Percentage utilization	
	store	time	number of subscripts		number of nonzeros		store	factor	solve	lower	upper
			lower	upper	lower	upper		time	time		
10	3074	0.133	128	327	204	444	2823	0.117	0.017	69.61	47.07
11	3860	0.200	224	613	507	1083	4268	0.300	0.050	67.85	51.89
12	17624	1.067	2671	6062	11160	20204	44292	14.100	0.700	55.27	27.96
13	3123	0.150	195	511	481	993	3636	0.267	0.050	55.51	45.72
14	1915	0.083	171	347	381	764	2404	0.250	0.050	74.54	39.01
15	19926	1.167	1696	4657	6942	16269	36773	11.583	0.550	73.93	50.45
16	13037	0.717	871	2593	3129	8035	19901	3.100	0.317	53.95	37.06
17	11944	0.633	480	2041	1304	5405	14701	1.167	0.200	60.58	31.51
18	26370	1.400	1717	5706	4675	13303	36284	3.850	0.533	53.67	30.94
19	3475	0.233	238	566	710	1522	4283	0.700	0.083	96.48	75.89
20	5393	0.267	488	1110	1355	2471	7617	1.050	0.117	80.37	51.19
21	4767	0.517	146	492	1325	8058	11455	1.233	0.200	78.49	5.85
22	12595	0.533	134	1076	174	1274	9955	0.100	0.100	89.08	78.73
23	17050	1.100	1235	4896	4358	12341	26827	5.233	0.383	53.85	36.58
24	26650	1.400	1396	5703	5189	16703	38037	7.317	0.567	71.71	43.69
25	22560	1.350	1957	5536	7614	16484	37546	9.933	0.550	86.77	44.86

Table 4.8: **Symbolic factorization algorithm with subscript compression (SFC-A). (The transpose is used and dense rows in the transpose are withheld in determining a column ordering.)**

are other possibilities in determining "good" column orderings for A. We are currently designing efficient algorithms for generating these column orderings. We will also compare these column orderings with the minimum degree column ordering obtained from A^TA. Third, an important, but difficult, problem is the identification of "dense" rows. This is a problem common to many sparse matrix computations. A related problem is to investigate when we should work with A^T instead of A. Finally, if the matrix A is reducible, then A can be permuted symmetrically so that the permuted matrix has a block triangular form. In this case, it is only necessary to decompose the diagonal blocks. The ideas described in this paper can of course be applied to these diagonal blocks. Efficient implementation of this generalization is under consideration.

6. References

[1] I.S. DUFF, "On algorithms for obtaining a maximum transversal", *ACM Trans. on Math. Software*, **7** (1981), pp. 315-330.

[2] I.S. DUFF, R.G. GRIMES, J.G. LEWIS, AND W.G. POOLE, JR., "Sparse matrix test problems", *ACM SIGNUM Newsletter*, **17**(2) (1982), p. 22.

[3] J.A. GEORGE AND J.W.H. LIU, "Algorithms for matrix partitioning and the numerical solution of finite element systems", *SIAM J. Numer. Anal.*, **15** (1978), pp. 297-327.

[4] J.A. GEORGE AND J.W.H. LIU, "An optimal algorithm for symbolic factorization of symmetric matrices", *SIAM J. Comput.*, **9** (1980), pp. 583-593.

[5] J.A. GEORGE AND E.G.Y. NG, "Orthogonal reduction of sparse matrices to upper triangular form using Householder transformations", Research report CS-84-05, Department of Computer Science, University of Waterloo (1984). (submitted to SIAM J. Sci. Stat. Comput.)

[6] J.A. GEORGE AND E.G.Y. NG, "An implementation of Gaussian elimination with partial pivoting for sparse systems", *SIAM J. Sci. Stat. Comput.*, **6** (1985), pp. 390-409.

[7] J.W.H. LIU, "On multiple elimination in the minimum degree algorithm", Technical Report No. 83-03, Department of Computer Science, York University, Downsview, Ontario (1983).

[8] A.H. SHERMAN, "On the efficient solution of sparse systems of linear and nonlinear equations", Research Report #46, Dept. of Computer Science, Yale University (1975).