AN AUTOMATIC PARTITIONING AND SOLUTION SCHEME FOR SOLVING LARGE SPARSE POSITIVE DEFINITE SYSTEMS OF LINEAR ALGEBRAIC EQUATIONS

by
Alan George
and
Joseph W.H. Liu

Research Report CS-75-17
June 1975

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

Work by the first author was supported in part by Canadian National Research Council Grant A8111. The second author's work was supported in part by an Ontario Graduate Fellowship.

ABSTRACT

We describe an implementation of Cholesky's method for banded or envelope (profile) ordered linear systems which requires substantially less storage than standard implementations. We then present a new ordering algorithm designed to enhance the effectiveness of our implementation. Finally, we provide numerical experiments comparing our ordering/solution package against what we regard to be its strongest competitors.

§1. Introduction

Let $Ax = b$ be a given N by N sparse symmetric positive definite system of linear algebraic equations. It is well known that for sparse systems, the way in which the equations are ordered (numbered) can sometimes drastically affect the storage and computation required for the direct solution of that system, using Gaussian elimination. When A is symmetric and positive definite, no pivoting (row and/or column interchange) is required to maintain numerical stability; that is, symmetric Gaussian elimination or Cholesky's method applied to PAP^T, where P is any N by N permutation matrix, is numerically stable [17]. Thus, we may choose P to achieve other objectives, such as reduced storage and/or computation.

A common choice for P is one for which PAP^T has a small bandwidth or envelope (profile). For a given symmetric matrix M, its bandwidth $\beta(M)$ and envelope Env(M) are defined as follows.

$$\beta(M) = \max_{M_{ij} \neq 0} |i-j|. \tag{1.1}$$

$$\mathrm{Env}(M) = \{(i,j) \mid i \geq f_j \text{ and } j \geq f_i\}, \tag{1.2}$$

where $f_i = \min\{j \mid M_{ij} \neq 0\}$, $1 \leq i \leq N$.

When Gaussian elimination is applied to a sparse matrix, it normally suffers "fill"; that is, the triangular factors have nonzero components in positions which are zero in the original matrix. It is easy to verify that when no pivoting is performed, this fill is confined within Env(M), and since $(i,j) \in \mathrm{Env}(M) \Rightarrow |i-j| \leq \beta(M)$, fill is also confined within the band.

Implicit in the use of these ordering strategies is the assumption that zeros outside the band or envelope are to be exploited, and zeros within the band or envelope are not exploited. Efficient storage schemes for utilizing such orderings have been described in [10,13].

It is becoming increasingly well known that if we are prepared to exploit all zeros, then orderings which yield a small band $\beta(A)$ or $|Env(A)|$ may be far from optimal in the least-fill or least arithmetic sense [6, 15]. However, robust algorithms for generating optimal or near-optimal orderings for general sparse matrices are not yet well developed. Moreover, in many situations, the relative simplicity of the (perhaps far from optimal) band or envelope schemes are a reasonable compromise. More to the point, there are several band and/or envelope reduction algorithms available which appear to be quite effective for a large class of problems [1,4,8].

The objectives of this paper are twofold. First, using some ideas which have already been presented and analyzed elsewhere [3,7], we describe an implementation of Cholesky's method which substantially reduces the storage requirements over standard schemes, when these banded or envelope orderings are used. Second, we present a new ordering algorithm which is designed specifically to enhance the effectiveness of our implementation of Cholesky's method.

An outline of the paper is as follows. In section 2 we present a simple example which provides some motivation for both our implementation scheme and our ordering algorithm. In section 3 we review some basic graph theoretic notions as they relate to Cholesky's method. We also make the

connection between a level structure in a graph, which is a central construct in many ordering algorithms (including the one we propose in this paper), and the partitioning it induces in the matrix problem associated with the graph. We also review the significance of the class of graphs called trees in connection with matrix orderings, and extend some of the important ideas to block matrices.

In section 4 we describe the principal ideas behind two algorithms which are regarded as the most generally effective band/envelope reduction algorithms in current use. These are the so-called reverse Cuthill-McKee algorithm (RCM) [4 ,12] and the algorithm recently developed by Gibbs et al. (GPS) [8]. We then describe our ordering algorithm, and through some examples show how it differs from the other two algorithms.

Section 5 contains a description of the data structures used in the implementation of our Cholesky solver, together with an example.

Section 6 contains a substantial set of examples comparing various performance criteria for our ordering/solution package against several competitors. Section 7 contains our conclusions.

§2. A Motivating Example

Suppose we wish to solve $Ax = b$, where A is ordered so that $\beta(A) = m$, and for convenience we suppose that for some integer k $N = km$, and $k = O(\sqrt{N})$. We may regard A as a block tri-diagonal matrix as shown below, where the blocks are m by m.

$$A = \begin{pmatrix} A_1 & B_1 & & & & \\ B_1^T & A_2 & B_2 & & & \\ & B_2^T & A_3 & \ddots & & \\ & & \ddots & \ddots & \ddots & \\ & & & \ddots & \ddots & B_{k-1} \\ & & & & B_{k-1}^T & A_k \end{pmatrix}. \tag{3.1}$$

If we factor A into LL^T, where L is lower triangular, using Cholesky's method [17], we may again regard L as partitioned accordingly, as shown in (3.2).

$$L = \begin{pmatrix} L_1 & & & & \\ W_1^T & L_2 & & & \\ & W_2^T & L_3 & \ddots & \\ & & \ddots & \ddots & \\ & & & W_{k-1}^T & L_k \end{pmatrix}. \tag{3.2}$$

Here the blocks are defined by

$$A_1 = \bar{A}_1 = L_1 L_1^T, \tag{3.3}$$

$$W_i = L_i^{-1} B_i, \quad i = 1,2,\ldots,k-1, \tag{3.4}$$

and

$$(3.5) \qquad \bar{A}_i = L_i L_i^T = A_i - B_{i-1}^T A_{i-1}^{-1} B_{i-1},$$

$$= A_i - W_{i-1}^T W_{i-1}, \qquad i = 2,3,\ldots,k.$$

Band-oriented storage schemes usually treat the W_i and L_i as full lower triangular matrices. Thus, to store L we require $km^2 + O(N)$ storage locations.

Our strategy for saving storage utilizes some ideas which have already been described and analyzed elsewhere, in a somewhat different context [3, 7]. From (3.5), it is clear that as far as the factorization is concerned, we do not need W_{i-1} after $\bar{A}_i$ is computed, so we simply discard it rather than save it. Instead, we retain only B_{i-1}; when we need to use W_{i-1}, we simply multiply by B_{i-1} and then solve the appropriate triangular system whose coefficient matrix is L_{i-1}. For example, to compute $z = W_{i-1}y$, we compute $\bar{y} = B_{i-1}y$ and then solve $L_{i-1}z = \bar{y}$. Thus, we only store L_i, $i = 1,2,\ldots,k$ and B_i, $i = 1,2,\ldots,k-1$, rather than L_i, $i = 1,2,\ldots,k$ and W_i, $i = 1,2,\ldots,k-1$. This will often reduce storage requirements because in many situations the W_i are far less sparse than the B_i. For example, the total number of nonzeros in all the B_i may only be $O(N)$. If this is true, the storage requirement using this latter strategy is $\frac{1}{2}km^2 + O(N)$, rather than $km^2 + O(N)$.

In connection with the computation (3.5), in some situations it may require fewer arithmetic operations to compute $\bar{A}_i$ as $A_i - B_{i-1}^T(L_{i-1}^{-T}(L_{i-1}^{-1}B_{i-1}))$ rather than as $A_i - (B_{i-1}L_{i-1}^{-T})(L_{i-1}^{-1}B_{i-1})$, which is implied by (3.5). (Consider the case when W_i is full lower triangular but B_{i-1} and L_{i-1} are very sparse.) Since we intend to discard W_i anyway, this "asymmetric" version of the

computation usually requires less temporary storage than the first. That is, we can compute $\bar{A}_i$ one column at a time, discarding the column of $B_{i-1}^{-1}A_{i-1}^{-1}B_{i-1}$ as soon as it has been used. On the other hand, the calculation of $\bar{A}_i$ as implied by (3.5) requires temporary storage for the whole matrix W_{i-1}. We distinguish between these two methods of computation, by denoting the symmetric version by F_1 and the asymmetric version by F_2.

§3. Graph Theoretic Preliminaries

In this section we review some basic graph theoretic notions and introduce a few definitions that are related to our implementation of Cholesky's method. We also establish some preliminary results needed in subsequent sections.

A graph $G = (X,E)$ consists of a finite nonempty set X of nodes together with a prescribed edge set E of unordered pairs of distinct nodes. A graph $G' = (X',E')$ is a subgraph of $G = (X,E)$ if $X' \subset X$ and $E' \subset E$.

Nodes x and y are said to be adjacent if $\{x,y\}$ is an edge in E. For a subset Y of nodes, the adjacent set of Y is defined as

$$Adj(Y) = \{x \in X \backslash Y \mid \{x,y\} \in E \text{ for some } y \in Y\}.$$

If $Y = \{y\}$, we shall write $Adj(y)$ instead of the formally correct $Adj(\{y\})$. The degree of a node x is the number of nodes adjacent to x, denoted by $|Adj(x)|$. Sometimes, we shall refer to $y \in Adj(x)$ as a neighbor of the node x.

A path of length ℓ is a sequence of ℓ edges $\{x_0,x_1\},\{x_1,x_2\},\ldots,\{x_{\ell-1},x_\ell\}$ where all the nodes $x_0,x_1,\ldots,x_\ell$ are distinct except possibly x_0 and x_ℓ. If $x_0 = x_\ell$, it is called a cycle. A graph G is connected if there is a path connecting each pair of distinct nodes. If G is disconnected, it consists of two or more maximal connected subgraphs called components.

For a subset Y of nodes, the span of Y is defined as

$$Span(Y) = \{x \in X \mid \exists \text{ a path from } y \text{ to } x, \text{ for some } y \in Y\}.$$

If $Y = \{y\}$, $Span(Y)$ is simply the connected component that contains the node y. In case the graph is connected, the span of any nonempty subset is the node set X itself.

Unless otherwise specified, graphs in this paper are assumed to be connected. The distance $d(x,y)$ between two nodes x and y is the length of a shortest path joining them. The diameter of a graph G is $\delta(G) = \max\{d(x,y) | x,y \in X\}$. If $d(x,y) = \delta(G)$, the nodes x and y are called peripheral nodes [2].

A tree is a connected graph with no cycles, or equivalently, it is a graph where every pair of distinct nodes is joined by a unique path. For a tree $T = (X,E)$, $|X| = |E| + 1$.

A rooted tree is a tree $T = (X,E)$ with a distinguished node R, called the root of T. If $\{R,x_1\},\ldots,\{x,y\}$ is the (unique) path from the root R to the node y, x is said to be the father of y. We can then introduce the single-valued function:

$$\text{Father} : X\backslash\{R\} \rightarrow X$$

that maps a node to its father in the rooted tree. It is important to note that this function Father(x) uniquely characterises the rooted tree (Parter [14]). Thus, a rooted tree with N nodes can be represented in this form using $N-1$ storage locations. For convenience, we let Father(R) = 0 so that Father becomes a mapping from X to $X \cup \{0\}$.

For a graph $G = (X,E)$ with $|X| = N$, an ordering or numbering of G is a bijective mapping $\alpha:\{1,2,\ldots,N\} \rightarrow X$. We use G_α and X_α to denote the ordered graph and ordered node set respectively.

It is often convenient to view permutations on a sparse symmetric matrix as orderings on a corresponding graph structure. Let M be a symmetric matrix. We associate an undirected graph $G(M) = (X(M),E(M))$ with M, such that $X(M)$ is the set of nodes corresponding to and labelled as the rows of M,

and $\{x_i,x_j\} \in E(M)$ if and only if $M_{ij} \neq 0$ and $i \neq j$. Note that the graph $G(M)$ has an implicit ordering defined by the matrix M. Indeed, each ordering α on $G(M)$ identifies a permutation matrix P_α on the matrix M.

In [14], Parter studied the effect of Gaussian elimination on matrices associated with tree structures. In this context, he introduced the class of monotone orderings for rooted trees. In our notation, a monotone ordering α for a rooted tree is one that always numbers a node before its father; that is,

$$\alpha(j) = \text{Father}(\alpha(i)) \Rightarrow j > i.$$

Clearly, the root is always numbered last by a monotone ordering.

We can extend this definition to general trees. Let α be an ordering on a tree $T = (X,E)$, where $|X| = N$. We call α a monotone ordering for T if it is one for the tree rooted at the node $\alpha(N)$. The following lemma is due to Parter [14].

Lemma 3.1 Let A be an N by N symmetric matrix associated with a monotonely-ordered tree. If $A = LL^T$ where L is the triangular factor of A, then $A_{ij} = 0 \Rightarrow L_{ij} = 0$, for $i \neq j$. □

In other words, matrices associated with monotonely-ordered trees do not have any fill in Gaussian elimination. We now extend this lemma in the following form.

Lemma 3.2 Let A and L be as in Lemma 3.1. Then $L_{ij} = L_{jj}^{-1}A_{ij}$ for $i \neq j$.

Proof In Gaussian elimination, the components of L are given by:

$$L_{ij} = L_{jj}^{-1}\left(A_{ij} - \sum_{k=1}^{j-1} L_{ik}L_{jk}\right) \quad \text{for } i \neq j.$$

It is sufficient to show that $\sum_{k=1}^{j-1} L_{ik}L_{jk}$ is zero. Assume for contradiction that $L_{ik}L_{jk} \neq 0$ for some $k = 1,\ldots,j-1$. By Lemma 3.1, we have $A_{ik}A_{jk} \neq 0$, so that the node x_k is connected to both x_i and x_j. This contradicts the assumption that the associated tree is monotonely-ordered. □

An interesting consequence of Lemma 3.2 is that the triangular factor L of A can be stored implicitly as $\{L_{jj} | j = 1,\ldots,N\}$ and $\{A_{ij} | A_{ij} \neq 0 \text{ and } i \neq j\}$; each nonzero off-diagonal component of L can be generated readily when required. This observation becomes crucial when we extend the ideas to block matrices. A substantial amount of storage can be saved in using the implicit storage scheme if each A_{ij} corresponds to a sparse block submatrix rather than a nonzero component of A.

The example described in section 2 demonstrates how this implicit scheme can almost halve the storage requirement. In that example, if each diagonal block A_i is considered as a node, we obtain a graph which is a simple tree - a "chain". Furthermore, the block ordering defined implicitly by A is a monotone ordering on this chain.

To formalise these ideas, it is convenient to introduce the concept of quotient graphs, a notion motivated by the partitioning of a matrix into block submatrices. Given a graph G = (X,E), let P be a partition on the node set X

$$P = \{Y_1, Y_2, \ldots, Y_k\}.$$

That is, $\bigcup_{i=1}^{k} Y_i = X$ and $Y_i \cap Y_j = \phi$ for $i \neq j$. We define the <u>quotient graph</u> of G with respect to the partition P to be the graph

$$G/P = (P, \mathcal{E})$$

where $\{Y_i, Y_j\} \in \mathcal{E}$ if and only if $\text{Adj}(Y_i) \cap Y_j \neq \phi$. When G/$P$ is a tree, we call it a <u>quotient tree</u>.

An ordering α of G is said to be compatible with a partition P if for each $Y \in P$, α numbers nodes in Y consecutively. An ordering of G which is compatible with respect to P induces an ordering on the quotient graph G/P. Obviously, there is a class of compatible orderings for G that corresponds to any ordering on G/P.

We can derive a simple class of quotient trees using the concept of level structures [1]. Formally, a level structure of a graph $G = (X,E)$ is a partition

$$\mathcal{L} = \{L_1, L_2, \ldots, L_\ell\}$$

of X such that

$$Adj(L_i) \subset L_{i-1} \cup L_{i+1}, \quad i = 1,\ldots,\ell.$$

Here, L_0 and $L_{\ell+1}$ are assumed to be empty. The number ℓ is the length of the level structure. The quantity $\max\{|L_i| \mid i = 1,\ldots,\ell\}$ is called the width of $\mathcal{L}$. It should be clear that the corresponding quotient tree $G/\mathcal{L}$ is a simple "chain".

In practice, it is common to produce and work on rooted level structures. For a node $x \in X$, the rooted level structure at x is defined as the level structure:

$$\mathcal{L}(x) = \{L_1(x), L_2(x), \ldots, L_{\ell(x)}(x)\},$$

where $L_1(x) = \{x\}$,

$$L_i(x) = Adj\left(\bigcup_{j=1}^{i-1} L_j(x)\right), \quad i = 2,\ldots,\ell(x)$$

and $$\ell(x) = \max\{d(x,y) \mid y \in X\}.$$

We now make the connection between a level structure of a graph and the partitioning it induces on a matrix associated with the graph. Let G(A) be the graph associated with a symmetric matrix A and let $\mathcal{L}$ be a level structure in G(A). The quotient graph $G(A)/\mathcal{L}$ is a chain, so if we number the nodes in each level L_i consecutively from L_1 to L_ℓ, the levels in $\mathcal{L}$ induce a block tridiagonal partitioning on the permuted matrix. The example in section 2 can be regarded as such. Figure 3.1 contains another example.

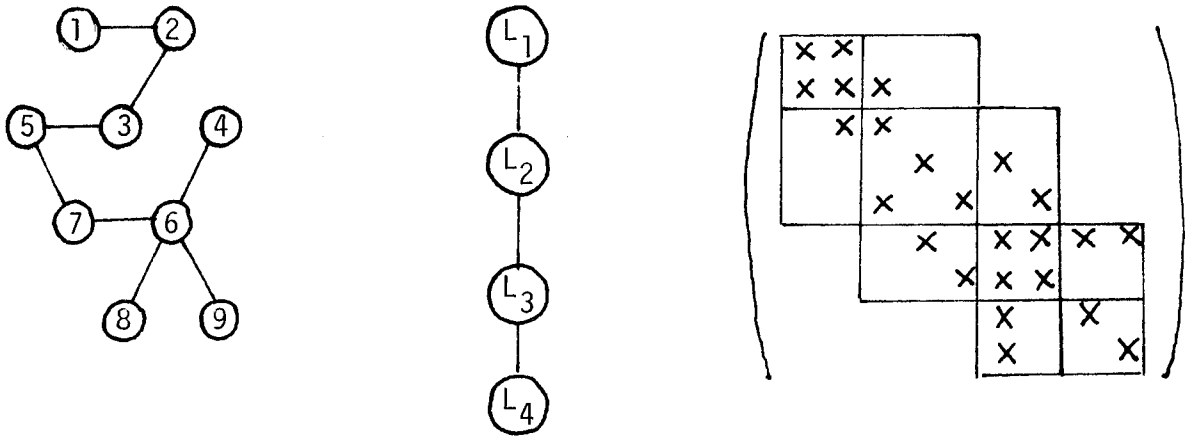


Figure 3.1 Block tridiagonal partitioning induced by a level structure

§4. Description of Ordering Algorithms

Gibbs et al. [8] have recently developed an ordering algorithm for band/envelope reduction. A novel feature in their algorithm (GPS) is the notion of pseudo peripheral nodes. Recall that nodes that are at maximum distance apart are called peripheral nodes. In general, it is time-consuming to find them.

Nodes at nearly maximum distance apart are pseudo peripheral nodes. In other words, if we let $\ell(x) = \max\{d(x,y) | y \in X\}$, then a node x is a pseudo peripheral node if $\ell(x)$ is close to the diameter $\delta(G)$ of the graph. In [8], Gibbs et al. have also provided an efficient way of finding pseudo peripheral nodes. Their approach is based on the following observation. Let $\mathcal{L}(x) = \{L_1(x), L_2(x), \ldots, L_{\ell(x)}(x)\}$ be the rooted level structure at x. Then for any $y \in L_{\ell(x)}(x)$, $\ell(x) \leq \ell(y)$.

Since our new ordering algorithm requires the determination of a pseudo peripheral node, for completeness we now describe their algorithm

Step 1 Find a node R of minimum degree.

Step 2 Generate the level structure at R:

$$\mathcal{L}(R) = \{L_1(R), L_2(R), \ldots, L_{\ell(R)}(R)\}$$

Step 3 Sort $L_{\ell(R)}(R)$ in order of increasing degree.

Step 4 For each $x \in L_{\ell(R)}(R)$, in order of increasing degree, generate the level structure $\mathcal{L}(x)$ at x. If $\ell(x) > \ell(R)$, reset R to x and go to step 3.

Step 5 Stop. R is a pseudo peripheral node.

At the termination of the algorithm, we note that $\ell(x) = \ell(y)$ for all the nodes $x,y \in \{R\} \cup L_{\ell(R)}(R)$. In other words, $\{R\} \cup L_{\ell(R)}(R)$ is a set of pseudo peripheral nodes. We may pick any one from this set to meet a specific objective; for example, minimal level width.

We now describe the essential ideas behind the GPS algorithm. For details, the reader is referred to [8]. The objective of the GPS algorithm is to determine a general level structure with small level width. It first finds a pseudo peripheral node R and a node $R' \in L_{\ell(R)}(R)$ where the width of $\mathscr{L}(R')$ is smallest among those defined by nodes in $L_{\ell(R)}(R)$. The algorithm then combines the rooted level structures $\mathscr{L}(R)$ and $\mathscr{L}(R')$ to form a new structure whose width is usually smaller than that of $\mathscr{L}(R)$ and $\mathscr{L}(R')$. The nodes in the graph are then numbered level by level in accordance with this new level structure.

Another effective band/profile reduction scheme is the reverse Cuthill-McKee (RCM) algorithm [4,5,12]. As noted by Gibbs et al., the original version of the algorithm performs a time-consuming search for a reasonably good starting node. Since the RCM algorithm is similar in spirit to the GPS algorithm, it is reasonable to start the former by using a pseudo peripheral node. Our new algorithm described later in this section uses the RCM algorithm to reorder subgraphs, so we now include a formal description of this scheme. Let G be a connected graph. The RCM ordering of G will be given by $x_1, x_2, \ldots, x_N$ after the execution of the following algorithm.

Step 1 Determine a pseudo peripheral node R and assign it to x_1.

Step 2 For $i = 1,2,\ldots,N$, find all the unnumbered neighbors of the node x_i and number them in increasing order of degree.

Step 3 Reverse the order $x_1, x_2, \ldots, x_N$.

Step 4 Stop.

It is straightforward to extend the algorithm for disconnected graphs. Implicit in the scheme is the use of the rooted level structure at R

$$\mathcal{L}(R) = \{L_1(R), L_2(R), \ldots, L_{\ell(R)}(R)\}.$$

Indeed, the Cuthill-McKee ordering numbers nodes in $L_{i-1}(R)$ before those in $L_i(R)$, so that the reverse ordering numbers the levels bottom up.

The renumbering procedure implies that the GPS and the RCM orderings are compatible (see section 2) with their respective level structures. In both cases, the induced ordering on the quotient tree $G/\mathcal{L}$ is monotone. Thus, when the graph is associated with a symmetric matrix, we may incorporate the implicit storage method described in sections 2 and 3 to both schemes.

To use the implicit storage scheme effectively, it is apparent that we want to get a partition $P = \{Y_1, Y_2, \ldots, Y_k\}$ with as many members as possible and consistent with the property that G/P remains a quotient tree. This is the motivation of our algorithm, which may be viewed as a refinement on the levels of a rooted level structure.

Let $\mathcal{L} = \{L_1, L_2, \ldots, L_\ell\}$ be a rooted level structure of a graph $G = (X,E)$. For each $j = 1, \ldots, \ell$, we define the subgraph $G_j = (X_j, E_j)$, where $X_j = \bigcup_{i=j}^{\ell} L_i$, and $E_j = \{\{x,y\} \in E \mid x,y \in X_j\}$. Then, each level L_j can be refined as:

$$\{Y \mid Y = L_j \cap C \text{ for some connected component } C \text{ in subgraph } G_j\}.$$

Lemma 4.1 Let $P = \{Y_1, Y_2, \ldots, Y_k\}$ be the partition defined by the refinements of the levels. Then G/P is a quotient tree. □

The Refined Quotient Tree(RQT) algorithm described below determines this refined partition. Note that we do not determine the connected components of the subgraphs G_j explicitly. A stack is used to store those partially formed partition members.

Step 1 Find a pseudo peripheral node R and its rooted level structure $\mathcal{L}(R) = \{L_1, L_2, \ldots, L_{\ell(R)}\}$. Empty the stack. Set $\ell \leftarrow \ell(R)$; pick a node x in the last level $L_{\ell(R)}$ and set $S \leftarrow \{x\}$.

Step 2 Determine the set $Y = Span(S)$ in the subgraph L_ℓ.

Step 3 If all the nodes in $Adj(Y) \cap L_{\ell+1}$ have been selected in some partition member, then go to step 5.

Step 4 Otherwise, push the set S onto the stack. Pick one such node $y_{\ell+1} \in Adj(Y) \cap L_{\ell+1}$. Trace a path $y_{\ell+1}, y_{\ell+2}, \ldots, y_{\ell+t}$ where $y_{\ell+i} \in L_{\ell+i}$ and $Adj(y_{\ell+t}) \cap L_{\ell+t+1} = \phi$. Let $S \leftarrow \{y_{\ell+t}\}$ and $\ell \leftarrow \ell+t$; go to step 2.

Step 5 Determine the set $S \leftarrow Adj(Y) \cap L_{\ell-1}$. Set $\ell \leftarrow \ell-1$. If $\ell = 0$, stop.

Step 6 If the node set T on the top of the stack belongs to L_ℓ, pop T from the stack and set $S \leftarrow S \cup T$. Go to step 2.

It is interesting to note that the order in which the partition members Y are formed defines a monotone ordering on the final quotient tree G/P. The following examples illustrate the working of the algorithm.

Example 4.1

Consider the graph in Figure 4.1. The rooted level structure at node 1 is shown in Figure 4.2. On applying the algorithm to $\mathcal{L}(1)$, we obtain a refined quotient tree with ten nodes.

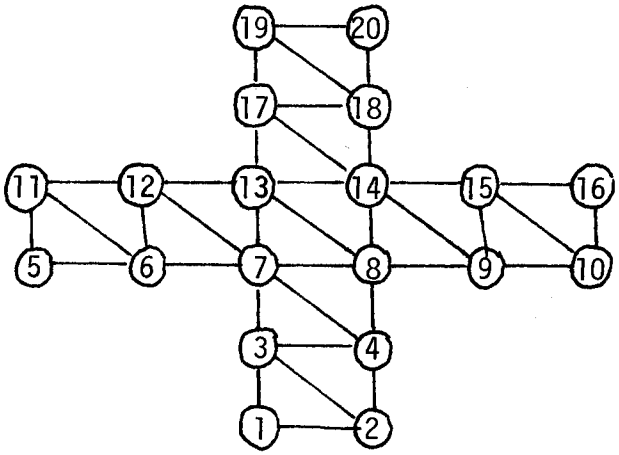


Figure 4.1 A '+' shaped graph

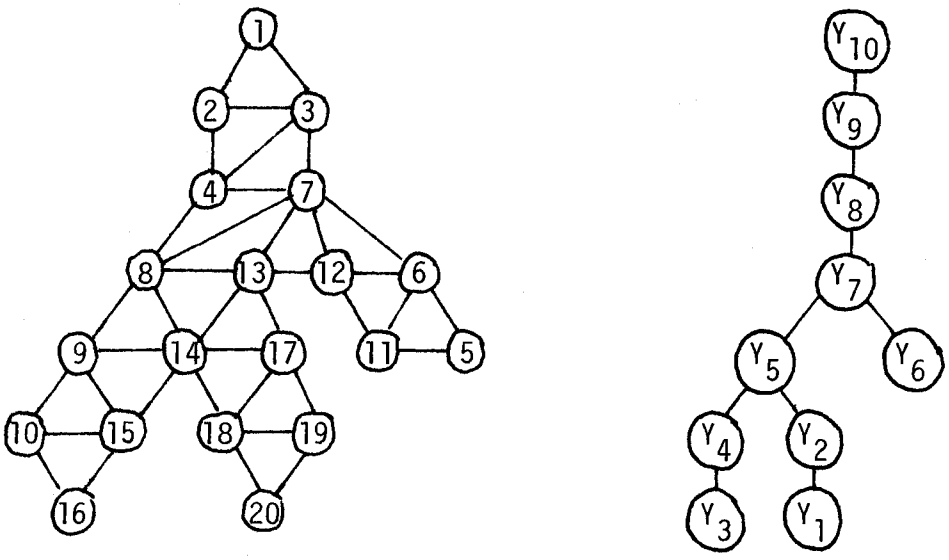


Figure 4.2 Rooted level structure and its refinement

In this example, $Y_1 = \{20\}$, $Y_2 = \{18,19\}$, $Y_3 = \{16\}$, $Y_4 = \{10,15\}$, $Y_5 = \{9,14,17\}$, $Y_6 = \{5,11\}$. So, we have $L_5 = Y_5 \cup Y_6$, $L_6 = Y_2 \cup Y_4$ and $L_7 = Y_1 \cup Y_3$.

Example 4.2

Consider any tree $T = (X,E)$. In this limiting case, the refined partition is the trivial one

$$P = \{\{x\} | x \in X\},$$

so that the quotient tree T/P is identical to T itself. The monotone ordering defined on T/P, and hence on T, is, in fact, a postorder traversal (Knuth [10,p.334]

The ordering algorithm will be complete if we specify how the nodes within each partition member are numbered. Consider a partition member $Y \in P$ where $Y \subset L_\ell$. Let S be the set

$$\{y \in Y | Adj(y) \cap L_{\ell+1} \neq \phi\}.$$

We first number nodes in the subgraph $Y \setminus S$ using the RCM ordering scheme. Note that the subgraph $Y \setminus S$ may be disconnected. The nodes in S are then numbered arbitrarily. This internal numbering strategy is chosen so as to reduce the profile of the diagonal block defined by Y. Example 4.3 illustrates the effect of this internal ordering.

Example 4.3

Consider the mesh M in Figure 4.3. Here, nodes in each triangle are assumed to be pairwise adjacent in the corresponding graph G. The rooted level structure at the node 1

$$\mathcal{L}(1) = \{L_1, L_2, L_3, L_4, L_5\}$$

has 5 levels. No refinement can be done on the levels so that $P = \{Y_1, Y_2, Y_3, Y_4, Y_5\}$, where $Y_i = L_{6-i}$. The new ordering in Figure 4.4 is obtained using the internal numbering strategy on the Y's. The correspondingly permuted matrix associated with G is also given in Figure 4.4.

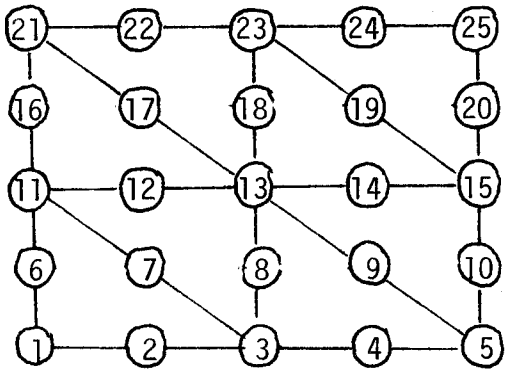
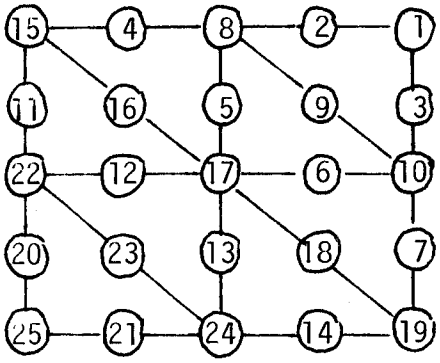


Figure 4.3 Mesh M of example 4.3



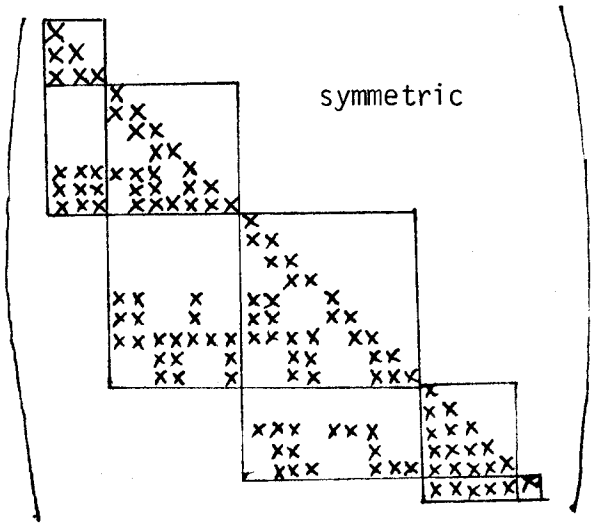


Figure 4.4 Reordered mesh and its corresponding matrix structure

As already pointed out in section 3, there is a class of orderings compatible with P. More careful reordering within partition members may further reduce the profile of the diagonal blocks and the arithmetic operations.

§5. Implementation of the Implicit Storage Scheme

In this section we provide some implementation details about our subroutines for solving sparse positive definite systems. The code is designed specifically to handle the quotient tree orderings such as those provided by the algorithms described in section 4. Our code accepts the problem $Ax = b$, along with a partitioning of A, and a) factors A into LL^T, saving the diagonal blocks of L and the nonzero components in the off-diagonal blocks of A, and b) solves the triangular systems $Ly = b$ and $L^Tx = y$. The code uses the following arrays to store the necessary structural information along with the actual components of the matrices. Here n_b is the number of diagonal blocks in the partitioning, n_w is the number of nonzero components in the lower off-diagonal blocks of A, and n_d is the number of components of the diagonal of L which we store, including possibly some zeros.

Array	Description
S :	an integer vector of length n_b. The number S(i) indicates the diagonal block j, $j > i$ to which block i is connected. If no such block exists, S(i) is zero.
p :	an integer vector of length n_b+1, where p(i) is the row number of A corresponding to the first row of the i-th diagonal block. For programming convenience, we have $p(n_b+1) = N+1$.
ℓ :	a real array of length n_d. After the factorization it contains the components of the envelopes of the diagonal blocks of L, stored row by row.
d :	an integer vector of length N. If we view the diagonal blocks L_i, $1 \le i \le n_b$ as a single N by N block diagonal matrix $\bar{L}$, then d(i) points to the position in ℓ where the i-th diagonal component of $\bar{L}$ resides.

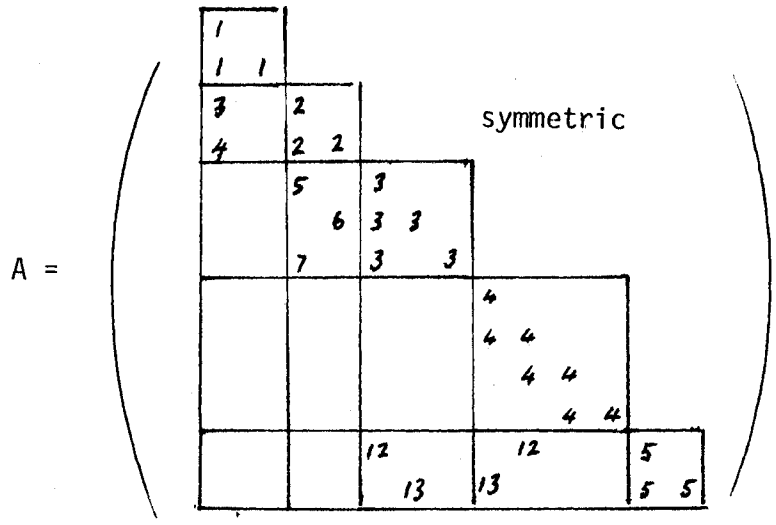
w : a real vector of length n_w, containing the nonzero components (outside the diagonal blocks) of consecutive rows of the lower triangle of A.

j : an integer vector of length n_w, where j(i) is the column subscript of the component stored in w(i).

v : an integer vector of length N. The number v(i) is the first position in w where the components of row i of A are stored. If v(i+1)-v(i) = 0, then no components of row i are in w.

Figure 5.1 is an example of a matrix stored using the arrays just described, with $n_b = 5$, $n_w = 9$, and $n_d = 22$. The array $\mathcal{Q}$ contains the components of diagonal blocks of A, which are overwritten by those of L during the factorization.

The storage required for the vectors j, d, v, s, and p must be regarded as "overhead" storage, since it is not used to store actual data. In addition, in our implementation of the F_2 version of the factorization (see section 2) we found it convenient to have an extra auxiliary real vector of length N. In the F_1 version, we need temporary storage for the largest W_i which occurs, along with another integer vector of length equal to the maximum number of columns in any of the W_i. This overhead storage is reported in the tables in section 6.



S : 2 3 5 5 0

p : 1 3 5 8 12 14

ℓ : 1 1 1 2 2 2 3 3 3 3 0 3 4 4 4 4 4 4 4 5 5 5

d : 1 3 4 6 7 9 12 13 15 17 19 20 22

j : 1 1 3 4 3 5 9 6 8

w : 3 4 5 6 7 12 12 13 13

v : 1 1 1 2 3 4 5 5 5 5 5 6 8 10

Figure 5.1 Example showing the arrays used in the implicit storage scheme

§6. Experimental Results

We described in section 3 how any ordering scheme which generates a level structure, and then numbers the levels consecutively induces a natural block tri-diagonal partitioning in the corresponding matrix. Two such algorithms, which experience has shown to be effective in producing small bandwidths and envelopes, are the reverse Cuthill-McKee (RCM) ordering algorithm and the algorithm recently developed by Gibbs et al. (GPS). We described their basic features in section 3. Recall that our implementation of the RCM algorithm uses the technique developed by Gibbs et al. to find the endpoint of a pseudo-diameter (pseudo peripheral node) as a starting node.

We are interested in answers to the following questions:

a) Does the use of our implementation of Cholesky's method (BLKSLV) significantly reduce storage requirements over using the ordinary envelope scheme (ENVSLV) proposed by Jennings [10], for the orderings produced by RCM and GPS?

b) How does the combination of our ordering algorithm (RQT) and BLKSLV compare with the combinations in a)?

c) Which of the two possible ways of performing the computation in BLKSLV (F_1 or F_2) appears to be more efficient for the class of problems we are considering?

In order to obtain answers to these questions, we report on the performance of the following ordering-algorithm/solver combinations:

1) RCM - ENVSLV

2) RCM - BLKSLV (F_1)

3) RCM - BLKSLV (F_2)

4) GPS - ENVSLV

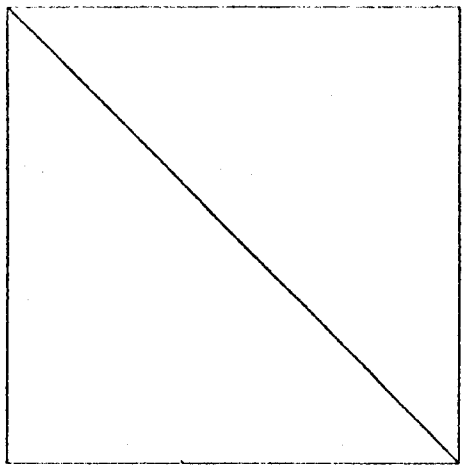
5) GPS - BLKSLV (F_1)

6) GPS - BLKSLV (F_2)

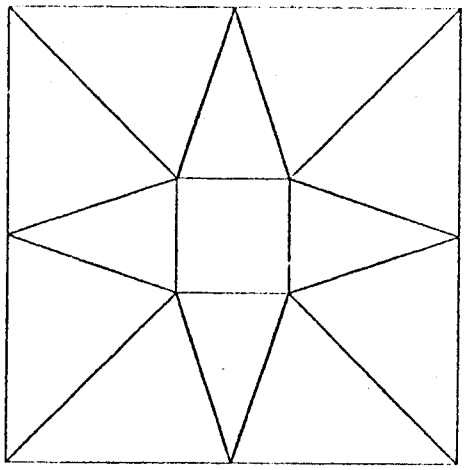
7) RQT - BLKSLV (F_1)

8) RQT - BLKSLV (F_2)

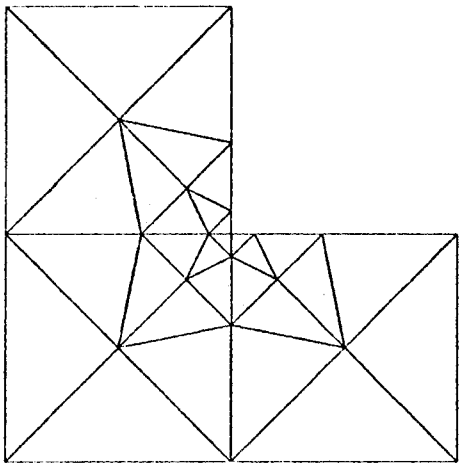
Our test problems consist of six two-parameter planar mesh problems typical of those arising in structural analysis. The basic meshes shown in Figure 6.1 are subdivided by the factor α in the obvious way, yielding a mesh having $\alpha(\alpha+1)/2$ times as many triangles as the original mesh, as shown in the examples in Figures 6.2-6.3 for various values of α. The second parameter μ governs the distribution of nodes on the mesh, and corresponds essentially to the degree of certain piecewise polynomial bases used in finite element applications [9,16,18]. For any $\mu \geq 1$, there is one node at each vertex of the mesh, $\mu-1$ nodes along each edge, and $(\mu-1)(\mu-2)/2$ nodes in the interior of each element (triangle). For our purposes, the coefficient matrix of the corresponding symmetric positive definite matrix problem $Ax = b$ has the property that $A_{ij} \neq 0 \iff x_i$ and x_j are associated with nodes which belong to the same mesh triangle.



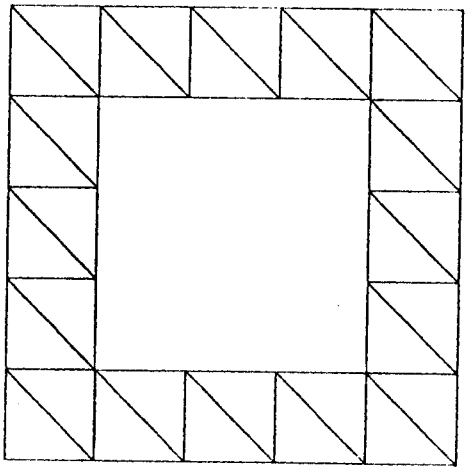
a) Square



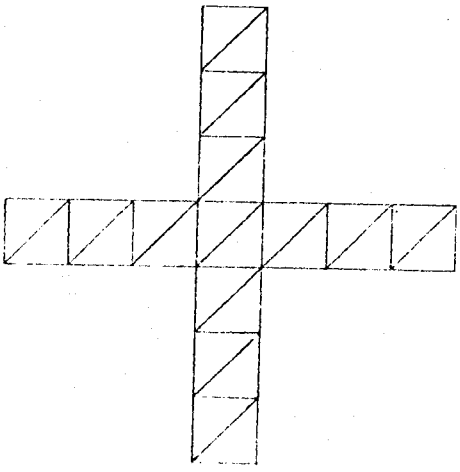
b) Hollow square (small hole)



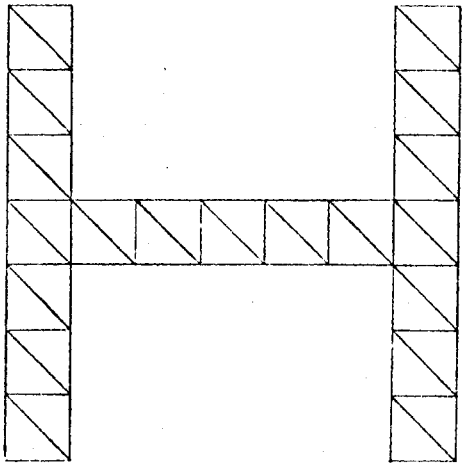
c) Graded L



d) Hollow square (large hole)



e) + shaped domain



f) H-shaped domain

Figure 6.1 Mesh Problems with $\alpha=1$

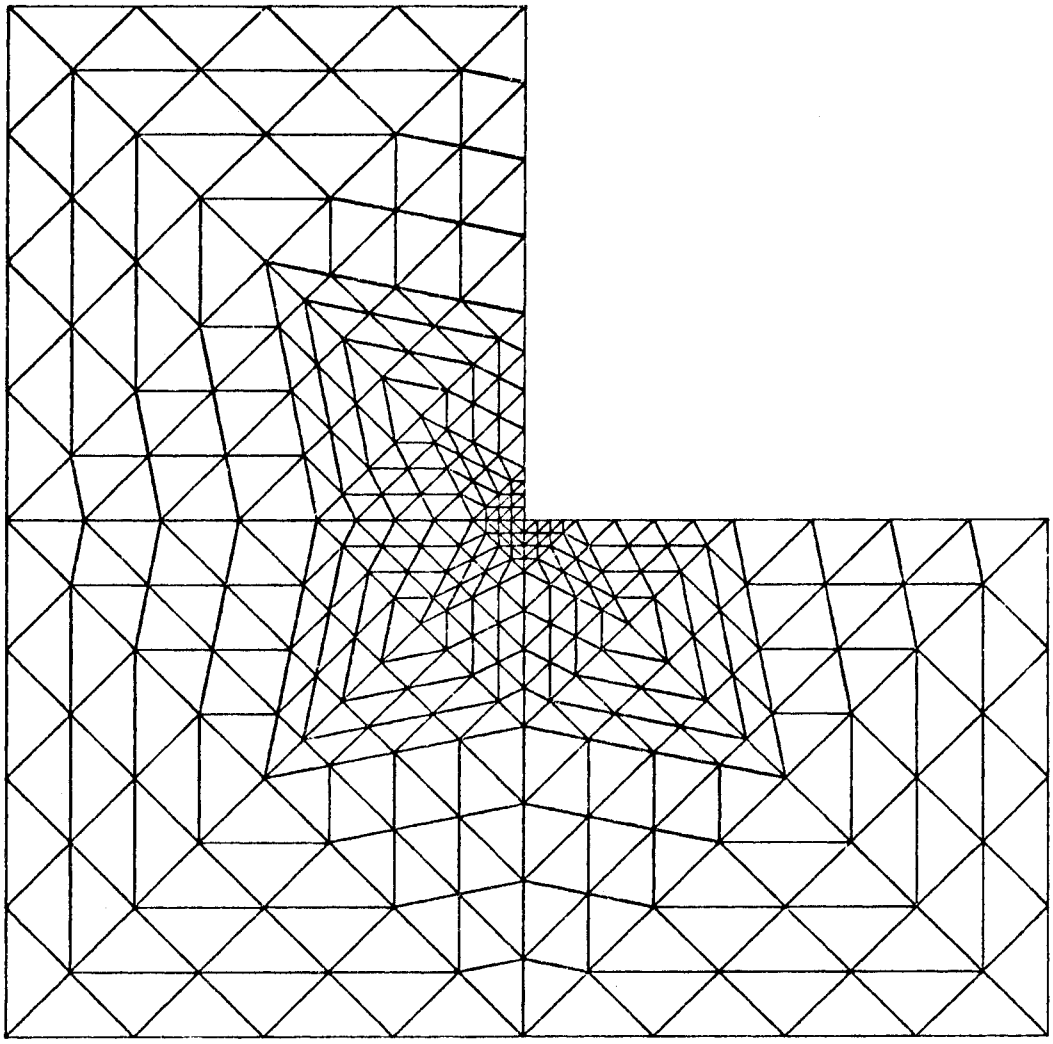


Figure 6.2 Graded L domain with α=4

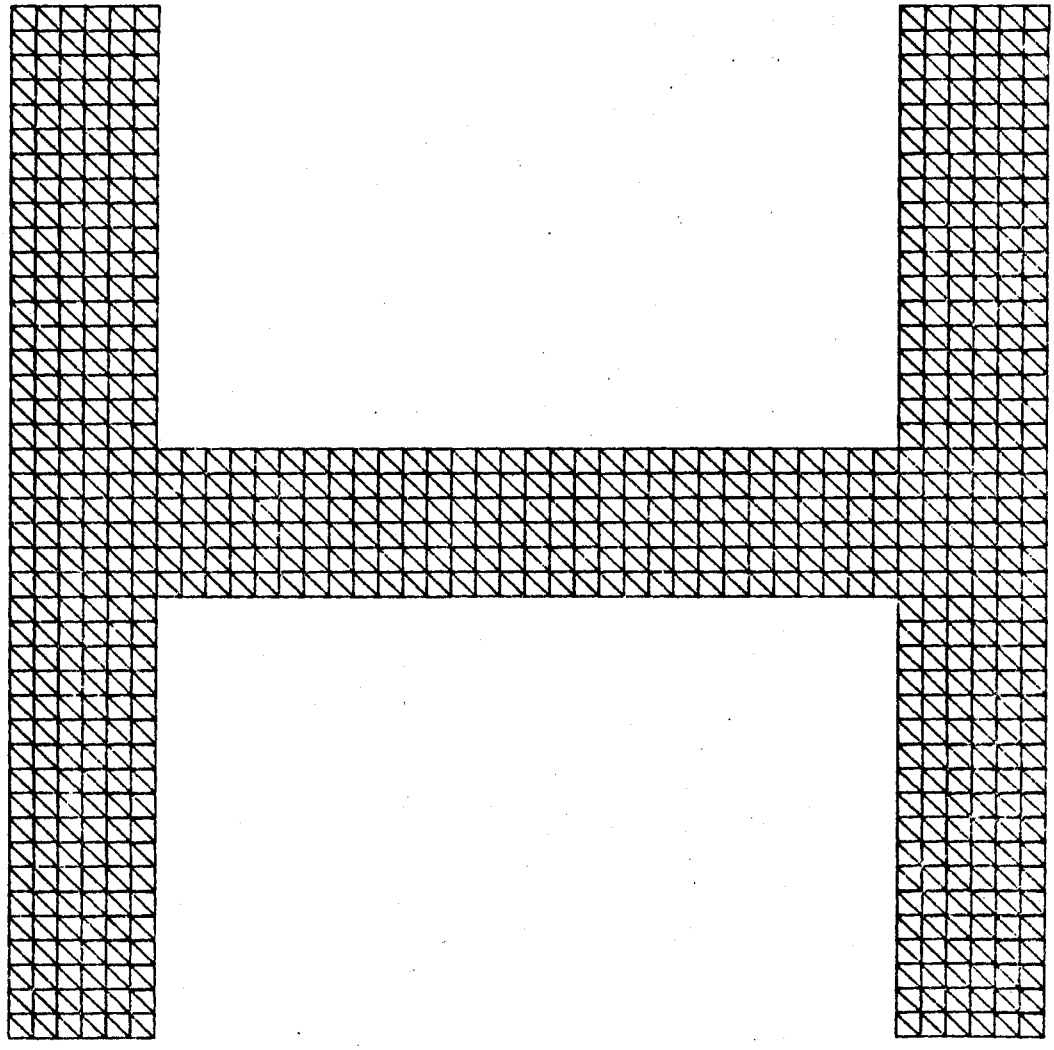


Figure 6.3 H-shaped domain with α=6

$N = 1089$ $\mu = 1$ $\alpha = 32$	Ordering			Storage†			Factorization		Solving	
	Time	Bandwidth	Profile	Primary	Overhead	Total	Operations	Time	Operations	Time
RCM-ENVSLV	0.60	33	25553	25553	1092	26645	344608	3.52	51106	0.46
RCM-BLKSLV(F_1)				14641	5543	20184	344608	4.41	54338	0.71
RCM-BLKSLV(F_2)				14641	5510	20151	560032	6.58	54338	0.63
GPS-ENVSLV	1.01	33	25553	25553	1092	26645	344608	3.56	51106	0.44
GPS-BLKSLV(F_1)				14641	5543	20184	344608	4.50	54338	0.64
GPS-BLKSLV(F_2)				14641	5510	20151	560032	6.70	54338	0.64
RQT-BLKSLV(F_1)	0.67	33	25553	14641	5543	20184	344608	4.50	54338	0.68
RQT-BLKSLV(F_2)				14641	5510	20151	560032	6.78	54338	0.67
$N = 961$ $\mu = 2$ $\alpha = 15$										
RCM-ENVSLV	0.81	65	23800	23800	964	24764	334114	3.34	47600	0.40
RCM-BLKSLV(F_1)				14865	8077	22942	334114	3.87	54838	0.58
RCM-BLKSLV(F_2)				14865	5256	20121	547581	5.99	54838	0.60
GPS-ENVSLV	1.12	63	37310	37310	964	38274	-	-	-	-
GPS-BLKSLV(F_1)				20365	7895	28260	845503	8.41	76708	0.74
GPS-BLKSLV(F_2)				20365	5316	25681	1221301	11.82	76708	0.74
RQT-BLKSLV(F_1)	1.11	119	31606	14106	8077	22183	494755	5.09	51802	0.55
RQT-BLKSLV(F_2)				14106	5256	19362	619059	6.47	51802	0.56

Table 6.1 Tabulated results for the square domain

† Primary storage is the storage required to store entries in the matrix; overhead storage is used to store subscripts, pointers, etc. Total storage is the amount of storage used, excluding that needed for the right-hand side.

$N = 936$ $\mu = 1$ $\alpha = 12$	Ordering			Storage			Factorization		Solving	
	Time	Bandwidth	Profile	Primary	Overhead	Total	Operations	Time	Operations	Time
RCM-ENVSLV	0.47	27	22753	22753	939	23692	301788	2.99	45506	0.39
RCM-BLKSLV(F_1)				12967	4733	17700	301788	3.78	48266	0.56
RCM-BLKSLV(F_2)				12967	4707	17674	489665	5.69	48266	0.56
GPS-ENVSLV	0.74	27	22753	22753	939	23692	301788	2.99	45506	0.39
GPS-BLKSLV(F_1)				12967	4733	17700	301788	3.81	48266	0.57
GPS-BLKSLV(F_2)				12967	4707	17674	489665	5.74	48266	0.59
RQT-BLKSLV(F_1)	0.51	28	23061	12967	4733	17700	310029	3.86	48266	0.57
RQT-BLKSLV(F_2)				12967	4707	17674	493141	5.84	48266	0.56
$N = 936$ $\mu = 2$ $\alpha = 6$										
RCM-ENVSLV	0.88	58	24842	24842	939	25781	356872	3.41	49684	0.42
RCM-BLKSLV(F_1)				15367	6911	22278	356872	4.06	57002	0.61
RCM-BLKSLV(F_2)				15367	5091	20458	590312	6.88	57002	0.61
GPS-ENVSLV	1.11	56	36608	36608	939	37547	-	-	-	-
GPS-BLKSLV(F_1)				19628	6959	26587	786126	8.12	73932	0.73
GPS-BLKSLV(F_2)				19628	5139	24767	1103268	11.30	73932	0.75
RQT-BLKSLV(F_1)	1.26	103	33772	14706	6911	21617	548162	5.31	54358	0.55
RQT-BLKSLV(F_2)				14706	5091	19797	683297	7.12	54358	0.58

Table 6.2 Tabulated results for the hollow square domain (small hole)

N = 1009 μ = 1 α = 8	Ordering			Storage			Factorization		Solving	
	Time	Bandwidth	Profile	Primary	Overhead	Total	Operations	Time	Operations	Time
RCM-ENVSLV	1.32	35	30028	30028	1012	31040	487992	4.54	60056	0.50
RCM-BLKSLV(F_1)				16732	5183	21915	487992	5.60	63006	0.67
RCM-BLKSLV(F_2)				16732	5070	21802	796388	8.58	63006	0.67
GPS-ENVSLV	0.98	33	25802	25802	1012	26814	362124	3.56	51604	0.43
GPS-BLKSLV(F_1)				14717	5199	19916	362124	4.46	54946	0.62
GPS-BLKSLV(F_2)				14717	5086	19803	596075	6.90	54946	0.62
RQT-BLKSLV(F_1)	1.41	35	29673	16717	5183	21900	476036	5.55	62946	0.69
RQT-BLKSLV(F_2)				16717	5070	21787	788944	8.40	62946	0.67
N = 1009 μ = 2 α = 4										
RCM-ENVSLV	1.19	69	27976	27976	1012	28988	422357	4.29	55952	0.48
RCM-BLKSLV(F_1)				17635	8792	26427	422357	4.85	65672	0.71
RCM-BLKSLV(F_2)				17635	5511	23146	720559	7.74	65672	0.72
GPS-ENVSLV	1.42	67	38530	38530	1012	39542	-	-	-	-
GPS-BLKSLV(F_1)				22355	8792	31147	825303	8.99	84552	0.80
GPS-BLKSLV(F_2)				22355	5511	27866	1147497	11.69	84552	0.84
RQT-BLKSLV(F_1)	1.68	120	38822	16807	8792	25599	676549	6.96	62360	0.68
RQT-BLKSLV(F_2)				16807	5511	22318	844221	8.70	62360	0.67

Table 6.3 Tabulated results for the graded L domain

N = 1440 μ = 1 α = 9	Ordering			Storage			Factorization		Solving	
	Time	Bandwidth	Profile	Primary	Overhead	Total	Operations	Time	Operations	Time
RCM-ENVSLV	0.69	21	28218	28218	1443	29661	300226	3.14	56436	0.49
RCM-BLKSLV(F_1)				16526	7259	23785	300226	4.44	60630	0.80
RCM-BLKSLV(F_2)				16526	7239	23765	484736	6.64	60630	0.84
GPS-ENVSLV	1.56	21	28218	28218	1443	29661	300226	3.42	56436	0.55
GPS-BLKSLV(F_1)				16526	7259	23785	300226	4.17	60630	0.76
GPS-BLKSLV(F_2)				16526	7239	23765	484736	6.25	60630	0.77
RQT-BLKSLV(F_1)	0.77	22	28245	16526	7259	23785	300496	4.47	60630	0.84
RQT-BLKSLV(F_2)				16526	7239	23765	485600	6.33	60630	0.79
N = 1152 μ = 2 α = 4										
RCM-ENVSLV	0.76	44	22200	22200	1155	23355	232144	6.95†	44400	0.40
RCM-BLKSLV(F_1)				14684	6407	21091	232144	7.85	53358	0.65
RCM-BLKSLV(F_2)				14684	6227	20911	392004	11.91	53358	0.66
GPS-ENVSLV	1.37	40	32943	32943	1155	34098	-	-	-	-
GPS-BLKSLV(F_1)				19524	6439	25963	518798	19.06	72644	0.78
GPS-BLKSLV(F_2)				19524	6259	25783	750097	23.81	72644	0.78
RQT-BLKSLV(F_1)	1.28	71	30400	14166	6407	20573	360926	5.28	51286	0.59
RQT-BLKSLV(F_2)				14166	6227	20393	455338	7.39	51286	0.60

Table 6.4 Tabulated results for the hollow square domain (large hole)

† The apparently inconsistent timing in this column is due to underflow interrupts.

N = 1180 μ = 1 α = 9	Ordering			Storage			Factorization		Solving	
	Time	Bandwidth	Profile	Primary	Overhead	Total	Operations	Time	Operations	Time
RCM-ENVSLV	0.82	31	25860	25860	1183	27043	332412	3.26	51720	0.42
RCM-BLKSLV(F_1)				15013	5940	20953	332412	4.04	55604	0.64
RCM-BLKSLV(F_2)				15013	5910	20923	539721	6.10	55604	0.64
GPS-ENVSLV	1.17	21	22630	22630	1183	23813	235208	2.48	45260	0.41
GPS-BLKSLV(F_1)				13296	5939	19235	235208	3.33	48714	0.65
GPS-BLKSLV(F_2)				13296	5919	19215	379424	4.77	48714	0.64
RQT-BLKSLV(F_1)	0.79	235	19759	9645	6028	15673	123215	2.07	34132	0.53
RQT-BLKSLV(F_2)				9645	6000	15645	190873	2.95	34132	0.52
N = 945 μ = 2 α = 4										
RCM-ENVSLV	1.04	58	19862	19862	948	20810	241666	2.41	39724	0.36
RCM-BLKSLV(F_1)				12965	7095	20060	241666	2.92	47522	0.52
RCM-BLKSLV(F_2)				12965	5070	18035	409569	4.76	47522	0.54
GPS-ENVSLV	1.47	40	20697	20697	948	21645	254761	2.72	41394	0.38
GPS-BLKSLV(F_1)				14639	5497	20136	254761	3.51	54098	0.59
GPS-BLKSLV(F_2)				14639	5110	19749	534489	6.41	54098	0.61
RQT-BLKSLV(F_1)	1.45	222	19355	8769	6517	15286	138166	1.88	30738	0.41
RQT-BLKSLV(F_2)				8769	5110	13879	179606	2.53	30738	0.41

Table 6.5 Tabulated results for the + shaped domain

$N = 1377$ $\mu = 1$ $\alpha = 8$	Ordering			Storage			Factorization		Solving	
	Time	Bandwidth	Profile	Primary	Overhead	Total	Operations	Time	Operations	Time
RCM-ENVSLV	0.79	27	21682	21682	1380	23062	195063	2.08	43364	0.39
RCM-BLKSLV(F_1)				13062	6967	20029	194028	2.87	47078	0.63
RCM-BLKSLV(F_2)				13062	6942	20004	309558	4.23	47078	0.64
GPS-ENVSLV	1.57	27	21682	21682	1380	23062	195063	2.18	43364	0.38
GPS-BLKSLV(F_1)				12212	6642	18854	125039	2.01	44328	0.60
GPS-BLKSLV(F_2)				12212	6617	18829	210521	2.94	44328	0.60
RQT-BLKSLV(F_1)	0.86	189	18857	10124	7047	17171	106173	1.98	35326	0.58
RQT-BLKSLV(F_2)				10124	7022	17146	164757	2.89	35326	0.60
$N = 805$ $\mu = 2$ $\alpha = 3$										
RCM-ENVSLV	0.73	43	10996	10996	808	11804	85875	0.99	21992	0.20
RCM-BLKSLV(F_1)				7787	4841	12628	84524	1.32	27522	0.37
RCM-BLKSLV(F_2)				7787	4314	12101	140990	1.98	27522	0.37
GPS-ENVSLV	1.00	42	13611	13611	808	14419	137205	1.45	27222	0.25
GPS-BLKSLV(F_1)				9307	5340	14647	136525	1.84	33542	0.40
GPS-BLKSLV(F_2)				9307	4339	13646	234751	2.85	33542	0.40
RQT-BLKSLV(F_1)	1.13	132	12021	6399	4871	11270	69679	1.11	21970	0.33
RQT-BLKSLV(F_2)				6399	4344	10743	92318	1.49	21970	0.33

Table 6.6 Tabulated results for the H shaped domain

§7. Concluding Remarks

To a large extent, the experimental results appearing in Tables 6.1-6.6 speak for themselves. We offer the following observations and remarks.

1) Our ordering algorithm/solution package appears to be as good as its competition in all our examples, and substantially superior when the domain has appendages and/or when high order elements ($\mu > 1$) are used. This latter aspect seems particularly important, since the trend in finite element applications seems to be towards the use of these more sophisticated elements.

2) Even for problems of around 1000 equations, the time spent in finding the ordering of the equations is still a nontrivial fraction of the overall time spent in solving the problem.

In this connection, in all three algorithms, a substantial portion of the ordering time (typically 40-50 percent) was spent finding the pseudo-peripherial node(s) from which the rooted level structures were generated.

3) As described earlier, we reported primary and overhead storage separately. In our implementation one storage location was used for each overhead item (subscript, pointer, etc.). On machines which have a large word size, it might make sense to pack such data two or three items to a word. This would mean our scheme would lead to much more dramatic savings in storage requirements than is apparent from our tables.

Along this line, we contend that storage reduction should be regarded as at least as important as processor time, perhaps more so. Since main storage continues to remain a relatively expensive hardware resource, computing center charging algorithms are usually designed to discourage large

main storage demands. A typical charging scheme is cost = (processor time) × p_q(storage used), where $p_q(x)$ is a polynomial of degree q, with $q \geq 1$. It is frequently the case that for large problems, halving the storage requirements more than compensates for doubling the processor time.

4) The algorithm GPS appeared to do an excellent job of finding an ordering having a small bandwidth, as it was designed specifically to do. Its performance as a profile minimizer was less consistent, however; for our problems the RCM algorithm appeared to be at least as effective, particularly for the problems with appendages and/or those involving high order elements ($\mu > 1$).

5) With regard to the variations F_1 and F_2 of BLKSLV (see section 2), for our examples, the F_1 version was significantly more efficient in terms of execution time, and for the low order elements ($\mu = 1$), required only slightly more storage. However, for $\mu > 1$, the F_2 version required substantially less storage.

References

[1] I. Arany, W.F. Smyth and L. Szoda, "An Improved Method for Reducing the Bandwidth of Sparse Symmetric Matrices", in Information Processing 71: Proceedings of IFIP Congress, North-Holland, Amsterdam, 1972.

[2] C. Berge, The Theory of Graphs and its Applications, John Wiley & Sons Inc., New York, 1962.

[3] James R. Bunch, and D.J. Rose, "Partitioning, Tearing and Modification of Sparse Linear Systems", J. Math. Anal. and Appl., 48(1974), pp.574-593.

[4] E. Cuthill, "Several Strategies for Reducing the Bandwidth of Matrices", in Sparse Matrices and Their Applications, D. Rose and R. Willoughby, eds., Plenum Press, New York, 1972.

[5] E. Cuthill and J. McKee, "Reducing the Bandwidth of Sparse Symmetric Matrices", Proc. 24th Nat. Conf., Assoc. Comput. Mach., ACM Publ. P-69, 1122 Ave. of the Americas, New York, N.Y., 1969.

[6] Alan George, "Nested Dissection of a Regular Finite Element Mesh", SIAM J. Numer. Anal., 10(1973), pp.345-363.

[7] Alan George, "On Block Elimination for Sparse Linear Systems", SIAM J. Numer. Anal., 11(1974), pp.585-603.

[8] N.E. Gibbs, W.G. Poole, and P.K. Stockmeyer, "An Algorithm for Reducing the Bandwidth and Profile of a Sparse Matrix", SIAM J. Numer. Anal., to appear.

[9] M.J.L. Hussey, R.W. Thatcher and M.J.M. Bernal, "Construction and Use of Finite Elements", JIMA, 6(1970), pp.262-283.

[10] A. Jennings, "A Compact Storage Scheme for the Solution of Symmetric Linear Simultaneous Equations", Computer J. 9, 281-285 (1966).

[11] D.E. Knuth, The Art of Computer Programming, vol.I (Fundamental Algorithms) Addison Wesley, 1968.

[12] W.H. Liu and A.H. Sherman, "Comparative Analysis of the Cuthill-McKee and the Reverse Cuthill-McKee Ordering Algorithms for Sparse Matrices", SIAM J. Numer. Anal., to appear.

[13] R.S. Martin and J.H. Wilkinson, "Solution of Symmetric and Unsymmetric Band Equations and Calculation of Eigenvectors of Band Matrices", Numer. Math., 9(1967), pp.279-301.

[14] S.V. Parter, "The Use of Linear Graphs in Gauss Elimination", SIAM. Rev., 3(1961), pp.364-369.

[15] D.J. Rose, "A Graph-theoretic Study of the Numerical Solution of Sparse Positive Definite Systems of Linear Equations", in Graph Theory and Computing, edited by R.C. Read, Academic Press, New York, 1972.

[16] P. Silvester, "High-order Polynomial Triangular Finite Elements for Potential Problems", Internat. J. Engrg. Sci., 7(1969), pp.849-861.

[17] James H. Wilkinson, The Algebraic Eigenvalue Problem, Clarendon Press, Oxford, 1965.

[18] O.C. Zienkiewicz, The Finite Element Method in Engineering Science, McGraw Hill, London, 1970.