The Multifrontal Solution of Indefinite Sparse Symmetric Linear Equations

I. S. DUFF and J. K. REID
AERE Harwell, England

We extend the frontal method for solving linear systems of equations by permitting more than one front to occur at the same time. This enables us to develop code for general symmetric systems. We discuss the organization and implementation of a multifrontal code which uses the minimum-degree ordering and indicate how we can solve indefinite systems in a stable manner. We illustrate the performance of our code both on the IBM 3033 and on the CRAY-1.

Categories and Subject Descriptors: G.1.3 [**Numerical Analysis**]: Numerical Linear Algebra—*linear systems (direct methods), sparse and very large systems*; G.4 [**Mathematics of Computing**]: Mathematical Software—*algorithm analysis, efficiency, reliability and robustness*

General Terms: Algorithms, Experimentation, Performance

Additional Key Words and Phrases: Sparse matrices, indefinite symmetric matrices, linear equations, frontal methods, vector processing, minimum-degree algorithm, generalized elements

1. INTRODUCTION

We consider the direct solution of sparse symmetric sets of n simultaneous linear equations

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

If A is positive definite, then any choice of pivots from the diagonal is numerically stable so they can be chosen on sparsity grounds alone. Choosing pivots from the diagonal preserves symmetry and conventional practice is to choose the diagonal pivots by symbolic processing on the sparsity pattern. The numerical factorization of A and the solution of Eq. (1.1) can then be performed within a known sparsity pattern that accommodates the fill-ins. Prominent examples of codes that implement this approach are the Yale Sparse Matrix Package (YSMP) [9], [10] and SPARSPAK [18]. An excellent description of a variety of pivotal strategies and storage schemes is provided in the book of George and Liu [17].

This approach may fail if A is indefinite; for instance, the very first pivot may be zero. Duff et al. [7] overcame this difficulty by following Bunch and Parlett [1] in using a mixture of 1×1 and 2×2 pivots chosen during the numerical factorization of A, and demonstrated that code based on this approach is com-

Authors' address: Computer Science and Systems Division, AERE Harwell, Oxford OX11 0RA, England.

petitive with the Harwell code MA17A [22] both in speed and storage. However, an additional subroutine, MA17E, has since been added to the MA17 package to choose pivots symbolically and is significantly faster. Even this is not competitive with the symbolic phases of the YSMP and SPARSPAK codes, which store the fill-ins implicitly rather than explicitly.

The aim of this work is to use the ideas of frontal elimination [20, 19] to permit stable numerical factorization to be performed based on a symbolically chosen pivotal sequence. An outline of our approach is given in Section 2; we describe the symbolic phase of our algorithm in Sections 3–5, the numerical factorization in Section 6, and the numerical solution in Section 7. We refer to these phases as ANALYZE, FACTOR, and SOLVE, and they are represented in our code by three separate entries. Finally, in Section 8 we give the results of runs on both a scalar machine (IBM 3033) and a vector machine (CRAY-1, actually a pipeline machine able to process vectors efficiently). A side advantage of our approach is that inner loops of the numerical phases are vectorizable, which gives a speed advantage, particularly on large problems.

When performing the numerical experiments for this paper, we used sets of test matrices covering a wide range of sparsity patterns and applications. Among the test matrices used (the order of the matrices in the tables together with an acknowledgment of the source follows each in parentheses) were a bandlike matrix from eigenvalue calculations in atomic physics (147, Johansson, Lund), a matrix from electrical circuit analysis (1176, Erisman, Boeing), normal equations matrices from surveying (85, 292, Ashkenazi, Nottingham), a Jacobian from a stiff system of ordinary differential equations in laser research (130, Curtis, Harwell), a parameterized set of problems from the triangulation of an L-shaped region (406, 1009, 1561, George and Liu [15]), matrices from the analysis of ship structures (503, 1005, Everstine [14]), a power network matrix (274, Lewis, Boeing), and the matrix from the 9-point discretization of the Laplacian on a 30 × 30 grid (900). We used many more matrices in our experiments, but the above sample is representative of these and is sufficient to illustrate our points.

2. OVERALL STRATEGY

It is convenient to discuss our overall strategy in terms of finite-element problems, although it is applicable to general systems. Finite-element problems occur naturally with matrices of the form

$$A = \sum_l B^{(l)}, \tag{2.1}$$

where each $B^{(l)}$ is the contribution from a single finite element and is zero except in a small number of rows and columns. Each $B^{(l)}$ may conveniently be stored as a small full matrix together with a vector of indices to label where each row and column of this packed matrix fits into the overall matrix A. In the frontal method, advantage is taken of the fact that elimination steps

$$a_{ij} := a_{ij} - a_{ik}\, a_{kk}^{-1} a_{kj} \tag{2.2}$$

do not have to wait for all the assembly steps

$$a_{ij} := a_{ij} + b_{ij}^{(l)} \tag{2.3}$$

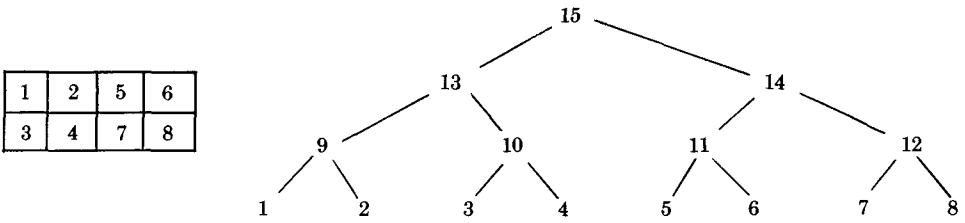

Fig. 1. A finite-element problem and its assembly tree.

from (2.1) to be complete. The operation (2.2) can be performed as soon as the pivot row and column is fully summed, that is, as soon as all operations (2.3) have been completed for them, for if any of a_{ik}, a_{kj}, and a_{kk} are not fully summed, then the wrong quantity will be subtracted in (2.2).

Irons' frontal method [20] treats the symmetric positive-definite case. An assembly order is chosen a priori, and each variable is eliminated as soon as it is fully summed, that is, after its last occurrence in a $B^{(l)}(l = 1, 2, \ldots)$. Hood [19] extended this idea to unsymmetric matrices with a symmetric sparsity pattern, and Duff [4], [5] has provided an implementation that treats unsymmetric patterns. Pivots are chosen on numerical grounds from the submatrix of fully-summed rows and columns. Some pivots are now off the main diagonal, and all the fully-summed variables are not necessarily eliminated at once. In both cases all operations are performed in a full matrix (called the frontal matrix) that corresponds to the submatrix of rows and columns belonging to variables that have appeared in one or more of the $B^{(l)}$ so far assembled, but have not yet been eliminated. After each elimination the pivot row is stored ready for use in back substitution. In the unsymmetric case, if numerically satisfactory pivots cannot be found for all the fully-summed variables, then the frontal matrix passed on to the next step is a little larger. In practice the extra storage and work requirements are slight.

The frontal method can be generalized to correspond to any bracketing of the summation (2.1). It is convenient to represent such a bracketing by an assembly tree. A simple example is shown in Figure 1 and corresponds to the summation

$$[[B^{(1)} + B^{(2)}] + [B^{(3)} + B^{(4)}]] + [[B^{(5)} + B^{(6)}] + [B^{(7)} + B^{(8)}]].$$

The summation is now performed from inner brackets outward, and again we may eliminate any variable immediately it is fully summed. In general, we now have several frontal matrices, corresponding to inner brackets that we have processed and have had to set aside temporarily. Notice that these frontal matrices are just like the original matrices $B^{(l)}$ and they are often called "generalized element matrices," though here we call them "generated element matrices" because they are generated by pivotal steps. The advantage of this approach is that the extra generality permits choices which are more economical in arithmetic operations. A prominent example of such a grouping is nested dissection (also known as nested substructuring). Here the overall problem is divided into parts and the parts are further subdivided, etc.; each node of the tree or bracket in the bracketed sum corresponds to a substructure.

Less obviously, the general-purpose and remarkably successful pivotal strategy of "minimum degree" can be implemented in this way. Here each pivot is chosen from the diagonal of the current reduced matrix in a row with least nonzeros. The number of nonzeros in a row is just the number of distinct variables in all the elements that contain the corresponding variable (we assume that each $B^{(l)}$ is a full submatrix). Furthermore, we can represent the elimination of a chosen variable by the assembly of these elements followed by its elimination within the generated element matrix. Following this, if any variable is within the new element and no others, then its degree must be the new minimum degree (one less than the previous minimum), and so it may be eliminated at once. By similar reasoning we may eliminate all such variables in turn. This leaves us with a reduced matrix of the form (2.1) to which the same argument may be applied. Normally this continues until we are left with a single generated element, and we may represent the minimum-degree order as an assembly tree with this generated element as its root, other generated elements as internal nodes, and original elements as terminal nodes. Exceptionally, and only if A is reducible (permutable to block-diagonal form), some null generated elements may also be left. In this case we have a forest of trees, each representing an independent computation. Though care must be taken to allow for this in a general-purpose code, it will happen rarely in practice. We do not discuss it further in this paper.

Even if the original matrix is not a finite-element matrix, this approach can be used. We may begin with an ordinary coefficient matrix A and treat each nonzero on the diagonal as a finite element involving one variable and each nonzero off the diagonal as a finite element involving two variables. During symbolic processing, this has the merit of never needing more storage than was required for the original matrix, since each generated element $B^{(l)}$ is represented by an index list formed by merging the index lists of the original row and the other generated elements involved, none of which are needed later. A further advantage during symbolic processing is that of speed because of working with linear index lists instead of square arrays for the matrices $B^{(l)}$. It is these considerations which give YSMP and SPARSPAK their advantage over older codes, such as Harwell's MA17.

There are advantages, too, in the numerical phases when using this approach. Performing all arithmetic operations on full matrices gives a performance advantage for large problems, particularly on a vector machine (see Section 8). Also the indefinite problem can be handled by permitting some numerical pivoting during the numerical factorization, following the ideas of Hood [19]. Off-diagonal pivots would destroy symmetry, so we use instead a mixture of 1×1 and 2×2 pivots. Details of our pivotal strategy are given in Section 6. We do, however, permit the front to grow slightly over what it would be in the positive-definite case. Our experience (see Section 6, Table VII) is that the effect is not very severe.

An alternative to starting with artificial finite elements with 1 or 2 variables is to look for higher order dense submatrices and thus reduce the number of terms in the sum (2.1).

The following algorithm, expressed in an informal version of ALGOL 60, constructs a set of element matrices $b^{(l)}$, $l = 1, 2, \ldots$, whose combined patterns span the sparsity pattern of a given matrix A.

Table I. Some Statistics on Artificial Generation of Elements

Matrix order	147	1176	292	85
Number of original nonzeros	1298	9864	1250	304
Number of generated elements	157	907	373	102
Number of reals in generated elements	2970	11,286	2029	481
Time to sort nonzeros and to generate elements (IBM 3033 seconds)	0.039	0.198	0.031	0.008
Ordering time (final code, MA27) (IBM 3033 seconds)	0.038	0.295	0.075	0.019

Algorithm 2.1

Flag all nonzeros a_{ij} as unused;
for each row i **do**
  while row i contains a flagged nonzero **do**
  begin add an artificial element containing just variable i;
    unflag a_{ii};
    for each flagged nonzero a_{ij} in row i **do**
    if $a_{kj} \neq 0$ for all variables k of the current element
    then add variable j to the element and unflag both a_{kj} and a_{jk} for all variables k now in the element
  end;

Given a problem that comes from a finite-element application, this algorithm usually constructs artificial elements that each span one or more natural elements, each "seeded" from an off-diagonal nonzero in a natural element. Thus the number of elements is usually near minimal (the minimal representation may be regarded as the natural representation that the algorithm seeks to recover). However, the following example illustrates that it does not always achieve the minimum number. If the natural elements involve variables (1, 2, 4), (1, 3, 5), (2, 3, 6), then the algorithm first generates the artificial element (1, 2, 3), then finds the three actual elements.

This scheme has the merit of some elegance. A preprocessing phase converts to a finite-element form and we then use a finite-element approach. However, the results in practice were disappointing. We show a few statistics in Table I from running our test problems. The number of generated elements was about n, and the number of reals in them was sometimes significantly greater than in the original matrix (of course, the number cannot be less). Finally, we were not happy with the time taken. We show in Table I how long this preprocessing takes compared with the whole ordering by our final code. It can be seen that the mere generation of the artificial elements can be quite a substantial overhead and we were unable to take advantage of this form during subsequent processing. We therefore abandoned this approach.

Given an assembly tree, or a set of brackets for the summation, there remains some choice for the order in which the operations are performed. For instance, any of the inner brackets may be processed first. We have found advantages in basing the order on the following variant of a depth-first tree search:

											8			
				4				6		7	7	12		
	2		3	3	10		5	5	11	11	11	11	14	
1	1	9	9	9	9	13	13	13	13	13	13	13	13	15

Fig. 2. Successive stack contents for the Figure 1 example.

```
Mark all nodes unused
Set node i to the root
while node i is unused do
  begin if i has an unused son
    then replace i by its leftmost unused son
    else
    begin mark i used
      if i has a right-hand brother
      then replace i by its next right-hand brother
      else replace i by its father
    end
  end
```

At each point where we return to a father from his rightmost son, we perform the assemblies of the elements corresponding to his sons and store the resulting generated element. For instance, such a search of the tree in Figure 1 will pass the nodes in the order 15, 13, 9, 1, 2, **9**, 10, 3, 4, **10**, **13**, 14, 11, 5, 6, **11**, 12, 7, 8, **12**, **14**, **15**, where we have bold-faced the son–father steps at which operations are performed. Because of using a depth-first tree search order, whenever we return to a father, the generated elements corresponding to his sons will be available in store. Because each is used just once we may free its space after use. The depth-first search means that the wanted elements are the most recently generated that are still in store, so the simple storage mechanism of a stack (first-in, last-out) is suitable for temporary storage of the generated elements. We show in Figure 2 the successive stack contents for the example of Figure 1.

To save storage we thought it might be worthwhile to order the sons by degree so that the leftmost son has least degree. Since generated elements belonging to leftmost sons are stacked for longer, we hoped that this would reduce the amount of stack storage needed. However, in practice the total storage was never much reduced and sometimes even increased, so we abandoned this idea.

We are not the first to use an element model for Gaussian elimination on symmetric matrices. Variants of this approach have been suggested by Sherman [23], Eisenstat et al. [11], [12], Peters [21], and George and Liu [16] and, as we remarked earlier, the ANALYZE codes of YSMP and SPARSPAK make heavy use of this model. Indeed, although our work, which owes its genesis to a report by Speelpenning [24], has been developed independently, we owe much to presentations of these ideas at conferences. Although Eisenstat et al. [12] comment on the organization of the tree search, we believe ours is the first detailed description of such an organization and associated data structures for all three phases of solution. The most novel feature of our algorithm is the incorporation of numerical pivoting (Section 6) and the demonstration of the vectorization of both FACTOR and SOLVE on the CRAY-1 (Section 8).

3. MINIMUM-DEGREE ORDERING

In the last section we sketched how the pivotal strategy of minimum degree may be implemented for a general matrix A with the help of an assembly tree. Here we explore the algorithmic details needed for its efficient execution. We have found it best to assume that the diagonal is entirely nonzero and to start with lists of column indices for all the off-diagonal nonzeros in each row. Each column index may be regarded as pointing to a finite element of order 2 (the other variable corresponds to the diagonal).

For speed it is important to avoid an expensive search for the pivot at each stage of the elimination. We have followed the procedure used in Harwell code MA28 (see [3] and [6]) of holding doubly-linked chains of variables having the same degree. Any variable that is active in a pivotal step can then be removed from its chain without any searching, and once its new degree has been computed it can be placed at the head of the corresponding chain. This makes selecting each pivot particularly easy. It is always at the front of the current minimum-degree chain.

Once the pivot has been chosen we run through what remains of the original matrix row and all the index lists of matrices $B^{(l)}$ that involve the pivot. By merging these lists, omitting the pivot itself, we obtain the index list of the new generated element. These merges are best done with the aid of a full vector of flags so that we can avoid including any variable twice. The degrees of all these variables (but no others) may change, so we remove them from their chains, recalculate their degrees, and insert them in the chains appropriate for their new degrees. This degree computation is the most expensive part of the whole calculation and we have found it well worthwhile to recognize identical rows (for example, compare cases (a) and (c) in Table II). The degree itself provides a convenient hashing function, particularly since all those of the same degree are chained together. Variables corresponding to identical rows can be amalgamated into a "supervariable," a similar concept to the indistinguishable nodes of George and Liu [17] or the prototype vertices of YSMP. It is convenient to name the supervariable after one of its components and treat the rest as dummies. The rows of the dummies can be omitted from the data structure, and by suitably flagging them we can allow them to be ignored (and removed) whenever they are later encountered in an index list.

To calculate the degree of a variable we need to run through the index lists of its original row and all the generated elements it is in. Again a full array of flags is needed to avoid counting any variable twice. Of course, all the variables whose degrees we are calculating are in the new generated element, so we begin with the flags set for the variables of this element and the integer in which the degree is accumulated set to the number of variables in the new element. The flags permit us to check that each of the other elements has at least one variable distinct from those of the new element. Any that do not can be added into the new element without extra fill-in. They may be treated just like the elements involving the pivot, that is, removed from the structure after updating the tree. We found that including such a check adds little overhead and occasionally leads to very worthwhile savings (for example, compare cases (a) and (b) in Table II). The most dramatic change is on the test example of order 130. This may be permuted

Table II. The Effect on ANALYZE Times of Algorithmic Changes in Minimum-Degree Ordering

Order:	147	1176	292	85	130	406	503
Number of nonzeros in upper triangular part of A:	1298	9864	1250	304	713	1561	3265
ANALYZE times (IBM 3033 seconds)							
a. Final code	0.068	0.53	0.13	0.034	0.14	0.20	0.36
b. Without element amalgamations	0.071	0.60	0 14	0.034	0.42	0.21	0.38
c. Without supervariable amalgamations	0.189	0 89	0.21	0.039	0.14	0.45	1.12
d. Without elements at front of lists	0 073	0.56	0.14	0.036	0.18	0.20	0.38

to have the form

$$A = \begin{bmatrix} D & E \\ E^{\mathrm{T}} & F \end{bmatrix}$$

where D is diagonal and F is nearly full but of quite low order. The rows of E have only a few different sparsity patterns, so there are only a few different patterns in the elements generated by pivoting in D. The device we have just described permits duplicate elements to be dropped quickly from the data structure, and it is this feature which leads to much better execution time than that of YSMP or SPARSPAK for this problem (see Table X, case of order 130).

These degree calculations and the construction of the index list for the pivot row both need pointers from the variables to the elements they are in. A very convenient way to provide them is to use (as in YSMP) the same numerical name for the generated element as for the variable that was eliminated when it was created. Where this variable name occurs in a row index list, it is now interpreted as an element name, which is exactly what is needed.

Further savings on the degree calculations (for example, compare cases (a) and (d) in Table II) are possible by permuting the element numbers ahead of the variable numbers in each row list. This ensures that the element lists are searched first. Once we reach the list of variables we may remove any that have already been included in the degree calculation, since they represent off-diagonal nonzeros of the original matrix that have been overlaid by nonzeros of the generated elements.

To illustrate the advantages of the devices we have just discussed we compared runs of our final code with versions each with one of the devices removed. The results varied from making little difference to changing the computing time by a factor of 3 or more. Results illustrating all the cases we ran are shown in Table II. The times are not consistent with those of Section 8 because we ran with the Harwell statement profiler OEO2 in use.

An early version of our code was so dominated by the degree calculation that we decided to terminate it abruptly at a threshold. Once the minimum degree reached this threshold, we increased it and recalculated the degree of all the outstanding variables, terminating the calculations at the new threshold. However, the incorporation of supervariable amalgamation, element absorption, and

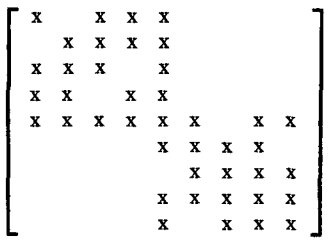

Fig. 3 The pattern of a matrix of order 9.

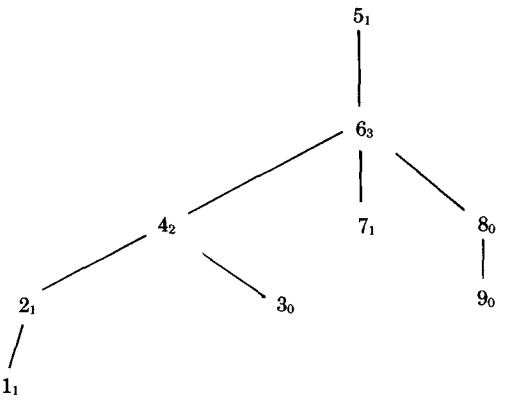

Fig. 4. The assembly and variable amalgamation tree for the matrix of Figure 3.

deletion of indices that are no longer needed so improved the calculation that using this threshold was not justifiable.

A single tree (forest in the reducible case) with n nodes suffices to record all the supervariable amalgamations and element absorptions. As each pivot is chosen we add son–father pointers for all the old elements absorbed into the new generated element. When variables are included in supervariables, we record son-to-father pointers for all the dummies pointing to the master variable. We need also to accumulate the number of variables in each supervariable in order to calculate degrees correctly, and the value zero provides a convenient flag to indicate dummies. The assembly tree (or forest in the reducible case) may be obtained from all the son–father pointers not flagged as dummies. For example, the matrix of Figure 3 might give rise to the tree of Figure 4, with numbers of variables in supervariables given as subscripts to the node numbers. This tree might be constructed by application of the minimum-degree ordering thus:

(i) Eliminate variable 1 of degree 4. The fills in rows 3 and 4 make them identical. Generated element 1 has supervariables (4, 3), (5).
(ii) Eliminate variable 2 of degree 4. Element 2 has supervariables (4, 3), (5) and can absorb element 1.
(iii) Eliminate supervariable (4, 3) of degree 3, after assembling element 2. Element 4 has supervariable (5) only.
(iv) Eliminate variable 7 of degree 4. The fills in rows 6 and 9 make rows 6, 8, 9 identical. We suppose 9 was found identical to 8 and then (8, 9) to 6. Generated element 7 has supervariable (6, 8, 9) only.
(v) Eliminate supervariable (6, 8, 9) of degree 4 after assembling element 7. Generated element contains variable 5 only and absorbs element 4.
(vi) Eliminate variable 5 after assembling element 6.

4. MANAGEMENT OF THE TREE SEARCH

The depth-first tree search cannot be performed until the tree is complete and local ordering strategies, such as minimum degree, construct the tree from the

terminal nodes toward the root. Since each node has at most one father, it is convenient to record the tree in the form of son-father pointers only. Of course, we must have a means of finding all the sons of each node if we are to perform a depth-first search efficiently. However efficiently it is done, it is sensible to do it just once, rather than repeatedly for each set of new numerical values. We therefore discuss in this section how the search may be managed and how the result may be recorded for convenient subsequent numerical processing.

The depth-first search determines a tentative pivotal sequence, which may be modified for numerical stability. This sequence divides naturally into blocks, each corresponding to a node of the assembly tree. During numerical factorization we hold the generated elements temporarily on a stack and each (block) pivotal step involves merging finite elements with one or two variables corresponding to nonzeros on and off the diagonal in the pivotal rows of the original matrix with a certain number (sometimes none) of generated elements from the top of the stack. It seems natural therefore to hold the pivotal sequence and for each block pivotal step, the number of variables eliminated and the number of stack elements assembled. For instance, depth-first search of the tree of Figure 4 yields the following steps

(1) Eliminate variable 1 after assembling elements from row 1; stack resulting generated element 1.
(2) Eliminate variable 2 after assembling elements from row 2 and stacked element 1; stack the resulting element 2.
(3) Eliminate variables 4, 3 after assembling elements from rows 4, 3 and stacked element 2; stack the resulting element 4.
(4) Eliminate variable 7 after assembling elements from row 7; stack the resulting element 7.
(5) Eliminate variables 6, 8, 9 after assembling elements from rows 6, 8, 9 and stacked elements 4 and 7; stack the resulting element 6.
(6) Eliminate variable 5 after assembling remaining element in row 5 with stacked element 6.

Where variable-supervariable amalgamations are present, as in this example, the correct pivotal sequence is obtained from a depth-first tree search only if nodes representing variable-supervariable amalgamations that are brothers of nodes representing eliminations are to the right of them. We begin with just son-father pointers, which, of course, say nothing of relative positions of sons. We therefore construct father-son and adjacent brother pointers in two passes, one for the amalgamation nodes and one for the assembly nodes, thus ensuring that amalgamation nodes are to the right.

We use the extra pointers to perform the depth-first search and construct the pivotal sequence. For each assembly node encountered we also record the number of eliminations and the number of assemblies. It is also convenient to record the degree. This permits us to recognize when an elimination is really taking place entirely within a single generated element. Here the node has just one assembly son, and its degree is identical to the degree of the element stacked by the son. In such a case, the two steps can be combined without extra fill-in. For example, the steps corresponding to nodes 5 and 6 of the tree in Figure 4 can be combined.

Table III. Times and Space Forecasts for Two Modes of Running Space Evaluation Routine

Order	147	1176	199	292	130	406	503
Nonzeros:	1298	9864	536	1250	713	1561	3265
Time (IBM 3033 seconds)							
Exact	0.005	0.054	0.007	0.010	0.006	0.012	0.019
Estimate	0.002	0 025	0.006	0.007	0.004	0.008	0.009
Forecast of real space required (×1000)							
Exact	2.6	14	1.3	2 8	1.0	6 8	13
Estimate	3.2	20	1 5	3.0	1 6	7.0	14
Forecast of integer space required (×1000)							
Exact	1.5	11	1.3	2.2	1.0	3.4	5.2
Estimate	2.3	15	1.4	2 6	1.6	3 8	6.8

Here we have a full matrix of degree 4 at node 6. The recording of degrees also permits storage and operation counts to be calculated easily.

Greater efficiency during numerical processing is possible if the pivotal block sizes are large. It may even be preferable to tolerate more operations and fill-in for the sake of bigger blocks. We have therefore built into our tree-search code the option of combining adjacent steps when the first stacks a generated element that is immediately used by the second, provided the numbers of eliminations performed in both steps lie below a given threshold. The effect of using this threshold is discussed in Section 7 (see Table IX).

An important feature of our approach, which feature is also shared by SPARSPAK, is that we can predict the storage and arithmetic requirements of the subsequent numerical factorization of a definite matrix. This is particularly important since such a prediction may well determine whether it is computationally feasible or attractive to continue with the direct solution of the system. It is a relatively simple matter to postprocess the output from the tree search to obtain this information, although the original input must be preserved if exact values are required. Of course, the forecast may be optimistic for systems where numerical pivoting is required, but we show in Section 6 (Table VII) that this inaccuracy is slight. Since we are keen to allow the ANALYZE phase to require only a minimal amount of storage, we have designed our space-evaluation routine so that, when the user has allowed the input data to be overwritten, an upper bound for the space necessary for the factorization of definite systems will be calculated. In this case, the space evaluation will be quicker since a scan through the original nonzeros is not performed. We present the times and space estimates provided by these two modes of operation in Table III. Since the time spent in the space evaluation routine is usually less than 10 percent of the ANALYZE time, and since the estimates can be over 50 percent too high, we choose to calculate the exact quantities whenever possible.

5. GENERATING A TREE FROM A GIVEN PIVOTAL SEQUENCE

It has been established (e.g., see [13]) that if no permutations are performed, then the pattern of row i of the Cholesky factor U may be obtained by merging

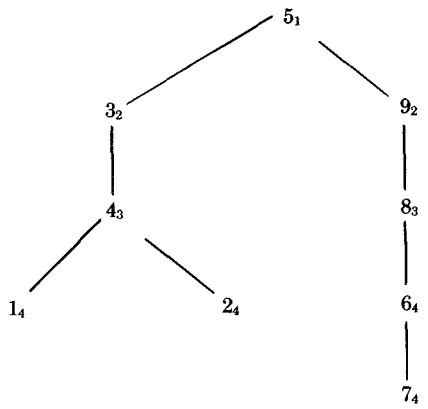

Fig. 5. Tree generated for matrix of Figure 3 with pivotal sequence 1, 2, 4, 3, 7, 6, 8, 9, 5 with degrees shown as subscripts

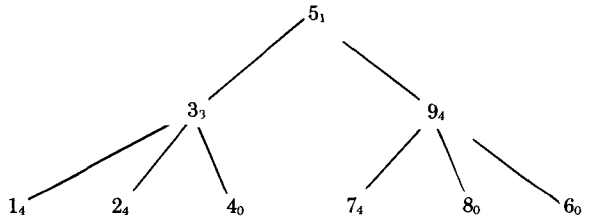

Fig. 6. Tree of Figure 5 after application of Section 4 tree search

the pattern of row i of A with the patterns of those earlier rows of U that have their first off-diagonal nonzero in column i. Expressed in the terms we have been using in this paper, this is because with variables reordered according to a given pivotal sequence, we know that each generated element will be required when the first of its variables is pivotal.

This makes the construction of the assembly tree very straightforward. We may now start with the pattern of the rows of the upper triangular part of A, rather than of both upper and lower triangular parts. The generated elements may be chained together according to their leading variable, in preparation for use when that variable is pivotal. Since each row of A is searched only once, and the list of each generated element is searched only once, it is not worthwhile to look for identical rows or elements that can be absorbed into other elements. This means that we obtain an assembly tree with n nodes, almost certainly much greater than is really necessary. However, the tree search techniques described in Section 4 yield node amalgamations where this is appropriate so there is no loss of efficiency. For instance, in Figure 5 we show the tree for the matrix of Figure 3 with the pivotal sequence generated in Section 3, and its condensed counterpart in Figure 6. Note that the pivotal sequence eventually used is not necessarily the same as that given, though the numerical operations performed are the same as they would have been (it is just that some independent operations are performed in a different order). Of course, the extra simplicity of the code for the case with a known pivotal sequence leads to better execution times, as illustrated in Table IV, though such is the success of variable amalgamation and element absorption

Table IV ANALYZE Times (IBM 3033 seconds) for Minimum-Degree Ordering and for Using the Same Sequence As a Known Pivotal Order

Order:	147	1176	292	130	406	1009	1561	1005	900
Nonzeros:	1298	9864	1250	713	1561	3937	6121	4813	4322
Pivotal sequence									
Unknown	0.039	0.30	0.077	0.073	0.115	0 28	0 43	0.40	0.30
Known	0.034	0.23	0.042	0.019	0 063	0.17	0.29	0.20	0 17

in the minimum-degree algorithm (see Section 3) that the gain is sometimes quite slight.

6. THE NUMERICAL FACTORIZATION

As we discussed in Section 4, it is convenient to postprocess the output from the ordering routines to produce assembly and elimination information for the numerical factorization. This information is independent of the actual numerical values of the matrix entries, and so need only be generated once for each sparsity pattern. In this section, we discuss how we combine this symbolic information with numerical pivoting in order to achieve a stable decomposition, even when the matrix is not definite.

The assembly and elimination information from the tree search (Section 4) can be conveniently passed to the numerical factorization routines in three arrays, two of which hold the number of eliminations and stack element assemblies at each assembly tree node and a third array holding the pivot order so that the variables being eliminated at each stage can be identified. For the example of Figure 6, these arrays might be

Number of eliminations:	1	1	2	1	3	1			
Stack assemblies:	0	0	2	0	1	2			
Pivot order:	1	2	4	3	7	6	8	9	5

One of the main implementation difficulties in this phase of the computation is the efficient assembly of generated elements and original rows prior to pivot selection and elimination within the full frontal matrix corresponding to the new generated element. The use of a stack to hold previously generated elements greatly facilitates the data organizaton, but we must still decide how to hold the frontal matrix and how to subdivide workspace between the frontal matrix, the stack, the matrix factors, and the original rows. It is useful, at this stage, to give the skeleton of our factorization algorithm.

for each assembly tree node **do**
begin
Assemble the appropriate number of stack elements from the top of the stack into the frontal matrix. Assemble those original rows whose corresponding variables are pivot candidates at this stage. Choose pivots from these and any earlier variables in the tentative sequence not yet used on numerical grounds. Perform elimination operations within the full frontal matrix. Store the pivot rows in storage allocated for the matrix factors and place the remainder of the element generated at this stage on the top of the stack.
end

Table V. A Comparison of Storage Requirements (in Thousands of Words) for the Two Different Frontal Matrix Organizations When No Garbage Collections Are Allowed

Order:	147	1176	292	406	1009	1561
Nonzeros·	1298	9864	1250	1561	3937	6121
Real storage required						
Fixed frontal matrix	4.1	23.5	4 0	8.4	24.0	42.3
Dynamic frontal matrix	3.8	20.4	4 0	8.3	23.8	42.1
Integer storage required						
Fixed frontal matrix	2.1	15.9	3.6	5.5	13.5	21 5
Dynamic frontal matrix	2 1	13.5	3 1	4.5	11.4	18.0
Integer storage for factors						
Fixed frontal matrix	0.8	5.9	2.3	3.9	9.4	15 1
Dynamic frontal matrix	0.8	3.6	1.8	2.9	7.3	11.7
Real storage for factors (same in both cases)	2 4	10.4	2.6	6.0	18.3	33.0

We consider the assembly of stack elements and original rows before discussing the numerical pivoting. Since the output from the tree search of Section 4 can be used to calculate the maximum size of the frontal matrix, we could allocate a fixed amount of storage for it and maintain mapping vectors identifying variables with their position in the front to facilitate assembly. We originally designed our factorization code using this scheme (which we hereafter call the fixed-front approach), but were unhappy with committing a fixed amount of storage to the front which may for most of the time be much smaller than its maximum size. With this fixed-front approach the number of permutations within the frontal matrix required to enable elimination to be performed efficiently was very high.

Because of these misgivings, we tried an approach using a dynamic allocation for the frontal matrix, generating it afresh at every stage. We only need storage for the current frontal matrix and can ensure that pivot candidates are in the first rows and columns, thereby reducing the number of permutations required. That is, the sort is done during the assembly rather than after it. However, in order to maintain an efficient assembly, we found that we would stack a generated element even if it were going to be assembled at the very next stage, which our experience has shown to be a common occurrence.

It is possible to construct small examples to favor either approach, and since it was not clear to us which would be superior, both methods were coded and runs were performed on our sets of test examples. We found the times for both factorization and solution to be very similar (within 5 percent), with the dynamic front approach having a slight edge. We are also concerned about the total storage required by the two approaches both during factorization and for the integer arrays passed to the solution routines. The dynamic-front approach will never require less space for the stack, but does not have the overhead of requiring more space for the frontal matrix than is actually needed. We compare these storage requirements in Table V, where it is clear that the dynamic-front approach is on the whole better, particularly with respect to real storage during factorization and integer information passed to the solution routines. We have therefore chosen

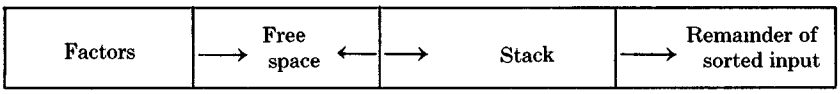

Fig. 7. Storage scheme for numerical factorization where the arrows indicate the directions in which the boundaries move.

to use a dynamically allocated front approach and now discuss the implementation of this further.

Since we follow our normal procedure (for example, used also in Harwell code MA28, [3]) of allowing the user to input the nonzeros in any order, we must first sort the matrix to avoid gross inefficiencies in the factorization. We have chosen to sort by rows in the pivot order provided by the tree search since this enables a simple reuse of the storage occupied by the sorted matrix (see Figure 7). Our principal choice lay between performing an in-place sort or a slightly simpler direct sort. The in-place sort requires only n extra real storage locations above that for the input matrix, whereas the direct sort requires twice the storage of the input matrix. The real storage needed in the direct sort can be greater than that required by the factorization itself. For example, in the case of order 1176 only 13,773 words of real storage are required for the factorization, whereas the direct sort needs 19,728 words. Since the in-place sort is less than 10 percent slower than the direct sort, we have chosen to implement this strategy in our code. As in the case of the ANALYZE entry, we give the user the option of allowing the integer information for the input matrix to be overwritten. This possibility of saving integer storage would not be possible if we had implemented a direct sort.

After sorting the user's input data, we perform the factorization using the information passed from the earlier routines. The storage organization is illustrated in Figure 7 where the arrows indicate the directions in which the boundaries move. We have chosen this scheme because we wish to use a single array for the numerical processing and we also want to keep garbage collections infrequent and simple. We know in advance the space required for the numerical factorization when no pivoting is necessary, and additionally we know how much less space is required if we allow the stack to overwrite the already processed input when the free space is exhausted. This is the only form of garbage collection which we have and it is particularly simple. Table VI shows the real and integer storage needed to avoid garbage collections, the minimum storage needed and the number of collections when minimum storage is used. The timings are not shown because the differences were less than the uncertainty of the IBM timer, thanks to the few collections needed and their simplicity.

Our other refinement on the scheme of Figure 7 is that when a stack element is being assembled we allow the new generated element to overlay the top element of the stack. This permits a slight reduction in the storage required during factorization.

We now discuss our strategy for numerical pivoting within the frontal matrices. These matrices are symmetric and we only store the upper triangular part. Since there are many cases when it is known a priori that the decomposition is numerically stable for any sequence of diagonal pivots (for example, when the

Table VI. The Effect of Running with Minimum Space

Order:	147	1176	292	406	1009	1561	900
Nonzeros:	1298	9864	1250	1561	3937	6121	4322
Number of garbage collections							
Real store	7	3	6	3	3	2	2
Integer store	13	11	13	11	7	11	11
Storage (words ×1000)							
Real storage							
No collections	3.8	20	4.0	8.3	24	42	24
Maximum collections	2.6	14	2.8	6.8	20	36	20
Integer storage							
No collections	2.2	14	3.1	4.5	11	18	11
Maximum collections	1.3	10	1.9	3.0	8	12	7

matrix is definite), we provide an option which does no numerical tests other than ensuring that the pivots are all nonzero and flagging if the pivots are not all of the same sign. If this option is not invoked, then we require that a diagonal pivot must satisfy the threshold criterion

$$|a_{kk}| > u \cdot \max_j |a_{kj}| \tag{6.1}$$

where u is a preset parameter in the range [0, 1) and the maximum is over all columns of the current frontal matrix (row k must, of course, be fully summed for a_{kk} to be available as a pivot candidate). This numerical test has been widely used in direct methods for sparse matrices and has been found satisfactory, the parameter u permitting an adequate balance between stability and sparsity preservation to be maintained.

Since we are restricting pivoting to the diagonal (to preserve symmetry and so reduce storage and arithmetic), it is entirely possible that a_{kk} is zero and so can never satisfy (6.1). To handle such cases and indeed to provide a stable decomposition when the system is indefinite, we use block pivots of order 2 as advocated originally (for full systems) by Bunch and Parlett [1]. We have already examined (Duff et al. [7]) the stability of this procedure when used with a threshold test of the form

$$\left\| \begin{bmatrix} a_{kk} & a_{k,k+1} \\ a_{k+1,k} & a_{k+1,k+1} \end{bmatrix} \right\|_\infty \max_{j \neq k,k+1} [\max(|a_{kj}|, |a_{k+1,j}|)] \leq u^{-1}, \tag{6.2}$$

with u now in the range $[0, \frac{1}{2})$, in the solution of sparse systems (the limit $\frac{1}{2}$ is needed to be sure that a pivot is available). This analysis also extends to our multifrontal scheme, although there are many different ways of implementing this pivoting strategy.

One possibility is to scan the entire fully-summed block searching for 1×1 pivots before looking for any block pivots of order 2. This could be done very efficiently, but we have rejected this approach because any failure to use a 1×1 pivot requires a second pass to find 2×2 pivots, which will repeat much of the work of the first. Therefore, when we scan a potential pivot row to test its

Table VII. Time (IBM 3033 seconds) and Storage Requirements with and without Pivoting for Stability

Order:	147	1176	292	406	1009	1561	274
Nonzeros:	1298	9864	1250	1561	3937	6121	943
Time for factorization							
No pivoting	0.076	0.46	0.085	0.19	0.59	1.22	0.053
With pivoting	0.079	0.49	0.092	0.20	0 64	1.24	0.055
Storage (words ×1000)							
For factorization							
Real							
No pivoting	2.6	13 8	2.8	6.8	20.0	36.1	1 47
With pivoting	2.7	14.0	2.9	7.0	20.5	36.6	1 51
Integer							
No pivoting	1.32	10.2	1.9	3 0	7.5	12.0	1.34
With pivoting	1.32	10.1	1.8	2 9	7.4	11.7	1.28
For solution							
Real							
No pivoting	2.35	10.4	2 6	6.0	18.3	33.0	1.35
With pivoting	2.38	10.7	2.7	6.1	18.7	33.5	1.39
Integer							
No pivoting	0.84	3.6	1 8	2.9	7.3	11.7	1.28
With pivoting	0.84	3.6	1 8	2.8	7.1	11.4	1 22

diagonal entry for stability, we also find a potential 2×2 pivot, defined by the largest off-diagonal in the fully summed part of the row. If the diagonal element fails the 1×1 stability test, we immediately test this 2×2 pivot, first against the largest entry in the current row and then, if it passes that test, against the largest entry in the other row of the block pivot. If the test (6.2) is not satisfied by this 2×2 block, we continue by searching the next fully-summed row. We also tried the strategy of testing the other diagonal entry of the 2×2 block for suitability as a 1×1 pivot, both before and after testing the 2×2 pivot. This gave a bias toward 1×1 pivots, but was not any more efficient, particularly because it increased greatly the number of interchanges within the frontal matrices. We also tried the strategy of only passing once through the frontal matrix in an attempt to find pivots, and found a slight increase in storage requirements with no compensating decrease in factorization time. We therefore, at each stage, search the fully-summed part of the front exhaustively for pivots.

The results in Table VII indicate that there is only slightly more storage required when numerical stability considerations delay pivoting and increase the front size. Thus the estimates from the analysis and tree search are a good guide to that actually required. The real storage is likely to increase by no more than about 2 percent, while, because of the possibility of several delayed eliminations occurring together, the integer storage might actually decrease. The only case we have seen where the estimate was further adrift was when the diagonal entries were very small (Ericsson and Ruhe, private communication).

7. NUMERICAL SOLUTION

In this section, we consider the efficient solution of sets of linear equations using the matrix factors produced by the numerical factorization of Section 6. Since we

Table VIII. Times (IBM 3033 seconds) for Solution Techniques Using Direct and Indirect Addressing

Order	406	1009	1561	147	1176	292	900
Nonzeros	1561	3937	6121	1298	9864	1250	4322
Solution times							
Direct	0.032	0.086	0.146	0.011	0.056	0.017	0.083
Indirect	0.025	0 073	0.129	0.009	0.048	0.013	0.072
Hybrid	0.024	0.067	0.118	0.009	0.043	0.012	0.066

have chosen to hold original row and column indices with the block pivot row, we can perform all operations on the input right-hand-side vector to produce the solution vector without requiring any auxiliary vectors. We call this single vector the right-hand-side vector, although it will be altered during the computation and finally holds the solution. We can either work with the right-hand-side vector using indirect addressing in the innermost loop or can load the appropriate components into a full vector of length the front size, perform inner-loop operations using direct addressing, and then unload back into the right-hand-side vector. The first option is similar to that used in many general sparse matrix codes, may double the inner-loop time over that required for the basic numerical operations, and will not vectorize. However, the second option will carry a high overhead if only a few eliminations are done at each stage. We have found empirically on the IBM 3033 and on the CRAY-1, by actual runs on versions of our code as well as times on simple loops, that for a given number of pivots, direct addressing is better when the order of the frontal matrix is greater than a threshold value. We hold these values in an array and include code for both forms of addressing, switching between the two depending on the number of pivots at each stage. In Table VIII we illustrate the advantage of such a hybrid approach by comparing it with the use of solely direct or indirect addressing on the IBM 3033.

On other machines the switchover point between indirect and direct addressing will be different, particularly on a machine which vectorizes the direct-addressing inner loop. The only change we need make is to alter the values in our switch array to match the characteristics of the machine.

In YSMP (Eisenstat et al. [9]), integer storage for the factors is greatly reduced by observing that column indices of successive rows of the factors have considerable overlap and if a second pointer array (of length the order of the system) is employed, much of this information can be reused. We illustrate this on the example in Figure 8. However, this is exactly the type of saving which we get when our block pivot is of order greater than 1. Additionally, during our discussion of the tree search, we remarked that it is easy to increase the order of the block pivots at the cost of a little more arithmetic and storage. In Table IX we show the storage requirements for the solution routine for various levels of node amalgamation. We see that the integer storage for the factors is reduced dramatically, although there is a more than compensatory increase in real storage. Notice, however, that the total storage for the factors remains about constant, although the storage required during factorization rises slowly. We do not show times in Table IX, since there is little variation over the range considered. They

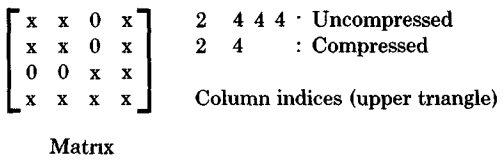

Fig 8. Illustration of YSMP compressed storage scheme.

Table IX Storage Required (Thousands of Words) at Various Levels of Node Amalgamation

Order:	147	1176	292	130	1561	1005	900
Nonzeros.	1298	9864	1250	713	6121	4813	4322
Real storage (for factors)							
Node amalgamation level							
1	2.4	10.4	2.6	0.7	33	21	18
3	2.4	11.3	2.9	0.8	35	23	19
5	2.5	11.8	3 2	0.9	37	25	20
Integer storage (for factors)							
Node amalgamation level							
1	0.8	3.6	1 8	0.7	12	9	7
3	0.8	2.8	1.3	0.4	9	6	5
5	0 6	2.5	1.0	0 3	8	5	5
YSMP	0.9	4.1	1 9	0.7	13	9	8
YSMP (with our order)	0.9	4.0	1.8	0.3	13	9	7
Total storage required by factorization (CDC mode)							
Node amalgamation level							
1	5.8	38	6.5	3.0	55	40	32
3	5.9	39	6.8	3.1	57	42	33
5	6.1	40	7 1	3.2	59	46	34

decrease slightly (about 2–5 percent) from the times in Table X until a node amalgamation level of about 10, and increase gradually thereafter. In view of these results we make node amalgamation available to the user of our code only by special request. We also show in Table IX the storage required by YSMP using the same pivotal sequence and the one it chooses. The storage shown for YSMP includes $2n$ for the pointers and row lengths (included also in the figures for our code). We have, however, not included storage for the permutation vector required by YSMP but not needed by our code because we hold original indices. A complete count of the storage required by us and YSMP is given in Table XI. We see that our block pivot scheme is more economic in storage than the compressed scheme, particularly if nodes are amalgamated. We were also surprised to find that giving our pivot sequence to YSMP saved it some storage, presumably because the order produced by tree searching leads to more repeated nonzero patterns in adjacent rows of the factorized matrix.

Table X. Times (IBM 3033 seconds) for Three Phases of Codes on Definite Systems

Order·	147	1176	292	130	406	1009	1561	1005	900
Nonzeros:	1298	9864	1250	713	1561	3937	6121	4813	4322
ANALYZE									
MA27	0 038	0.295	0.075	0.072	0.112	0.275	0.428	0.382	0.300
YSMP	0.053	0.449	0.082	0.286	0.107	0.280	0.439	0.512	0.279
SPARSPAK	0.083	0.647	0.140	0.386	0 193	0.492	0.812	0.827	0.568
SPARSPAK + input	0.151	1.088	0.203	0.441	0.260	0.599	0.956	0.955	0.683
FACTOR									
MA27	0.061	0.341	0.070	0 024	0.170	0.541	1.10	0.897	0.575
YSMP	0.056	0 378	0 046	0.015	0.138	0.548	1.04	1.059	0.578
SPARSPAK	0.056	0.353	0.051	0 012	0 144	0.632	1.27	0.781	0.576
SPARSPAK + input	0.096	0.720	0.087	0.032	0.191	0.748	1.45	0.945	0.701
SOLVE									
MA27	0.009	0.043	0.012	0 004	0.024	0.067	0.118	0.078	0.066
YSMP	0 007	0.037	0.009	0.003	0.020	0.063	0.109	0.078	0.063
SPARSPAK	0.009	0.049	0 012	0 004	0.024	0 074	0.128	0 078	0 069

8. GENERAL REMARKS

In the design of our code, called MA27 in the Harwell Subroutine Library, we had three main goals, namely,

(i) to develop a general-purpose sparse solver for symmetric positive-definite systems competitive with other routines available;
(ii) to solve indefinite symmetric systems stably with little overhead above the definite case;
(iii) to develop a general-purpose sparse code which vectorizes well.

In this section, we examine how well these goals have been satisfied.

It is difficult to compare the performance of different codes, partly because each divides the solution to the problem in a different way. We have chosen to measure separately the three phases of nonnumeric preprocessing (ANALYZE), numerical input and factorization (FACTOR), and solution using the factors (SOLVE) since these correspond to the operations required once for a given structure, once for given matrix values, and once for each right-hand side, respectively. We present the times in double precision on the IBM 3033 for these three phases in Table X and the storage required in Table XI.

The MA27 ANALYZE times in Table X reflect the total time for the nonnumeric phase, including matrix reordering. However, because input to SPARSPAK requires a subroutine call for each row of the matrix, we give both the time including input and for the ordering alone. Indeed, the form of input to each package is different. SPARSPAK permits input an entry at a time, a row at a time, or by finite elements (full submatrices of reals together with identifying indices). MA27 requires arrays giving the row and column indices of each entry. These entries can be in any order and duplicates are allowed. For entry to YSMP the matrix must be ordered by rows with column indices for each entry and we do not include times needed for this ordering. The effect of these different input

Table XI. Storage in Thousands of FORTRAN Words Required by Sparse Matrix Codes

Order:		147	1176	292	130	406	1009	1561	1005	900
Nonzeros.		1298	9864	1250	713	1561	3937	6121	4813	4322
ANALYZE										
IBM	MA27[a]	2.0	16	2.7	1.4	3.6	9	14	10	9
	YSMP	7.1	54	7.4	4.1	9.4	24	37	28	25
	SPARSPAK	2.9	22	3.2	1.7	4.5	11	18	13	11
CDC	MA27[a]	3.8	29	4.8	2.5	6 4	16	25	18	16
	YSMP	7.1	54	7.4	4.1	9.4	24	37	28	25
	SPARSPAK	5.6	43	5.9	3 2	7.8	20	31	22	20
FACTOR										
IBM	MA27[a]	7.0	40	7.6	3.0	16.1	46	82	60	47
	YSMP	10.6	63	12.9	5 2	22.6	65	107	77	64
	SPARSPAK	6.3	32	8.5	2.8	16.1	50	85	52	46
CDC	MA27[a]	5.8	38	6.5	3 0	11 5	32	55	40	32
	YSMP	6.8	41	8.7	3.6	14.6	41	67	47	40
	SPARSPAK	4.2	21	6 4	2 2	11.2	33	55	35	31
SOLVE										
IBM	MA27	5.5	26	6.9	2.2	14.6	43	76	50	42
	YSMP	6.5	32	8.8	2.9	17.5	52	88	62	50
	SPARSPAK	6.3	32	8.5	2 8	16.1	50	85	52	46
CDC	MA27	3.4	16	4.9	1.7	9 5	27	47	32	26
	YSMP	3.9	19	5.6	2.0	10.7	31	52	36	30
	SPARSPAK	4.2	21	6.4	2.2	11.2	33	55	35	31

[a] Assumes overwriting of input data.

forms is significant on the FACTOR entry so, in order to avoid penalizing the more flexible interfaces, we give in Table X FACTOR times for the numerical factorization only. To illustrate the overhead we give the times for SPARSPAK, including the input/sort.

In Table XI we have included all the storage required, including overhead and permutations and have given values in words for both the IBM, where reals occupy two words and some integers (for example, row and column indices) occupy only half a word, and the CDC, where reals and integers both occupy one word. Both versions are available in MA27 and SPARSPAK, but YSMP does not offer a half-word integer version, although comments are included to allow rapid conversion to double precision. Additionally, the data structure used by parts of the YSMP code involves pointers which require full integers (unless the number of nonzeros in the factors is severely limited), so full-word integer storage has been assumed throughout. We have, however, not included storage for the matrix reals in the YSMP ANALYZE because minor changes to some statements in a sort routine yield a version that does not need the real arrays. SPARSPAK does not recover storage between numerical factorization and solution and so the storage for these is the same.

In Table X we see that our ordering times are generally superior to those of YSMP and SPARSPAK, largely because of the refinements to our code which we discussed in Section 3. This is particularly noticeable in the laser problem of order 130. The overheads of loading into full matrices or vectors before performing

Table XII. A Comparison of MA27 with MA28 and an Older 2 × 2 Code of Munksgaard

Order	147	1176	292	199	130	406	900
Nonzeros:	1298	9864	1250	536	713	2716	4322
Time (IBM 3033 seconds)							
ANALYZE/FACTOR							
MA27	0.12	0.7	0.17	0.09	0.11	0.3	0.9
MA28	0.60	4.6	0.58	0.20	0.16	3.4	7.4
Munksgaard	0.81	6.8	0.61	0.31	0.16	1.9	13.6
FACTOR							
MA27	0 08	0 5	0 09	0.045	0.040	0.2	0.6
MA28	0 17	1.3	0.14	0.052	0.048	0.5	1.0
Munksgaard	0 46	3.4	0.32	—	0.068	1.0	5.5
SOLVE							
MA27	0.008	0 043	0.012	0.006	0.005	0.022	0.067
MA28	0.011	0.053	0.014	0.007	0.005	0.029	0.055
Munksgaard	0.010	0.055	0.013	—	0.005	0.026	0.079
Storage (thousands of words)							
FACTOR							
IBM MA27	7	40	8	4	3	16	47
IBM MA28	15	71	17	7	5	43	108
CDC MA27	6	38	6	3	3	12	32
CDC MA28	14	75	17	8	6	40	75
SOLVE							
IBM MA27	6	26	7	3	2	15	42
IBM MA28	13	60	16	7	4	39	73
CDC MA27	3	16	5	3	2	10	26
CDC MA28	11	49	13	6	4	31	59

the arithmetic operations are reflected in the slower FACTOR times on the smaller problems, although these times are more competitive on larger problems. The SOLVE times are comparable with SPARSPAK but are slightly worse than YSMP because our use of block pivots makes the next to innermost loops more complicated.

In Table XI we see that our code is extremely competitive with respect to the amount of storage required. It is comparable in its requirements for the FACTOR phase and is consistently the best for both IBM and CDC storage modes in the ANALYZE and SOLVE phases.

We should point out that the storage figures are a little flattering to MA27, which has a particular provision for overwriting the input matrix. If this must be preserved, then its storage must be added to the MA27 figures and to the YSMP figures for ANALYZE.

The only other code which we are aware of that solves sparse symmetric indefinite codes efficiently and stably is that of Munksgaard. His work is based on Duff et al. [7], has a similar storage scheme to Harwell Subroutine MA17E, and performs a numerical factorization at the same time as the analysis. Of course, a code for unsymmetric systems, for example, MA28 [3], could also be used to solve indefinite systems, although the symmetry of the original problem would be destroyed. In Table XII we compare our new code with that of

Table XIII. Times (CRAY-1 seconds) for MA27 and YSMP on the CRAY-1

Order.	147	1176	292	406	1561	1005	900
Nonzeros:	1298	9864	1250	1561	6121	4813	4322
Factorization times							
MA27	0.019	0.096	0 024	0.050	0.255	0.193	0.144
YSMP	0.026	0 174	0.023	0 060	0.420	0.410	0.237
Solution times							
MA27	0.0029	0.016	0.005	0.008	0.035	0.024	0.021
YSMP	0.0033	0.016	0.004	0.008	0.044	0.031	0.025

Munksgaard and with MA28, where the test matrices are symmetric indefinite. Our fast ANALYZE times have no counterpart in more general codes, and the better FACTOR times for MA27 reflect the fact that we need only store and operate on half of the matrix because our block diagonal pivoting preserves symmetry. The SOLVE times are comparable with MA28 although, because we preserve symmetry, the storage is much less. Munksgaard's times are high because in both ANALYZE and FACTOR he dynamically adjusts his data structure to allow for fill-in.

Finally, we demonstrate, in Table XIII, the vectorization of our factor and solve codes by comparing them with YSMP on the CRAY-1 computer. YSMP uses indirect addressing throughout and hence does not vectorize, which is apparent if we compare the results with those in Table X. In Table XIII we see that the problem size at which MA27 becomes competitive is very much reduced because of vectorization, although because of the short length of the vectors in the innermost loop and the other overheads the effect is a little disappointing.

Most routines which vectorize better on the CRAY-1 than MA27 do so at the expense of many more multiplications. Codes based on variable band or frontal techniques fall into this category. We ran the Harwell frontal code MA32 [4, 5] on the examples in our tables and found that it also was substantially slower than MA27 for both FACTOR and SOLVE.

Thus, we see from the results of this section that we have achieved the objectives stated earlier, our only cost being a slight loss of efficiency on small definite systems.

ACKNOWLEDGMENT

We would like to thank the referee for his careful reading of the paper and constructive comments.

REFERENCES

1. Bunch, J.R., and Parlett, B.N. Direct methods for solving symmetric indefinite systems of linear equations *SIAM J. Numer. Anal. 8* (1971), 639–655.
2. Bunch, J.R., and Rose, D.J. (Eds) *Sparse Matrix Computations.* Academic Press, New York, 1976.
3. Duff, I.S. MA28—A set of Fortran subroutines for sparse unsymmetric linear equations Harwell Rep. AERE R. 8730, HMSO, London, 1977.
4. Duff, I.S Design features of a code for solving sparse unsymmetric linear systems out-of-core. *SIAM J. Sci Stat Comput. 4* (1983).

5 DUFF, I.S. MA32—A package for solving sparse unsymmetric systems using the frontal method. Harwell Rep AERE R. 10079, HMSO, London, 1981.
6. DUFF, I.S., AND REID, J.K. Some design features of a sparse matrix code. *ACM Trans. Math. Softw. 5*, 1 (Mar. 1979), 18–35.
7. DUFF, I.S., REID, J.K., MUNKSGAARD, N., AND NIELSEN, H.B. Direct solution of sets of linear equations whose matrix is sparse, symmetric and indefinite. *J. Inst. Maths. Appl. 23* (1979), 235–250
8. DUFF, I.S., AND STEWART, G.W. (Eds.) *Sparse Matrix Proceedings 1978.* SIAM Press, Philadelphia, Pa., 1979.
9. EISENSTAT, S.C., GURSKY, M C., SCHULTZ, M H., AND SHERMAN, A H. The Yale sparse matrix package, I The symmetric codes, and, II The non-symmetric codes. Reps. 112 and 114, Dept. Computer Science, Yale Univ., New Haven, Conn , 1977.
10. EISENSTAT, S.C., GURSKY, M C., SCHULTZ, M.H , AND SHERMAN, A.H. Yale sparse matrix package I The symmetric codes. *Int. J. Numer. Meth. Eng 18* (1982), 1145–1151.
11 EISENSTAT, S.C., SCHULTZ, M.H., AND SHERMAN, A H. Applications of an element model for Gaussian elimination. In *Sparse Matrix Computations*, J. R. Bunch and D. J. Rose (Eds.), Academic Press, New York, 1976, pp. 85–96.
12. EISENSTAT, S.C., SCHULTZ, M.H., AND SHERMAN, A.H. Software for sparse Gaussian elimination with limited core storage. In *Sparse Matrix Proceedings 1978*, I.S. Duff and G.W. Stewart (Eds.), SIAM Press, Philadelphia, Pa., 1979.
13. EISENSTAT, S C., SCHULTZ, M.H., AND SHERMAN, A.H. Algorithms and data structures for sparse symmetric Gaussian elimination. *SIAM J. Sci. Stat Comput. 2* (1981), 225–237.
14. EVERSTINE, G.C. A comparison of three resequencing algorithms for the reduction of matrix profile and wavefront. *Int. J Numer. Meth. Eng. 14* (1979), 837–853.
15. GEORGE, J.A., AND LIU, J.W.H. An automatic nested dissection algorithm for irregular finite element problems. *SIAM J. Numer. Anal. 15* (1978), 1053–1069.
16. GEORGE, J.A., AND LIU, J W.H. A minimal storage implementation of the minimum degree algorithm. *SIAM J Numer. Anal. 17* (1980), 282–299
17. GEORGE, A., AND LIU, J.W. *Computer Solution of Large Sparse Positive Definite Systems.* Prentice-Hall, Englewood Cliffs, N. J., 1981.
18. GEORGE, A., LIU, J.W., AND NG, E. User guide for SPARSPAK: Waterloo sparse linear equations package. Res. Rep. CS-78-30 (Rev. Jan. 1980), Dept. Computer Science, Univ. of Waterloo, Waterloo, Ont., Canada, 1980.
19. HOOD, P. Frontal solution program for unsymmetric matrices. *Int. J. Numer Meth. Eng. 10* (1976), 379–400.
20. IRONS, B.M. A frontal solution program for finite element analysis. *Int. J. Numer. Meth Eng. 2* (1970), 5–32
21. PETERS, F.J. Sparse matrices and substructures Mathematical Centre Tracts 119, Mathematisch Centrum, Amsterdam, The Netherlands, 1980.
22 REID, J.K. Two Fortran subroutines for direct solution of linear equations whose matrix is sparse, symmetric and positive definite. Harwell Rep AERE R. 7119, HMSO, London, 1972.
23. SHERMAN, A.H On the efficient solution of sparse systems of linear and nonlinear equations. Res Rep. 46, Dept. Computer Science, Yale Univ., New Haven, Conn , 1975.
24. SPEELPENNING, B. The generalized element method. Private communication, 1973; also, issued as Rep. UIUCDCS-R-78-946, Dept. Computer Science, Univ. of Illinois at Urbana-Champaign, 1978.

Received July 1982; revised January 1983; accepted March 1983