AERE - R 10533

AERE - R 10533

United Kingdom Atomic Energy Authority

HARWELL

MA27 – A Set of Fortran Subroutines for Solving Sparse Symmetric Sets of Linear Equations

I.S. Duff and J.K. Reid
Computer Science and Systems Division
AERE Harwell, Oxfordshire

July 1982

Price £3.00 net from H.M. Stationery Office

C13

AERE R-10533

MA27 - A set of Fortran subroutines for solving sparse symmetric sets of linear equations

I. S. Duff and J. K. Reid

ABSTRACT

In this report we describe a set of Fortran subroutines for solving sparse symmetric sets of linear equations whose matrix may be indefinite. The matrix pattern is analyzed using a predetermined amount of storage and the storage requirements for the definite case are predicted during this analysis. The indefinite case is handled by additional interchanges and the use of some 2x2 pivots, neither of which greatly increase storage demands or execution times.

Computer Science and Systems Division,
AERE Harwell.

July 1982

HL82/2225 (C13)

CONTENTS

1. Introduction 2

2. Overall strategy 6
   2.1 MA27D/E/F : common blocks 8

3. Analysis of sparsity pattern 10
   3.1 MA27A : driver subroutine for analysis 10
   3.2 MA27G : sort prior to minimum degree analysis 13
   3.3 MA27H : minimum degree analysis 15
   3.4 MA27I : compress data structure during analysis 18
   3.5 MA27J : sort prior to analysis given a pivot sequence 19
   3.6 MA27K : analysis given pivot sequence 20
   3.7 MA27L : depth-first tree search 21
   3.8 MA27M : computer storage requirements and multiplication counts for factorization 22

4. The numerical factorization 24
   4.1 MA27B : driver subroutine for numerical factorization 25
   4.2 MA27N : sort prior to factorization 25
   4.3 MA27O : actual factorization 27
   4.4 MA27P : compress data structure during factorization 33

5. Solution of a set of equations 34
   5.1 MA27C : driver subroutine for solution of equations 35
   5.2 MA27Q : forward substitution 35
   5.3 MA27R : back substitution 37

References 38

Appendix : Specification document 39

ISBN 0-7058-0985-4

1. Introduction

The purpose of this report is to describe a collection of subroutines for solving sparse symmetric sets of n linear equations

$$A \underline{x} = \underline{b} \tag{1.1}$$

by Gaussian elimination. The package is called MA27 and contains 15 subroutines and three common blocks. Each subroutine is associated with just one of the three tasks

(a) analyze the sparsity pattern of the matrix and prepare for handling numerical values,

(b) factorize numerically a matrix whose pattern has already been analyzed, and

(c) solve a set of equations whose matrix A has been factorized.

For each of these tasks we devote a section of this report and for each subroutine we devote a subsection. The common blocks are used during all the tasks, so these are described as a subsection of Section 2, in which our overall strategy is explained. This structure should aid the reader in relating this report to the actual code and means that the contents page provides an index to the purposes of the routines. To give an idea of the varying complexity of the different routines, we show, in Table 1.1, statistics on the length of each subroutine when the double length versions were compiled on the IBM 3033 using the H-extended enhanced compiler at an optimization level of 3. Note that this compiler counts a logical IF as two statements.

We expect only three of the routines to be called directly by users. These are MA27A (to analyze a pattern), MA27B (to factorize a matrix) and MA27C (to solve a set of equations). These driver subroutines have been kept relatively short. Their main function is to subdivide workspace and provide most of the diagnostic printing. The data manipulation and numerical computation is performed by the other (auxiliary) subroutines

which are called by the drivers. Details of how to call the driver subroutines are given in the library specification reproduced in the appendix.

To give the reader a feel for the relative expense of the various subroutines and to illustrate some effects which are problem dependent, we show a breakdown of execution times for subroutines in the MA27 package in Table 1.2.

A companion paper (Duff and Reid, 1982) introduces the algorithms and compares them with alternatives, but does not describe the code other than in broad outline. Our aim is for this paper to be self-contained, though we advise the reader to study the companion paper first. Here we concentrate on what the code does rather than why we chose to write it this way. We do not describe the code in fine detail. For such detail the code itself and its comments must be studied.

During most of this report we refer to just one version of the code, which uses single precision reals and has some of its integer arrays of IBM type INTEGER*2. This may be converted into a Fortran 66 version that satisfies the PFORT verifier (Ryder, 1974) by deleting "*2" from each INTEGER*2 statement. Other versions are for double-length arithmetic, complex and double complex arithmetic and for greater efficiency on the CRAY-1. In all cases the changes are simple and not very numerous. The actual runs that yielded the data in Tables 1.1 and 1.2 used the double precision version.

Subroutine	Number of cards	Number of Fortran statements (as counted by IBM H-compiler)	Length of double precision compiled code (bytes IBM H-extended enhanced compiler, opt=3)
MA27A	130	90	3150
MA27B	150	140	3440
MA27C	90	70	1740
MA27D/E/F	70	10	0
MA27G	180	140	2060
MA27H	380	240	3580
MA27I	60	40	640
MA27J	170	130	2000
MA27K	130	70	1300
MA27L	130	90	1380
MA27M	180	100	1940
MA27N	210	140	2150
MA27O	650	500	6640
MA27P	40	30	590
MA27Q	170	120	1700
MA27R	180	130	2070
Total for MA27 package	2920	2000	34400

Table 1.1 Code lengths for MA27 package (rounded to a multiple of 10)

Order	147	1176	130	1009
Non zeros	1298	9864	713	3937
Subroutine				
MA27A	.082	.576	.099	.512
MA27G	.017	.127	.009	.050
MA27H	.015	.116	.057	.208
MA27I	.008	.033	.007	.054
MA27J	.013	.101	.007	.039
MA27K	.020	.108	.011	.133
MA27L	.004	.032	.004	.026
MA27M	.011	.102	.010	.055
MA27B	.085	.576	.039	.659
MA27N	.016	.123	.008	.048
MA27O	.069	.452	.030	.611
MA27P	-	-	-	-
MA27C	.009	.047	.005	.070
MA27Q	.004	.022	.002	.033
MA27R	.005	.024	.002	.036

Table 1.2 Breakdown of time (in seconds on an IBM 3033) spent in MA27 subroutines (double length versions)

2. Overall strategy

We regard the matrix A as having the form

$$A = \sum_{\ell} B^{(\ell)} \qquad (2.1)$$

where each matrix $B^{(\ell)}$ is zero except in a small number of rows and columns. We refer to each $B^{(\ell)}$ as an "element matrix" by analogy with finite element problems where each $B^{(\ell)}$ is associated with a finite element. We use the frontal method, in which advantage is taken of the fact that elimination steps

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

do not have to wait for all the assembly steps

$$a_{ij} := a_{ij} + b_{ij}^{(\ell)} \qquad (2.3)$$

from (2.1) to be complete. It is only necessary that the pivot row and column is fully summed.

At a typical step we use a full matrix of workspace (the frontal matrix) to accumulate all those $B^{(\ell)}$ that have non-zeros in the next pivot row or column. An associated index list indicates the correspondence between rows and columns in this full matrix and rows and columns in the overall sparse matrix. The actual elimination operations (2.2) may then be performed in this full matrix. The pivot row and column are then stored elsewhere for use in forward and back substitution and the remaining part of the frontal matrix is treated as another matrix $B^{(\ell)}$. Therefore the rest of the matrix (the reduced matrix) also has the form (2.1). We refer to matrices $B^{(\ell)}$ created by elimination as "generated element matrices".

The whole process may be represented by a tree. Each terminal node is associated with one of the original element matrices $B^{(\ell)}$ and each non-terminal node is associated with a generated element matrix $B^{(\ell)}$. The sons of a node correspond to the matrices accumulated prior to the elimination that created its associated matrix $B^{(\ell)}$.

The operations may be performed in any order that leads to all sons of a node being treated before the node itself, since the only differences are the very minor roundoff effects from summing quantities in different orders. We use an order determined by a depth-first search of the tree because then the generated element matrices may be stored on a stack since they will always be wanted on a last-in first-out basis. Note incidently that this implies that when the user provides a pivot order, the order actually used is not necessarily exactly the order given though it does lead to results which are identical except for minor roundoff effects.

So far we have assumed that just one elimination is performed for each non-terminal tree node. This can lead to nodes having only one son and an implementation in which generated elements are stored on the stack and immediately retrieved for further use without modification. We obtain worthwhile gains by recognising this situation and permitting more than one elimination to be associated with non-terminal tree nodes.

To obtain an initial set of matrices $B^{(\ell)}$ for a given matrix A, we simply associate a $B^{(\ell)}$ with each diagonal entry a_{ii} in A and a $B^{(\ell)}$ with each off-diagonal pair a_{ij}, a_{ji}.

The analysis subroutines (see Section 3) construct the tree and perform a depth-first search of it, thereby finding a pivot sequence. They pass the pivot sequence and the tree to the factorization routines (see Section 4). The tree is passed as an operator list with one entry (consisting of a pair of integers) for each non-terminal node. The entries are in depth-first search order and indicate the number of generated matrices to be retrieved from the stack and the number of eliminations to be performed. Perhaps surprisingly, this is sufficient information. If the non-zeros of the upper triangular part of the permuted matrix are stored by rows, then those in the rows of the variables to be eliminated may be regarded as original element matrices $B^{(\ell)}$ and assembled with those indicated as needed from the stack.

During elimination the pivots are checked for size and if necessary interchanges are performed and possibly 2x2 pivots are used. Sometimes the generated elements are bigger than anticipated during analysis because

some eliminations are delayed. These changes can lead to more operations being performed than were anticipated during analysis and more storage may be needed, but our experience is that such increases are never great (indeed integer storage requirements are often less). No such increases happen for positive (or negative) definite matrices.

2.1 MA27D/E/F : Common blocks

The MA27 package contains the following common blocks

i) MA27D contains four parameters whose default settings in BLOCK DATA are usually adequate. However the user may wish to reset the relative pivot threshold U used by MA27O (see Section 4.3) or the parameters LP, MP, LDIAG that control printed output. For further details, see Section 2.2 of the specification in the appendix.

ii) MA27E contains variables which are set by the MA27 subroutines to hold information likely to be of interest to the user, including operation count, space requirements, number of data structure compresses, number of 2x2 pivots and extra error information. For details, see Section 2.2 of the specification in the appendix.

iii) MA27F contains variables whose values may depend on the machine but are unlikely to depend on the problem being solved. They are therefore not likely to be of interest to the user and are not detailed in the specification in the appendix. They are given default values by BLOCK DATA and are as follows:-

(a) IOVFLO (default value 32639) is the largest integer such that all integers i in the range $-\text{IOVFLO} \leqslant i \leqslant \text{IOVFLO}$ can be handled by the shortest integer type in use.

(b) NEMIN (default value 1) controls the amalgamation of tree nodes at which few eliminations are performed (see Section 3.7 for further details).

(c) IFRLVL is an array of length 20 that controls whether direct or indirect addressing is performed during forward and back substitution (see Sections 5.2 and 5.3 for details, including default settings).

3. Analysis of sparsity pattern

In this section we describe those subroutines which analyze the sparsity pattern of a symmetric matrix. Note that numerical values are not passed and that a stable factorization of any symmetric matrix having this pattern can subsequently be obtained. This approach has the advantage that the storage and operation counts for numerical factorization can be estimated. The analysis itself is relatively fast and its storage requirements are known in advance.

3.1 MA27A : Driver subroutine for analysis

MA27A is the driver subroutine called directly by the user. For his convenience we ask for the positions of the non-zeros to be provided as pairs IRN(K), ICN(K) of entries in two INTEGER*2 arrays and we do not demand that they be in any particular order. This means that MA27A must first perform a sort. Unfortunately the requirements of the minimum-degree analysis subroutine (MA27H), which is called if the user does not provide a pivot order, differ from those for the analysis subroutine (MA27K) called if a pivot order is given, so separate sort routines (MA27G and MA27J) have been written for the two cases. Therefore MA27A calls MA27G and MA27H if the sequence is not given and MA27J and MA27K if it is given. In both cases it then calls MA27L to perform a depth-first search of the tree and finally MA27M is called to calculate the storage requirements and number of multiplications needed for actual numerical factorization. The two analysis routines may call MA27I to compress the data structure. We summarize the call trees in Figures 3.1 and 3.2.

It should be noted that the storage requirements of MA27A depend only on the order, n, and number of non-zeros, nz. There is no possibility of failure even if there is severe fill-in during numerical factorization. On the other hand the storage needed by MA27B is not predictable from n and nz. For this reason we have included the subroutine MA27M which calculates the minimal storage requirements both with and without compresses of the data

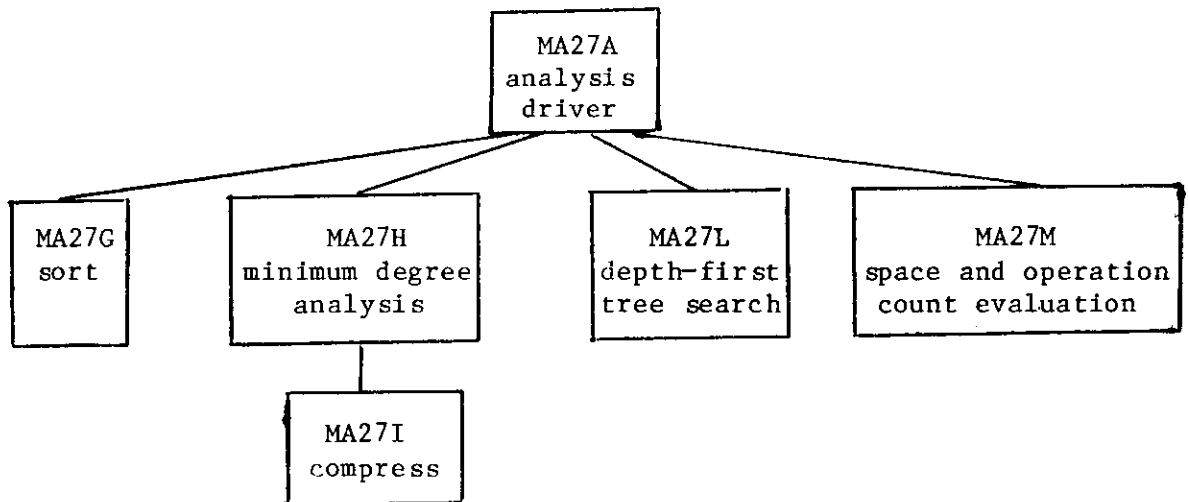

Figure 3.1 Call tree from MA27A, pivot sequence not given.

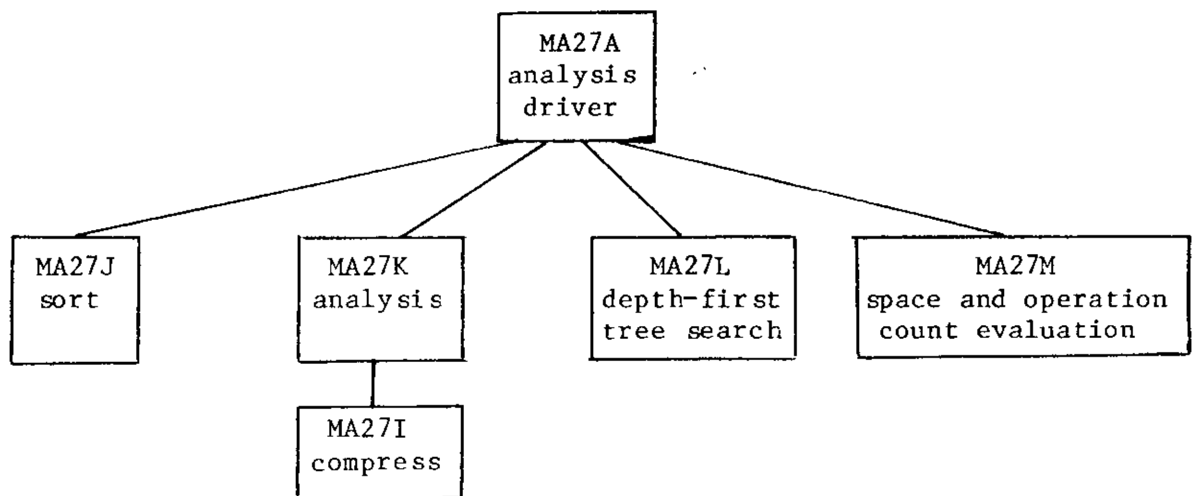

Figure 3.2 Call tree from MA27A, given pivot sequence

structure (calls of MA27P). These values apply to the positive-definite case but our experience of non-definite cases is that the requirements at worst only increase slightly and may even decrease.

The original sparsity pattern is needed by MA27M if it is to calculate the space requirements exactly. In any case we expect the user to want it preserved since it is needed again for the call to MA27B. However, if he wishes to save storage and is willing to regenerate the pattern, then he may equivalence the arrays IRN and ICN which hold the pattern with the work-array IW. In this case MA27M calculates estimates of storage requirements which are high (safe). An automatic test for equivalence is made by setting IW(1) = IRN(1)-1 before calling MA27M and testing IW(1) and IRN(1) for equality at the start of MA27M.

The remaining roles played by MA27A are to subdivide the workspace arrays dynamically, provide two levels of diagnostic printing on entry and exit and to print error messages. We have tried to avoid including any printing in the routines it calls. The only printing is of warning messages about out-of-range indices when they are found by either of the sort routines. We show details of how the arrays are subdivided in Figures 3.3

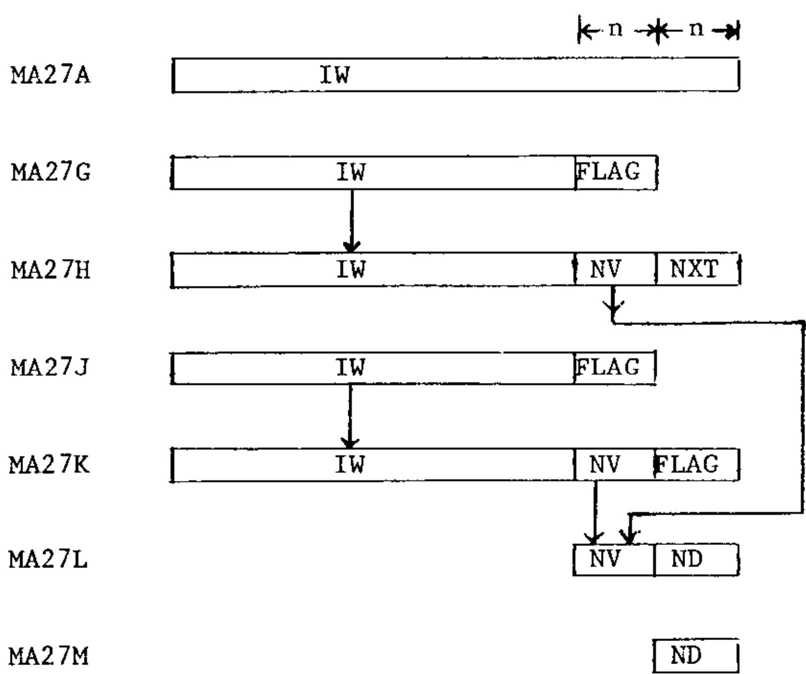

Figure 3.3 Sharing the workspace IW

and 3.4. Where a subarray is used to pass information from one subroutine to another, we show this by an arrow. In most such cases we use the same names, but sometimes different names were needed for clarity within the auxiliary routines. Further details of the purposes of the subarrays are given in the subsections for the individual subroutines. The reason for having three arrays is because only IKEEP is left containing information needed for matrix factorization and because IW can be an INTEGER*2 array on IBM machines provided n $\leqslant$ 32639 (INTEGER*2 limit under WATFIV) whereas IW1 must be an INTEGER array unless the number of non-zeros is to be severely limited.

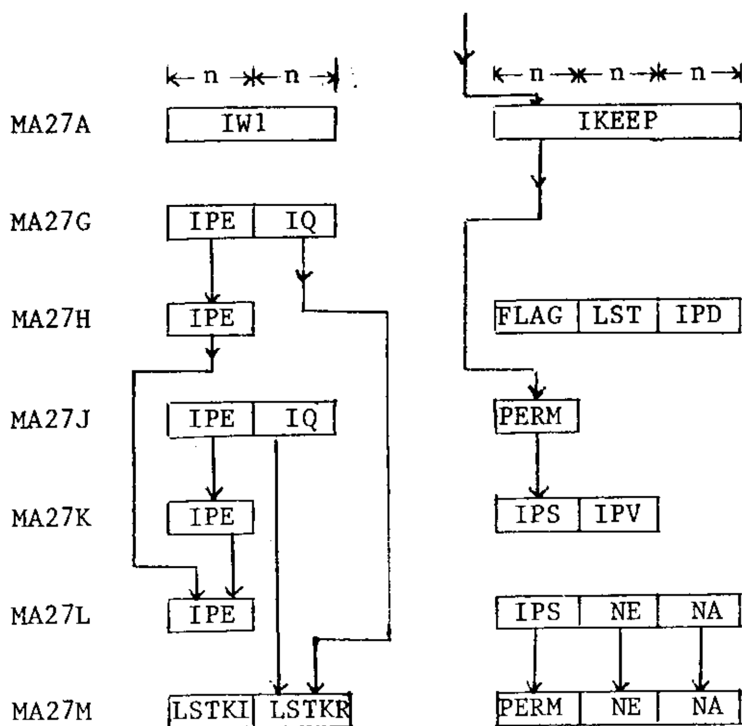

Figure 3.4 Sharing the arrays IW1 and IKEEP

3.2 MA27G : sort prior to minimum degree analysis

Our minimum degree code (MA27H, Section 3.3) assumes that diagonal entries are always suitable as pivots (the pivotal sequence is altered slightly during factorization, see Section 2 (or Section 4 for more detail), if a diagonal element is later found to be unsuitable). We therefore assume for the purpose of analysis that the diagonal elements are all present, which means that there is no need to store them explicitly. Following (2.1)

we can thus define the pattern of A

$$A = \sum_{\ell} B^{(\ell)} \tag{3.1}$$

where each $B^{(\ell)}$ corresponds to an off-diagonal pair a_{ij}, a_{ji}. Rapid access is needed to i given j and vice-versa, so we sort all the off-diagonal entries of A by rows and include both a_{ij} and a_{ji} in each case. Ordering within each row is unnecessary.

As explained at the beginning of Section 3.1, we require the user to specify each non-zero as a pair IRN(K), ICN(K) of entries in two INTEGER*2 arrays. Each pair i = IRN(K), j = ICN(K) is taken to specify both a_{ij} and a_{ji}. We begin by scanning the pairs to count the number of off-diagonal non-zeros in each row. If any index lies outside the range [1,n] then we ignore the pair after printing a warning message (for the first ten cases) and setting appropriate error flags. We also ignore diagonal entries. The indices i and j for each valid pair are transferred to a work array IW.

Once the numbers of non-zeros in the rows are known we may accumulate them to set up pointers to row starts in the sorted version. This permits an in-place sort to be performed on the non-zeros, now held in array IW. At an intermediate stage in the sort, part of each row will be in its final position. These sorted parts are held at the front of the space that eventually holds the whole row and pointers to the starts of the remaining parts of the rows are kept. Actual moves are made in groups that we call chains. The first non-zero in a chain is moved to just behind the sorted part of its row; if this position holds an unsorted non-zero then it is treated similarly; the sequence continues until a position that does not hold a non-zero is reached. The pointers are updated as each chain is traversed and the non-zeros sorted in the chain are flagged (by the signs of the row indices). The heads of the chains are found by a simple search for unflagged non-zeros.

This would complete the sort were it not that duplicate elements must be ignored for the row counts needed by the minimum degree algorithm. It seemed sensible to eliminate duplicates during the sort so that the row

counts were as required and so that storage is saved. To identify duplicates we use an array of n INTEGER*2 flags, initially set to zero. We then scan the rows in turn. For a non-zero a_{ij} we set FLAG(j) = i unless it already has this value which indicates a duplicate that can be removed. Note that by using a different flag value for each row (Gustavson, 1978) we avoid the need to reinitialize the flags when treating a fresh row. If any duplicates are found, then the row counts are revised and the non-zeros are moved to fill in the gaps.

The reason for transferring the indices to the work array IW and performing an in-place sort there is that this preserves the user's input arrays, which are certainly needed for actual numerical factorization and are also needed if exact calculations of space requirements are to be made by MA27M (see Section 3.8) later. If the user is willing to regenerate his data and wishes to save space then he may associate IRN and ICN with parts of IW (for details see the specification in the appendix). Care has been taken to ensure that the code works correctly in this case.

3.3 MA27H : minimum degree analysis

If the user does not provide a pivot order, then one is constructed using the algorithm of "minimum degree". The degree of a variable is the number of non-zeros in the corresponding row of the matrix, and each pivot is chosen in turn to minimize its degree in the current reduced problem. Note that the degrees of all the variables active (in the front) during a pivotal step may change because of the eliminations performed and the fill-ins that they may cause.

Our minimum degree analysis subroutine, MA27H, uses an INTEGER array IPE of length n to hold pointers to lists held in an INTEGER*2 work-array IW. Each list is headed by its length. Initially the lists contain the column indices of the non-zeros in the rows and each index may be regarded as a pointer to an element matrix $B^{(\ell)}$. They are later revised to contain a mixture of pointers to generated and original element matrices $B^{(\ell)}$. When a variable is eliminated we replace the list of elements with which it is associated by a list of variables in the element matrix generated by its elimination.

For speed it is important to avoid an expensive search for the pivot at each stage of the elimination. We therefore hold doubly-linked chains of variables having the same degree. Any variable that is active in a pivotal step and whose degree may therefore change can then be removed from its chain without any searching and once its new degree is known 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. The INTEGER*2 work-arrays IPD, NEXT, LST, each of length n, are used for this purpose.

When we eliminate a variable we replace the list of elements with which it is associated by a list of variables in the element generated. It is therefore convenient to use as a name for the generated element the index (i, say) of the variable eliminated. This has the further advantage that if another row (j, say) originally contained a non-zero a_{ij}, then the entry i in its list now (correctly) acts as a pointer to the generated element.

The main loop begins by choosing the next variable for elimination, an easy task given the chains of variables of equal degree. We then remove it from its chain. We next search its index list in IW. Entries are either dummy (for reasons to be explained) or point to element matrices $B^{(\ell)}$. We construct a new index list by merging the index lists of all these element matrices. The old lists can be discarded so no more storage will be needed than previously, though some data compression may be required (MA27I, Section 3.4) since we always insert the new list in the free space at the end of IW. For each generated element ℓ merged in, we record a son-father tree pointer. Son-father pointers for original element matrices $B^{(\ell)}$ are not needed explicitly because we can assume that when variable i is eliminated all original matrices $B^{(\ell)}$ involving elements a_{ij} of row i are to be included.

We use an INTEGER*2 array FLAG of length n to avoid duplicates in the merged list. As variable i is inserted in the list FLAG(i) is set to zero so that if i is encountered again the zero value of FLAG(i) will indicate that i has already been included.

The degrees of all the variables in the generated element (but no others) may change, so we remove them from their chains.

Once the new list of variables is constructed we run through it revising the associated lists of elements $B^{(\ell)}$ and calculating the new degrees. Each variable in the list must be associated with at least one $B^{(\ell)}$ absorbed into the new element. We therefore revise the list of each variable encountered to remove all those elements absorbed but to add the new element matrix. The new degree of a variable i is the number of variables in the new element plus the number of other variables in other elements associated with variable i. We use a different flag value for each degree calculation but take advantage of the flags already set to zero for the variables in the new element, starting each calculation from the known number of variables in the new element.

This degree calculation is potentially very expensive so we have included some other devices to aid it. If two rows have identical sparsity pattern (either originally or because fill-ins make them so) then they will have the same degree until one is eliminated. One will be eliminated only if it has minimum degree and following its elimination the other will have degree one less, which will certainly be minimum. It follows that we can treat such a pair together and eliminate them together. In general any number of rows may have an identical pattern and may be treated together. We regard the corresponding variables as grouped into a "supervariable". We name the supervariable after one of its variables (its principal variable, i say) hold a list of element matrices $B^{(\ell)}$ only for it and calculate only its degree. We indicate the association of another variable, j say, with i by setting the pointer IPE(j) to i. We also need to store the number of variables in each supervariable because this will now be needed in each degree calculation. We use an INTEGER*2 array NV for this purpose. The degree calculation is used to help recognize identical rows. All variables active at a pivot step and having equal degrees are in any case linked together at the front of a same-degree chain of variables. When each new variable is added to the front of a chain the array FLAG will have been set during the preceding degree calculation to flag the variables in the matrix row corresponding to the new variable. Other variables in the chain can be

tested quickly to see if they have any different variables in their matrix rows.

Another device we use to speed the degree calculation is to watch out for generated element matrices whose variables are all in the list of the newly created element. Such an element matrix can be merged into the new element before the elimination without causing any extra fill. By doing this we simplify the later degree calculations for all its variables. It is easy to build a suitable test into the degree calculation. Remember that the flags of all the variables in the new element are zero and that we begin the degree calculation as if this element had been searched. If a search of another element matrix increases the degree we know that it must have at least one other variable. If the degree does not increase, then we search its list of variables to see if any have non-zero flags.

The absorption of variables into supervariables and elements into other elements may leave some dummy entries in lists. We use the value -1 in FLAG to flag this but remove dummies only when they are later encountered.

3.4 MA27I : compress data structure during analysis

Both the analysis routines (MA27H and MA27K) hold lists of integers in the INTEGER*2 array IW. Each list is headed by its length and is accessed via the INTEGER array IPE of length n holding pointers to the starts of the lists.

For simplicity, new lists are always written to the start of the free space at the end of IW and no immediate attempt is made to reuse the space occupied by the old list. This means that occasionally the storage will need to be "compressed" so that all free space is at the end of IW. Such compresses are performed by MA27I for both MA27H and MA27K. Our experience is that these compresses are relatively infrequent and do not contribute greatly to the overall cost (see Table 1.2) even when minimal storage is allocated to IW. Note that discarding explicit non-zeros on the diagonal usually yields plenty of "elbow room".

It is assumed that all the integers held in IW are non-negative and the routine begins by overwriting the lengths at the head of each list by -(its index), after having saved the length in the corresponding position in array IPE. Then IW is scanned. When a negative number is found, this yields an index and the entry length is recovered from the corresponding position in array IPE. The entry is then copied forward and a pointer to it is set in IPE. The scan is then continued from the end of the old position of the list. Thus a single scan of the array IW is all that is needed.

3.5 MA27J : sort prior to analysis given a pivot sequence

The analysis code for the case when a pivot sequence is given (MA27K) assumes, as does MA27H, that diagonal entries are always suitable as pivots and for the purpose of analysis we do not store them explicitly. Again we commence with

$$A = \sum_{\ell} B^{(\ell)} \tag{10.1}$$

where each $B^{(\ell)}$ corresponds to an off-diagonal pair a_{ij}, a_{ji}. However access is now required only from whichever of i and j appears earlier in the given pivot sequence. This means that the work array IW need only be about half as long as before and a somewhat different sort to that of MA27G is needed.

As in MA27G we begin by scanning the non-zeros, checking for invalid indices and counting the numbers of non-zeros in the rows. Again we warn about out-of-range indices and ignore diagonal entries, but now we transfer only row indices into IW and only count the off-diagonal entries as belonging in the upper triangular part of the permuted matrix. The dummy value zero is placed in IW to indicate an entry to be ignored.

The row counts are then accumulated to give pointers to row ends and an in-place sort is performed in IW, similar to that in MA27G (Section 3.2). One difference is that within the sort itself we can now only use IW(i), i = 1,2,...,nz because we have declared to the user that ICN(1) may be equivalenced with IW(i), i > nz. It is therefore not safe to leave gaps

ahead of each list ready to hold the list length. Instead we must move the lists forward in storage after the sort.

Another difference is that duplicate entries in the rows are normally innocuous in MA27K, so we do not remove them in MA27J. The only exception is if the number of entries in a row with duplicates is too large for INTEGER*2 storage. This cannot happen without duplicates because we assume that n is small enough to be stored as an INTEGER*2 variable. Therefore we remove duplicates only in this case and for simplicity in the code do it for all rows if we need to do it for any.

3.6 MA27K : analysis given pivot sequence

Knowing the pivot sequence makes analysis much easier. There is no longer any need to calculate and revise the degrees of the variables. The point at which a generated element matrix will be wanted is known at the time of its creation, since this will be when the first of its variables (in pivot order) becomes pivotal. We therefore chain together all generated elements having the same first variable. At each pivotal step we merge the index list of the row of the upper triangular part of A (i.e. all the original element matrices $B^{(\ell)}$ needed at that step) with the index lists of all the generated element matrices whose first variable is now pivotal. Notice that the original matrix rows and the generated index lists are each searched just once. There is therefore no point in looking for rows identical to other rows or elements that may be absorbed since any such tests would involve extra searches of the lists. One consequence is that every internal node of the tree corresponds to the elimination of a single variable, which would not be very efficient during factorization. However this is rectified during the depth-first search (MA27L, see next subsection).

As in the case of minimum degree analysis (MA27H), each list is constructed by merging old lists, so there will always be sufficient storage available, though occasional compresses (by MA27I) may be needed.

3.7 MA27L : depth-first tree search

The analysis routines (MA27H and MA27K) construct an assembly tree held in the array IPE as son-father pointers. MA27H also stores variable-supervariable pointers in IPE. Another array, NV, is used to flag which variables represent their supervariables (and have generated elements with the same numerical name). For these variables NV actually holds their degrees just before their elimination.

A depth-first search, which we describe in detail in Duff and Reid (1982), requires ready access from a node to all its sons. We therefore change the way the tree is stored. Another INTEGER array IPS is used for pointers from the nodes to their 'eldest' sons, and if node i has a younger brother then IPE(i) is changed to point to him. By making these changes in two passes, the first for nodes representing variables absorbed into supervariables and the second for nodes at which eliminations take place, we ensure that elimination nodes are always elder brothers of the others.

The depth-first search is organized in n steps. Each begins by moving from father to son to grandson, etc. as far as possible. The node reached then represents the next variable in pivot sequence and all the father-son links are removed immediately after use (to prevent reuse). We then move to the next younger brother if there is one or failing this go back to the father. This completes the moves of the step. The ordering of brothers ensures that when a node with k eliminations is reached, the last k-1 nodes visited will be nodes without eliminations representing variables absorbed into a supervariable. It is easy to count their number as we go along. It is also easy to accumulate the numbers of elimination node sons each node has. These numbers are needed by the factorization subroutine to tell it how many generated element matrices to retrieve from the stack at each stage. If there is just one and the difference in degrees between the father and his one son equals the number of eliminated variables, then the father and son nodes can be amalgamated without any additional fill-in. This amalgamation is important for efficiency in the factorize subroutine when analysis has been performed by MA27K since this subroutine does not absorb variables into supervariables. It can be useful also after MA27H

since not quite all identical rows are recognised (only rows active in a step are checked).

We also combine two steps if each involves less than NEMIN eliminations, where NEMIN is a parameter in common block MA27F. Our hope was that the extra efficiency of eliminating several variables together would compensate for the consequent increase in fill-in. Our experience has been that any gains when running on the IBM have been slight, although on the CRAY-1 (see Duff and Reid, 1982) gains (sometimes as much as 30%) have been recorded at a NEMIN level of 8. We therefore set the default value of NEMIN to 1 for the IBM and 8 for the CRAY-1, which means that such amalgamations are not done on the IBM.

The output from MA27L consists of a pivotal order (set in IPS), the number of elimination stages (in NSTEPS) and for each stage the number of variables eliminated (in NE), the number of generated element matrices assembled from the stack (in NA) and the degree (in ND). This provides sufficient information for storage and operation counts to be made by MA27M and for actual factorization later.

3.8 MA27M : computer storage requirements and multiplication counts for factorization

As we have stated earlier, an important feature of the MA27 package is that the analysis can be performed in a predetermined amount of storage (independent of fill-in) and will yield a forecast of the storage requirements and number of multiplications in a subsequent numerical factorization. If the matrix is definite and storage was not saved by equivalencing IRN and ICN with IW, the forecast will be exact. For indefinite systems, experience has shown that little extra storage is needed and that sometimes the integer storage requirement is actually reduced (see Duff and Reid, 1982). We now discuss subroutine MA27M which calculates these forecasts.

The information computed by MA27M is returned to the user in common block MA27E. Values for real and integer storage for the factorization (both with and without compresses) and for the factors themselves are so

returned in addition to the number of multiplications required to factorize a definite matrix. When IW(1) has been equivalenced to IRN(1) to save storage, MA27M can only provide an estimate for the minimum real and integer space. These will always be over-estimates, so it is quite safe to use them in a subsequent factorization, but they are often rather pessimistic (see Duff and Reid, 1982).

We discussed, in Section 3.1, the method we use to discover automatically whether IRN(1) has been equivalenced to IW(1). If no equivalence has been employed, MA27M first scans the original input and calculates and stores (in array LSTKI) the number of non-zeros in each row in the upper triangle of the permuted matrix. When an equivalence has been used the total number of non-zeros in each row (passed from MA27G or MA27J) is known and this is held in permuted row order in LSTKI.

We then, in the main loop of MA27M, run through the tree nodes at each stage updating the storage requirements for the factors, the frontal matrix, the stack and the remaining original matrix rows. A running total of the number of multiplications required is also kept.

At each tree node we know (from the output in arrays NA, NE and ND passed from MA27L) the number of stack elements, NSTK, and original rows assembled, NELIM, the number of eliminations performed, NELIM, and the order of the assembled frontal matrix, NFR. Since at most one element is stacked at each tree node and the construction of the tree by MA27L has ensured that at least one original row is assembled (and eliminated) at each node, we can store the lengths of the stacked elements and the lengths of the original rows in the same array (LSTKI). At each node we remove NSTK elements from the stack, perform NELIM eliminations on a matrix of order NFR and stack an element of order NFR-NELIM. Since NELIM original rows are assembled at each step, it is a trivial matter to keep track of the storage required for the remaining rows by subtraction from a running total.

Finally, we must ensure that our storage forecasts are sufficient for the sort routine MA27N, although in the vast majority of cases the storage required for the actual factorization will dominate.

4. The numerical factorization

In this section, we are concerned with the numerical factorization of the matrix. Calls to the factorization routines must follow calls to the analysis routines (Section 3) and must precede calls to the solve routines (Section 5).

The driver subroutine for numerical factorization is MA27B which in turn uses the auxiliary routines MA27N, NA27O and MA27P to effect the factorization. We show the call tree for numerical factorization in Figure 4.1 and discuss the driver and its auxiliary routines in subsections 4.1 to 4.4.

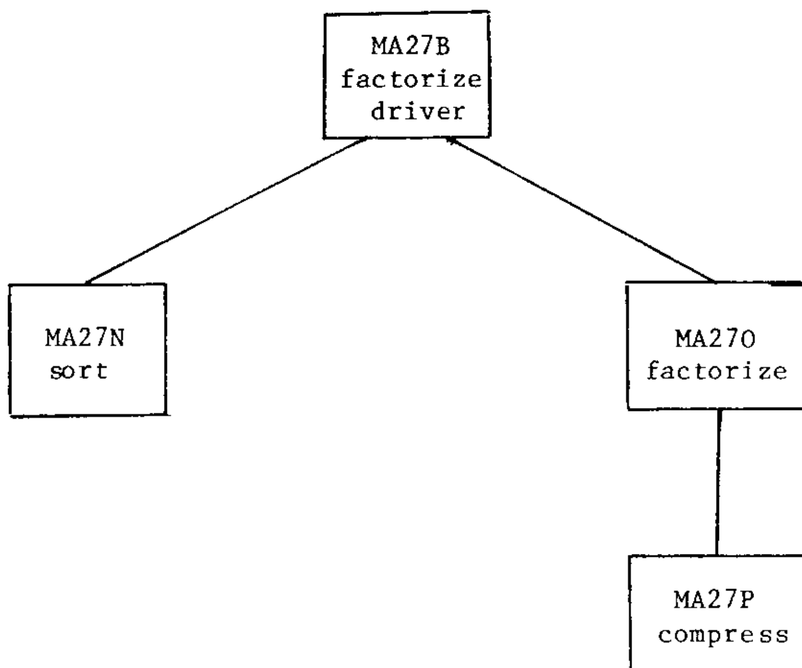

Figure 4.1 Call tree for numerical factorization

4.1 MA27B : driver subroutine for numerical factorization

MA27B is the driver subroutine called by the user to factorize a given matrix, following the analysis of its pattern by MA27A.

The non-zeros may again be in any order so the first task necessary is to sort them. This sort is quite similar to that needed by MA27A when the pivot sequence is known but the added presence of the reals and the fact that diagonal entries are needed here and not there made it impossible to use the same code without loss of speed. Therefore a third sort routine, MA27N, is called here.

Following this, the factorization routine MA27O is called, which itself makes occasional use of a very simple routine (MA27P) for compressing the data structure.

As for MA27A, this driver provides two levels of diagnostic printing on entry and exit and prints error messages. The only printing outside it is of warning messages about out-of-range indices in the sort routine MA27N and of unexpected changes in signs of pivots in the factorization routine MA27O.

We call the order passed to MA27B the "tentative" pivot order because in the non-definite case it may be modified during numerical factorization. Similarly the number of eliminations at each stage is tentative and may be changed. However we treat the number of stack assemblies as firm, even though some may, strictly speaking, become redundant. We do not, however, actually alter any of the arrays output from MA27A. Thus, the analysis information can be used for a sequence of matrices with the same sparsity pattern.

4.2 MA27N : sort prior to factorization

The sort prior to numerical factorization, performed by subroutine MA27N, is similar to that of MA27J although in MA27N we must also sort the real values and include entries on the diagonal. For convenience later in MA27O, we place the diagonals first in their rows.

As in MA27J, we first pass through the user's input data copying row indices to our work array (IW) and accumulating the number of non-zeros in each row in the upper triangle of the matrix permuted according to the tentative pivot order passed to us from MA27A. Entries out-of-range are flagged with a zero in the appropriate entry of IW, a diagnostic flag is set, and up to ten warning messages are printed. During this scan diagonal entries are accumulated and the IW entry for these set to zero also. We do the accumulation at this stage because it avoids problems later with the identification of diagonal entries which are not present (we will hold them as explicit zeros), saves storage and facilitates placing the diagonal first in each row of the sorted matrix.

As in the MA27G sort, another reason for transferring row indices to IW and using this work array for the subsequent sort is that the user's input data may be preserved. We point out, however, that this data is not needed for any further reasons by MA27B or MA27C (and their auxiliary routines). Thus, unless there are external reasons for preserving the input information (for example, should iterative refinement be in use), we expect the user to associate the input arrays IRN and ICN with parts of IW (for details see the specification in the appendix). We have taken care to ensure that the code works correctly in this case.

We then perform an in-place sort in an identical fashion to MA27J only this time reals as well as integers are moved. This in-place sort does not involve the diagonals which are placed directly in position immediately afterwards since after the in-place sort the running pointers will point to the beginning of each row. At the same time as placing the diagonals in position, the column index (equal to the row index) is flagged negative. This enables MA27O to avoid generating an inverse permutation array since the only requirement for such a permutation is to identify the original index of each row when it is assembled. The only complication in placing the diagonals occurs if the initial number of entries is less than the sorted number because of missing diagonals. In this case, the in-place sort is performed without leaving room for the diagonals. The entries in the sorted matrix are then moved by rows to leave space for the diagonals. In this sort we need not concern ourselves with duplicate off-diagonal

entries. These will be summed automatically during the factorization MA27O and since we do not hold row lengths we are not troubled (as we were in the MA27J sort) if there are more than n entries in a single row.

Finally the sorted matrix is moved to the end of the arrays in preparation for the main factorization subroutine MA27O.

4.3 MA27O : actual factorization

The numerical factorization subroutine, MA27O, is very long. We would have liked to subdivide MA27O but were unable to find a way to do this without fear of substantial run-time overhead. The difficulty is that the main loop, which spans almost the whole subroutine, performs one step of the decomposition (that is all operations associated with one tree node), and the number of steps can be quite large. Any subdivision therefore risks a large number of subroutine calls with the overhead this would involve.

At each major step, we first assemble the frontal matrix, then perform any eliminations and finally place the remainder of the frontal matrix on the stack. The movement of data is shown in Figures 4.2 and 4.3.

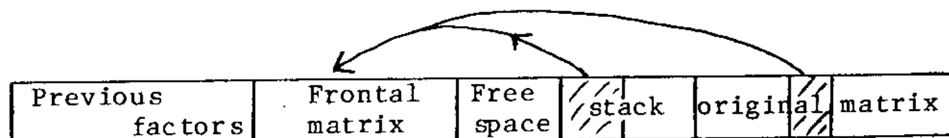

Figure 4.2 Assembly of frontal matrix

Figure 4.3 Placing remainder of front on stack

For the assembly phase we use the information passed from MA27A which gives at each stage the number of stacked elements assembled (in NSTK) (which may be zero) and the number of original matrix rows assembled (in NELIM) which, in the definite case, is equal to the number of eliminations. The original rows have been ordered by MA27N to pivot order and are stored with a corresponding integer list of column indices with the first entry, which will be the diagonal, flagged negative. At a typical stage, say the kth, we need to assemble the original element matrices $B^{(\ell)}$ which have not so far been assembled and which have an entry in any of the next NELIM(k) rows. These correspond exactly to the non-zeros in the upper-triangular parts of the rows of A and we will talk of "assembling the rows". The stacked elements are held as full upper-triangular matrices stored by rows. For each stacked element the integer information consists of the number of columns (and rows) in the element followed by the column indices. Because of our depth-first search (see Section 3.7), the NSTK(k) stacked elements required will be at the top (left hand end in Figures 4.2 and 4.3) of the stack and so identification of the elements required is particularly simple. Although the columns in the stacked elements are in pivotal order, the columns within each original row are not. Since we wish our assembled frontal matrix to be in tentative pivot order, we do a sort at the same time as the assembly. We effect this by doing the assembly in two stages, the first calculates indexing information for the frontal matrix and the second moves the reals themselves. An added advantage of splitting the assembly in this way is that the storage required for the frontal matrix is known at the end of the first stage (which only uses an auxiliary array) and so we can make sure that there is sufficient space (by data compression using MA27P, if necessary) before commencing the actual assembly. We call this first stage the symbolic assembly.

We begin the symbolic assembly with the stacked elements, if any, since their columns are ordered and the assembly does not involve any repeated searching. During this symbolic assembly we hold the column indices so far placed in the front in a linked list (in array IW2). A scan of this linked list in phase with the list of column indices of the next stacked element is satisfactory at each stage because both lists are in tentative pivotal order. When performing the subsequent symbolic assembly

of the original rows, we first check the IW2 entry to see if the column is in the front and, if it is not, then scan the chain to place it in the correct position. At the end of the symbolic assembly, we know the size of the new frontal matrix and its column indices in pivotal order.

We first make sure there is sufficient space in our arrays to hold the new frontal matrix and perform a very simple compress using MA27P if necessary. It is possible that there will still be insufficient space, in which case an error return is provoked, but the amount of storage to factorize a definite system was output from MA27A and our experience is that the requirements for indefinite systems differ only slightly. We then copy the integer and real information directly into place as indicated in Figure 4.2. The only subtlety here is that the linked list entries are converted into relative positions during the integer copy to facilitate the move of the reals.

We can now choose pivots in the frontal matrix and perform the eliminations. The maximum number of pivots will be the number of tentative eliminations plus the number of variables in stacked elements which were not previously eliminated because of stability considerations. Since we assembled in tentative pivot order, these rows and columns will come first. If the user has indicated that the matrix is definite (by setting $U \leqslant 0.0$, see Section 2.2 of the specification sheet) we do no stability tests but perform the eliminations as long as the pivot is non-zero and has not changed sign. A zero pivot (when the matrix has been declared definite) raises an error condition as does a change in sign when U is set to zero. When U is set negative, changes in sign are flagged and a warning is printed but the decomposition continues.

The frontal matrix, at an intermediate stage, has the form shown in Figure 4.4. When $U > 0$ we test the next potential pivot for stability in the following manner. We first search the whole of the potential pivot row determining the largest entry in the potential pivot columns (shaded in Figure 4.4) and the largest entry in the rest (unshaded in Figure 4.4). Searches of this kind (and swops) are complicated by the fact that we store only the upper triangle of the frontal matrix and so, except for the first

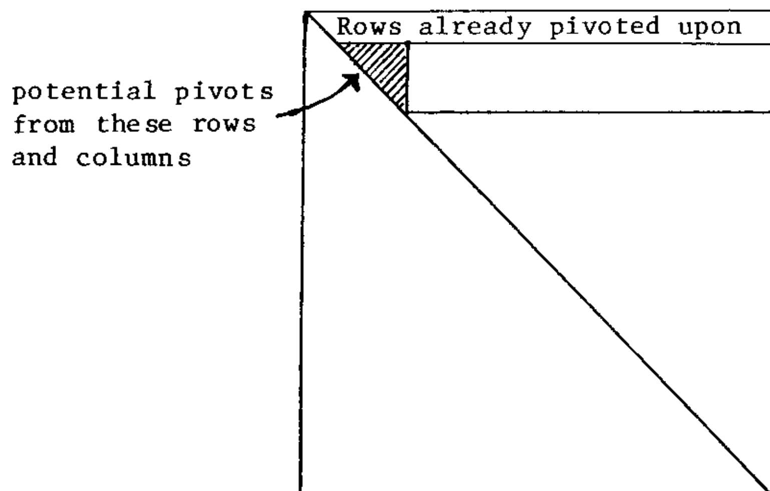

Figure 4.4 Frontal matrix during pivoting and elimination

uneliminated row, we must perform the search in two parts, by columns for the lower triangle and by rows for the upper triangle. The diagonal entry (a_{jj}, say) is then tested against the largest in all the row (a_{jk}, say) to see if the inequality

$$|a_{jj}| > U \cdot |a_{jk}| \, . \tag{4.1}$$

holds. If so then a_{jj} is used as pivot. If not, and entry $a_{j\ell}$ is the largest entry in the first part of row j (shaded region in Figure 4.4), we then test the 2x2 pivot

$$\begin{bmatrix} a_{jj} & a_{j\ell} \\ a_{j\ell} & a_{\ell\ell} \end{bmatrix} \tag{4.2}$$

for stability. To do this test, we first check the inequality

$$\left\| \begin{bmatrix} a_{jj} & a_{j\ell} \\ a_{j\ell} & a_{\ell\ell} \end{bmatrix}^{-1} \right\|_{\infty} < \quad |a_{jk}|/U \tag{4.3}$$

and, if this is satisfied we scan row to find the largest entry, $a_{\ell m}$ say, and accept the 2x2 pivot (4.2) if it satisfies the inequality

$$\left\| \begin{bmatrix} a_{jj} & a_{j\ell} \\ a_{j\ell} & a_{\ell\ell} \end{bmatrix}^{-1} \right\|_{\infty} < \quad |a_{\ell m}|/U \tag{4.4}$$

The stability test implied by the inequalities (4.3) and (4.4) is exactly that recommended by Duff et al (1979), who noted that with U in the range $(0,\frac{1}{2}]$ it is always possible to find a 1x1 or a 2x2 pivot.

If both the 1x1 and 2x2 pivot from row j fail the stability test, then our search for a pivot continues from the next potential pivot row. If we are successful in choosing a pivot we symmetrically permute it to the front of the uneliminated region (shaded in Figure 4.4) before performing the elimination operation themselves. We then look similarly for further pivots, cycling from the first row unsearched last time.

After the eliminations on the frontal matrix are complete, the rows from which pivots have been chosen will be at the beginning of the frontal matrix (see Figure 4.3) and so can be left in place as the next block pivot row in the factors. The remainder of the frontal matrix must be placed on the stack. A data compress may be needed to create room for its associated integer list. The illustration of data movement in Figure 4.3 is expanded in Figure 4.5 where the integer storage for each block pivot row is explicitly shown. The only added complication in this integer storage is

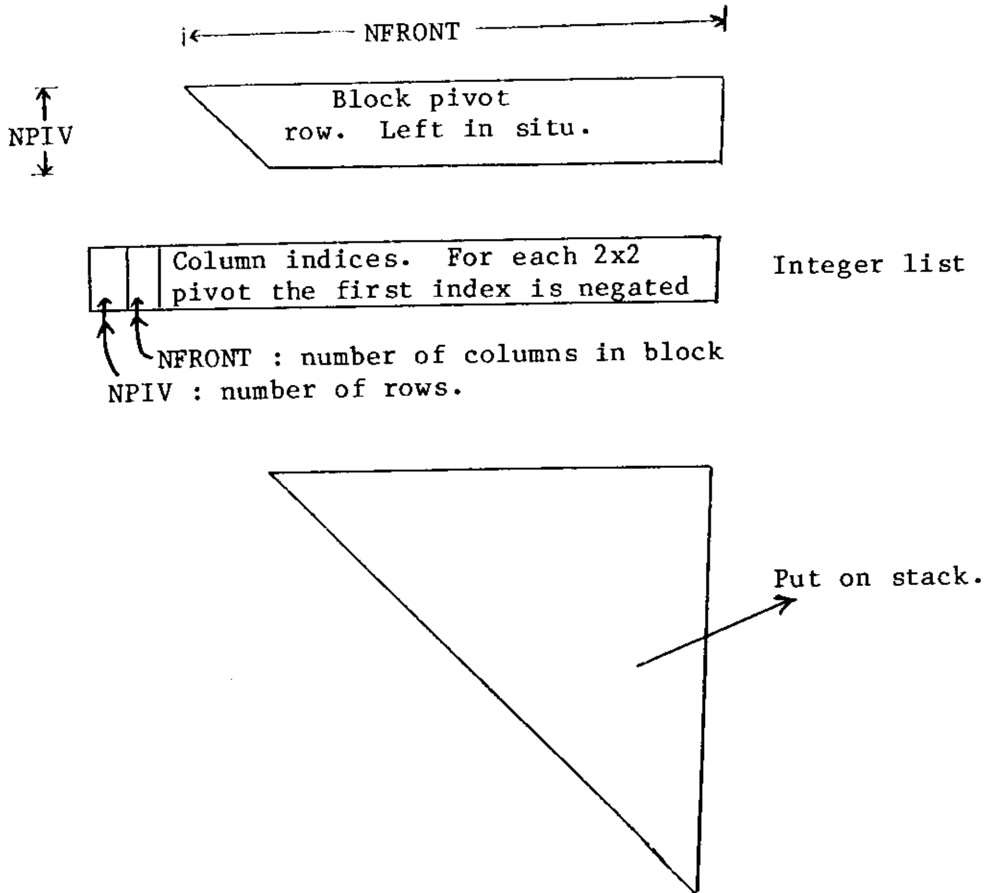

Figure 4.5 Disposition of frontal matrix

that, if NPIV is 1, it is omitted and the list of column indices is preceded only by the number of rows (negated to indicate that NPIV is 1). It is important, when comparing MA27 to other codes, to realise that this is all the integer information that needs to be preserved between calls to MA27B and subsequent calls to MA27C. There are no pointer arrays or permutation vectors required.

Once the frontal matrix has been disposed of as shown in Figure 4.5, we are ready to continue with further assemblies.

Notice that any null rows or columns or any zero block will effectively be swept to the end of the factored form and will not ever need to be stored. The total number of pivots (NTOTPV) will, in this case, be less than n, the order of the system, but the decomposition will be valid for the nonsingular submatrix of order NTOTPV.

4.4 MA27P : compress data structure during factorization

The subroutine, MA27P, for compressing the data structure during factorization is even simpler than that used by the analyse routines.

As we discussed in Section 4.3, the only compress necessary is to move the data in the stacked elements to overwrite original rows which have already been treated. We illustrate this in Figure 4.6.

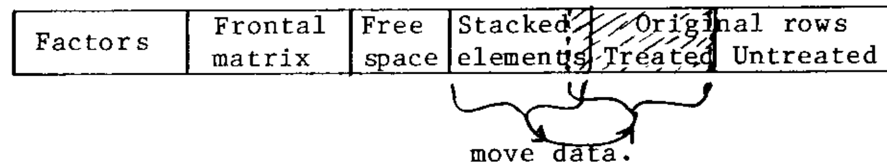

Figure 4.6 Illustration of compression during factorization

Since we do not use any pointers in the data structure the relative position of the data within the array in Figure 4.6 is unimportant. Thus a simple move (in reverse order to avoid potential overwriting) is all that is required.

The data structures for reals and integers (and when compresses are required on them) are quite independent. We choose to implement compresses for both types in the one subroutine although the code for each is quite separate.

5. Solution of a set of equations

After the analysis and numerical factorization of the coefficient matrix by MA27A and MA27B respectively, we now use the factors which can be written

$$P\ A\ P^T = U^T\ D\ U \qquad (5.1)$$

to solve the system

$$A\ \underline{x} = \underline{b} \qquad (5.2)$$

The solution to (5.2) using the decomposition (5.1) is effected in two steps. The first (forward substitution) solves the system

$$U^T \underline{y} = P\ \underline{b} \qquad (5.3)$$

and the second (back substitution) solves the system

$$D\ U\ (P\underline{x}) = \underline{y} \qquad (5.4)$$

Because the original column indices are held in the integer information for the factors the permutations represented by the matrix P in (5.3) and (5.4) will be performed implicitly when accessing components of $\underline{b}$ or storing components of $\underline{x}$. Thus we do not need any storage for P and so do not refer again to the permutations in this section.

The driver subroutine called by the user for the solve entry is MA27C and the call tree for this phase of the computation is shown in Figure 5.1.

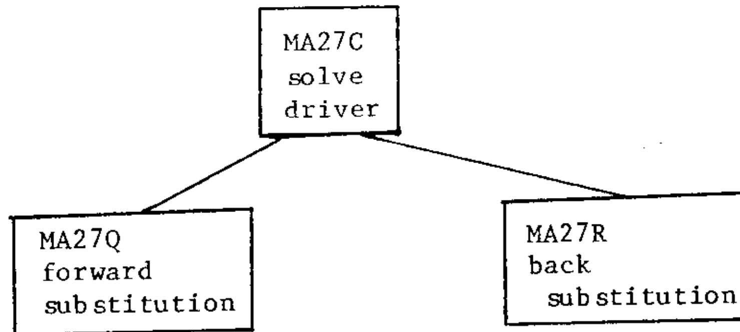

Figure 5.1 Call tree from MA27C

5.1 MA27C : driver subroutine for solution of equations

The driver subroutine MA27C is very simple. As for the other drivers, it provides two levels of optional diagnostic printing on entry and exit. It checks for the degenerate case that A is a zero matrix and, in this case, sets the solution to the zero vector. Otherwise it calls MA27Q for forward substitution and MA27R for back substitution. These subroutines have no error returns or diagnostic printing.

5.2 MA27Q : forward substitution

Subroutine MA27Q uses the factors produced by MA27O to effect the forward substitution. That is, MA27Q solves the system

$$U^T \underline{y} = \underline{b}$$

where the solution, $\underline{y}$, overwrites $\underline{b}$ held in array RHS. The subroutine performs the solution a block pivot row at a time, the number of passes through the main loop (NBLK) being equal to the number of block pivots.

For each block pivot (stored in A and IW), we first determine whether it is computationally better to use direct or indirect addressing in the innermost loop. For direct addressing we first load an auxiliary vector (W) with the components of RHS which correspond to columns in the block pivot row. Computations for each row of the block pivot row are then performed on W using direct addressing. Finally, W is copied back to the

appropriate components of RHS. When using indirect addressing, operations in the inner loop are performed on array entries of RHS itself, the indirect addressing being required to identify the appropriate entries in RHS. Since direct addressing requires the load and unload of W only once for the whole block pivot, the choice of whether direct or indirect addressing is faster will depend both on the number of rows in the block pivot (NPIV) as well as the number of columns (LIELL). This choice will also be machine dependent. For example, a computer like the CRAY-1 will execute loops involving only direct addressing much faster than if indirect addressing is present. The IBM 3033 is also faster when all addressing is direct but less markedly so than the CRAY-1. We report on experiments on these two computers in Duff and Reid (1982).

To avoid machine dependence in the main body of the code, we hold information in the array IFRLVL of common block MA27F to determine whether direct or indirect addressing is faster. This can be reset by the user to match the characteristics of the machine being used. Our experience is that the break-even points depend both on the number of eliminations in the block and on its number of columns. We therefore hold in IFRLVL(i), the minimum number of columns required in the block pivot for direct addressing to be better when the number of block pivot rows is equal to i, i = 1,2,...10. If there are more than 10 block pivot rows, we use IFRLVL(10) in our test. In the main Harwell code for use on the IBM the default values for IFRLVL(i), i = 1,2,...10 have been set, in a block data subprogram, to:

32639, 32639, 32639, 32639, 14, 9, 8, 8, 9, 10

where 32639 is the value of IOVFLO and means that indirect addressing is always used when NPIV $\leqslant$ 4.

For each block pivot we thus first test to see whether direct or indirect addressing in the innermost loop is preferable. The code for each form of addressing is quite independent. In each case we first determine (by looking for negative flags in IW) whether the current pivot is 1x1 or 2x2 and then use an innermost loop appropriate to the pivot size. Since our experience is that block pivots often have only one row, we avoid an added loop when using indirect addressing by performing the operations for

only one pivot at each pass through the main loop. A simple check (on NPIV) at the beginning of the main loop determines whether we fetch the next block pivot row or continue processing the present one. Because we wish to load and unload W only once when using direct addressing, there is a loop on the number of pivots in the block in this case.

When we scan A and IW to identify and use each block pivot row in turn, we set an auxiliary pointer array (IW2) to point to the first entry in IW corresponding to each block pivot row and record the position of the last entry in the factors held in A (in the scalar LATOP). We do this to facilitate MA27R which requires access to block pivot rows in reverse order.

5.3 MA27R : back substitution

Subroutine MA27R uses the matrix factors in A and IW to solve the system

$$D\,U\,\underline{x} = \underline{y}$$

where the solution $\underline{x}$ overwrites the output $\underline{y}$ from MA27Q in array RHS. As in MA27Q, the subroutine performs the solution a block pivot row at a time this time using the factors in reverse order.

Again we first choose whether direct or indirect addressing in the innermost loop is better by comparing the number of columns in the block pivot with an entry in IFRLVL appropriate to the number of rows in the block pivot. Because the innermost loop is a scalar product rather than an addition of a multiple of one vector to another, we cannot use the same IFRLVL values as we used for the tests in MA27Q. We store appropriate values in IFRLVL(i), i = 11,12,...20 and, if there are NPIV rows in the block pivot, we perform the test against entry 10 + min(10,NPIV) of IFRLVL. The default values for IFRLVL(i), i = 11,12,...20 for the IBM are

32639, 32639, 32639, 32639, 24, 11, 9, 8, 9, 10.

As in MA27Q, we use separate innermost loops for 1x1 or 2x2 pivots and, when using indirect addressing, loop only on the main loop if there is more than one row in the block pivot. At any stage, we know the position of the last non-zero in the current block pivot row (APOS-1) and can access the appropriate indexing information directly from the array IW2 set by MA27Q.

References

1. Duff, I.S., Munksgaard, N., Nielsen, H.B. and Reid, J.K. (1979). Direct solution of sets of linear equations whose matrix is sparse, symmetric and indefinite. J. Inst. Maths. Applics. 23 pp 235-250.

2. Duff, I.S. and Reid, J.K. (1982). The multifrontal solution of indefinite sparse symmetric linear systems. Harwell Report CSS122.

3. Gustavson, F.G. (1978). Two fast algorithms for sparse matrices : multiplication and permuted transposition. ACM Trans. Math. Software 4 pp 250-269.

4. Ryder, B.G. (1974). The PFORT verifier. Software Practice and Experience 4 pp 359-377.

1 SUMMARY

To **solve a sparse symmetric system of linear equations.** Given an $n \times n$ sparse symmetric matrix $\mathbf{A} = \{a_{ij}\}$ and an n-vector **b**, this subroutine solves the system **Ax=b**. The matrix **A** need not be definite.

The method used is a direct method based on a sparse variant of Gaussian elimination and is discussed further by Duff and Reid, AERE R.10533 (1982).

ATTRIBUTES — Versions: MA27A, MA27AD **Language:** Fortran (standard available). **Date:** June 1982. **Size:** 34,400 bytes; 2,920 cards. **Origin:** I.S.Duff and J.K.Reid, Harwell. **Conditions on external use:** (i), (ii), (iii) and (iv).

2 HOW TO USE THE ROUTINE

2.1 Argument lists and calling sequences

There are three entries:

(a) MA27A/AD accepts the pattern of **A** and chooses pivots for Gaussian elimination using a selection criterion to preserve sparsity and subsequently constructs subsidiary information for actual factorization by MA27B/BD. The user may input his own pivot sequence in which case only the necessary information for MA27B/BD will be generated.

(b) MA27B/BD factorizes a matrix **A** using the information from a previous call to MA27A/AD. The actual pivot sequence used may differ slightly from that produced by MA27A/AD if **A** is not definite.

(c) MA27C/CD uses the factors generated by MA27B/BD to solve a system of equations **Ax=b**.

A call to MA27C/CD must be preceded by a call to MA27B/BD which in turn must be preceded by a call to MA27A/AD. Since the information passed from one subroutine to the next is not corrupted by the second, several calls to MA27B/BD for matrices with the same sparsity pattern but different values may follow a single call to MA27A/AD, and similarly MA27C/CD can be used repeatedly to solve for different right hand sides **b**.

To perform symbolic manipulations

The single precision version:

```
      CALL MA27A(N,NZ,IRN,ICN,IW,LIW,
     *           IKEEP,IW1,NSTEPS,IFLAG)
```

The double precision version:

```
      CALL MA27AD(N,NZ,IRN,ICN,IW,LIW,
     *            IKEEP,IW1,NSTEPS,IFLAG)
```

N is an **INTEGER variable** which must be set by the user to the order n of the matrix **A**. It is not altered by the subroutine. **Restriction** (IBM version): $1 \leqslant n \leqslant 32639$.

NZ is an **INTEGER variable** which must be set by the user to the number of non-zeros being input. It is not altered by the subroutine. **Restriction:** NZ $\geqslant$ 0.

IRN, ICN are **INTEGER*2** *(or* **INTEGER** *for Fortran 66 version)* **arrays** of length NZ. The user must set them so that each off-diagonal non-zero a_{ij} is represented by IRN(k)=i and ICN(k)=j or by IRN(k)=j and ICN(k)=i. These arrays will be unaltered by the subroutine unless the user wishes to conserve storage which can be done by equivalencing IRN(1) to IW(1) and ICN(1) to IW(k), k > NZ.

IW is an **INTEGER*2** *(or* **INTEGER** *for Fortran 66 version)* **array** of length LIW. This is used as workspace by the subroutine. Its length must be at least 2*NZ+3*N+1 (or NZ+3*N+1 if the pivot order is specified in IKEEP), but we recommend that it should be at least 20% greater than this (see NCMPA in section 2.2).

LIW is an **INTEGER variable**. It must be set by the user to the length of array IW and is not altered by the subroutine.

IKEEP is an **INTEGER*2** *(or* **INTEGER** *for Fortran 66 version)* **array** of length 3N. It need not be set by the user and must be preserved between a call to MA27A/AD and subsequent calls to MA27B/BD. If the user wishes to input his own permutation, the position of variable i in the pivot order should be placed in IKEEP(i), i=1,2,...,n and IFLAG should be set to 1. Note that the given order may be replaced by another that gives virtually identical numerical results.

IW1 is an **INTEGER array** of length 2N. It is used as workspace by the subroutine.

NSTEPS is an **INTEGER variable**. It need not be set by the user on input and should be passed unchanged when later calling MA27B/BD.

IFLAG is an **INTEGER variable** which the user must set to zero if he wishes a suitable pivot order to be chosen automatically or to 1 if he wishes the pivot order he has set in IKEEP to be used. On exit from MA27A/AD, a value of zero indicates that the subroutine has performed successfully. For non-zero values, see section 2.3.

To factorize a matrix

The single precision version:

```
      CALL MA27B(N,NZ,IRN,ICN,A,LA,IW,LIW,
     *           IKEEP,NSTEPS,MAXFRT,IW1,IFLAG)
```

The double precision version:

```
      CALL MA27BD(N,NZ,IRN,ICN,A,LA,IW,LIW,
     *            IKEEP,NSTEPS,MAXFRT,IW1,IFLAG)
```

N is an **INTEGER variable** which must be set by the user to the order n of the matrix **A**. It is not altered by the subroutine. **Restriction** (IBM version): $1 \leqslant n \leqslant 32639$.

NZ is an **INTEGER variable** which must be set by the user

to the number of entries in the matrix **A**. It is not altered by the subroutine. **Restriction:** NZ $\geq$ 0.

IRN, ICN, A. IRN and ICN are **INTEGER*2** *(or* **INTEGER** *for Fortran 66 version)* **arrays** of length NZ and A is a **REAL** *(or* **DOUBLE PRECISION** *for D version)* **array** of length LA. These must be set by the user to hold the non-zeros. A diagonal non-zero a_{ii} is held as A(k)=a_{ii}, IRN(k)=ICN(k)=i and a pair of off-diagonal non-zeros $a_{ij}=a_{ji}$ is held as A(k)=a_{ij} and IRN(k)=i, ICN(k)=j or vice-versa. Multiple entries are summed and any with IRN(k) or ICN(k) out of range are ignored. On exit array A will hold the non-zero entries of the factors of the matrix **A**. These entries in A must be preserved by the user between calls to this subroutine and subsequent calls to MA27C/CD. IRN and ICN will be unaltered by the subroutine unless the user wishes to conserve storage by equivalencing IRN(1) to IW(1) and ICN(1) to IW(k), k > NZ.

LA is an **INTEGER variable** which must be set by the user to the length of array A. It must be at least as great as NRLNEC of common block MA27E/ED (see section 2.2), set by MA27A/AD. It is advisable to allow a slightly greater value because the use of numerical pivoting might increase storage requirements marginally. It is not altered by the subroutine.

IW is an **INTEGER*2** *(or* **INTEGER** *for Fortran 66 version)* **array** of length at least as great as NIRNEC as output from MA27A/AD in common block MA27E/ED (see section 2.2). A slightly greater value is recommended because numerical pivoting may increase storage requirements marginally. IW, which need not be set by the user, is used as workspace by MA27B/BD and on exit holds integer indexing information on the matrix factors. It must be preserved by the user between calls to this subroutine and MA27C/CD.

LIW is an **INTEGER variable** which must be set by the user to the length of array IW. It is not altered by MA27B/BD.

IKEEP is an **INTEGER*2** *(or* **INTEGER** *for Fortran 66 version)* **array** of length 3N which must be passed unchanged since the last call to MA27A/AD. It is not altered by MA27B/BD.

NSTEPS is an **INTEGER variable** which must be passed unchanged since the last call to MA27A/AD. It is not altered by MA27B/BD.

MAXFRT is an **INTEGER variable** which need not be set by the user and should be passed unchanged to subsequent calls of MA27C/CD.

IW1 is an **INTEGER array** of length N. It is used as workspace by the subroutine.

IFLAG is an **INTEGER variable**. On exit from MA27A/AD, a value of zero indicates that the subroutine has performed successfully. For non-zero values, see section 2.3.

To solve equations using the factors from MA27B/BD

The single precision version:

```
      CALL MA27C(N,A,LA,IW,LIW,W,MAXFRT,
     *           RHS,IW1,NSTEPS)
```

The double precision version:

```
      CALL MA27CD(N,A,LA,IW,LIW,W,MAXFRT,
     *            RHS,IW1,NSTEPS)
```

N is an **INTEGER variable** which must be set by the user to the order n of the matrix **A**. It is not altered by the subroutine.

A is a **REAL array** *(or* **DOUBLE PRECISION** *for D version)* of length LA which must be unchanged since the last call to MA27B/BD. It is not altered by the subroutine.

LA is an **INTEGER variable** which must be set by the user to the length of **array** A. It is not altered by the subroutine.

IW is an **INTEGER*2** *(or* **INTEGER** *for Fortran 66 version)* **array** of length LIW which must be unchanged since the last call to MA27B/BD. It is not altered by the subroutine.

LIW is an **INTEGER variable** which must be set by the user to the length of **array** IW. It is not altered by the subroutine.

W is a **REAL array** *(or* **DOUBLE PRECISION** *for D version)* of length MAXFRT as output from MA27B/BD (see section 2.2). This value will always be less than n. W is used as workspace by MA27C/CD.

MAXFRT is an **INTEGER variable** which must be passed unchanged since the last call to MA27B/BD. It is not altered by MA27C/CD.

RHS is a **REAL array** *(or* **DOUBLE PRECISION** *for D version)* of length **n**. On entry, RHS(i) must hold the ith component of the right hand side of the equations being solved. On exit it will be equal to the corresponding entry of the solution vector.

IW1 is an **INTEGER array** of length NSTEPS as output from MA27A/AD. This value will be at most n. IW1 is used as workspace by MA27C/CD.

NSTEPS is an **INTEGER variable** which must be passed unchanged since the last call to MA27A/AD. It is not altered by the subroutine.

2.2 Common blocks used

There are three common blocks used by the MA27 routines. The first one, MA27D/DD, allows the user to input parameters to control the solution process and diagnostic printing and the second, MA27E/ED, provides information on the decomposition. The third, MA27F/FD, allows control of certain system dependent parameters and is unlikely to be of concern to many users.

The first common block can be declared as:

The single precision version:

```
      COMMON/MA27D/U,LP,MP,LDIAG
```

The double precision version:

```
      COMMON/MA27DD/U,LP,MP,LDIAG
```

where all the values are given default values by a block data subprogram. None of these values is altered by the subroutines.

U is a **REAL** *(or* **DOUBLE PRECISION** *for D version)* **variable** which is used by the subroutine to control

numerical pivoting. Values greater than 0.5 are treated as 0.5 and less than –0.5 as –0.5. Its default value is 0.1. If U is positive, numerical pivoting will be performed. If U is non-positive, no pivoting will be performed and the subroutine will fail if a zero pivot is encountered. If U is non-positive and not all pivots are of the same sign a flag (see section 2.3) will be set and the factorization will continue if U is zero and will exit immediately a sign change is detected if U is less than zero.

If the system is definite, then setting U to zero will decrease the factorization time while still providing a stable decomposition. For problems requiring greater than average numerical care a higher value than the default would be advisable.

LP is an **INTEGER variable** used by the subroutines as the output stream for error messages. If it is set to zero these messages will be suppressed. The default value is 6.

MP is an **INTEGER variable** used by the subroutines as the output stream for diagnostic printing and for warning messages. If it is set to zero then messages are suppressed. The default value is 6.

LDIAG is an **INTEGER variable** used by the subroutines to control diagnostic printing. If LDIAG is equal to zero (the default), no diagnostic printing will be produced, a value of 1 will print scalar parameters (both in argument lists and in common blocks) and a few entries of array parameters on entry and exit from each subroutine while LDIAG equal to 2 will print all parameter values on entry and exit.

The second common block is:

The single precision version:

```
COMMON/MA27E/OPS,IERROR,NRLTOT,NIRTOT,
*NRLNEC,NIRNEC,NRLADU,NIRADU,NRLBDU,
*NIRBDU,NCMPA,NCMPBR,NCMPBI,NTWO
```

The double precision version:

```
COMMON/MA27ED/OPS,IERROR,NRLTOT,NIRTOT,
*NRLNEC,NIRNEC,NRLADU,NIRADU,NRLBDU,
*NIRBDU,NCMPA,NCMPBR,NCMPBI,NTWO
```

OPS is a **REAL** *(or* **DOUBLE PRECISION** *for D version)* **variable**. On output from MA27A/AD, OPS will be set to the number of multiply-add pairs of operations required by the factorization if no pivoting is performed. Numerical pivoting in MA27B/BD may increase the number of operations slightly.

IERROR is an **INTEGER variable**. In the case of an error condition, extra information is placed here. For details, see section 2.3.

NRLTOT,NIRTOT are **INTEGER variables**. On exit from MA27A/AD, they give the total amount of REAL (or DOUBLE PRECISION for the D version) and INTEGER*2 (or INTEGER for Fortran 66 version) words respectively required for a successful completion of MA27B/BD without the need for data compression provided no numerical pivoting is performed. The actual amount required may be higher because of numerical pivoting, but probably not by more than 3%.

NRLNEC,NIRNEC are **INTEGER variables**. On exit from MA27A/AD, they give the amount of REAL (or DOUBLE PRECISION for the D version) and INTEGER*2 (or INTEGER for Fortran 66 version) words required respectively for successful completion of MA27B/BD allowing data compression (see NCMPBR), again provided no numerical pivoting is performed. Numerical pivoting may cause a higher value to be required, but probably not by more than 3%. If storage was conserved by equivalencing IW(1) with IRN(1). NRLNEC and NIRNEC cannot be calculated exactly but instead an upper bound will be returned. Experience has shown that this can overestimate the exact values by 50% although the tightness of the bound is very problem dependent. For example, a tight bound will generally be obtained if there are many more non-zeros in the factors than in the input matrix.

NRLADU,NIRADU are **INTEGER variables**. On exit from MA27A/AD, they give the number of REAL (or DOUBLE PRECISION for the D version) and INTEGER*2 (or INTEGER for Fortran 66 version) words required to hold the matrix factors if no numerical pivoting is performed by MA27B/BD. Numerical pivoting may change this slightly.

NRLBDU,NIRBDU are **INTEGER variables**. On exit from MA27B/BD, they give the amount of REAL (or DOUBLE PRECISION for the D version) and INTEGER*2 (or INTEGER for Fortran 66 version) words actually used to hold the factorization.

NCMPA is an **INTEGER variable**. On exit from MA27A/AD, NCMPA will hold the number of compresses of the internal data structure performed by MA27A/AD. If this is high (say > 10), the performance of MA27A/AD may be improved by increasing the length of array IW.

NCMPBR,NCMPBI are **INTEGER variables**. On exit from MA27B/BD, they will hold the number of compresses of the real and integer data structure respectively required by the factorization. If either of these is high (say > 10), then the speed of the factorization may be increased by allocating more space to the arrays A or IW as appropriate.

NTWO is an **INTEGER variable**. On exit from MA27B/BD, this gives the number of 2×2 pivots used during the factorization.

Although the third common block is not of interest to the general user, it is discussed more fully by Duff and Reid (AERE R–10533, 1982). If the user is working in an environment where all common blocks must be declared in the calling program, a suitable declaration would be:

The single precision version:

```
COMMON/MA27F/IDUMMY(22)
```

The double precision version:

```
COMMON/MA27FD/IDUMMY(22)
```

where

IDUMMY is an **INTEGER array** of length 22.

2.3 Error diagnostics

A successful return from MA27A/AD or MA27B/BD is indicated by a value of IFLAG equal to zero. There are no error returns from MA27C/CD. Possible non-zero values for IFLAG are given below. In each case an identifying

message is output on unit LP (errors) or MP (warnings).

- −1 Value of N out of range. Either N < 1 or N > 32639 (IBM version only) (MA27A/AD and MA27B/BD entries).
- −2 Value of NZ out of range. NZ < 0. (MA27A/AD and MA27B/BD entries).
- −3 Failure due to insufficient space allocated to array IW (MA27A/AD and MA27B/BD entries). IERROR in COMMON is set to a value that may suffice.
- −4 Failure due to insufficient space allocated to array A (MA27B/BD entry only). IERROR in COMMON is set to a value that may suffice.
- −5 Matrix is singular (MA27B/BD entry only). IERROR in COMMON is set to the pivot step at which singularity was detected.
- −6 A change of sign of pivots has been detected when U was negative. IERROR in COMMON is set to the pivot step at which the change was detected. (MA27B/BD entry only).

A positive flag value is associated with a warning message which will be output on unit MP.

- +1 Index (in IRN or ICN) out of range. Action taken by subroutine is to ignore any such entries and continue (MA27A/AD and MA27B/BD entries). IERROR in COMMON is set to the number of faulty entries. Details of the first ten are printed on unit MP.
- +2 Pivots have different signs when factorizing a supposedly definite matrix (input value of U is zero) (MA27B/BD entry only). IERROR in COMMON is set to the number of sign changes. Note that this warning will overwrite an IFLAG=1 warning. Details of the first ten are printed on unit MP.
- +3 Matrix is rank deficient. In this case, a decomposition will still have been produced which will enable the subsequent solution of consistent equations (MA27B/BD entry only). IERROR in COMMON will be set to the rank of the matrix. Note that this warning will overwrite an IFLAG=1 or IFLAG=2 warning.

3 GENERAL INFORMATION

Use of common: The subroutines use common blocks MA27D/DD, MA27E/ED, MA27F/FD (see section 2.2).

Workspace:

MA27A/AD:
IW (**INTEGER*2**) of length LIW
IW1 (**INTEGER**) of length 2*N

MA27B/BD:
IW1 (**INTEGER**) of length N

MA27C/CD:
W (**REAL** *(or* **DOUBLE PRECISION** *for D version)*) of length at most N
IW1 (**INTEGER**) of length at most N

Other routines: All the subroutines called by the principal subroutines are in the MA27 package. They are called MA27G/GD, MA27H/HD, MA27I/ID, MA27J/JD, MA27K/KD, MA27L/LD, MA27M/MD, MA27N/ND, MA27O/OD, MA27P/PD, MA27Q/QD, MA27R/RD

Input/output: Error warning and diagnostic messages only. Error messages on unit LP and warning and diagnostic messages on unit MP. These have default value 6, and printing of these messages is suppressed if LP or MP is set to 0.

Restrictions:

$N \geqslant 1$,
$N \leqslant 32639$ (IBM version),
$NZ \geqslant 0$.

Portability: If all INTEGER*2 declarations are changed to INTEGER, the resulting subroutines satisfy the PFORT verifier, a portable Fortran closely approximating the ANSI standard of 1966. In this case, the restriction $n \leqslant 32639$ (INTEGER*2 limit under WATFIV) is removed.

4 METHOD

A version of sparse Gaussian elimination is used.

The MA27A/AD entry (with IFLAG=0) chooses pivots from the diagonal using the minimum degree criterion employing a generalized element model of the elimination thus avoiding the need to store the filled-in pattern explicitly. The elimination is represented as an assembly and elimination tree with the order of elimination determined by a depth first search of the tree.

The MA27B/BD entry factorizes the matrix by using the assembly and elimination ordering generated by MA27A/AD. At each stage in the multifrontal approach pivoting and elimination are performed on full submatrices and, when diagonal 1×1 pivots would be numerically unstable, 2×2 diagonal blocks are used. Thus MA27B/BD can be used to factor indefinite systems and will perform well on machines capable of vectorization.

The MA27C/CD entry uses the factors from MA27B/BD to solve systems of equations either by loading the appropriate parts of the vectors into an array of the current front-size and using full matrix code or by indirect addressing at each stage, whichever performs better.

A fuller account of this method is given by Duff and Reid (AERE-R.10533, 1982).

5 EXAMPLE OF USE

We illustrate the use of the package on the solution of the single set of equations

$$\begin{pmatrix} 2 & 3 & & & \\ 3 & 0 & 4 & & 6 \\ & 4 & 1 & 5 & \\ & & 5 & 0 & \\ & 6 & & & 1 \end{pmatrix} \mathbf{x} = \begin{pmatrix} 8 \\ 45 \\ 31 \\ 15 \\ 17 \end{pmatrix}$$

We have set LDIAG to 2 so that all the information passed to and from the package is displayed in this small case. Note that this example does not illustrate all the facilities.

Program

```
C SIMPLE EXAMPLE OF USE OF MA27 PACKAGE
      INTEGER*2 IRN(10),ICN(10),IW(40),IKEEP(15)
      INTEGER IW1(10)
      DOUBLE PRECISION A(30),W(5),RHS(5),U
      COMMON /MA27DD/ U,LP,MP,LDIAG
C
C STORE ARRAY LENGTHS
      LIW=40
      LA=30
C
C ASK FOR FULL PRINTING FROM MA27 PACKAGE
      LDIAG=2
C
C SET IFLAG TO INDICATE PIVOT SEQUENCE IS TO BE FOUND BY MA27AD
      IFLAG=0
C
C READ MATRIX AND RIGHT-HAND SIDE
      READ(5,10)N,NZ
10    FORMAT(2I2,F4.1)
      READ(5,10)(IRN(I),ICN(I),A(I),I=1,NZ)
      READ(5,20)(RHS(I),I=1,N)
20    FORMAT(5F4.1)
C
C ANALYZE SPARSITY PATTERN
      CALL MA27AD(N,NZ,IRN,ICN,IW,LIW,IKEEP,IW1,NSTEPS,IFLAG)
C
C FACTORIZE MATRIX
      CALL MA27BD(N,NZ,IRN,ICN,A,LA,IW,LIW,IKEEP,NSTEPS,MAXFRT,
     *            IW1,IFLAG)
C
C SOLVE THE EQUATIONS
      CALL MA27CD(N,A,LA,IW,LIW,W,MAXFRT,RHS,IW1,NSTEPS)
      STOP
      END
```

Data

```
 5 7
 1 1 2.0
 1 2 3.0
 2 3 4.0
 2 5 6.0
 3 3 1.0
 3 4 5.0
 5 5 1.0
  8. 45. 31. 15. 17.
```

Output

```
 ENTERING MA27AD WITH        N       NZ      LIW   IFLAG
                             5        7       40       0
 MATRIX NON-ZEROS        1       1       1       2       2       3       2       5
                         3       3       3       4       5       5
```

```
LEAVING MA27AD WITH NSTEPS   IFLAG     OPS IERROR NRLTOT NIRTOT
                         4       0      4.     0     19     25
                    NRLNEC NIRNEC NRLADU NIRADU  NCMPA
                        14     20      9     14     0
IKEEP(.,1)=     5     4     3     2     1
IKEEP(.,2)=     1     1     1     2
IKEEP(.,3)=     0     0     2     1


ENTERING MA27BD WITH        N       NZ       LA     LIW NSTEPS        U
                            5        7       30      40      4     0.10
MATRIX NON-ZEROS          0.2000D 01      1      1      0.3000D 01      1     2
                          0.4000D 01      2      3      0.6000D 01      2     5
                          0.1000D 01      3      3      0.5000D 01      3     4
                          0.1000D 01      5      5
IKEEP(.,1)=     5     4     3     2     1
IKEEP(.,2)=     1     1     1     2
IKEEP(.,3)=     0     0     2     1

LEAVING MA27BD WITH MAXFRT  IFLAG NRLBDU NIRBDU NCMPBR NCMPBI   NTWO IERROR
                         3      0     10     13      0      0      1      0
BLOCK PIVOT =         1 NROWS =        1 NCOLS =        2
COLUMN INDICES =     5     2
REAL ENTRIES .. EACH ROW STARTS ON A NEW LINE
 1.00000000D 00 -6.00000000D 00
BLOCK PIVOT =         2 NROWS =        2 NCOLS =        3
COLUMN INDICES =    -4     3     2
REAL ENTRIES .. EACH ROW STARTS ON A NEW LINE
-4.00000000D-02  2.00000000D-01 -8.00000000D-01
 0.00000000D-01 -0.00000000D-01
BLOCK PIVOT =         3 NROWS =        2 NCOLS =        2
COLUMN INDICES =     2     1
REAL ENTRIES .. EACH ROW STARTS ON A NEW LINE
-2.77777778D-02  8.33333333D-02
 4.44444444D-01


ENTERING MA27CD WITH        N       LA     LIW MAXFRT  NSTEPS
                            5       30      40      3      4
BLOCK PIVOT =         1 NROWS =        1 NCOLS =        2
COLUMN INDICES =     5     2
REAL ENTRIES .. EACH ROW STARTS ON A NEW LINE
 1.00000000D 00 -6.00000000D 00
BLOCK PIVOT =         2 NROWS =        2 NCOLS =        3
COLUMN INDICES =    -4     3     2
REAL ENTRIES .. EACH ROW STARTS ON A NEW LINE
-4.00000000D-02  2.00000000D-01 -8.00000000D-01
 0.00000000D-01 -0.00000000D-01
BLOCK PIVOT =         3 NROWS =        2 NCOLS =        2
COLUMN INDICES =     2     1
REAL ENTRIES .. EACH ROW STARTS ON A NEW LINE
-2.77777778D-02  8.33333333D-02
 4.44444444D-01
RHS  8.0000000D 00   4.5000000D 01   3.1000000D 01   1.5000000D 01   1.7000000D 01


LEAVING MA27CD WITH
RHS  1.0000000D 00   2.0000000D 00   3.0000000D 00   4.0000000D 00   5.0000000D 00
```