AERE - R 8730
(1980 revision)

AERE - R 8730
(1980 revision)

United Kingdom Atomic Energy Authority
HARWELL

MA28 – A set of Fortran subroutines for sparse unsymmetric linear equations

I.S. Duff
Computer Science and Systems Division
AERE Harwell, Oxfordshire
November 1980

PRICE £5.00 net from H.M. Stationery Office

C13

AERE-R.8730
(1980 revision)

MA28 - A SET OF FORTRAN SUBROUTINES FOR SPARSE UNSYMMETRIC LINEAR EQUATIONS

I.S. Duff

Abstract

In this report we present a suite of subroutines for the solution of sparse unsymmetric sets of linear equations using a variant of Gaussian elimination. The subroutines are divided into three distinct phases. The first phase optionally preorders the matrix to block triangular form and then performs a sparsity oriented factorization, the second factorizes a matrix of a similar sparsity pattern, while the third uses these decompositions to solve the equations.

In this revised edition, the actual Fortran listings have been replaced by a reference to their availability in machine readable form. Other changes to the first edition are minor. This revision is essentially a reprint of the 1979 revision, the only changes being that the MA28A specification sheet has been typeset and has a few minor changes and the existence of a version of the package for complex equations is announced.

Computer Science and Systems Division
AERE Harwell

November 1980

HL80/3459

CONTENTS

ISBN-0-70-580593-X

1.1 General introduction

The purpose of this report is to present a set of subroutines which implement a sparse variant of Gaussian elimination for the solution of unsymmetric sets of linear equations.

It is sometimes possible (particularly in network analysis or linear programming) to reorder the equations and variables so that the coefficient matrix is in block triangular form. We illustrate this by

$$P_1 A Q_1 = \begin{pmatrix} A_{11} & & & \\ A_{21} & A_{22} & & \\ A_{31} & \ddots & \ddots & \\ \vdots & & \ddots & \\ A_{N1} & \cdots\cdots & \cdots & A_{NN} \end{pmatrix} \qquad (1.1)$$

where A is the coefficient matrix of the system before reordering and P_1 and Q_1 are permutation matrices corresponding to the reordering of the equations and variables respectively. The matrices $A_{ii}(i=1,\ldots,N)$, henceforth called "diagonal blocks", are square and, by convention, cannot be further permuted to block triangular form. The "off-diagonal blocks" A_{ij} $(j<i)$ are rectangular and some of them can be zero matrices.

Our routines (see section 2.2) can perform this preordering. We consider this facility to be worthwhile because it is cheap and can greatly facilitate the solution process since we observe that the system

$$(P_1 A Q_1)\, \underline{x} = \underline{b} \qquad (1.2)$$

can be readily solved by partitioning $\underline{x}$ and $\underline{b}$ similarly to the block form

of (1.1). We then solve (1.2) by the block forward substitution

$$\left.\begin{aligned} A_{11}\,\underline{x}_1 &= \underline{b}_1 \\ A_{ii}\,\underline{x}_i &= \underline{b}_i - \sum_{k=1}^{i-1} A_{ik}\,\underline{x}_k \qquad i=2,3,\ldots,N \end{aligned}\right\} \tag{1.3}$$

Clearly, the matrices $A_{ii}(i=1,\ldots,N)$ can be considerably smaller than the complete system.

We expect the user to call the three subroutines MA28A/B/C. These subroutines manage the data and call other subroutines to perform the real work. The effect of calling MA28A/B/C is as follows.

1. MA28A finds permutation matrices P' and Q' such that

$$P'AQ'$$

is in block lower triangular form (see Figure (1.1)) and then performs Gaussian elimination within each diagonal block A_{kk} to obtain the factorizations

$$P_k A_{kk} Q_k = L_k U_k, \qquad k=1,2,\ldots,N \tag{1.4}$$

where L_k is unit lower triangular and U_k is upper triangular. If we combine the preordering permutations with those used in the decomposition on the diagonal blocks to form the permutation matrices P and Q, then we can express our decomposition as

$$PAQ = \begin{pmatrix} L_1U_1 & & & \\ A_{21} & L_2U_2 & & \\ A_{31} & A_{32} & \ddots & \\ \vdots & \vdots & & \\ A_{N1} & A_{N2} & \cdots\cdots & L_NU_N \end{pmatrix} \qquad (1.5)$$

It is possible to omit the first process and perform a full decomposition of the original matrix.

2. MA28B uses information from a previous call to MA28A to factorize a new matrix of similar sparsity structure.

and

3. MA28C solves sets of equations

$$A\underline{x} = \underline{b} \quad \text{or} \quad A^T\underline{x} = \underline{b}$$

using the factorization obtained by MA28A or MA28B.

It is expected that the user will find the code easier to use and more efficient than the previous Harwell subroutines, MA18A/B/C. In addition to having a more flexible user interface (see section 1.3) and being in a more portable subset of Fortran, these subroutines employ an entirely different data structure for the internal representation of the sparse matrix (see section 2.3). The block triangularization facility was not present in MA18 and these new routines also allow the user to factorize rectangular matrices and obtain a solution of a rectangular system. In such cases, the matrix must be made square by bordering it with a zero matrix (see the MA28 specification sheet).

Some details of the algorithms employed and their differences from MA18 (Curtis and Reid (1971)) are given by Duff and Reid (197). Here we concentrate on describing the subroutine organisation and code itself. The reader will find the paper by Duff and Reid (197) helpful though not essential to an understanding of this text.

The subroutines in the MA28 package (see section 5.2) are in IBM Fortran although special comment cards (with / as the last non-blank character in columns 2 to 72) have been inserted to indicate the statement for statement replacement necessary to produce a standard deck. Harwell subroutine OE04A automates this statement exchange and the standard Fortran code thus produced has been checked by the Bell Telephone Laboratories Fortran verifier (Ryder,1974). The main difference between IBM and standard Fortran codes lies in the use of INTEGER*2 arrays. If the reader is contemplating using the code on another machine range on which a short length integer is available, then he is strongly urged to consider replacing the INTEGER*2 declarations by the appropriate short integer. Great care has been taken in designing the subroutines so that the use of half-word integer arrays imposes a restriction on the order of the matrix and not on the number of non-zeros. The savings in storage by using such a technique is quite appreciable (see section 1.4). In order to make the code easier to read we have avoided any backward jumps by programming all loops using the DO statement. This additionally removes one possible cause of infinite looping during program testing and design and makes a complexity analysis much simpler. All DO loops longer than three or four statements end with a CONTINUE statement and CONTINUE statements are not used for any other purpose.

If anyone wishes to understand all the technicalities of the code, he can do so from comments embedded in the code itself. This report is

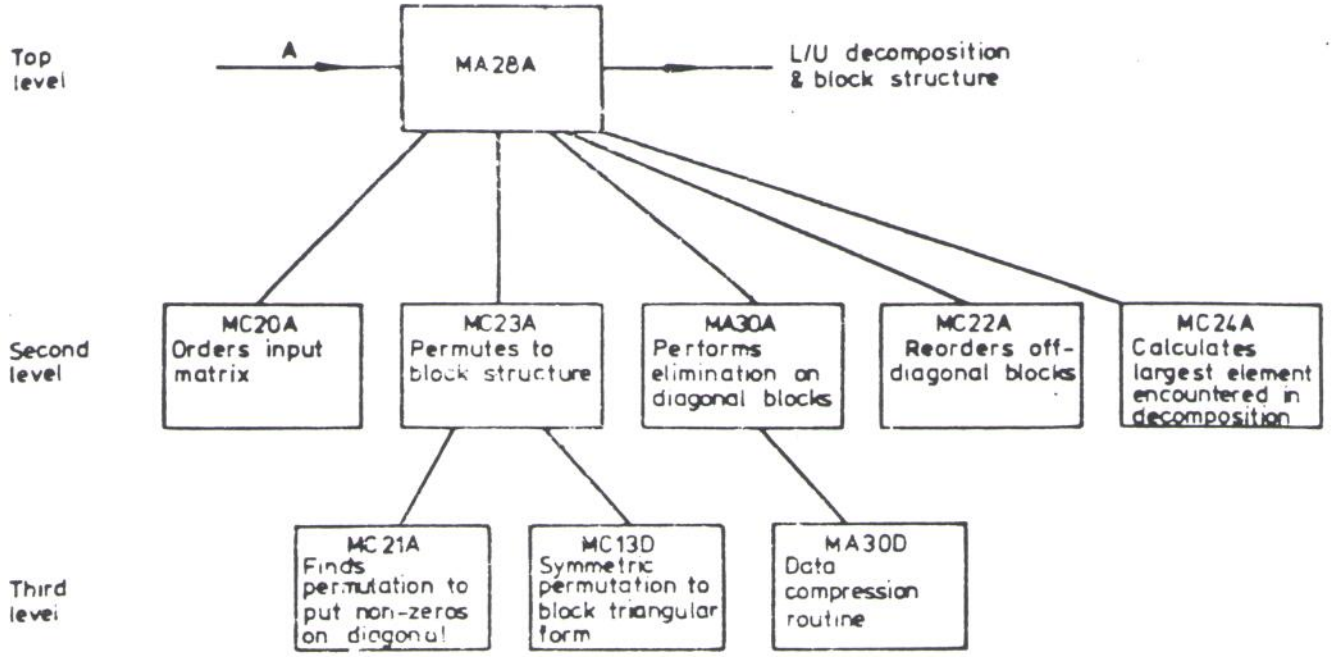
designed to satisfy the curiosity of the interested user and so, in the following text, we make general rather than detailed comments on the code. Although we only explicitly refer to the single length versions throughout this paper (except Table 2), the comments must be understood to apply equally to the double precision version of the subroutines.

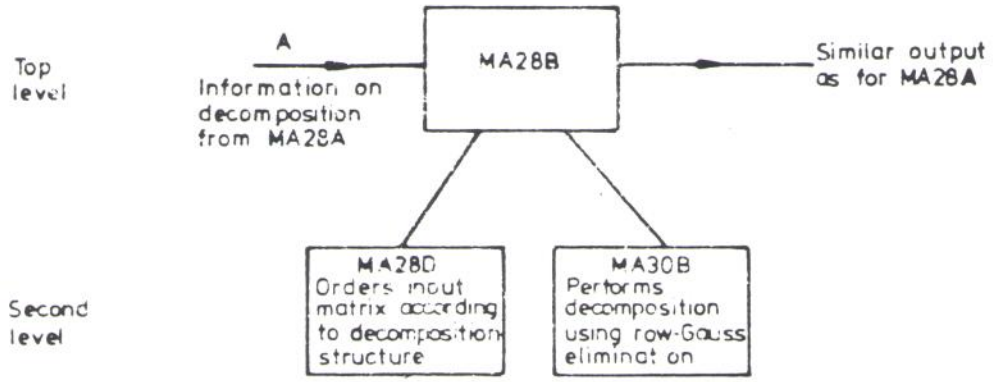
Because of the complexity of the total package (see section 1.4), we have divided this report into several sections with each basic driver routine and its dependents being described in a separate section. In the remainder of this section, we discuss the organisation and hierarchy of subroutines in the package, describe the user oriented data structure, give some details of the size of the code and its storage requirements, and provide a rough guide to the time and space required for practical problems. In sections 2,3 and 4 we describe the drivers MA28A,MA28B and MA28C respectively at the same time discussing any subroutines used by them. The final section, section 5, presents specification sheets for the subroutines involved in the package.

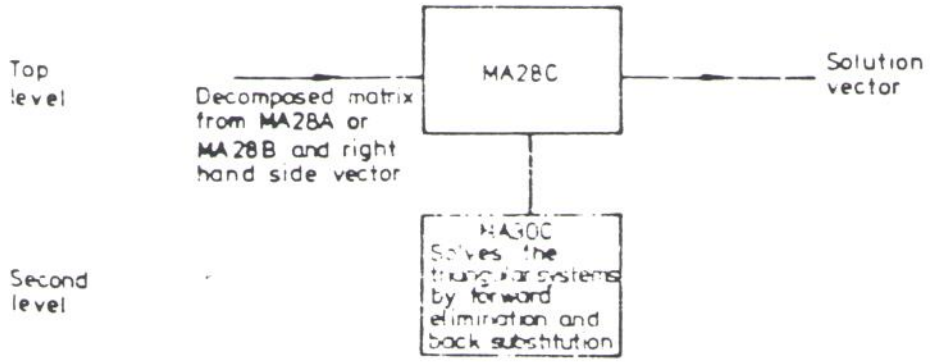
The author would like to thank J.K. Reid for reading a draft of the original report and making some helpful suggestions. Subroutine MA20A was written by A.W. Westerberg and J.K. Reid.

1.2 Subroutine organisation

The following diagram indicates the hierarchical interrelationships between the subroutines in the MA28 package. The diagram is really in three separate parts according to whether we are describing the inter-subroutine relationships in MA28A,MA28B or MA28C. The flow from left to right represents the sequence of calls in the code while flow down the page represents hierarchy with the top level being nearest to the top of the page. Note that this diagram can be used to aid in designing a program for overlay when appropriate (see section 1.4).


Top level
A
MA28A
L/U decomposition & block structure
Second level
MC20A Orders input matrix
MC23A Permutes to block structure
MA30A Performs elimination on diagonal blocks
MC22A Reorders off-diagonal blocks
MC24A Calculates largest element encountered in decomposition
Third level
MC21A Finds permutation to put non-zeros on diagonal
MC13D Symmetric permutation to block triangular form
MA30D Data compression routine


Top level
A
Information on decomposition from MA28A
MA28B
Similar output as for MA28A
Second level
MA28D Orders input matrix according to decomposition structure
MA30B Performs decomposition using row-Gauss elimination


Top level
Decomposed matrix from MA28A or MA28B and right hand side vector
MA28C
Solution vector
Second level
MA30C Solves the triangular systems by forward elimination and back substitution

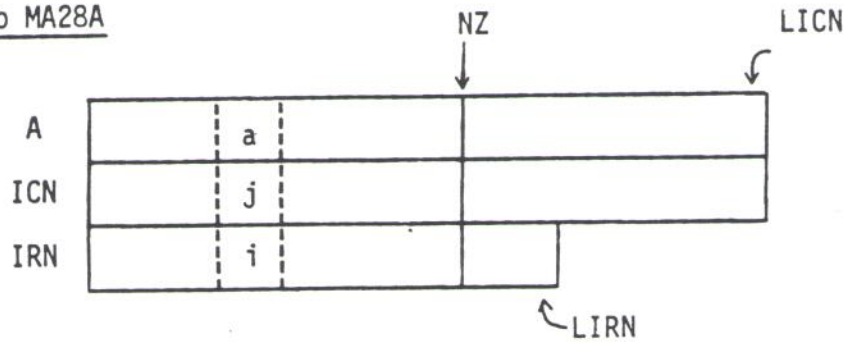
1.3 Top level data structure

The MA28 input data structure is designed for ease of use and is not compatible with efficient execution of the elimination itself. This means that data reorganisation is needed prior to calling the subroutines at the second and third levels which perform the real work.

For entry to MA28A and MA28B, the user need only specify the order of the matrix A and its non-zeros with their row and column indices. The elements may be in any order (see Figure 1). A check is performed on all row and column indices and, if any index is out of range, the routine will exit after completing this data check. However, the calculation will proceed if there is more than one element with the same row and column index. In such circumstances, the data entry causing duplication is identified to the user (although this identification can be switched off), its value is added to the sum of previous ones with the same indices and, on exit, a flag informs the user of such an occurrence. After this data check the subroutines must then reorder the data to facilitate the row and column access required by Gaussian elimination. This is further discussed in sections 2 and 3.

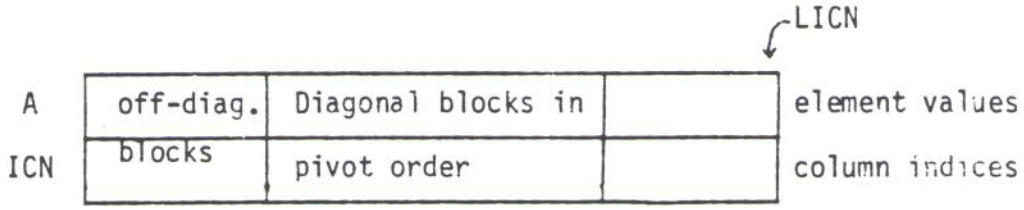
On output from MA28A or MA28B, the permuted and decomposed matrix is held by rows with the column indices in array ICN and the corresponding real values in A. Note that these are column indices in the permuted matrix PAQ. Any off-diagonal blocks precede the diagonal blocks, the rows being in pivotal order with no space within or between the rows (see Figure 1). The length of the part of the rows in the diagonal blocks, the off-diagonal blocks, and the lower triangular part of the decomposition are held in array IKEEP (positions 1 to n, 4n+1 to 5n, and 3n+1 to 4n respectively) which also holds the permutation arrays (row permutation in positions n+1 to 2n, column permutation in 2n+1 to 3n) and

Input to MA28A

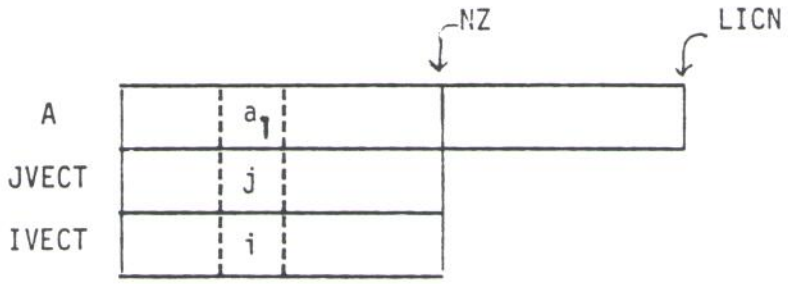


where element (i,j) of matrix has value a.

Output from MA28A (and MA28B)



User's input to MA28B (ICN,IKEEP input preserved from previous calls)



where element (i,j) of new matrix has value a_1.

Figure 1 Top level data structure

information on the block structure and matrix singularity (by negating elements of row and column permutations respectively).

On entry to MA28C, the user need only pass this information over unchanged (thus he need not concern himself with its structure) and input his right-hand side in unpermuted form in a real vector. The real vector yields the solution on exit, again in unpermuted form. Note that the user need not concern himself with the permutations although the factorization is actually held in permuted form.

1.4 Code lengths and storage requirements

This package is not particularly designed for use on small machines. Its 1795 Fortran statements require about 37K bytes of storage on our IBM 370/168 machine when compiled using the Fortran H compiler with OPT=2, although the structure shown in the figure in section 1.2 can be used for overlay with little loss of efficiency and a reduction in program storage (determined by the path MA28A,MC20A,MA30A and MA30D) to approximately 16.5K bytes. Comparable figures for MA18A are 15.5K bytes overlaying to 10.2K bytes.

Additionally, it is possible for the sophisticated user to design his own driver routine and hence possibly avoid some of the overheads inherent in a robust general purpose code. As a guide to the size and core requirements of the package, we present the appropriate figures for each subroutine in Table 1.

One of the slight drawbacks to a general purpose sparse matrix handler is the amount of integer overhead required in addition to the storage for the non-zero themselves. In Table 2, we give the storage required by each subroutine when handling a matrix of order n with NZ non-zeros. For comparison, the storage requirements for MA18 are given.

Subroutine	Number of cards in deck	Number of Fortran statements	Length of compiled code: H(OPT=2) on IBM 370/168 in bytes
MA28A	352	183	4072
MA28B	120	64	1742
MA28C	51	12	648
MA28D	131	107	2650
MA30A	794	689	11204
MA30B	120	95	1996
MA30C	229	153	3026
MA30D	46	36	716
MC23A	202	135	3118
MC20A	103	72	1546
MC21A	139	104	2574
MC13D	131	78	2144
MC22A	78	56	1304
MC24A	46	34	880
Total Package	2542	1818	37620

Table 1 Code lengths for subroutines in MA28 package

In Table 2, LICN must be as great as the number of non-zeros in the factors of the decomposition (often in the range 2 to 4 times NZ). LIRN must be as great as the maximum number of non-zeros in any active matrix (often need be only n or 2n greater than NZ). Although LIRN must be at least as great as NZ for the MA28A entry, it need only be as large as the maximum number of non-zeros in the active part of a diagonal block and so may be less than NZ for entry to MA30A. OFF is the number of non-zeros in the off-diagonal blocks and is always less than NZ. Any overheads independent of n or NZ have been omitted.

Subroutine	Real words	Integer words	Short integer words	Bytes: 8/real 4/integer 2/short integer	
				Permanent	Temporary
MA28AD	LICN+n*	Incl.in short words	LICN+LIRN+15n+	10n+10LICN	28n*+2LIRN
MA28BD	LICN+n	4n	LICN+2NZ+5n+	10n+10LICN	24n+4NZ
MA28CD	LICN+2n	-	LICN+5n+	18n+10LICN	8n
MA30AD++	LICN	2n	LICN+LIRN+10n	10n+10LICN	18n+2LIRN
MA30BD++	LICN+n	n	LICN+4n	8n+10LICN	12n
MA30CD++	LICN+2n	-	LICN+5n	18n+10LICN	8n
MC20AD	NZ	n	2 NZ	4n+10NZ	2NZ
MC23AD	NZ+n	2n	NZ+10n	8n+10NZ	28n
MC21A	-	n	NZ+7n	8n+2NZ	10n
MC13D	-	n	NZ+6n	10n+2NZ	6n
MC22AD	OFF	2n	2 OFF+3n	6n+10 OFF	8n+2 OFF
MC24AD	LICN+n	-	LICN+2n	4n+10LICN	8n
MA18AD	LICN	2 LICN	13n	26n+16LICN	Work arrays are not separated from main storage
MA18BD**	LICN+2n	-	LICN+5n	26n+10LICN	
MA18CD**	LICN	-	LICN+13n	26n+10LICN	

Table 2 Storage requirements for MA18 and MA28 (double precision versions)

*If an estimate of the largest element encountered during the decomposition is not requested on a call to MA28AD, then n less reals and 8n less bytes of temporary storage are needed.

+If the user is not using the block triangularization option, then n less short integers and 2n less bytes of permanent storage are needed.

**In MA18, the MA18CD entry is an entry point within subroutine MA18BD. Additionally, MA18CD performs a similar function to MA28BD(viz. FACTOR) while the operate function is performed by MA28CD and MA18BD.

++Strictly speaking, if block triangularization has been performed prior to calling these MA30 entries, then the size of most of the arrays can be based on the dimension of the largest block rather than the system as a whole.

1.5 A simple minded guide to time and space requirements

It must be stressed at the outset that any figures presented in this subsection are to be taken as a rough and ready guide rather than a hard and fast rule. Hopefully, they will be of some assistance to the MA28 user but they should in no way override the healthy intuition of a sophisticated or experienced user.

Let us assume that the coefficient matrix is of order n and has NZ non-zeros. Then, in the absence of other information, LICN might be set to 3NZ and LIRN to NZ+2n, if space permits. Over a wide variety of matrices, the number of arithmetic operations performed has been empirically observed to be about $\tau^2/4n$ where τ is the number of non-zeros in the decomposed form.

To estimate the time required for the three phases of ANALYZE (MA28A), FACTOR (MA28B), and OPERATE (MA28C) we again refer to empirical results. In each case, there is a wide variation in figures but, as a rough guide, we can suggest values of 25 and 10 microseconds per arithmetic operation for ANALYZE and FACTOR respectively (on our IBM 370/168), while the time for OPERATE is clearly proportional to τ and is, on average, 4 microseconds per decomposed matrix non-zero. Our experimentally based results are summarised in Table 3.

	ANALYZE (MA28A)	FACTOR (MA28B)	OPERATE (MA28C)
Storage (bytes)	42n + 32NZ	34n + 34NZ	26n + 30NZ
Time (micro-seconds)	$25/4\ \tau^2/n$	$5/2\ \tau^2/n$	4τ

Table 3 Rough estimate of storage and time (on the IBM 370/168) requirements for a matrix of order n, with NZ and τ non-zeros in the original matrix and its decomposed form respectively.

Notice that our storage figures assume the values for LICN,LIRN mentioned above. A more detailed estimate of storage requirements can be obtained from Table 2. Also, the value of τ is generally not known a priori. Experience has shown that a value 5/2 NZ is a satisfactory estimate for τ although an estimate of order nlogn is more realistic for problems arising from PDEs in 2 dimensions.

2.1 Subroutine MA28A

Subroutine MA28A is a data management driver routine which calls other subroutines to perform the work of block triangularization and decomposition.

On entry to the subroutine the elements of the matrix together with their row and column indices (arrays A(I), IRN(I),ICN(I),I=1,...,NZ) can be in any order (see section 1.3). Although this is a very convenient user interface, it is not suitable for performing Gaussian elimination where we will require access to the matrix by both rows and columns. After an initial check on the input data, we therefore use MC20A, whose specification sheet is included in section 5, to reorder the matrix by rows. The output from this subroutine (a description of which can be found at the end of its specification sheet) consists of the matrix non-zeros and their column indices held in corresponding positions of a real array A and an integer array ICN. The entries in the same row are contiguous and there is no wasted space between the rows. The entries corresponding to row I are held in positions IW(I) to IW(I+1)-1 for I=1,....N-1 and in positions IW(N) to NZ for row N. The subroutine then checks for any duplicate entries, summing the values of any such elements and adjusts the arrays and pointers IW accordingly. This pass through the matrix is also used to calculate the value of the maximum modulus of the matrix entries. This will be used later to calculate a bound for the growth in the size of matrix elements during decomposition.

Control then passes to MC23A (see section 2.2) which determines a permutation to block lower triangular form and permutes the matrix accordingly before passing to MA30A (see section 2.3) for the decomposition phase.

Alternatively, if the logical variable LBLOCK was set by the user to .FALSE. , then no block triangular form will be computed although the matrix information must be moved to the end of arrays A and ICN and the permutations set to the identity before entry to MA30A.

After decomposition using MA30A (described in section 2.3), the part of the matrix corresponding to the diagonal blocks, which may be the whole matrix, is held in pivotal order by rows. For each row the elements of L_K (see equation 1.4) are in pivotal order and precede the elements of U_K which can be in any order except that the pivot comes first. The column indices of all these elements are those of the permuted matrix, so that the column index of the pivot in row I is I, I=1,...,N. However, the off-diagonal blocks are quite unaltered by MA30A and so before proceeding to use the decomposition to solve sets of equations we must first permute the rows of the off-diagonal blocks and change the column indices to those of the permuted matrix. This is done by MC22A which is a simplified form of MC20A. Its action is adequately described at the end of its specification sheet (see section 5).

Finally, unless the user sets the logical variable GROW to .FALSE. , the routine will use MC24A to obtain a bound on the largest element encountered during LU decomposition. When this bound is added to the largest element modulus of the original matrix then it gives an estimate (an upper bound) for the value of a in equation

$$|e_{ij}| \leq 3.01.a.\varepsilon.m_{ij}$$

where

ε is the machine precision

m_{ij} is the number of operations on position (i,j) during the decomposition

and

e_{ij} is the $(i,j)^{th}$ element of E where

$$A+E = LU$$

a slight extension of a result which Reid (1971) has shown to hold when the multipliers during the elimination are not constrained to be less than 1.

The reason for calculating an estimate for a in this manner is that we can avoid an extra overhead in the innermost loop of the calculation and thus increase the speed of our code (more apparent in MA28B, of course, see Table 11 in section 3.1). MC24A calculates this bound by using Hölder's generalization of Schwarz's inequality on the relation

$$a_{ij}^{(k)} = a_{ij} - \sum_{m=1}^{k} \ell_{im} u_{mj} \qquad k<i\leq n, \quad k<j\leq n \ .$$

This was discussed by Erisman and Reid (1974).

Notice that this argument and that for the numerical criterion for selection of pivots in section 2.3, implicitly assumes that the elements of the original matrix do not differ too widely in magnitude. Although we believe that the "best" scaling can often be performed by the person with a knowledge of the original problem and so do not incorporate scaling procedures within MA28, we give an example in the specification sheet for this subroutine (see section 5) of how one might use the scaling routine MC19A with this package.

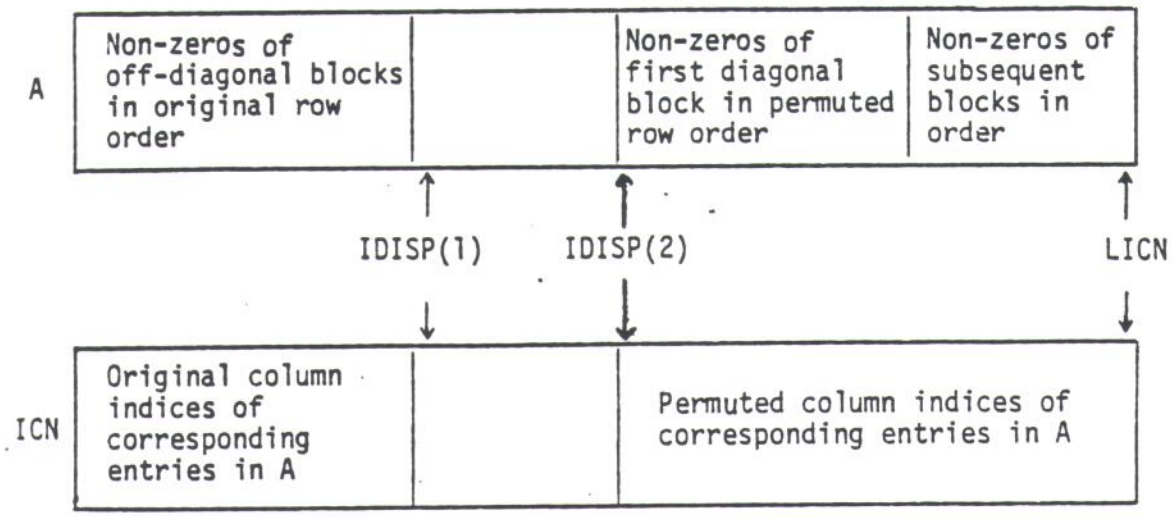
2.2 MC23A ... the block triangularization phase

Subroutine MC23A is, like MA28A, to a large extent a data handling routine. The real work in computing the permutation is performed by MC21A and MC13D.

MC21A described by Duff (1978) calculates a row permutation (held in array IP) which, when performed on the original matrix, ensures that the new diagonal is zero-free. Elements (IP(I),I) will be non-zero unless the matrix is structurally singular in which case N-NUMNZ of them will be zero. The pointers to the beginning of the rows (IW(.,1)) and the row lengths (LENOFF) are then explicitly permuted (the permutations being held in IW(,2) and LENR, respectively) before entry to MC13D. This subroutine, described by Duff and Reid (1978a,1978b), then finds a symmetric permutation (held in IQ) to permute the matrix to block lower triangular form. Duff (1977) justifies this two-stage approach and, in particular, proves that the final block form is essentially independent of the particular row permutation which made the diagonal zero-free. The remainder of subroutine MC23A then merely reorganises and reorders the matrix according to the combined row permutations (now held in IP) and the column permutation IQ.

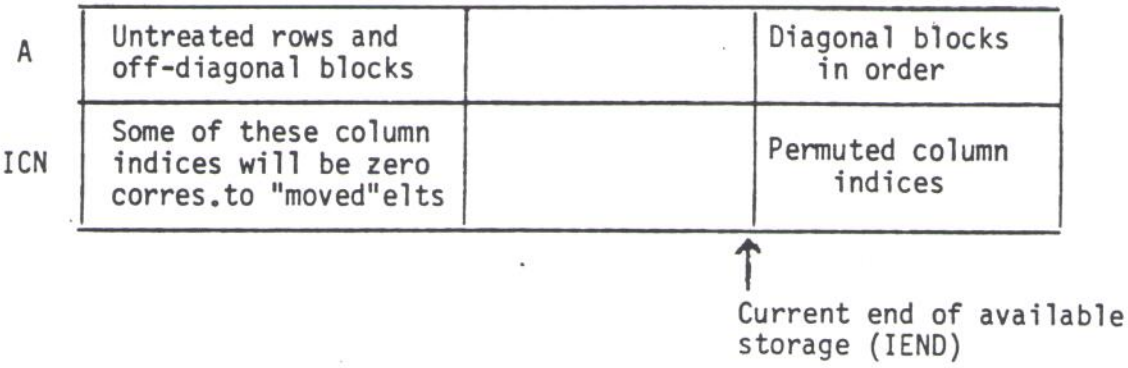
On entry to MC23A, the non-zero values of the matrix occupy the first positions in A and ICN while, on entry to MA30A, the diagonal blocks should occupy the last positions (see section 2.3). If there is only one block, not only are the two permutations set to the identity but the NZ non-zeros are moved to the last NZ positions in A and ICN before exiting from the subroutine.

Otherwise the reordering of arrays A and ICN is performed so that the output is in the form



Note that the off-diagonal blocks remain unpermuted while the diagonal blocks are permuted. This is done so that after the decomposition of the diagonal blocks, the off-diagonal blocks can easily be permuted according to the pivot permutation and additionally only one single permutation (of the original matrix) need be held at any stage. It is worthwhile to describe this reordering in some detail.

Each row is treated in turn working through the permuted matrix in the reverse order. The row is scanned also in reverse order and any columns corresponding to entries lying in the diagonal blocks are placed at the current end of available storage, their column indices changed according to the permutation and their present position set to zero to indicate that the space is now freed. Any time an off-diagonal block entry is encountered we leave it in position. Counts of the number of non-zeros in the diagonal and off-diagonal parts of the row are kept. At an intermediate state, the storage allocation is as follows:



If the current end of available storage comes too close to the storage allocated to the untreated rows and off-diagonal blocks, then danger of overwriting occurs which is avoided by simply moving information towards the beginning of the arrays overwriting any entries with zero column index (called a "compression"). The pointers to the beginning of each row have then to be adjusted since we require access to intermediate rows (in the original ordering) during the main part of the algorithm.

Finally, after all the entries in diagonal blocks have been separated and placed at the end of the storage, a final compression-like pass is required to remove all the copied diagonal block information (identified by zero column indices), which has not previously been compressed, from the off-diagonal storage.

For this separation routine to succeed, LICN (the length of arrays A and ICN) need only be of length RMAX+NZ where RMAX is the maximum number of non-zeros in a row and NZ the initial number of non-zeros. However, increasing the storage allocated should decrease the number of compresses required and thus decrease the running time as is illustrated by the results in Table 4.

Order of matrix Number of non-zeros	199 701	363 2454	822 4841	Total over 32 different matrices
LICN equal to RMAX+NZ	110	180	1540	16340
LICN equal to 2n+NZ	40	110	310	3890
LICN equal to 10000	30	80	250	3190

Table 4 Some timings (in milliseconds on an IBM 370/168) of MC23A with varying storage allocations

2.3 MA30A ...the ANALYZE phase

Subroutine MA30A is the kernel of the MA28 package. It performs a sparse variant of Gaussian elimination choosing pivots from either the entire matrix or pivoting and eliminating only within externally defined diagonal blocks. We first explain how the subroutine identifies the block structure and then examine in detail the main loop of the algorithm which performs the decomposition within the blocks.

The matrix is input to MA30A in the same structure as was output from MC23A (see previous section). The block structure is recognised by negative values in the ordering array IP indicating the last row of each block (unnecessary for the last block). The length of each row must also be supplied. MA30A first calculates the position of the beginning of each row (necessary because they are not accessed in order) and counts the number of blocks. Then the decomposition proceeds on each block in turn.

Within this main loop, the subroutine first examines the structure of the block before performing the decomposition within an inner loop. Firstly, the order of the block is calculated and, if this is 1, special simple action can be taken (really just moving one element to the appropriate part of the storage) and we proceed with the next block. For a larger block, we obtain the pattern of non-zeros by columns in the following manner. We first calculate the number of non-zeros in each column and then generate an array of pointers IPC such that IPC(I) gives the address in the proposed column file array (IRN) of the first position after the last non-zero in column I. We then scan the part of the row file corresponding to the current block, placing each element in the appropriate position in the column file and decrementing the corresponding IPC pointer by one. Thus, the column orientation and column counts are obtained after only two passes through the block in a number of operations proportional to n and τ, where n is the order of the block and τ its number of non-zeros. We then generate doubly linked lists in preparation for the pivot selection. These lists are described further in the following paragraph.

We use the pivotal strategy of Markowitz (1957) with the additional constraint that no pivot be less than u times the largest element in the active part of its row (i.e. U_K part of row only). The quantity u is a user set parameter which we will discuss at greater length in the next paragraph. Markowitz's strategy is to select as pivot that non-zero for which the product of the number of other non-zeros in its row and its column is minimized. If the search for this non-zero is done crudely, we might examine every non-zero of the active matrix at each stage, giving a most unsatisfactory $O(n\tau)$ operation count. To facilitate the search for such an element, we hold doubly linked lists (i.e. with backward and forward pointers) of all rows and columns having the same number of non-

zeros. A simple implementation of this results in 6 vectors of length n (3 each for rows and columns) but we can reduce this to 5 by utilising only one array to point to the beginning of both lists and by chaining together the lists for rows and columns. Thus we have

IFIRST: If there are any rows with I non-zeros, then IFIRST(I) is the row index of the first row in the chain; otherwise if there are any columns with I non-zeros, then -IFIRST(I) is the column index of the first column in the chain. If there are no rows or columns with I non-zeros IFIRST(I) is zero.

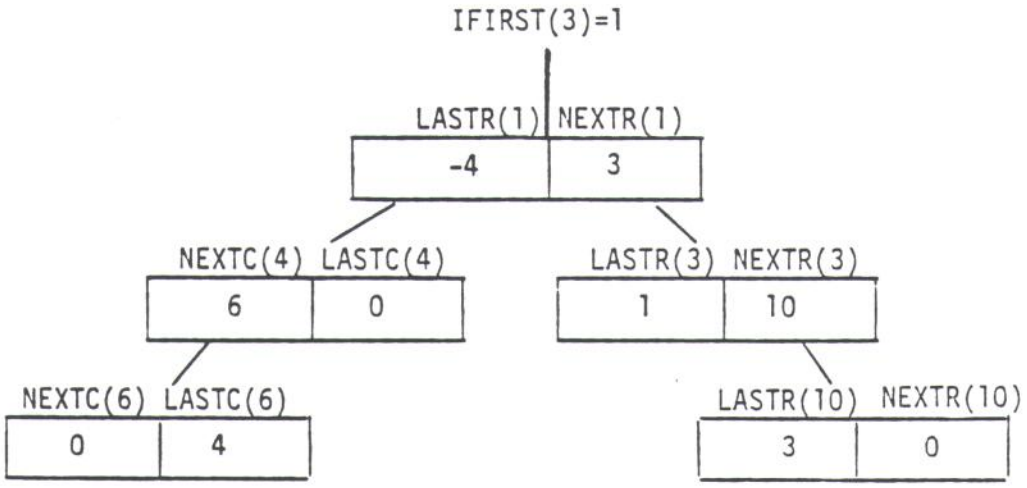
NEXTR: NEXTR(I) is the row index of the next row in the chain with the same number of non-zeros as row I, or is zero at the end of the chain.

LASTR: LASTR(I) is the row index of the previous row in the chain with the same number of non-zeros as row I. For the first row in the chain -LASTR(I) is the index of the first column with the same number of non-zeros as row I and is zero if no such columns exist.

NEXTC: NEXTC(I) is the column index of the next column in the chain with the same number of non-zeros as column I, or is zero at the end of the chain.

LASTC: LASTC(I) is the column index of the previous column in the chain with the same number of non-zeros as column I, or is zero at the beginning of the chain.

Thus, if rows 1,3,10 and columns 4,6 all have 3 non-zeros we can represent the above data structure in the following manner:



These arrays enable us to search the rows and columns in increasing number of non-zeros stopping whenever we know that there cannot be a non-zero later in the sequence with a better Markowitz count than the present potential pivot. Normally, we will not have to search many rows and columns as is illustrated by the results in Table 5. Once we have selected the pivot, we remove all rows with non-zeros in the pivot column and all columns with non-zeros in the pivot row from the linked lists because the number of non-zeros in these rows and columns will normally be altered. After the elimination, these rows and columns (with the exception of the pivot row and column itself) are reinserted in the lists in positions corresponding to their new numbers of non-zeros.

Order of matrix	147	57	199	292
Number of non-zeros	2449	281	701	2208
Number of rows/columns searched for possible pivot	1199	158	2879	895
Average length of search	8.2	2.8	14.5	3.1

Table 5 Profiler counts on MA30A indicating amount of searching to find pivot

In addition to choosing our pivot to minimize the Markowitz count, we must also test it for stability by comparing its magnitude with those in its row in columns which have not yet been pivotal (the active part of the row). We use the acceptance test

$$|\text{POSSIBLE PIVOT}| > u. \max_{\text{active row}} |\text{ELT.IN ROW}|$$

with strict inequality because when handling singular matrices we may get a row of all zeros and we wish to recognize this fact. Notice that with the test in this form, u cannot have a value of 1.0 since otherwise we would be unable to choose an element in a singleton row as pivot. For this reason, values of u greater than or equal to 1.0 are reset to .9999 (.99999999 in double precision). A value of u near to 1.0 will bias the algorithm towards partial pivoting with its consequent good stability properties although the fill-in (and hence the size of LICN) may then be very high. A value of u equal to 0.0 will cause the pivots to be chosen on the sparsity criterion alone with danger of severe numerical instability. We have found, in practice, that a compromise value of u in the range 0.1 to 0.25 preserves the sparsity while still providing a numerically satisfactory decomposition. Because of the facility for handling rectangular systems, we must also test elements in singleton columns for stability (such an element must be chosen in the square case) and have observed this necessity in practice. For example, on a consistent rectangular system of dimensions 313x176, a relative residual of 10^{6} was reduced to 10^{-10} when such a test was employed. Since we are not storing the values of the non-zeros by columns, the search of a column for possible pivots can be quite expensive since for each non-zero a row scan is required to test for stability. We attempt to reduce this work by biasing our pivot selection to testing possible pivots during a row scan. This is easily achieved through using the structure of row and column ordering vectors described earlier in this section. Each time we examine rows and columns with I non-zeros we first examine the rows (through IFIRST, then NEXTR) before examining the columns. The strength of this bias is illustrated in Table 6.

Order of matrix	147	57	199	292
Number of non-zeros	2449	281	701	2208
Number of rows examined for possible pivot	790	131	2554	548
Number of columns examined for possible pivot	409	27	325	347

Table 6 Profiler counts illustrating bias towards searching rows for pivots.

Nevertheless, the loop for checking the stability of potential pivots encountered in a column scan still has one of the largest execution counts (see Table 7). Fortunately, this loop contains only two Fortran statements.

Order of matrix	147	199	292
Number of non-zeros	2449	701	2208
Number of elimination operations of type $a_{ij} \leftarrow a_{ij} + a_{ik} a_{kk}^{-1} a_{kj}$	63722	2772	25864
Stability check for pivots (during column scan)	85946	2001	7267
Scan of non-pivot rows in elimination step	86952	5204	38955
Scan of non-pivot rows to locate multiplier	40013	2780	18495

Table 7 Some of the highest profiler counts for MA30A

It is reassuring to note that one of the major counts is that of the number of arithmetic operations which is a measure of the actual work which must be done to decompose the system. However, it is worthwhile noting that it may be significantly more efficient when decomposing an mxn matrix A (with m>n) to perform the factorization on A and not A^T since we wish our rows rather than our columns to have few non-zeros.

The acceptance test and the extra loop in which the Markowitz pivot selection procedure is embedded ensure that for any singular block non-zeros in the successive active matrices are first chosen as pivots , then explicitly held zeros, and finally when the active matrix is entirely null the two ordering arrays for the present block are completed, any parts of the unpivoted rows in already pivoted columns (these exist for overdetermined systems, for example) are moved to the appropriate part of storage, and the decomposition proceeds with the next block. Notice that since the active matrix must all be zero when such singularities are found no numerical operations need be done until the next block. It is also of interest to note that if a call to MC23A preceded this call, then all the structural singularity of the original matrix will appear as structurally singular blocks of dimension 1.

There are two basic ways of adding one sparse vector (which in our algorithm will be a multiple of the pivot row) to another. Since this operation is at the innermost loop of our code, it is extremely important to do this efficiently. The first method relies on having the column indices in sequence. We then scan the two rows simultaneously taking appropriate action on encountering equal or unequal indices. We reject this because of the overheads required to keep the column indices in order and the number of comparisons in the inner loop. These overheads are illustrated by the profiler counts in Table 8. The difference between the

Order of matrix	147	199	292	822
Number of non-zeros	2449	701	2208	4841
Number of elimination operations of the form $a_{ij} \leftarrow a_{ij} + a_{ik} a_{kk}^{-1} a_{kj}$	51316	2535	28097	6266
Number of comparisons to keep row/columns in order	102993	9557	72329	79687
Number of comparisons when adding sparse-vectors	71306	3805	39316	16398

Table 8 Profiler counts indicating overheads of keeping columns in order in MA18A

number of arithmetic operations for runs on identical matrices in Tables 7 and 8 is caused by the different tie-breaking strategies used by MA18A and MA28A when determining the non-zero with lowest Markowitz count.

To overcome these objections we could first load the pivot row into a full vector w, previously set to zero, so that our algorithm becomes:

(i) scan the pivot row i setting $w_j = a_{ij}$ for each non-zero

(ii) scan the non-pivot row k making the modification $a'_{kj} = a_{kj} - \alpha w_j$ (if α is the current multipler) and setting $w_j = 0$

(iii) scan the pivot row again looking for non-zeros a_{ij} for which $w_j \neq 0$; for each we have a fill and add $(-\alpha w_j)$ to the set of non-zeros in the non-pivot row, then setting $w_j = 0$.

In fact the double scan of the pivot row can be avoided except for the first non-pivotal row by restoring w to the expanded pivotal row instead of the zero vector in step (iii). However, although this avoids the

necessity for keeping column indices in order and can greatly reduce the number of comparisons, it is still unsatisfactory for the following reason. At stage (iii), a "non-zero" in the pivot row which happened to have the value zero would not be recognised and could not cause any fill-in. While this does not create any problems in MA28A, the structure of the decomposition could be altered thus causing a failure in a subsequent entry to MA28B for which the previously numerically zero element in the pivot row is now non-zero. For this reason, the following ruse is adopted.

When a pivot has been selected, the non-zeros and column indices of the pivot row are first moved to the beginning of the available storage but the original now redundant copy of this row is not all immediately freed. We use the entries in the active part of this row (excluding the pivot) and the integer array IQ of length N, known to have all its entries positive, in the following way

(i) scan the active part of the pivot row; if its p^{th} non-zero is in column j then we store IQ(j) in the position of the corresponding column index in the now redundant pivot row and reset IQ(J) to the value -p

then for each non-pivot row we

(ii) scan the non-pivot row; we make no change to a non-zero in column j if IQ(J)>0 but if IQ(j)<0 we can use its value to find the appropriate real element of the pivot row and after performing the required operation we change the sign of IQ(j);

(iii) scan the pivot row again looking for non-zeros a_{ij} for which IQ(j)<0; for each we have a fill-in whose real value can be calculated. If for any non-zero a_{ij} in the pivot row IQ(j)>0, then its sign is changed so that, at the end of this step IQ has been reset as in (i) ready for operations on the next non-pivotal row.

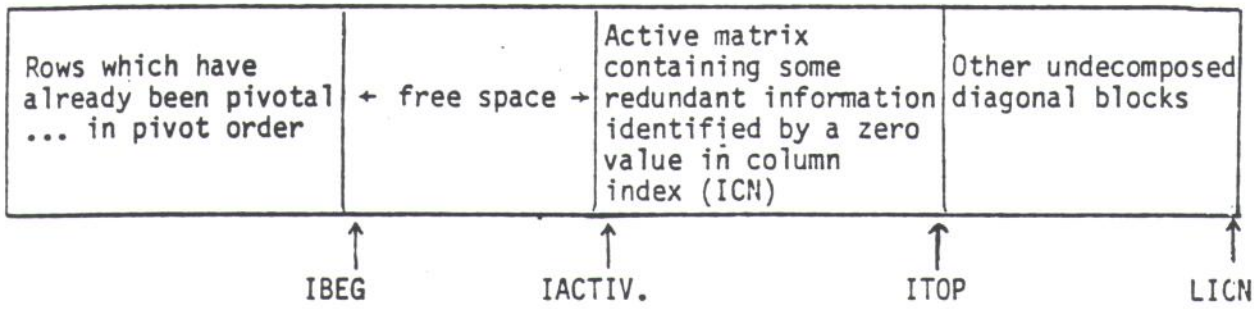
At the end of the pivot step it is an easy matter to reset the values of IQ by a final scan of the pivot row.

We obtain one other useful benefit when we use these methods of adding two sparse vectors. At the beginning of stage (iii), when we have modified elements in the non-pivot row we can tell immediately how many fill-ins there will be to that row. This enables us to examine positions surrounding the present position of the non-pivot row in arrays A and ICN to see if the fill-in can be accommodated by putting the fill-in elements immediately after the current row or by moving the row slightly forward to enable this to be done. In this way, as the counts in Table 9 illustrate, we often avoid having to create a new copy of a row at the beginning of our data structure.

Order of matrix	147	199	292	822
Number of non-zeros	2449	701	2208	4841
Number of times there is fill-in to non-pivot row	545	442	1010	914
Number of times no move is required at all	308	253	598	561
Number of times row is moved only locally	108	88	200	201
Number of times it must be moved to beginning of active block	129	101	212	152

Table 9 Profiler statistics on fill-in to non-pivot rows.

At an intermediate stage in the calculation, the data structure is:



Thus any time that a row must be moved to locations just prior to IACTIV because of fill-in the "free space" available to hold the decomposed matrix is reduced. When there is not enough "free space" to accommodate a row that has just been pivotal or a non-pivot row which must be moved, we recover some space from the active matrix by compressing the structure by overwriting the redundant information. This is done by MA30D/DD in one pass as follows. First a pass through the pointers to the beginnings of each row in the active part of the matrix enables these positions to be marked by overwriting the column index by the negation of the row index after storing the column index in the pointer array. We then scan the active block, starting from ITOP, working backwards and overwriting any redundant information thus compressing the data to the upper end of the storage. Whenever a negative "column index" is found we obtain the actual column index from the pointer array and reset that entry to its new value.

A similar strategy and compression technique is used on the arrays containing the row indices of the active matrix by columns. However, here we do not explore the possibility of keeping filled-in columns "in-place" quite so exhaustively. This is because the array of row indices (IRN) will not grow in a fashion similar to the column index file since only the active part of the matrix is stored at each stage. Some statistics related to these "compressions" are given in Table 10.

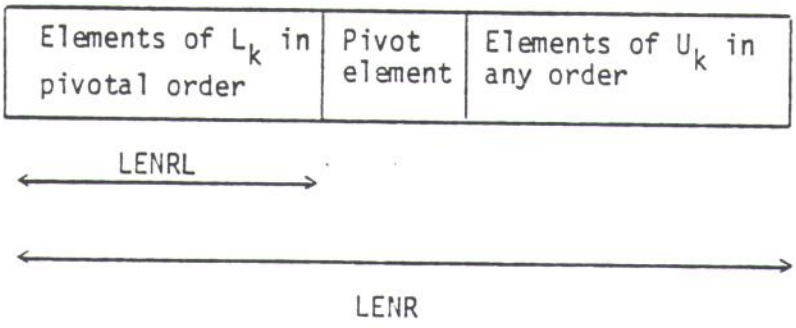
Order of matrix	147	199	292	822
Number of non-zeros	2449	701	2208	4841
Space required for row file	5812	1528	5747	6681
Space required for column file (active matrix)	2607	701	2208	4841
Compresses when run at minimum space — row: file	261	37	66	53
Compresses when run at minimum space — column: file	66	8	32	1
Time when run with minimum space	5260	380	1540	2170
Compresses when space increased by n (2n for row file) from minimum — row: file	100	3	10	3
Compresses when space increased by n (2n for row file) from minimum — column: file	18	3	15	1
Time for these runs	3000	260	1040	1790
Time for row file length 10,000) col.file length 5,000)	1540	240	830	1760

Table 10 Some statistics on compression. (Time in milliseconds on an IBM 370/168)

It is, of course, possible that even after a compression there is insufficient space to hold the "moved" row or column. We must abandon the algorithm if this occurs in the file of row indices (IRN) (with error flag -3 or -6) but we have an extra option if this happens in the main file (ICN). Here we can (IF ABORT3 has been left at its default value of .FALSE.) overwrite the already computed pivot rows (by setting IBEG=1). This will usually enable us to complete the decomposition. Although we lose some of the factors by doing this and thus cannot continue

with MA30C or MA30B, we do now know exactly how much space would be required for a successful decomposition. We envisage this feature as being very useful in cases where the user has given too low a value for LICN on his initial run.

During the decomposition, we retain the column indices of the ordering of the matrix on entry to MA30A/AD. As each multiplier is calculated (element of an L_k) it is swopped with the element in the row immediately after all previously calculated multipliers and when a row becomes pivotal the pivot element is similarly swopped. Thus, after a pivot row has been moved to the front of the storage, it is held as



but the column indices held are not those of the pivot order. This permutation would be impossible to do, since, at the time the row is moved, we do not know the appropriate permutations to apply to the elements of U_k. Therefore, after the decomposition has been completed, we replace all column indices by their permuted values (thus the pivot in row I has column index I and the column indices of the lower triangular part of the row are in ascending order), update our global permutation arrays IP,IQ by the pivot permutation held in LASTR and LASTC respectively, and permute LENR and LENRL accordingly. We also record zero pivots by negating corresponding entries in the IQ array before exiting from the subroutine.

3.1 Subroutine MA28B

MA28B is intended principally for the factorization of a matrix whose sparsity pattern is identical to that decomposed by a previous call to MA28A, although we will see that MA28B can handle a slightly extended set of matrices. In addition to the new matrix we require information on the structure of the decomposition and the pivotal ordering from that earlier call to MA28A.

The matrix is input the same way as for the MA28A entry and similar checks are performed on the validity of the user's data. The remainder of the subroutine is in three parts. First a call to MA28D performs a reordering of the input matrix (next paragraph), then the matrix is factorized by MA30B (section 3.2) and finally MC24A calculates a bound for the growth of element size during decomposition in the same fashion as for the MA28A entry (section 2.1).

The reordering performed by MA28D must be considered an integral part of our FACTOR entry and indeed as much time is sometimes spent in this reordering as in the numerical factorization itself (see Table 11). The logic of the main loop in this subroutine is very similar to that of subroutine MC20A. On each pass through the loop one non-zero of the input matrix (the current element) is placed in its correct position in the A/ICN array. (Note that no change is made to the ICN array). If this would overwrite any unprocessed input element in A then the next pass has this non-zero as its current element. At each pass, the current element is tested to see whether its indices are within range and whether it lies in a legitimate block of the block triangular permutation. If it is in the off-diagonal blocks or the U_K part of the diagonal blocks, then its true position is found by a linear search of the appropriate row. However, since the column indices of the L_K part of each diagonal block are in order, a binary search suffices to find the position of each non-zero in that part of the matrix. Once all the non-zeros of the input matrix have been processed,

a final pass through A and ICN sets the numerical value of any unaccessed entry to zero (in A) and calculates the largest element of the input matrix.

Notice that we do not require the input matrix to have the same sparsity as the matrix originally decomposed but only that all of its non-zeros are also non-zeros of the final factored form. Additionally, since only the non-zero element values in A are moved, the indexing arrays are unchanged on exit from this subroutine.

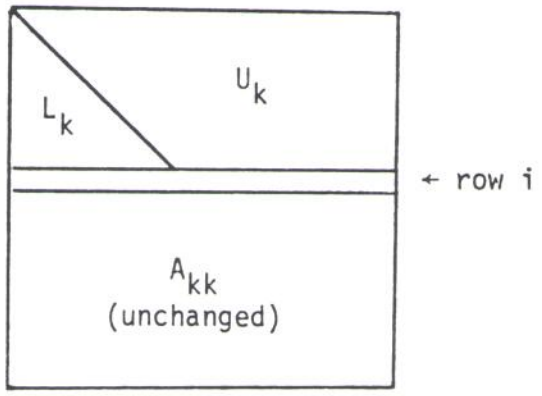
Order of matrix	147	199	292	822
Number of non-zeros	2449	701	2208	4841
Time to preorder matrix (MA28D)	70	20	70	120
Time to factorize (MA30B)	210	30	140	60
Time to factorize (with calculation of growth in inner loop)	290	40	180	70
Time to calculate growth in element size using MC24A	13	3	13	17

Table 11 Some statistics on MA28B (times in milliseconds on an IBM 370/168)

One important feature is that, in the event of failure due to duplicate elements or non-zeros outside the legitimate sparsity structure, the information returned to the user refers to the original row and column indices of his input matrix.

3.2 MA30B...the FACTOR phase

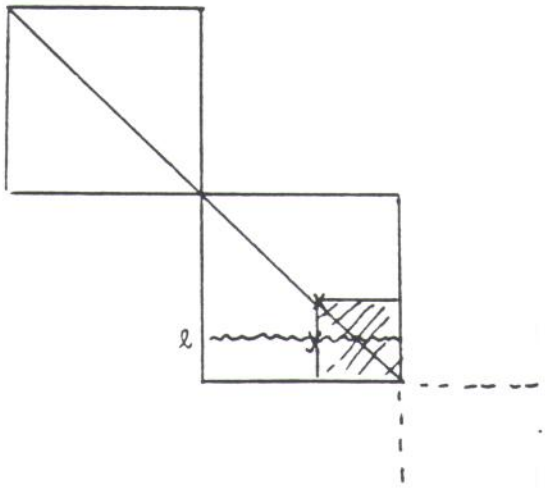
The task of the FACTOR subroutine MA30B is much simpler than that of MA30A, since here we already know the pivot sequence and the structure of the LU decomposition. Here we use the decomposition sometimes referred to as row Gauss elimination. We decompose each diagonal block in turn and at stage I in the decomposition, block k has the form



and during this stage row i is transformed from a row of A_{kk} into a row of L_k and U_k. This transformation is done in the following way. First row i of A_{kk} is expanded into a full vector W upon which all arithmetic operations are subsequently performed. The sparsity structure of row i of L_k is then examined in order to identify which previous rows modify row i. The modifications are then performed on W and after all have been completed it only remains for us to transfer the information

back into the sparse data structure by a final pass through the sparsity structure for L_k and U_k (held in ICN).

An added complication in the MA30B code lies in detecting singularities. We wish MA30B to decompose a singular matrix if and only if the singularities occur in the same place as they did during the previous decomposition by MA30A. However, we wish to remove such checks from the inner loop of the code. If, at stage i, no singularity had been found by MA30A, we merely check to see if the pivot is zero and then optionally compute the ratio of the pivot to the largest element in the same row of U_k. If, however, there was a singularity found by MA30A at this stage, then we must examine row i to ensure that no non-zeros are present in the part of the matrix which should be zero or null. For example, in the figure



if the pivots from the position of the x to the end of the block were zero (or null) in the MA30A entry then the shaded region must contain no non-zeros in the MA30B entry.

To allow for the processing of rows like row ℓ in the figure above, when the zero element y is held explicitly (x is also zero), zero pivots are set to one when first encountered and are then reset to zero on completion of the decomposition of the block.

4.1 Subroutine MA28C

Subroutine MA28C is merely used as an encapsulating interface to MA30C. This embedding serves to facilitate the user interface by achieving compatability with the calls to MA28A and MA28B and by shielding the user from the sub-division of workspace. Since all the work is done by MA30C, we now discuss this subroutine in subsection 4.2.

4.2 MA30C...the OPERATE phase

MA30C is effectively two separate routines. The first half solves $A\underline{x}=\underline{b}$ given the decomposition of A by MA30A or MA30A and MA30B, the second half solves $A^T\underline{x}=\underline{b}$ given the same decomposition.

We will now discuss the $A\underline{x}=\underline{b}$ entry (MTYPE=1) before making a few comments on the second half of the subroutine. We hold the decomposition shown in figure 1.5 where the permutations corresponding to P and Q are held in arrays IP and IQ, respectively. We first perform the permutation P on the input vector and then perform the forward elimination corresponding to the rows of L_1 in the first diagonal block. On reaching the end of the block back substitution is performed using the appropriate elements of U_1. The procedure for subsequent blocks is similar except that prior to the forward elimination using a row of the lower triangular part of a diagonal block, a forward substitution should be performed using the non-zeros in the part of the row in any sub-diagonal blocks. The only added complication occurs if any pivot is zero due to singularities in A. This will have been recorded by MA30A by negating the appropriate entry in IQ. When such a negative entry is found during the back substitution phase, no operations need be performed and the value of the entry in the solution vector is the residual corresponding to that unsatisfied equation. This is used to update the

value of the maximum such residual after which it is set to zero (the choice is arbitrary) in order to avoid problems associated with scaling. Finally, a permutation corresponding to Q^{-1} secures the solution.

For the transpose entry (MTYPE≠1), we can express our decomposition as

$$Q^{-1}AP^{-1} = \begin{pmatrix} U_1^T L_1^T & A_{21}^T & A_{31}^T & \cdots & A_{N1}^T \\ & U_2^T L_2^T & & & \\ & & \ddots & & \\ \bigcirc & & & & U_N^T L_N^T \end{pmatrix}$$

and so the sequence is now as follows:

(i) Preorder using IQ (correct since Q operated on columns and now Q^{-1} operates on rows).

For each block,

(ii) Perform forward elimination using appropriate elements from the U_i updating maximum residual in the event of singularities.

(iii) Perform back substitution using columns of the L_i^T and $A_{ij}^T (j<i)$ in reverse order, first using elements in the diagonal blocks and then in the off-diagonal blocks,

and finally

(iv) Order the solution vector using IP^{-1}.

Notice that, in the transpose entry, the forward elimination is performed using the U_i^T which are effectively held by columns, since the U_i are held by rows. We can use this to take advantage of any zeros in the right hand side since, during the forward elimination, if we are performing the elimination using the k^{th} column of the U_i^T and the k^{th} component of the modified right hand side is zero, then no operations need be done and we can

immediately proceed to the next column of the U_i^T. There is, however, a slight compensation in the former entry since the inner loops can be organised to reduce the number of basic operations on array elements (see code). Indeed on runs on our 32 test matrices, the total time for MA28C with MTYPE=1 was 610 milliseconds whereas the time for MTYPE≠1 was 630 milliseconds (on our IBM 370/168).

5. Specification sheets and availability of code

Specification sheets for the subroutines described in sections 1-4 now follow. The order in which the subroutines are so described is:

MA28A/B/C

MA30A/B/C

MC13D

MC20A

MC21A

MC22A

MC23A

MC24A

The codes for MC13D and MC21A are given by Duff and Reid (1978b) and Duff (1978) respectively. Machine readable versions of all the codes in the MA28A package can be obtained by writing to S. Marlow, Harwell Subroutine Library, Building 8.9, A.E.R.E. Harwell, Oxon OX11 ORA.

A similar code for solving sparse unsymmetric equations whose coefficient matrix is complex is also in the Harwell Subroutine Library. The corresponding subroutines are ME28A/B/C, ME30A/B/C, ME20A, ME22A and ME23A. Subroutine ME24A exists but is not called from the ME28 package. Subroutines MC21A and MC13D are called by ME23A. Further details on ME28 can be found in Duff (1980).

SPECIFICATION SHEETS

FOR

SUBROUTINE

MA28A

1 SUMMARY

To **solve a sparse system of linear equations.** Given a sparse matrix $\mathbf{A} = \{a_{ij}\}_{n\times n}$ this subroutine **decomposes A into factors,** solves $\mathbf{Ax} = \mathbf{b}$ (or optionally $\mathbf{A}^T\mathbf{x} = \mathbf{b}$). It will decompose a new matrix having the same sparsity pattern as a previous one by using the same pivotal sequence taking much less processing time than the original factorization. The matrix **A** is also allowed to be singular or rectangular.

The method is a variant of Gaussian elimination for sparse systems and further information is given in Duff, AERE R.8730, (1977).

ATTRIBUTES — Remark: Supersedes MA18A. **Versions:** MA28A, MA28AD. **Calls:** MA30A, MC20A, MC22A, MC23A, MC24A. **Language:** Fortran IV (standard available) **Date:** April 1977. **Size:** 9,600 bytes; 653 cards. **Origin:** I.S.Duff, 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) MA28A decomposes **A** into factors using a pivotal strategy designed to compromise between maintaining sparsity and controlling loss of accuracy through roundoff.

(b) MA28B factorizes a new matrix **A** of the same pattern, using the pivotal sequence determined by an earlier entry to MA28A.

(c) MA28C uses the factors produced by MA28A (or MA28B) to solve $\mathbf{Ax}=\mathbf{b}$ or $\mathbf{A}^T\mathbf{x}=\mathbf{b}$.

MA28B is much faster than MA28A. In some applications it is expected that there will be many calls to MA28B for each call to MA28A. Also, it is expected that MA28C may be called with many different vectors for the same matrix **A**.

We first describe the argument list for MA28A. Reference should be made to this description for information on paramaters which are common to MA28A and MA28B/C.

To decompose a matrix

The single precision version:

```
CALL MA28A(N,NZ,A,LICN,IRN,LIRN,ICN,U,IKEEP,IW,W,IFLAG)
```

The double precision version:

```
CALL MA28AD(N,NZ,A,LICN,IRN,LIRN,ICN,U,IKEEP,IW,W,IFLAG)
```

N is an **INTEGER*4 variable** which must be set by the user to the order n of the matrix **A**. It is not altered by the subroutine. **Restriction:** $1 \leqslant n \leqslant 32767$.

NZ is an **INTEGER*4 variable** which must be set by the user to the number of non-zeros in the matrix **A**. It is not altered by the subroutine. **Restriction:** NZ$\geqslant$1.

A is a **REAL*4 array** *(or* **REAL*8** *when using D version)* of length LICN and A(K), K=1,NZ must be set by the user to hold the non-zero elements of the matrix **A**. On exit, A holds the non-zero elements in the factors of the matrix **A**. It should be preserved by the user between calls to this routine and MA28C.

LICN is an **INTEGER*4 variable** which must be set by the user to the length of arrays A and ICN. Since the decomposition is returned in A and ICN, LICN should be large enough to accommodate this and should ordinarily be 2 to 4 times as large as NZ (see section 2.4). It is not altered by the subroutine. **Restriction:** LICN$\geqslant$NZ.

IRN is an **INTEGER*2 array** of length LIRN. On entry to MA28A, IRN(K) must hold the row index of the non-zero stored in A(K), K=1,NZ. It is used as workspace by MA28A, is altered by MA28A, and need not be preserved for any subsequent calls.

LIRN is an **INTEGER*4 variable** which must be set by the user to the length of array IRN. LIRN need not be as large as LICN, normally it will not need to be very much greater than NZ. It is not altered by the subroutine. **Restriction:** LIRN$\geqslant$NZ.

ICN is an **INTEGER*2 array** of length LICN. On entry ICN(K) must hold the column index of the non-zero stored in A(K), K=1,NZ. On output, it holds the column indices of the factors of the matrix **A**. These entries should be unaltered by the user between a call to this subroutine and subsequent calls to MA28B or MA28C.

U is a **REAL*4 variable** *(or* **REAL*8** *when using D version)* . On input to MA28A, the user should set U to a value between zero and one to control the choice of pivots. A value of 0.10 has been found to work well on test examples. The subroutine will not fail if U is outside the above range; values of U less than zero are treated as zero and values of U greater than one are treated as one. It is unaltered by the subroutine.

IKEEP is an **INTEGER*2 array** of length 5*N. It need never be referenced by the user and should be preserved between calls to this subroutine and MA28B or MA28C.

IW is an **INTEGER*4 array** of length 5*N. It is used as workspace by the subroutine.

W is a **REAL*4 array** *(or* **REAL*8** *when using D version)* of length N. W is used as workspace and an estimate of the largest element encountered during LU decomposition (see section 2.4) is output in W(1). If such an estimate is not requested, then W is not used at all by MA28A.

IFLAG is an **INTEGER*4 variable**. On exit from MA28A, a value of zero indicates that the subroutine has performed successfully. For non-zero values, see section 2.3.

To decompose a matrix which has a similar structure to that previously decomposed by MA28A

The single precision version:

```
CALL MA28B(N,NZ,A,LICN,IVECT,JVECT,ICN,IKEEP,IW,W,IFLAG)
```

The double precision version:

```
CALL MA28BD(N,NZ,A,LICN,IVECT,JVECT,ICN,IKEEP,IW,W,IFLAG)
```

The user must input his matrix in the same way in which he input the original matrix to MA28A. In this case, the parameters are as follows:

N **INTEGER*4 variable** equal to the order of the matrix. It is not altered by the subroutine.

NZ **INTEGER*4 variable** equal to number of non-zeros in the matrix. It is not altered by the subroutine.

A **REAL*4 array** *(or* **REAL*8** *when using D version)* of length LICN. The user must set A(K), K=1,NZ to hold the non-zero elements of the matrix **A**. On exit, A holds the non-zero elements of the factors of the matrix **A**. It must be preserved by the user between calls to this subroutine and MA28C.

LICN **INTEGER*4 variable** equal to length of arrays A and ICN. It is not altered by the subroutine.

IVECT,JVECT **INTEGER*2 arrays** of length NZ. IVECT(K) and JVECT(K) must contain respectively the row and column index of the non-zero stored in A(K), K=1,NZ. They are not altered by MA28B.

The other parameters are as follows:

ICN,IKEEP are the **INTEGER*2 arrays** (of lengths LICN, and 5*N, respectively) of the same names as in the previous call to MA28A. They should be unchanged since this earlier call and they are not altered by MA28B.

IW is an **INTEGER*4 array** of length 4*N used as workspace by MA28B.

W is a **REAL*4 array** *(or* **REAL*8** *when using D version)* of length N. It is used as workspace and, if an estimate of element growth is requested (see section 2.4), this will be output in W(1).

IFLAG is an **INTEGER*4 variable** which will be set to zero on successful exit from MA28B, otherwise it will have a non-zero value (see section 2.3).

To solve equations $Ax=b$ or $A^Tx=b$, using the factors of A from MA28A or MA28B

The single precision version:

```
CALL MA28C(N,A,LICN,ICN,IKEEP,RHS,W,MTYPE)
```

The double precision version:

```
CALL MA28CD(N,A,LICN,ICN,IKEEP,RHS,W,MTYPE)
```

Information about the factors of **A** is communicated to this subroutine via the parameters N, A, LICN, ICN and IKEEP where:

N **INTEGER*4 variable** equal to the order of the matrix. It is not altered by the subroutine.

A **REAL*4 array** *(or* **REAL*8** *when using D version)* of length LICN. It must be unchanged since the last call to MA28A or MA28B. It is not altered by the subroutine.

LICN is an **INTEGER*4 variable** equal to the length of arrays A and ICN. It is not altered by the subroutine.

ICN,IKEEP are the **INTEGER*2 arrays** (of lengths LICN and 5*N, respectively) of the same names as in the previous call to MA28A. They should be unchanged since this earlier call and they are not altered by MA28C.

The other parameters are as follows:

RHS is a **REAL*4 array** *(or* **REAL*8** *when using D version)* of length N. The user must set RHS(I) to contain the value of the Ith component of the right hand side (b_I) I=1,N. On exit, RHS(I) contains the Ith component of the solution vector (x_I), I=1,N.

W is a **REAL*4 array** *(or* **REAL*8** *when using D version)* of length N. It is used as workspace by MA28C.

MTYPE is an **INTEGER*4 variable** which the user must set to determine whether MA28C will solve

$\mathbf{Ax}=\mathbf{b}$ (MTYPE equal to 1) or $\mathbf{A}^T\mathbf{x}=\mathbf{b}$ (MTYPE≠1, zero say). It is not altered by MA28C.

2.2 Parameter usage summary

MA28A

INPUT	N, NZ, A(LICN), LICN, IRN(LIRN), LIRN, ICN(LICN), U
UNCHANGED BY MA28A	N, NZ, LICN, LIRN, U
OUTPUT	A, ICN, IKEEP(5*N), W(1) †, IFLAG
WORK–ARRAYS	IW(5*N), W(N) †.

MA28B

INPUT	(by user) NZ, A(LICN), IVECT(NZ), JVECT(NZ)
INPUT	(from MA28A) N, ICN(LICN), LICN, IKEEP(5*N)
UNCHANGED BY MA28B	N, NZ, ICN, LICN, IKEEP
OUTPUT	A, IFLAG, W(1)†
WORK–ARRAYS	IW(4*N), W(N).

MA28C

INPUT	(by user) RHS(N), MTYPE
INPUT	(from MA28A) N, ICN(LICN), LICN, IKEEP(5*N)
INPUT	(from MA28A or MA28B) A(LICN)
UNCHANGED BY MA28C	N, ICN, LICN, IKEEP, A, MTYPE
OUTPUT	RHS
WORK–ARRAYS	W(N)

† Optional ..... see sections 2.1 and 2.4.

2.3 Error diagnostics

A successful return from MA28A or MA28B is indicated by a value of IFLAG equal to zero. There are no error returns from MA28C. Possible non-zero values for IFLAG are given below. Unless otherwise stated error returns are for both MA28A and MA28B entries:

−14 to −8 Error in user's input matrix. The nature is specified in an output message.

−14 More than one non-zero in same position in matrix. Action taken is to proceed with value equal to sum of duplicate elements. (See common block variable MP in section 2.4).

−13 Non-zero was not present in factors after previous call to MA28A. (MA28B entry only).

−12 Row or column index out-of-range.

−11 $1 \leqslant N \leqslant 32767$ violated.

−10 $NZ \leqslant 0$

−9 LICN<NZ

−8 LIRN<NZ (MA28A entry only)

−7 Error encountered during block triangularization phase. (MA28A entry only)

−6 to −3 Storage allocation for decomposition is insufficient (see common block variables MINICN and MINIRN, section 2.4) (all MA28A only).

−6 LIRN and LICN too small....information available from MINICN (see section 2.4).

−5 LICN too small....increase to at least value given by common block variable MINICN (see section 2.4).

–4 LICN far too small. No useful information in MINICN.

–3 LIRN too small.

–2 Matrix numerically singular.

–1 Matrix structurally singular. This means that the non-zero pattern is such that the matrix will be singular for all possible numerical values of the non-zeros (MA28A only).

+1 Successful decomposition on a structurally singular matrix (MA28A only).

+2 Successful decomposition on a numerically singular matrix (MA28A only).

+I (I=1,2,..,N) Warning. Very small pivot in row I (MA28B only).

2.4 Common blocks used

In single precision version:

```
COMMON/MA28E/LP,MP,LBLOCK,GROW
COMMON/MA28F/EPS,RMIN,RESID,IRNCP,ICNCP,MINIRN,MINICN,IRANK,
*  ABORT1,ABORT2
```

In double precision version:

```
COMMON/MA28ED/LP,MP,LBLOCK,GROW
COMMON/MA28FD/EPS,RMIN,RESID,IRNCP,ICNCP,MINIRN,MINICN,IRANK,
*  ABORT1,ABORT2
```

LP,MP are **INTEGER*4 variables** used by the subroutine as the unit numbers for its warning and diagnostic messages. Default value for both is 6 (for line printer output). The user can either reset them to a different stream number or suppress the output by setting them to zero. While LP directs the output of error diagnostics from the subroutines themselves and internally called subroutines, MP controls only the output of a message which warns the user that he has input two or more non-zeros A(I), . . ,A(K) with the same row and column indices. The action taken in this case is to proceed using a numerical value of A(I)+...+A(K). In the absence of other errors, IFLAG will equal –14 on exit (see section 2.3).

LBLOCK is a **LOGICAL*4 variable** which controls an option of first preordering the matrix to block lower triangular form (using Harwell subroutine MC23A) (see section 2.6). The preordering is performed if LBLOCK is equal to its default value of .TRUE. If LBLOCK is set to .FALSE. , the option is not invoked and the space allocated to IKEEP can be reduced to 4*N+1.

GROW is a **LOGICAL*4 variable**. If it is left at its default value of .TRUE. , then on return from MA28A or MA28B, W(1) will give an estimate (an upper bound) of the increase in size of elements encountered during the decomposition. If the matrix is well scaled (see section 2.7), then a high value for W(1) indicates that the LU decomposition may be inaccurate and the user should be wary of his results and perhaps increase U for subsequent runs. We would like to emphasise that this value only relates to the accuracy of our LU decomposition and gives no indication as to the singularity of the matrix or the accuracy of the solution.

EPS,RMIN are **REAL*4 variables** *(or* **REAL*8** *when using D version)* . If, on entry to MA28B, EPS is less than one, then RMIN will give the smallest ratio of the pivot to the largest element in the corresponding row of the upper triangular factor thus monitoring the stability of successive factorizations. If RMIN becomes very large and W(1) from MA28B/BD is also very large, it may be advisable to perform a new decomposition using MA28A/AD.

RESID is a **REAL*4 variable** *(or* **REAL*8** *when using D version)* which on exit from MA28C/CD

gives the value of the maximum residual

$$\max_i | b_i - \sum_j a_{ij} x_j |$$

over all the equations unsatisfied because of dependency (zero pivots) (see section 2.5).

IRNCP, ICNCP are **INTEGER*4 variables** which monitor the adequacy of 'elbow room' in IRN and A/ICN respectively. If either is quite large (say greater than N/10), it will probably pay to increase the size of the corresponding array for subsequent runs. If either is very low or zero then one can perhaps save storage by reducing the size of the corresponding array.

MINIRN, MINICN are **INTEGER*4 variables** which, in the event of a successful return (IFLAG $\geqslant$ 0 or IFLAG=−14) give the minimum size of IRN and A/ICN respectively which would enable a successful run on an identical matrix. On an exit with IFLAG equal to −5, MINICN gives the minimum value of ICN for success on subsequent runs on an identical matrix. In the event of failure with IFLAG= −6, −4, −3, −2, or −1, then MINICN and MINIRN give the minimum value of LICN and LIRN respectively which would be required for a successful decomposition up to the point at which the failure occurred.

IRANK is an **INTEGER*4 variable** which gives an upper bound on the rank of the matrix.

ABORT1 is a **LOGICAL*4 variable** with default value .TRUE. If ABORT1 is set to .FALSE. then MA28A will decompose structurally singular matrices (including rectangular ones).

ABORT2 is a **LOGICAL*4 variable** with default value .TRUE. If ABORT2 is set to .FALSE. then MA28A will decompose numerically singular matrices.

2.5 Singular or rectangular matrices

Singular and rectangular matrices can be handled by resetting common block variables ABORT1 and/or ABORT2 (see section 2.4). Additionally, the user must set N to the largest dimension of his system. RESID (see section 2.4) will give useful information on the consistency of the equations.

2.6 Block lower triangular form

Many large unsymmetric matrices can be permuted to the form

$$\mathbf{PAQ} = \begin{pmatrix} \mathbf{A}_{1,1} & & & & & \\ \cdot & \mathbf{A}_{2,2} & & & & \\ \cdot & \cdot & \mathbf{A}_{3,3} & & & \\ \cdot & \cdot & \cdot & \cdot & & \\ \cdot & \cdot & \cdot & \cdot & \cdot & \\ \mathbf{A}_{k,1} & \cdot & \cdot & \cdot & \cdot & \mathbf{A}_{k,k} \end{pmatrix}$$

whereupon the system

$$\mathbf{Ax=b} \qquad (\mathbf{A}^T\mathbf{x=b})$$

can be solved by block-forward–(back–) substitution giving a saving in storage and execution time if the matrices $\mathbf{A}_{ii}$ are much smaller than **A**.

This facility is invoked as a default and information on the reordering is given by the variables in

Single precision version:

COMMON/MC23B/LP, NUMNZ, NUM, LARGE, ABORT

Double precision version:

```
COMMON/MC23BD/LP,NUMNZ,NUM,LARGE,ABORT
```

where the **INTEGER variables** NUMNZ, NUM, LARGE give the structural rank, number of diagonal blocks (K), and the order of the largest block respectively. ABORT is a logical variable set by MA28A to the value of ABORT1 (section 2.4).

If the user wishes to suppress this option he may do so by setting common block variable LBLOCK to .FALSE. He can then also reduce the length of IKEEP to 4N+1.

2.7 Badly-scaled systems

If the user's input matrix has elements differing widely in magnitude, then an inaccurate solution may be obtained and the estimate of the increase in element size given by MA28A or MA28B will not be very meaningful. In such cases, the user is advised to first use MC19A/AD to obtain scaling factors for his matrix and then explicitely scale it prior to calling MA28A. Thereafter, both left and right hand sides should be scaled as indicated in the code following the example in section 5.

3 GENERAL INFORMATION

Use of common: the subroutine uses common blocks MA28E/ED, MA28F/FD, MA28G/GD, see sections 2.4 and 2.6.

Workspace: W of length N (optional in MA28A/AD entry ...see 2.4).

Other subprograms: The following subroutines are called by the subroutines in this package. MC20A/AD, MC22A/AD, MC23A/AD, MC24A/AD, MA30A/AD.

Input/output: Errors and warning messages only. Error messages on unit LP, warning messages on unit MP. Both have default value 6, and output is suppressed if they are set to zero.

Restrictions:

$1 \leqslant n \leqslant 32767$,
$0 \leqslant$ NZ,
LICN $\geqslant$ NZ,
LIRN $\geqslant$ NZ.

Portability: special comments are included in the code to enable the library subroutine OE04A to generate a version that conforms to PFORT, a portable Fortran closely approximating the ANSI standard of 1966. For this version, the restriction $n \leqslant 32767$ is removed.

4 METHOD

These subroutines are really only data management routines. A description of the initial subroutines called is given by Duff (AERE Report R.8730,1977). The method used is a sparse variant of Gaussian elimination.

5 EXAMPLE OF USE

5.1 An example to solve sparse equations.

In the example code shown in Figure 1, we first decompose a matrix and use information from this decomposition to solve a set of linear equations. Then we factorize a matrix of a similar sparsity pattern and solve another set of equations.

Thus if, in this example, we wish to solve:

$$\begin{pmatrix} 3.14 & 7.5 & \\ 4.1 & 3.2 & 0.3 \\ & 1.0 & 4.1 \end{pmatrix} \mathbf{x} = \begin{pmatrix} 1.0 \\ 2.0 \\ 3.0 \end{pmatrix}$$

followed by the system:

$$\begin{pmatrix} 4.7 & 6.2 & \\ 3.2 & 0.0 & 0.31 \\ & 3.1 & 0.0 \end{pmatrix} \mathbf{x} = \begin{pmatrix} 1.1 \\ 2.1 \\ 3.1 \end{pmatrix}$$

we have as input

```
N,NZ:            3   7
IRN,ICN,A:       1 1   3.14
                 2 3   Ø.3Ø
                 3 3   4.1
                 2 1   4.1
                 1 2   7.5
                 3 2   1.Ø
                 2 2   3.2
X:      1.Ø   2.Ø  3.Ø
```

(and output would be X: 0.48858D+00 –0.71219D–01 0.74908D+00) followed by input

```
A:     4.7   Ø.31  Ø.Ø   3.2   6.2   3.1   Ø.Ø
X:     1.1   2.1   3.1
```

(with output now being X: –0.10851D+01 0.10000D+01 0.17975D+02)

5.2 Scaling

If the user wishes to scale his matrix (see section 2.8) he should insert the following lines

(a) Between reading the input matrix and calling MA28AD.

```
      CALL MC19AD(N,NZ,A,IRN,ICN,R,C,A(NZ+1))
      DO 34Ø I=1,N
      R(I)=EXP(R(I))
34Ø   C(I)=EXP(C(I))
      DO 35Ø II=1,NZ
      I=IRN(II)
      J=ICN(II)
```

```
350  A(II)=A(II)*R(I)*C(J)
```

(b) Before calling MA28CD

```
     DO 425 I=1,N
425  X(I)=X(I)*R(I)
```

(c) After calling MA28CD

```
     DO 475 I=1,N
475  X(I)=X(I)*C(I)
```

(d) and the following additional declarations are required:

```
     REAL R(50),C(50)
```

```
      DOUBLE PRECISION A(500),W(50),X(50),U
      INTEGER IW(250)
      INTEGER*2 ICN(500),IRN(300),IVECT(250),JVECT(250),IKEEP(250)
      LICN=500
      LIRN=300
C READ IN INPUT MATRIX.
      READ(5,100) N,NZ
 100  FORMAT(20I6)
      IF (N.GT.50.OR.NZ.GT.250) GO TO 500
      READ(5,200) (IRN(I),ICN(I),A(I),I=1,NZ)
 200  FORMAT(2I6,D12.5)
C COPY INDEX INFORMATION FOR USE IN SUBSEQUENT MA28BD CALL.
      DO 300 I=1,NZ
      IVECT(I)=IRN(I)
 300  JVECT(I)=ICN(I)
      U=0.10D0
C DECOMPOSE MATRIX INTO ITS FACTORS.
      CALL MA28AD(N,NZ,A,LICN,IRN,LIRN,ICN,U,IKEEP,IW,W,IFLAG)
      IF (IFLAG.LT.0) GO TO 500
C READ IN RIGHT HAND SIDE.
      READ(5,400) (X(I),I=1,N)
 400  FORMAT(5D13.5)
C SOLVE LINEAR SYSTEM.
      CALL MA28CD(N,A,LICN,ICN,IKEEP,X,W,1)
C PRINT OUT SOLUTION VECTOR.
      WRITE(6,400) (X(I),I=1,N)
C READ IN, DECOMPOSE, AND SOLVE SYSTEM OF SIMILAR EQUATIONS.
      READ(5,400) (A(I),I=1,NZ)
      READ(5,400) (X(I),I=1,N)
      CALL MA28BD(N,NZ,A,LICN,IVECT,JVECT,ICN,IKEEP,IW,W,IFLAG)
      IF (IFLAG.LT.0) GO TO 700
      CALL MA28CD(N,A,LICN,ICN,IKEEP,X,W,1)
      WRITE(6,400) (X(I),I=1,N)
      GO TO 700
 500  WRITE(6,600)
 600  FORMAT(13H ERROR RETURN)
 700  STOP
      END
```

Figure 1. Code to solve a set of sparse linear equations

SPECIFICATION SHEETS

FOR

SUBROUTINE

MA30A

1. Purpose

These subroutines perform operations pertinent to the solution of a general sparse N×N system of linear equations

$$\sum_{j=1}^{N} a_{ij} x_j = b_i, \quad i=1,2,\ldots,N,$$

(i.e. solve $A\underline{x}=\underline{b}$). Structurally singular matrices are permitted, including those with row or columns consisting entirely of zeros (i.e. including rectangular matrices). It is assumed that the non-zeros a_{ij} do not differ widely in size. If necessary a prior call of the scaling subroutine MC19A/AD may be made.

(a) MA30A/AD performs the LU decomposition of the diagonal blocks of the permutation PAQ of a sparse matrix A, where input permutations P_1 and Q_1 are used to define the diagonal blocks. There may be non-zeros in the off-diagonal blocks but they are unaffected by MA30A/AD. P and P_1 differ only within blocks as do Q and Q_1. The permutations P_1 and Q_1 may be found by calling MC23A/AD or the matrix may be treated as a single block by using $P_1=Q_1=I$. The matrix non-zeros should be held compactly by rows, although it should be noted that the user can supply his matrix by columns to get the LU decomposition of A^T.

(b) MA30B/BD performs the LU decomposition of the diagonal blocks of a new matrix PAQ of the same sparsity pattern, using information from a previous call to MA30A/AD. The elements of the input matrix must already be in their final positions in the LU decomposition structure. This routine executes about five times faster than MA30A/AD.

(c) MA30C/CD uses the factors produced by MA30A/AD or MA30B/BD to solve $A\underline{x}=\underline{b}$ or $A^T\underline{x}=\underline{b}$ when the matrix P_1AQ_1 (PAQ) is block lower triangular (including the case of only one diagonal block).

If the user requires a more convenient data interface then he should consult MA28A/A/B/C. These subroutines call MA30A/B/C after checking the user's input data and optionally using MC23A to permute the matrix to block triangular form.

2. Argument lists

We first describe the argument list for MA30A. This description should also be consulted for further information on most of the parameters of MA30B and MA30C.

```
CALL MA30A(N,ICN,A,LICN,LENR,LENRL,IDISP,IP,IQ,IRN,LIRN,LENC,IFIRST,
     LASTR,NEXTR,LASTC,NEXTC,IPTR,IPC,U,IFLAG)
```

N is an INTEGER variable which must be set by the user to the order of the matrix. Because of the use of INTEGER*2 arrays, the value of N should be less than 32768(2^{15}). It is not altered by MA30A.

ICN is an INTEGER*2 array of length LICN. Positions IDISP(2) to LICN must be set by the user to contain the column indices of the non-zeros in the diagonal blocks of P_1AQ_1. Those belonging to a single row must be continguous but the ordering of column indices within each row is unimportant. The non-zeros of row I precede those of row I+1, I=1,...,N-1 and no wasted space is allowed between the rows.

On output the column indices of the LU decomposition of PAQ are held in positions IDISP(1) to IDISP(2). The rows are in pivotal order, and the column indices of the L part of each row are in pivotal order and precede those of U. Again there is no wasted space either within a row or between the rows. ICN(1) to ICN(IDISP(1)-1), are neither required nor altered. If MC23A has been called, these will hold information about the off-diagonal blocks.

A is a REAL (DOUBLE PRECISION in D version) array of length LICN whose elements IDISP(2) to LICN must be set by the user to the values of the non-zero elements of the matrix in the order indicated by ICN. On output A will hold the LU factors of the matrix where again the position in the matrix is determined by the corresponding values in ICN. A(1) to A(IDISP(1)-1) are neither required nor altered.

LICN is an integer variable which must be set by the user to the length of arrays ICN and A. It must be big enough for A,ICN to hold all the non-zeros of L and U and leave some "elbow room". It is possible to calculate a minimum value for this by a preliminary run of MA30A (see section 6). The adequacy of the elbow room can be judged by the size of the common block variable ICNCP (see section 6). It is not altered by MA30A.

LENR is an INTEGER*2 array of length N. On input, LENR(I) should equal the number of non-zeros in row I, I=1,...,N of the diagonal blocks of P_1AQ_1. On output, LENR(I) will equal the total number of non-zeros in row I of L and row I of U.

LENRL is an INTEGER*2 array of length N. On output from MA30A, LENRL(I) will hold the number of non-zeros in row I of L.

IDISP is an INTEGER array of length 2. The user should set IDISP(1) to be the first available position in A/ICN for the LU decomposition while IDISP(2) is set to the position in A/ICN of the first non-zero in the diagonal blocks of P_1AQ_1. On output IDISP(1) will be unaltered while IDISP(2) will be set to the position in A/ICN of the last non-zero of the LU decomposition.

IP is an INTEGER*2 array of length N which holds a permutation of the integers 1 to N. On input to MA30A the absolute value of IP(I) should indicate the row of A which is row I of P_1AQ_1. A negative value for IP(I) indicates that row I is at the end of a diagonal block. On output from MA30A, IP(I) indicates the row of A which is I^{th} row in PAQ. IP(I) will still be negative for the last row of each block (except the last).

IQ is an INTEGER*2 array of length N which again holds a permutation of the integers 1 to N. On input to MA30A, IQ(J) should indicate the column of A which is column J of P_1AQ_1. On output from MA30A, the absolute value of IQ(J) indicates the column of A which is the J^{th} in PAQ. For rows, I say, in which structural or numerical singularity is detected IQ(I) is negated (see section 7).

IRN is an INTEGER*2 array of length LIRN used as workspace by MA30A.

LIRN is an INTEGER variable. It should be greater than the largest number of non-zeros in a diagonal block of P_1AQ_1 but need not be as large as LICN. It is the length of array IRN and should be large enough to hold the active part of any block, plus some "elbow room" the a posteriori adequacy of which can be estimated by examining the size of common block variable IRNCP (see section 6).

LENC,IFIRST,LASTR,NEXTR,LASTC,NEXTC
are all INTEGER*2 arrays of length N which are used as workspace by MA30A.

IPTR,IPC are INTEGER arrays of length N which are used as workspace by MA30A.

U is a REAL (DOUBLE PRECISION in D version) variable which should be set by the user to a value between 0. and 1.0. If less than zero it is reset to zero and if its value is 1.0 or greater it is reset to 0.9999 (0.999999999 in D version). It determines the balance between pivoting for sparsity and for stability, values near zero emphasizing sparsity and values near one emphasizing stabality. We recommend U=0.1 as a possible first trial value. The stability can be judged by a later call to MC24A/AD.

IFLAG is an INTEGER variable. It will have a non-negative value if MA30A is successful. Negative values indicate error conditions while positive values indicate that the matrix has been successfully decomposed but is singular (see section 5).

We now describe the argument list for MA30B.

CALL MA30B(N,ICN,A,LICN,LENR,LENRL,IDISP,IP,IQ,W,IW,IFLAG)

N is an INTEGER variable set to the order of the matrix.

ICN is an INTEGER*2 array of length LICN. It should be unchanged since the last call to MA30A. It is not altered by MA30B.

A is a REAL (DOUBLE PRECISION in D version) array of length LICN The user must set entries IDISP(1) to IDISP(2) to contain the elements in the diagonal blocks of the matrix PAQ whose column numbers are held in ICN, using corresponding positions. Note that some zeros may need to be help explicitly. On output entries IDISP(1) to IDISP(2) of array A contain the LU decomposition of the diagonal blocks of PAQ. Elements A(1) to A(IDISP(1)-1) are neither required nor altered by MA30B.

LICN is an INTEGER variable which must be set by the user to the length of arrays A and ICN.

LENR,LENRL are INTEGER*2 arrays of length N. They should be unchanged since the last call to MA30A. They are not altered by MA30B.

IDISP is an INTEGER array of length 2. It should be unchanged since the last call to MA30A. It is not altered by MA30B.

IP,IQ are INTEGER*2 arrays of length N. They should be unchanged since the last call to MA30A. They are not altered by MA30B.

W is a REAL (DOUBLE PRECISION in D version) array of length N which is used as workspace by MA30B.

IW is an INTEGER array of length N which is used as workspace by MA30B.

IFLAG is an INTEGER variable. On output from MA30B IFLAG has the value zero if the factorization was successful, has the value I if pivot I was very small (see section 5) and has the value -I if an unexpected singularity was detected at stage I of the decomposition (see section 5).

We finally describe the argument list for MA30C.

CALL MA30C(N,ICN,A,LICN,LENR,LENRL,LENOFF,IDISP,IP,IQ,X,W,MTYPE)

N is an INTEGER variable set to the order of the matrix.

ICN is an INTEGER*2 array of length LICN. Entries IDISP(1) to IDISP(2) should be unchanged since the last call to MA30A. If the matrix has more than one diagonal block, then column indices corresponding to non-zeros in sub-diagonal blocks of PAQ must appear in positions 1 to IDISP(1)-1. For the same row those entries must be contiguous, with those in row I preceding those in row I+1 (I=1,...,N-1) and no wasted space between rows. Entries may be in any order within each row. It is not altered by MA30C.

A is a REAL (DOUBLE PRECISION in D version) array of length LICN. Entries IDISP(1) to IDISP(2) should be unchanged since the last call to MA30A or MA30B. If the matrix has more than one diagonal block, then the values of the non-zeros in sub-diagonal blocks must be in positions 1 to IDISP(1)-1 in the order given by ICN. It is not altered by MA30C.

LICN is an INTEGER variable set to the size of arrays ICN and A.

LENR,LENRL are INTEGER*2 arrays of length N which should be unchanged since the last call to MA30A. They are not altered by MA30C.

LENOFF is an INTEGER*2 array of length N. If the matrix PAQ (or P_1AQ_1) mas more than one diagonal block, then LENOFF(I), I=1,...,N should be set to the number of non-zeros in row I of the matrix PAQ which are in sub-diagonal blocks. If there is only one diagonal block then LENOFF(1) may be set to -1, in which case the other elements of LENOFF are never accessed. It is not altered by MA30C.

IDISP is an INTEGER array of length 2 which should be unchanged since the last call to MA30A. It is not altered by MA30C.

IP,IQ are INTEGER*2 arrays of length N which should be unchanged since the last call to MA30A. They are not altered by MA30C.

X is a REAL (DOUBLE PRECISION in D version) array of length N. It must be set by the user to the values of the right hand side vector $\underline{b}$ for the equations he wishes to solve. On exist from MA30C it will be equal to the solution $\underline{x}$ required.

W is a REAL (DOUBLE PRECISION in D version) array of length N which is used as workspace by MA30C.

MTYPE is an INTEGER variable which must be set by the user. If MTYPE=1, then the solution to the system $A\underline{x}=\underline{b}$ is returned; any other value for MTYPE will return the solution to the system $A^T\underline{x}=\underline{b}$. It is not altered by MA30C.

3. Parameter usage summary

MA30A
INPUT by user. N,ICN(IDISP(1) to LICN),A(IDISP(1) to LICN), LICN, LENR(N),IDISP(2),IP(N),IQ(N),LIRN,U.

UNCHANGED by MA30A. N,LICN,LIRN.

WORK ARRAYS. IRN(LIRN),LENC(N),IFIRST(N),LASTR(N),NEXTR(N), LASTC(N),NEXTC(N),IPTR(N),IPC(N).

OUTPUT from MA30A. ICN,A,LENR,LENRL(N),IDISP,IP,IQ,IFLAG.

MA30B
INPUT by user. A(IDISP(1) to IDISP(2)).

INPUT (from MA30A) N,ICN(IDISP(1) to IDISP(2)),LICN,LENR(N),LENRL(N), IDISP(2).

UNCHANGED by MA30B. N,ICN,LICN,LENR,LENRL,IDISP.

WORK ARRAYS. W(N),IW(N).

OUTPUT from MA30B. A,IFLAG.

MA30C
INPUT by user. LENOFF(N),X(N),MTYPE,A(IDISP(1)-1),ICN(IDISP(1)-1).

INPUT (from MA30A/B). N,A(IDISP(1) to IDISP(2)),ICN(IDISP(1) to IDISP(2)),LICN,IDISP(2),LENR(N),LENRL(N),IP(N),IQ(N).

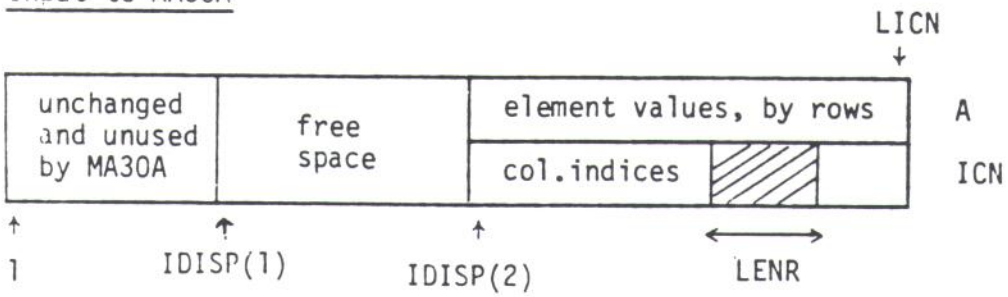
UNCHANGED by MA30C. N,A,ICN,LICN,IDISP,LENR,LENRL,LENOFF,IP,IQ,MTYPE.

WORK ARRAYS. W(N).

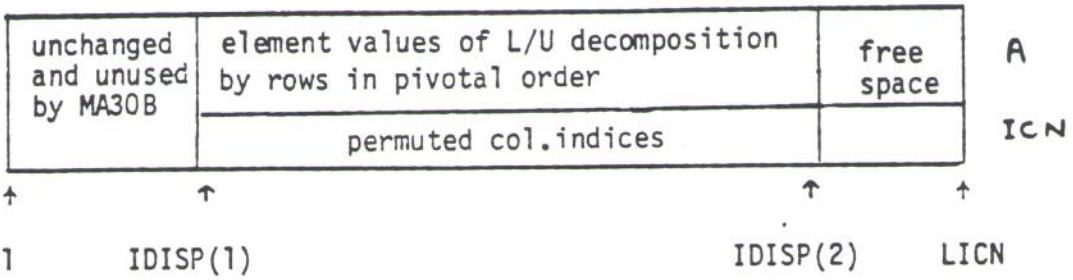
OUTPUT from MA30C. X.

4. Data structures summary

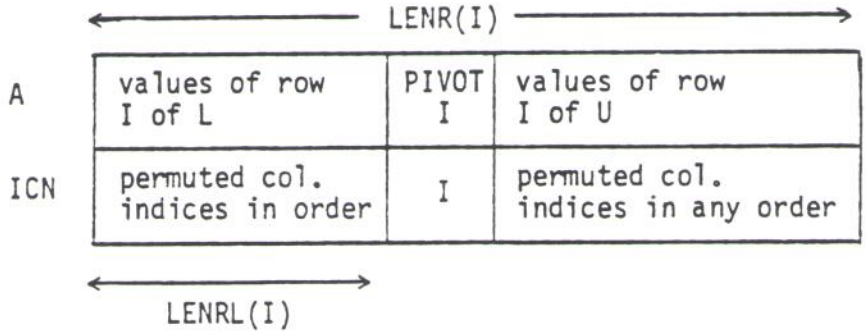
a) Input to MA30A



b) Output from MA30A and input to MA30B (A is overwritten)

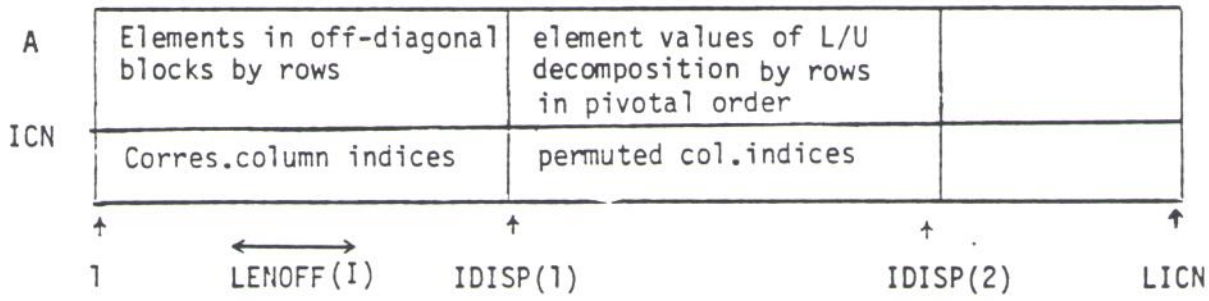


and for each row (row I, say)



N.B. Fill-ins can occur anywhere within the row.

c) Input to MA30C



5. Error diagnostics

MA30A

If the subroutine performs the LU decomposition without any complications or errors, the value of IFLAG will be non-negative on exit from MA30A. The complications which can arise are given below. Some are more disastrous than others and the user must decide in each instance what further action to take. In some cases messages are output on the line printer (unless supressed (LP=0) or switched to another stream, see section 6). Possible negative values for IFLAG are as follows:

-1 The matrix is structurally singular with rank given by IRANK in COMMON block MA30F (see section 6). The message:

ERROR RETURN FROM MA30A/AD BECAUSE MATRIX IS STRUCTURALLY SINGULAR

will be output.

If, however, the user wants the LU decomposition of a structurally singular matrix (see section 7) and sets the common block variable ABORT1 to .FALSE. , then in the event of singularity and a successful decomposition IFLAG is returned with the value +1 and no message is output.

-2 The matrix is numerically singular (it may also be structurally singular) with estimated rank given by IRANK in COMMON block MA30F (see section 6). The message:

ERROR RETURN FROM MA30A/AD BECAUSE MATRIX IS NUMERICALLY SINGULAR

will be output.

The user can choose to continue the decomposition even when a zero pivot is encountered by setting common block variable ABORT2 to .FALSE. If a singularity is encountered, IFLAG will then return with a value of +2 and no message is output if the decomposition has been completed successfully.

-3 LIRN has not been large enough to continue with the decomposition. Should this happen, the message:

ERROR RETURN FROM MA30A/AD BECAUSE LIRN NOT BIG ENOUGH

AT STAGE < > IN BLOCK < > WITH FIRST ROW < > AND LAST ROW < >
is output.

If the stage was zero then common block variable MINIRN (see section 6) gives the length sufficient to start the decomposition on this block and the message:

TO CONTINUE SET LIRN TO AT LEAST <MINIRN> is output.

For a successful decomposition on this block the user should make LIRN slightly (say about N/2) greater than this value.

-4 LICN has not been large enough to continue with the decomposition. Should this happen, the message:

ERROR RETURN FROM MA30A/AD BECAUSE LICN NOT BIG ENOUGH

AT STAGE < > IN BLOCK < > WITH FIRST ROW < > AND LAST ROW < >
is output.

-5 The decomposition has been completed but some of the LU factors have been discarded to create enough room in A/ICN to continue the decomposition. The variable MINICN in COMMON block MA30F (see section 6) then gives the size that LICN should be to enable the factorization to be successful. Each time any of the LU factors are destroyed, the message:

LU DECOMPOSITION DESTROYED TO CREATE MORE SPACE

is output.

If the user sets common block variable ABORT3 to .TRUE. , then the subroutine will exit immediately instead of destroying any factors and continuing.

-6 Both LICN and LIRN are too small. Termination has been caused by lack of space in IRN (see error IFLAG= -3), but already some of the LU factors in A/ICN have been lost (see error IFLAG= -5). MINICN gives the minimum amount of space required in A/ICN for decomposition up till this point and the message:

ERROR RETURN FROM MA30A/AD LIRN AND LICN TOO SMALL

is output.

MA30B

If MA30B performs a successful decomposition, then IFLAG will have a non-negative value on exit. Other values for IFLAG are now described.

IFLAG= -I : The routine has terminated at stage I for one of the following reasons.

(i) A zero pivot was found at stage I where MA30A found a non-zero pivot.

(ii) A non-zero was found in a part of row I which lay in a submatrix which was entirely zero after decomposition using MA30A (see diagram in section 7). With this error return the message:

ERROR RETURN FROM MA30B/BD SINGULARITY DETECTED IN ROW I

is output.

IFLAG= +I : Although the decomposition has been successful this return indicates that there is a small pivot in row I with possible consequent stability problems. The common block variable RMIN gives an idea of how severe this is. See the description of common block MA30G (section 6) for further details on this return.

MA30C

There are no error returns from this subroutine.

6. Common blocks

MA30A

There are two labelled common blocks:

COMMON/MA30E/LP,ABORT1,ABORT2,ABORT3.

The variables in this common block (used also by MA30B) which are used as controlling parameters by MA30A are:

The INTEGER LP is the unit number to which the error messages (described in section 5) are sent. A BLOCK DATA subprogram sets LP equal to 6. This default value can be reset by the user, if desired. A value of 0 suppresses all messages.

The logical variables ABORT1,ABORT2,ABORT3 are used to control the conditions under which the subroutine will terminate.

If ABORT1 is .TRUE. then the subroutine will exit immediately on detecting structural singularity.

If ABORT2 is .TRUE. then the subroutine will exit immediately on detecting numerical singularity.

If ABORT3 is .TRUE. then the subroutine will exit immediately when the available space in A/ICN is filled up by the previously decomposed, active, and undecomposed parts of the matrix.

The default values for ABORT1,ABORT2,ABORT3 are set in a BLOCK DATA subprogram to .TRUE. , .TRUE. and .FALSE. respectively.

COMMON/MA30F/IRNCP,ICNCP,IRANK,MINIRN,MINICN

The variables in this common block are used to provide the user with information on the decomposition.

The INTEGERS IRNCP and ICNCP are used to monitor the adequacy of the allocated space in arrays IRN and A/ICN respectively, by taking account of the number of data management compresses required on these arrays. If IRNCP or ICNCP is fairly large (say greater than N/10), it may be advantageous to increase the size of the corresponding array(s). IRNCP and ICNCP are initialized to zero on entry to MA30A and are incremented each time the compressing routine MA30D (see section 8) is entered.

ICNCP is the number of compresses on A/ICN.

IRNCP is the number of compresses on IRN.

IRANK is an INTEGER variable which gives an estimate (actually an upper bound) of the rank of the matrix. On an exit with IFLAG equal to 0, this will be equal to N.

MINIRN is an INTEGER variable which, after a successful call to MA30A, indicates the minimum length to which IRN can be reduced while still permitting a successful decomposition of the same matrix. If, however, the user were to decrease his file to that size the number of compresses (IRNCP) may be very high and quite costly. If LIRN is not large enough to begin the decomposition on a diagonal block, MINIRN will be equal to the value required to continue the decomposition and IFLAG will be set to -3 or -6 (see section 5). A value of LIRN slightly greater than this (say about N/2) will usually provide enough space to complete the decomposition on that block. In the event of any other failure MINIRN gives the minimum size of IRN required for a successful decomposition up to that point.

MINICN is an INTEGER variable which after a successful call to MA30A, indicates the size of LICN required to enable a successful decomposition. In the event of failure with IFLAG= -5 (see section 6), MINICN will, if ABORT3 is left set to .FALSE. (see beginning of this section), indicate the minimum length that would be sufficient to prevent this error in a subsequent run on an identical matrix. Again the user may prefer to use a value of ICN slightly greater than MINICN for subsequent runs to avoid too many compresses (ICNCP). In the event of failure with IFLAG equal to any negative value except -4, MINICN will give the minimum length to which LICN could be reduced to enable a successful decomposition to the point at which failure occurred.

Notice that, on a successful entry IDISP(2) gives the amount of space in A/ICN required for the decomposition while MINICN will usually be slightly greater because of the need for "elbow room".

If the user is very unsure how large to make LICN, the variable MINICN can be used to give him that information. A preliminary run should be performed with ABORT3 left set to .FALSE. and LICN about 1½ times as big as the number of non-zeros in the original matrix. Unless his initial problem is very sparse (when the run will be successful) or fills in extremely badly (giving an error return with IFLAG equal to -4), an error return with IFLAG equal to -5 should result and MINICN will give the amount of space required for a successful decomposition.

MA30B

Again there are two labelled COMMON blocks. The first

```
COMMON/MA30E/LP,ABORT1,ABORT2,ABORT3
```

is the same as in MA30A.

LP has the same meaning as in MA30A and the LOGICAL variables ABORT1,ABORT2 and ABORT3 are not referenced by MA30B.

```
COMMON/MA30G/EPS,RMIN
```

where

EPS is a REAL (DOUBLE PRECISION in D version) variable. It is used to test for small pivots. Its default value, set by a BLOCK DATA

subprogram, is 1.0E-4 (1.0D-4 in D version). If the user sets EPS to any value greater than 1.0, then no check is made on the size of the pivots. Although this would result in bad instability, such an eventuality can be checked by using MC24A, and MA30B should execute slightly faster.

RMIN is a REAL (DOUBLE PRECISION in D version) variable which gives the user some information about the stability of the decomposition. At each stage of the LU decomposition the magnitude of the pivot APIV is compared with the largest off-diagonal element currently in its row (row of U), ROWMAX say. If the ratio

$$\min_i \left(^{APIV}/_{ROWMAX}\right)$$

where the minimum is taken over all the rows, is less than EPS then RMIN is set to this minimum value and IFLAG is returned with the value +I (see section 5) where I is the row in which this minimum occurs.

If the user sets EPS greater than one, then this test is not performed. In this case, and when there are no small pivots RMIN will be set equal to EPS.

MA30C

This subroutine uses only one labelled COMMON block viz.

COMMON/MA30G/RESID

RESID is a REAL (DOUBLE PRECISION in D version) variable. In the case of singular or rectangular matrices its final value will be equal to the maximum residual ($|b_i - \sum_j a_{ij}x_j|$) for the unsatisfied equations; otherwise its value will be set to zero.

7. Singular or rectangular matrices

MA30A

MA30A can perform a decomposition on matrices which are singular or rectangular. This facility is controlled by the common block variables ABORT1 and ABORT2 (see section 6).

In any singular block of the matrix, MA30A will permute a null or zero block to the end...viz.

$$P_k A_{kk} Q_k = \begin{pmatrix} A_1 & C \\ D & 0 \end{pmatrix} \overset{r_k}{\longleftrightarrow},$$

performing an LU decomposition on the first r_k rows and columns, where the $r_k \times r_k$ matrix A_1 is non-singular. The final rank returned (IRANK) equals

$\Sigma_k r_k$ where the sum is over the diagonal blocks. For each row of A in which a singularity is detected the corresponding entry in IQ is made negative.

MA30B

MA30B will produce the LU decomposition of a singular or rectangular matrix provided that the matrix was successfully decomposed by MA30A and the same singularities are found by both the MA30B and the MA30A runs. Thus the zero block of the above figure must be zero after MA30B and the block corresponding to the first r_k rows and columns must be non-singular.

MA30C

If MA30A or MA30B have been used to decompose a singular or rectangular matrix then MA30C will still perform the solution process using the information in array IQ which has a negative value for any row in which a singularity occurs. Because of the mode of operation of these earlier codes, these singularities will always occur at the ends of blocks. The maximum residual for the unsatisfied equations is given by the common block variable RESID (see section 6).

8. Other subroutines called

MA30A calls subroutine MA30D which garbage collects on arrays compressing the useful information to the top end of the array freeing earlier locations for workspace or subsequent storage. MA30D need never be called directly by the user. MA30AD calls a similar routine MA30DD.

There are no subroutines called MA30B or MA30C.

9. Portability and handling of larger arrays

These subroutines have been written in IBM Fortran. If it is desired to use them on other machines appropriate special comment cards have been included in the source code so that the preprocessor OEØ4A can be used to convert them into standard Fortran. One effect of this is to change all INTEGER*2 declarations to INTEGER thus enabling the subroutine to handle matrices up to order $2^{31}-1$ with less than 2^{31} non-zeros.

10. Method

MA30A uses a sparse variant of Gaussian elimination to decompose each diagonal block into its LU factors.

MA30B utilizes knowledge of the pivotal sequence and the sparsity structure of the LU factors from a previous call to MA30A to factorize the new matrix into its LU factors by row Gauss elimination.

MA30C performs simple forward elimination and back substitution on each block in turn performing back substitution on the off-diagonal parts when required. The code for the solution of the direct problem and its transpose are entirely separate.

A discussion of the design of these subroutines is given by Duff and Reid (Harwell Report CSS 48, 1977) while fuller details of the implementation are given in Duff (Harwell Report A.E.R.E.-R.8730,1977).

March 1977

SPECIFICATION SHEETS

FOR

SUBROUTINE

MC13D

1. Purpose

Given the column numbers of the non-zeros in each row of a sparse matrix this subroutine finds a symmetric permutation that makes the matrix block lower triangular. It can also be interpreted as accepting the row numbers of the non-zeros in each column and symmetrically permuting to block upper triangular form.

2. Argument List

CALL MC13D(N,ICN,LICN,IP,LENR,IOR,IB,NUM,IW)

N is an INTEGER variable which must be set by the user to the order of the matrix. Because of the use of INTEGER*2 arrays in MC13D the value of N should be less than 32768 (2^{15}). It is not altered by MC13D.

ICN is an INTEGER*2 array of length LICN which must be set by the user to contain the column indices of the non-zeros. Those belonging to a single row must be contiguous but the ordering of column indices within each row is unimportant and wasted space between rows is permitted. It is not changed by MC13D.

LICN is an INTEGER which must be set by the user to the length of array ICN. It is unaltered by MC13D.

IP is an INTEGER array of length N and must be set by the user so that IP(I) contains the position in array ICN of the first column index of a non-zero in row I, for I=1,2,....N. It is not altered by MC13D.

LENR is an INTEGER*2 array of length N. The user must set LENR(I) equal to the number of non-zeros in row I, I=1,2,....N. It is not changed by MC13D.

IOR is an INTEGER*2 array of length N in which the permutation is output. IOR(I) gives the position in the original ordering of the row or column which is I^{th} in the permuted form.

IB is an INTEGER*2 array of length N used for output. IB(I) contains the row number in the permuted matrix of the beginning of the I^{th} block.

NUM is an INTEGER output variable which is set to the number of blocks in the permuted form.

IW is an INTEGER*2 work array of length at least 3*N.

3. Error returns

There are no error returns. However, if the user submits a matrix with zeros on the diagonal, MC13D might give a block triangular form which could be further reduced by unsymmetric permutations. To obtain the best results, the user is advised first to permute the matrix so that it has a zero-free diagonal. This can be done by subroutine MC21A.

4. Portability and handling of larger arrays

This routine has been written in IBM Fortran. If it is desired to use it on other machines appropriate special comment cards have been included in the source code so that the preprocessor OEØ4A can be used to convert MC13D into standard Fortran. One effect of this is to change all INTEGER*2 declarations to INTEGER thus enabling the subroutine to handle matrices up to order $2^{31}-1$ with less than 2^{31} non-zeros.

5. Subroutines called

MC13D calls MC13E, which never needs to be called directly by the user.

6. Method

The method used is that of Tarjan (SIAM J. Computing (1972), 1, pp.146-160) and is described by Duff and Reid (Harwell Report CSS 29, T976).

March 1976

SPECIFICATION SHEETS

FOR

SUBROUTINE

MC20A

1. Purpose

a) MC20A: To sort the non-zeros of a sparse matrix from arbitrary order to column order, unordered within each column.

b) MC20B: To sort the non-zeros within each column of a sparse matrix stored by columns.

2. Argument lists

```
CALL MC20A(NC,MAXA,A,INUM,JPTR,JNUM,JDISP)

CALL MC20B(NC,MAXA,A,INUM,JPTR)
```

NC (INTEGER) must be set by the user to the number of matrix columns, and must not exceed 32767+JDISP. It is not altered by MC20A/B.

MAXA (INTEGER) must be set by the user to the number of matrix non-zeros. It is not altered by MC20A/B.

A is a REAL (DOUBLE PRECISION in D version) array of length MAXA. For entry to MC20A the user must set it to contain the non-zeros in any order. On exit from MC20A they are reordered so that column 1 precedes column 2 which precedes column 3, etc, but the order within columns is arbitrary. This format is required for entry to MC20B. On exit from MC20B the non-zeros are also ordered within each column.

INUM is an INTEGER*2 array of length MAXA. On entry to and exit from MC20A/B the absolute value of INUM(K) is the row number of the element in A(K). The values, including signs, are moved so the user is at liberty to use these signs as flags attached to the non-zeros.

JPTR is an INTEGER array of length NC. It is not required to be set for entry to MC20A. On exit from MC20A and on entry to and exit from MC20B it contains the position in A of the first element of column J,J=1,2,...NC.

JNUM is an INTEGER*2 array of length MAXA. On entry to MC20A JNUM(K)+JDISP is the column number of the element held in A(K). It is destroyed by MC20A.

JDISP (INTEGER) must be set by the user to his required displacement for column numbers, in the range [0,32767]. Normally zero will be suitable, but positive values permit matrices with up to 65534 columns to be handled, although JNUM is an INTEGER*2 array. JDISP is not altered by MC20A.

3. Notes

It is expected that this subroutine will be called by other library subroutines but not by the user directly. There are no checks on the validity of the data and no error exits.

4. Method

MC20A is an in-place sort algorithm which handles each item to be sorted exactly 3 times, so it is of order MAXA. The number of elements in each column is first obtained by a counting pass. The space needed by each column is allocated. Each element in turn is made the "current element" and examined to see if it is in place. If not, it is put into the next location allotted for the column it occurs in, and the element displaced made the current element. This chain of displacing elements continues until the first element examined in the chain is located and stored. Then the next item is examined. A flag prevents an element being moved twice.

MC20B is a pairwise interchange algorithm of maximum order $r(r-1)/2$, for each column, where r is the number of elements in the column.

November, 1975

SPECIFICATION SHEETS

FOR

SUBROUTINE

MC21A

1. Purpose

Given the pattern of non-zeros of a sparse matrix this subroutine finds a row permutation that makes the matrix have no zeros on its diagonal.

2. Argument List

CALL MC21A(N,ICN,LICN,IP,LENR,IPERM,NUMNZ,IW)

N is an INTEGER variable which must be set by the user to the order of the matrix. Because of the use of INTEGER*2 arrays, the value of N should be less than 32768 (2^{15}). It is not altered by MC21A.

ICN is an INTEGER*2 array of length LICN which must be set by the user to contain the column indices of the non-zeros. Those belonging to a single row must be contiguous but the ordering of column indices within each row is unimportant and wasted space between rows is permitted. It is not altered by MC21A.

LICN is an INTEGER which must be set by the user to the length of array ICN. It is not altered by MC21A.

IP is an INTEGER array of length N and must be set by the user so that IP(I) contains the position in array ICN of the first column index of a non-zero in row I,for I=1,2,...,N. It is not altered by MC21A.

LENR is an INTEGER*2 array of length N. The user must set LENR(I) equal to the number of non-zeros in row I, I=1,2,...,N. It is not altered by MC21A.

IPERM is an INTEGER*2 array of length N in which the row permutation is output. IPERM(I) gives the position in the original matrix of row I in the permuted matrix, I=1,...,N.

NUMNZ is an INTEGER output variable which gives the number of non-zeros on the diagonal of the reordered matrix. If this is less than N, the matrix is structurally singular and will be a fortiori numerically singular. For an example of this, see section 3.

IW is an INTEGER*2 work array of length at least 5*N.

3. Error returns

There are no error returns. However, the user may input a matrix for which there is no permutation that makes the diagonal zero-free. An example of this is

$$\begin{pmatrix} X & 0 \\ X & 0 \end{pmatrix}.$$

In such instances the algorithm will produce a permutation which will put as many non-zeros on the diagonal as possible (1 in the above example). This number will be output in NUMNZ. The array IPERM will still hold a permutation of the integers 1,2,...,N but in this case N-NUMNZ of the elements (IPERM(I),I) will be zero.

4. Portability and handling of larger arrays

This routine has been written in IBM Fortran. If it is desired to use it on other machines appropriate special comment cards have been included in the source code so that the preprocessor OEØ4A can be used to convert MC21A into standard Fortran. One effect of this is to change all INTEGER*2 declarations to INTEGER thus enabling the subroutine to handle matrices up to order $2^{31}-1$ with less than 2^{31} non-zeros.

5. Subroutines called

MC21A calls MC21B which never needs to be called directly by the user.

6. Method

The method used is a simple depth first search with look ahead technique and is described by Duff (Harwell report CSS 49, 1977).

April 1977

SPECIFICATION SHEETS

FOR

SUBROUTINE

MC22A

1. Purpose

This subroutine replaces a sparse matrix A by the permuted matrix PAQ. A is held by rows where, corresponding to each non-zero value, the column index is held. The elements in a single row must be contiguous but can be in any order and, for the input matrix, row I must precede row I+1,I=1,...,N-1 with no wasted space between the rows. On output, PAQ will be held in the same arrays the rows being in the order determined by P, the elements within each row being in the same order with their column indices changed according to Q (see section 4 for a diagram of this data structure).

2. Argument list

CALL MC22A(N,ICN,A,NZ,LENROW,IP,IQ,IW,IW1)

N is an INTEGER variable which must be set by the user to the order of the matrix. Because of the use of INTEGER*2 arrays, the value of N should be less than 32768 (2^{15}). It is not altered by MC22A.

ICN is an INTEGER*2 array of length at least NZ. The first NZ entries of this array must be set by the user to contain the column indices of the non-zeros in the original matrix. Those belonging to a single row must be contiguous but the ordering of column indices within each row is unimportant. The non-zerosof row I precede those of row I+1,I=1,...,N-1 and no wasted space is allowed between the rows. On output the column indices of PAQ are held in positions 1 to NZ again without any wasted space between the rows.

A is a REAL (Double Precision in D version) array of length at least NZ whose elements must be set by the user to the values of the non zero elements of the matrix in the corresponding positions in ICN. On output the elements of A will be permuted in an exactly similar fashion to those of ICN.

NZ is an INTEGER variable which must be set by the user to the number of non-zeros in the matrix.

LENROW is an INTEGER*2 array of length N. On input, LENROW(I) should be set by the user to be the number of non-zeros in row I (I=1,...,N) of the original matrix. On output LENROW will be permuted so that LENROW(I) is the number of non zeros in row I of PAQ.

IP is an INTEGER*2 array of length N which must be set by the user so that row I of PAQ was row |IP(I)| of the original matrix A, I=1,...,N. The sign of IP(I) is immaterial and the array is not altered by MC22A.

IQ is an INTEGER*2 array of length N which must be set by the user so that column I of PAQ was column |IQ(I)| in the original matrix A, I=1,...,N. The sign of IQ(I) is immaterial and the array is not altered by MC22A.

IW is an INTEGER array of length 2*N which is used as workspace by MC22A.

IW1 is an INTEGER*2 array of length NZ which is used as workspace by MC22A.

3. Parameter usage summary

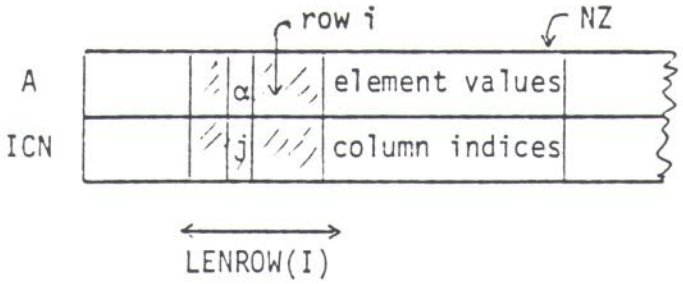
INPUT unchanged by MC22A N,NZ,IP,IQ.

INPUT changed by MC22A ICN,A,LENROW.

WORK ARRAYS IW(N), IWI(NZ)

OUTPUT from MC22A ICN,A,LENROW.

Data structure



Value of element (i,j) of A is α.

. Error returns

There are no error returns, if NZ or N are less than or equal to zero the subroutine immediately returns control to the calling program.

. Common blocks and other subroutines called

There are no common blocks or other subroutines used by MC22A.

7 Portability and handling of larger arrays

This routine has been written in IBM Fortran. If it is desired to use it on other machines appropriate special comment cards have been included in the source code so that the preprocessor OEØ4A can be used to convert MC22A/AD into standard Fortran. One effect of this is to change all INTEGER*2 declarations to INTEGER thus enabling the subroutine to handle matrices up to order 2^{31}-1 with less than 2^{31} non-zeros.

7. Method

MC22A is an in-place sort algorithm which performs the sort in O(NZ) operations.

A preliminary pass permutes LENROW, sets up a work array to identify which old row is in each new position, and calculates the amount (off-set) by which each row must be moved to achieve the desired permutation. Each position in array ICN is then examined in turn to see whether it contains the non-zero which should be there in the final form. If this is not the case, its element value and column index are stored temporarily and the element which should be in this position (accessed through the work array and corresponding offset) is placed there. The position from which this new element came now becomes the active position and the process continues in this chainlike fashion until it is found that the element which was in the original active position is required. At this stage, the information is taken from the temporary storage and the chain is complete. At each stage in the chain a flag is set in the work array to ensure the position is not processed during a subsequent scan and when an element is placed in its final position its column index is permuted according to the array IQ.

November 1976

SPECIFICATION SHEETS

FOR

SUBROUTINE

MC23A

1. Purpose

For a given matrix A, held by rows in a sparse storage format, this routine discovers if it is possible to permute the rows and columns so that the resulting matrix is in block lower triangular form. If it is possible, then the output matrix is reordered to the form PAQ, where P and Q are permutation matrices, so that non-zeros in the off-diagonal blocks preceed those in the diagonal blocks which are in order. It should be noted that if the user submits his matrix by columns, then this subroutine will produce the block upper triangular form and appropriate permutations.

2. Argument list

CALL MC23A(N,ICN,A,LICN,LENR,IDISP,IP,IQ,LENOFF,IW,IW2)

N is an INTEGER variable which must be set by the user to the order of the matrix. Because of the use of INTEGER*2 arrays, the value of N should be less than 32768(2^{15}). It is not altered by MC23A.

ICN is an INTEGER*2 array of length LICN which must be set by the user to contain the column indices of the non-zeros. Those belonging to a single row must be contiguous but the ordering of column indices within each row is unimportant. The non-zeros of row I preceed those of row I+1,I=1,...,N-1 and no wasted space is allowed between the rows. On a successful output, information is held in positions 1 to IDISP(1)-1 and IDISP(2) to LICN of array ICN. The permuted column indices of the rows in the diagonal part are held in IDISP(2) to LICN, with the rows in permuted order, and no spaces wasted between the rows. If there is more than one diagonal block, the original column indices of the parts of the rows in off-diagonal blocks will be found in original row order in positions 1 to IDISP(1)-1, again with no wasted space between rows.

A is a REAL (DOUBLE PRECISION in D version) array of length LICN whose elements must be set by the user to the values of the non-zero elements of the matrix in the corresponding positions in ICN. On output it will be reordered similarly to ICN.

LICN is an INTEGER variable which must be set by the user to the length of arrays ICN and A. It is not altered by MC23A.

LENR is an INTEGER*2 array of length N. The user must set LENR(I) equal to the number of non-zeros in row I,I=1,2,...,N. On output from MC23A,LENR(I) I=1,2,...,N is equal to the number of non-zeros in the part of row I (of the permuted matrix) in its diagonal block.

IDISP is an INTEGER array of length 2. On output from MC23A, IDISP(1) is the position in arrays ICN and A of the first location after the off-diagonal blocks (equal to one if there are no blocks) while IDISP(2) is the position in ICN and A of the first non-zero in the diagonal blocks. (See section 4). In the event of an error (see section 5) IDISP(1) is returned with a negative value.

IP is an INTEGER*2 array of length N. On output from MC23A |IP(I)|, I=1,...,N is the original row number of the row which is Ith in the permuted form. The last row in the permuted form of each diagonal block (except the last) is indicated by a negative value for IP.

IQ is an INTEGER*2 array of length N. On output from MC23A, IQ(I), I=1,2,...,N holds the original column number of the column which is Ith in the permuted form.

LENOFF is an INTEGER*2 array of length N. On output from MC23A,LENOFF(I), I=1,2,...,N holds the number of non-zeros in the part of row I in the off-diagonal blocks, reference being to the original row number. However, if the matrix has only one diagonal block LENOFF(1) is set to -1, the other entries being unimportant.

IW is an INTEGER*2 array of length 5*N and is used as workspace by MC23A.

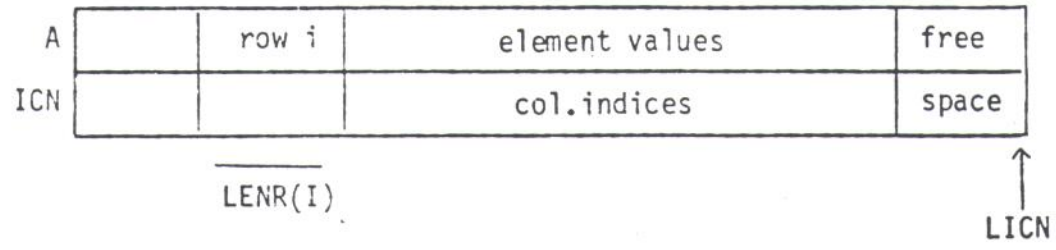
IW1 is an INTEGER array of length 2*N which is used as workspace by MC23A.

3. Parameter usage summary

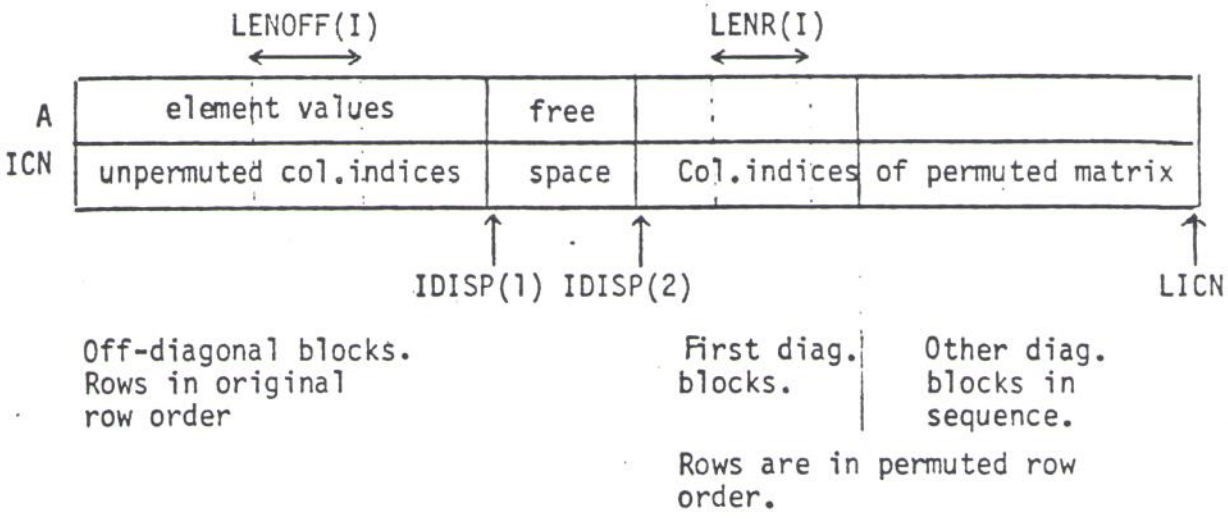
INPUT by user N,ICN,A,LICN,LENR.
UNCHANGED by MC23A. N,LICN.
WORK ARRAYS IW(5*N),IW1(2*N).
OUTPUT FROM MC23A. ICN,A,LENR,IDISP,IP,IQ,LENOFF.

4. Data Structures

On input



On output



5. Error return

There are two error returns indicated by a value of IDISP(1) equal to -1 or -2 although one of them is only invoked if the user changes value of the COMMON variable ABORT to .TRUE. (see section 6).

If ABORT equals .TRUE. and it is impossible to permute the matrix so that the diagonal of the permuted matrix is free from zeros, then the matrix is structurally singular (and a fortiori numerically singular) and the subroutine will terminate immediately setting IDISP(1) to -1 and printing the message

ERROR RETURN FROM MC23A BECAUSE MATRIX IS STRUCTURALLY SINGULAR,RANK=<r>.

where r is the calculated value of the structural rank, on unit LP (see section 6).

The second possible error return, which cannot be switched off by the user, occurs if there is insufficient space in A,ICN for the subroutine to perform its reordering. The subroutine does not require much "elbow room" needing a value of LICN equal to at most NZ+N, where NZ is the number of non-zeros in the matrix. If the user has allowed insufficient space the message:

ERROR RETURN FROM MC23A BECAUSE LICN NOT BIG ENOUGH INCREASE BY <N>

is output on unit LP (see section 6).

6. Common blocks

A labelled common block

```
COMMON/MC23B/LP,NUMNZ,NUM,LARGE,ABORT
```

is used. The INTEGER LP is used in the output described in section 5 and has a value of 6 set by a BLOCK DATA subprogram. This default value can be reset by the user if it is desired to surpress (LP=0) or reroute this output.

The INTEGER variable NUMNZ is set by MC23A to the number of non-zeros of A which can be permuted onto the diagonal of the matrix. If NUMNZ does not equal N, the matrix is structurally singular.

The INTEGER variable NUM is set by MC23A to the number of diagonal blocks present in the permuted form.

The INTEGER variable LARGE is set by MC23A to the order of the largest diagonal block in the permuted form.

The LOGICAL variable ABORT is used by the routine to decide if it should stop its reordering if the matrix is structurally singular. It will do so if ABORT has the value .TRUE. Its default value, set in a BLOCK DATA subprogram, is .FALSE.

7. Portability and handling of larger arrays

This routine has been written in IBM Fortran. If it is desired to use it on other machines appropriate special comment cards have been included in the source code so that the preprocessor OEØ4A can be used to convert MC23A/AD into standard Fortran. One effect of this is to change all INTEGER*2 declarations to INTEGER thus enabling the subroutine to handle matrices up to order $2^{31}-1$ with less than 2^{31} non-zeros.

8. Subroutines called

MC23A calls Harwell subroutines MC13D,MC13E,MC21A and MC21B. The same subroutines which need never be called directly by the user are called by MC23AD.

If the user is using this routine prior to solving sets of linear equations and wishes to have a better user interface which combines block triangularization with factorization of his matrix, then he should use MA28A/AD which calls MC23A/AD internally.

9. Subroutine MC23A is really a data handling routine. The real work is done by subroutine MC21A/B (Duff, Harwell report) which finds a column permutation to make the diagonal zero free and MC13D/E (Duff and Reid, Harwell Report CSS.29,1976) which finds a subsequent symmetric permutation to put the matrix into block lower triangular form.

February 1977

SPECIFICATION SHEETS

FOR

SUBROUTINE

MC24A

1. Purpose

To obtain an estimate of the largest element encountered during Gaussian elimination on a sparse matrix A of order n from the LU factors obtained. If the matrix has been previously scaled (by using, for example, MC19A/AD), then this estimate will give an indication of the numerical accuracy of the decomposition.

2. How to use the subroutine

2.1 Calling sequence and argument list

The single precision version:

```
CALL MC24A(N,ICN,A,LICN,LENR,LENRL,W)
```

The double precision version:

```
CALL MC24AD(N,ICN,A,LICN,LENR,LENRL,W).
```

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: $1 \leq n \leq 32767$.

ICN is an INTEGER*2 array of length LICN which must contain the column indices of the non-zeros in the decomposition. Each row (of L and U) is held contiguously. Row I precedes row I+1,I=1,...,N-1 and there is no wasted space between the rows. Although the column indices need not be in order, those in L must precede those in U with the pivot being the first entry in the row of U. It is not altered by the subroutine.

A is a REAL (DOUBLE PRECISION in D version) array of length LICN which contains the values of the non-zero elements in the LU decomposition. The non-zero held in A(K) is in column ICN(K). It is not altered by the subroutine.

LICN is an INTEGER variable which must be set by the user to be the length of arrays ICN and A. It is not altered by the subroutine.

LENR is an INTEGER*2 array of length N. LENR(I) must be set by the user to the combined number of non-zeros in rows I of L and U, I=1,...,N. It is not altered by the subroutine.

LENRL is an INTEGER*2 array of length N. LENRL(I) must be set by the user to the number of non-zeros in row I of L, I=1,2,...,N. It is not altered by the subroutine.

Available from

HER MAJESTY'S STATIONERY OFFICE

49 High Holborn, London, WC1V 6HB
P.O. Box 569, London, SE1 9NH (Trade and London area mail orders)

13a Castle Street, Edinburgh, EH2 3AR
41 The Hayes, Cardiff, CF1 1JW
Brazennose Street, Manchester, M60 8AS
Southey House, Wine Street, Bristol, BS1 2BQ
258 Broad Street, Birmingham, B1 2HE
80 Chichester Street, Belfast, BT1 4JY
or any bookseller

Printed in England

I.S.B.N.-0-70-580593-X