RAL-95-001

MA47, a Fortran code for direct solution of indefinite sparse symmetric linear systems

by

I. S. Duff and J. K. Reid

Abstract

We describe the design of a new code for the solution of sparse indefinite symmetric linear systems of equations. It is intended to complement the Harwell code `MA27`. The principal difference between the two codes lies in the exploitation by `MA47` of the additional sparsity available when the matrix has some zero diagonal entries. Other features have been included to enhance the execution speed, particularly on vector and parallel machines.

Central Computing Department,
Rutherford Appleton Laboratory,
Oxon OX11 0QX.

January 1995.

CONTENTS

1 Introduction 1

1.1 MA47I: initialization 2

2 The algorithm 3

2.1 The design of the factorized form 6

2.2 Analyse phase 7

2.3 Factorization 10

2.4 Solution 12

2.5 The singular case 13

2.6 The complex case 14

3 Analysis of the sparsity pattern 14

3.1 MA47A: driver for the analyse phase 14

3.2 MA47F: compress analyse data structure 15

3.3 MA47G: construction of the sparsity pattern by rows 16

3.4 MA47H: Markowitz analysis 17

3.4.1 The data structures used 17

3.4.2 The execution of the analysis code 19

3.5 MA47J: construction of the upper-triangular sparsity pattern by rows........ 21

3.6 MA47K: analysis for a given pivotal sequence 22

3.7 MA47L: depth-first search of the assembly tree 23

3.8 MA47M: calculate storage and operation counts 25

3.9 MA47N: construct map for sorting the entries 25

3.10 MA47T: recognition of supervariables 26

3.11 MA47V: raise the row-count threshold 27

3.12 MA47Z: reset the array of flags 27

4 Numerical factorization 27

4.1 MA47B: driver for the numerical factorization phase 29

4.2 MA47O: numerical factorization 29

4.2.1 Basic structure of MA47O 30

4.2.2 Some data structures 31

4.2.3 Step 1. Symbolic assembly........ 32

4.2.4 Step 2. Numerical assembly 33

4.2.5 Step 3. Definition of pivot and degree calculation 34

4.2.6 Step 4, first part. Selection of structured pivots........ 34

4.2.7 Step 4, second part. Selection of full pivots 35

4.2.8 Step 5. Structured pivot: column sort........ 35

4.2.9 Step 6. Reserving space for Schur update 36

4.2.10 Step 7, part 1. Generation of Schur update for a structured pivot 37
4.2.11 Step 7, part 2. Generation of Schur update for a full pivot 37
4.2.12 Step 8. Absorption of elements 38
4.2.13 Step 9. Stacking the generated element 38
4.2.14 Step 10. Stacking the fully-summed block 38
4.2.15 Step 11. Reorganization of the data structure for the solve routine 38
4.3 MA47P and MA47S: data compression routine........ 40
4.4 MA47U: print the factors 40
4.5 MA47W: postprocess the factors from full pivots........ 40
4.6 MA47X and MA47Y: auxiliary routines. 41

5 Solution 41
5.1 MA47C: driver for the solution phase 41
5.2 MA47Q: forward substitution........ 41
5.3 MA47R: back-substitution 42

6 Numerical Experience 43
6.1 Introduction........ 43
6.2 Effect of restricting pivot selection 45
6.3 Effect of change in pivot threshold 46
6.4 Effect of node amalgamation 47
6.5 Effect of change of block size for Level 3 BLAS during factorization 48
6.6 Effect of change of block size for level 2 BLAS during solution 49
6.7 Performance of MA47 and comparison with MA27 50

7 References 53

Appendix. The specification document 55

1 Introduction

This report describes the MA47 collection of Fortran subroutines for the direct solution of sparse symmetric sets of n linear equations

$$\mathbf{Ax}=\mathbf{b}, \tag{1.1}$$

when the matrix **A** is symmetric and indefinite. The code will work for the definite case but there are many opportunities for simplifications and efficiency improvements, so we plan to provide a separate code for this special case. The code uses a multifrontal method in a similar way to MA27 (Duff and Reid 1982 and 1983) but aims to take advantage of the additional sparsity when there are zeros on the diagonal of the matrix **A**.

To accommodate the indefiniteness, we use block pivots to produce a factorization

$$\mathbf{A}=\mathbf{P}\,\mathbf{L}\,\mathbf{D}\,\mathbf{L}^T\mathbf{P}^T \tag{1.2}$$

where **P** is a permutation, **L** is unit lower triangular, and **D** is block diagonal. Some of the blocks correspond to the use of 2×2 pivots (Bunch and Parlett 1971). A tentative choice of the permutation and blocking is made by working symbolically from the zero/nonzero structure of the matrix **A**. We call this the ***analyse*** phase. Given the results from this phase, we ***factorize*** a matrix of the given structure, including symmetric permutations for the sake of numerical stability, but trying to keep close to the tentative pivotal sequence and block structure. Finally, we use the factorization to ***solve*** for a particular vector **b**.

Throughout this paper, we use the term ***entry*** for a matrix coefficient that is nonzero or might be nonzero. Note that sometimes an entry may have the value zero but a coefficient that is not an entry is always zero. If a coefficient of the reduced matrix is obtained by modification of an entry, we regard the result as an entry even it it has the value zero since it might be nonzero for another matrix of the same pattern. Also, the user may find it convenient to treat a zero as an entry during the analyse phase in anticipation of a later matrix having a nonzero in the corresponding position.

The subroutines are named according to the naming convention of the Harwell Subroutine Library (Anon 1993). We describe the single-precision versions which have names that commence with MA47 and have one more letter. The corresponding double-precision versions have an additional letter D. The code itself is available from AEA Technology, Harwell; the contact is Libby Thick, Harwell Subroutine Library, B 8.19 Harwell, Didcot, Oxon OX11 0RA, tel (44) 235 432688, fax (44) 235 432989, email libby.thick@aea.orgn.uk, who will provide details of price and conditions of use.

MA47 accepts an $n \times n$ symmetric sparse matrix whose entries are stored in any order in the array A with their row and column indices stored in corresponding positions in the arrays IRN and JCN. Each diagonal entry a_{ii} is represented by A(k), IRN(k) = i, and JCN(k) = i, for some

k. Each pair of off-diagonal entries a_{ij} and a_{ji} is represented by `A`(k) with `IRN`(k) = i and `JCN`(k) = j or `IRN`(k) = j and `JCN`(k) = i, for some k. Multiple entries are permitted and are summed. This is the most user-friendly format that we have been able to devise, and is the same as that of `MA27`.

We describe the algorithm in Section 2. There are four subroutines that are called directly by the user:

Initialize. `MA47I` provides default values for the arrays `CNTL` and `ICNTL` that together control the execution of the package. This subroutine is described in Section 1.1.

Analyse. `MA47A` accepts the pattern of **A** and makes a tentative choice of block pivots. It also calculates other data needed for actual factorization. The user may provide a pivotal sequence, in which case the necessary data will be generated. This subroutine and those called by it are described in Section 3.

Factorize. `MA47B` accepts a matrix **A** together with a set of recommended block pivots. It performs the factorization, including additional permutations when they are needed for numerical stability. This subroutine and those called by it are described in Section 4.

Solve. `MA47C` uses the factorization produced by `MA47B` to solve the equations $\mathbf{Ax} = \mathbf{b}$. This subroutine and those called by it are described in Section 5.

Section 6 is devoted to our experience of the actual running of the code. The specification document is included as an appendix.

The most significant difference from `MA27` is the treatment of generated matrices as other than full. Another important difference is the use of BLAS, but we plan to provide an enhanced version of `MA27` that includes these.

1.1 MA47I: initialization

`MA27` makes extensive use of `COMMON` for parameters that control the actions or provide information for the user. Default values are set by `BLOCK DATA` so that the user has to take explicit action only to obtain particular information or to set a control parameter to a non-default value. This format is not well-matched to the requirements of parallel processing, where several copies of the routines may be executing at once. We have therefore changed to having array arguments for this purpose, with an initialization routine to provide default values.

Subroutine `MA47I` provides default values for the arrays `CNTL` and `ICNTL` that together control the execution of the package. Their purposes and default values are given in the appendix.

2 The algorithm

Our algorithm is based on the work of Duff, Gould, Reid, Scott, and Turner (1991). We use block pivots that may be

(i) of the form

$$\begin{pmatrix} \mathbf{0} & \mathbf{A}_1 \\ \mathbf{A}_1^T & \mathbf{0} \end{pmatrix} \tag{2.1}$$

with $\mathbf{A}_1$ square, which we call an ***oxo*** pivot;

(ii) of the form

$$\begin{pmatrix} \mathbf{A}_2 & \mathbf{A}_1 \\ \mathbf{A}_1^T & \mathbf{0} \end{pmatrix} \quad \text{or} \quad \begin{pmatrix} \mathbf{0} & \mathbf{A}_1 \\ \mathbf{A}_1^T & \mathbf{A}_2 \end{pmatrix} \tag{2.2}$$

with $\mathbf{A}_1$ square, which we call a ***tile*** pivot; or

(iii) of any other form, which we call ***full.***

We will use the term ***structured*** for a pivot that is either a tile or an oxo pivot. The blocks $\mathbf{A}_1$ and $\mathbf{A}_2$ are usually full and we always store them as full matrices.

The matrix modifications of a block pivotal step that lie outside the pivot rows and columns (the Schur update) are not applied at the time but are stored in a ***generated element matrix.*** This has entries only in a principal submatrix that after permutation has the general form

$$\begin{pmatrix} \mathbf{0} & \mathbf{B}_2 & \mathbf{B}_3 \\ \mathbf{B}_2^T & \mathbf{B}_1 & \mathbf{B}_4 \\ \mathbf{B}_3^T & \mathbf{B}_4^T & \mathbf{0} \end{pmatrix}, \tag{2.3}$$

where the blocks on the diagonal are square. This is the form of the submatrix altered by a pivotal step with an oxo pivot and is illustrated in Figure 2.1. The blocks $\mathbf{B}_1$, $\mathbf{B}_2$, $\mathbf{B}_3$, and $\mathbf{B}_4$ are usually full and we always store them as full matrices. A tile pivot produces the special case of this form where the first or last block row and column is null. It is illustrated in Figure 2.2. For a full pivot, the generated element is held as a full matrix, which is the special case where the first and last block row and column are both null.

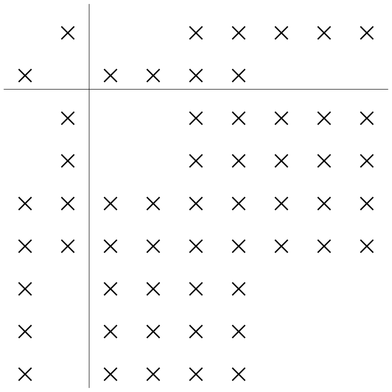

Figure 2.1. An oxo pivot, its pivot rows and columns, and fill-in pattern (generated element).

$$\begin{array}{cc|ccccccc}
\times & \times & & & \times & \times & \times & \times & \times \\
\times & & \times & \times & \times & \times & & & \\
\hline
 & \times & \times & \times & \times & \times & \times & \times & \times \\
 & \times & \times & \times & \times & \times & \times & \times & \times \\
\times & \times & \times & \times & \times & \times & \times & \times & \times \\
\times & \times & \times & \times & \times & \times & \times & \times & \times \\
\times & & \times & \times & \times & \times & & & \\
\times & & \times & \times & \times & \times & & & \\
\times & & \times & \times & \times & \times & & &
\end{array}$$

Figure 2.2. A tile pivot, its pivot rows and columns, and fill-in pattern (generated element).

We have chosen the multifrontal technique (Duff and Reid 1983) for the sake of efficiency during the analyse phase and to permit extensive use of full-matrix code and the BLAS (Basic Linear Algebra Subprograms: Lawson, Hanson, Kincaid, and Krogh 1979, Dongarra, Du Croz, Hammarling, and Hanson 1988, Dongarra, Du Croz, Duff, and Hammarling 1990) during factorization. We will use the notation $\mathbf{B}^{(l)}$ for the generated element matrix from the l–th (block) pivotal step and the notation $\mathbf{A}_k$ and $\mathbf{B}_k^{(l)}$ to denote the submatrices of $\mathbf{A}$ and $\mathbf{B}^{(l)}$ obtained by removing the rows and columns corresponding to the first k (block) pivotal steps. Following (block) step k, the reduced matrix is held as

$$\mathbf{A}_k + \sum_{l \in \mathbf{I}_k} \mathbf{B}_k^{(l)} \tag{2.4}$$

where $\mathbf{I}_k$ is the set of indices of element matrices that are active then. If $\mathbf{B}_{k-1}^{(l)}$ has entries only in the pivotal rows and columns, $\mathbf{B}_k^{(l)}$ will be zero and l is omitted from the index set $\mathbf{I}_k$. Other $\mathbf{B}_k^{(l)}$ may have entries that lie entirely within the pattern of the newly generated element $\mathbf{B}_k^{(new)}$; for efficiency, such a $\mathbf{B}_k^{(l)}$ is added into $\mathbf{B}_k^{(new)}$ and l is omitted from $\mathbf{I}_k$. We say that $\mathbf{B}_k^{(l)}$ is ***absorbed*** into $\mathbf{B}_k^{(new)}$.

Such absorption certainly takes place if the pivot is full and overlaps one or more of the diagonal entries of $\mathbf{B}_k^{(l)}$ since in this case the pivot row has an entry for every index of $\mathbf{B}_k^{(l)}$. If all the pivots are full, all generated elements are full and therefore any generated element that is involved in a pivotal step is absorbed. This is the situation for a definite matrix.

In the definite case, the whole process may be represented by a tree, known as the ***assembly tree***, which has a node for each block pivotal step. The sons of a node correspond to the element matrices that contribute to the pivotal row(s) and are absorbed in the generated element. Here, it is efficient to add all the generated elements from the sons and the pivot rows from the original matrix into a temporary matrix known as the ***frontal matrix,*** which can be held in a square array of size the number of rows and columns with entries involved. The rows and columns are known as the ***front.*** For a fuller description of this case, see Duff, Erisman, and Reid (1986), Sections 10.5 to 10.9.

When there are some structured pivots, we employ a similar assembly tree, but a generated element is not necessarily absorbed at its father node. Instead, it may persist for several generations, making contributions to several pivotal rows, until it is eventually absorbed. As an illustration of absorption not occurring, a simple 1×1 pivot might overlap the leading (zero) block of (2.3). In such a case, $\mathbf{B}_1$ and $\mathbf{B}_4$ are absorbed but the non-pivotal rows of $\mathbf{B}_2$ and $\mathbf{B}_3$ are inactive (unless made active by entries from elsewhere). Absorption occurs for a structured pivot if an off-diagonal entry of $\mathbf{B}_k^{(l)}$ overlaps the off-diagonal block $\mathbf{A}_1$ of the pivot. This is seen by regarding the structured pivot as a sequence of 1×1 pivots starting with the entry and its symmetrically placed partner. To handle the structured case efficiently, we sum only the contributions to the pivot rows, form the Schur update, and then add into it any generated elements from the sons that can be absorbed. The frontal matrix is thus more complicated, but we still refer to the set of rows and columns involved as the front.

Previously (Duff, Gould, Reid, Scott, and Turner 1991), we had anticipated working with generated elements (after permutation) of the forms

$$\begin{pmatrix} \mathbf{B}_1 & \mathbf{B}_2 \\ \mathbf{B}_2^T & \mathbf{0} \end{pmatrix} \quad \text{and} \quad \begin{pmatrix} \mathbf{0} & \mathbf{B}_3 \\ \mathbf{B}_3^T & \mathbf{0} \end{pmatrix}. \tag{2.5}$$

A tile generated element is of the first form and an oxo generated matrix can be represented as the sum of a matrix of the first form plus one of the second form:

$$\begin{pmatrix} \mathbf{0} & \mathbf{B}_2 & \mathbf{B}_3 \\ \mathbf{B}_2^T & \mathbf{B}_1 & \mathbf{B}_4 \\ \mathbf{B}_3^T & \mathbf{B}_4^T & \mathbf{0} \end{pmatrix} = \begin{pmatrix} \mathbf{0} & \mathbf{B}_2 & \mathbf{0} \\ \mathbf{B}_2^T & \mathbf{B}_1 & \mathbf{B}_4 \\ \mathbf{0} & \mathbf{B}_4^T & \mathbf{0} \end{pmatrix} + \begin{pmatrix} \mathbf{0} & \mathbf{0} & \mathbf{B}_3 \\ \mathbf{0} & \mathbf{0} & \mathbf{0} \\ \mathbf{B}_3^T & \mathbf{0} & \mathbf{0} \end{pmatrix}. \tag{2.6}$$

We have decided to use the form (2.3) because, in the case of an oxo generated element,

(i) the duplication of the index lists of the first and third blocks is avoided;

(ii) for a row of the first or third block, one rather than two elements involve it and need to be included in a list of elements associated with the row (such lists are needed during the analyse phase); and

(iii) a link would need to be maintained between the two parts of an oxo generated element in order to recognize that both parts of the element can be absorbed when an off-diagonal entry overlaps the off-diagonal block $\mathbf{A}_1$ of a structured pivot.

2.1 The design of the factorized form

Two considerations had a profound effect on our design for the factorized form:

(i) the wish to use block operations during both factorization and solution, and

(ii) the wish to be able readily to modify the factorization so that it is a factorization of a positive definite matrix.

For a full pivot, this is easy to achieve. If the first pivot $\mathbf{A}_{11}$ is full, we factorize it as

$$\mathbf{A}_{11} \;=\; \mathbf{L}\,\mathbf{D}\,\mathbf{L}^T, \tag{2.7}$$

where **L** is unit lower triangular and **D** is a block diagonal matrix with blocks of order 1 or 2, and have the matrix factorization

$$\begin{pmatrix} \mathbf{A}_{11} & \mathbf{A}_{12} \\ \mathbf{A}_{12}^T & \mathbf{A}_{22} \end{pmatrix} \;=\; \begin{pmatrix} \mathbf{L} & \\ \mathbf{M}^T & \mathbf{I} \end{pmatrix} \begin{pmatrix} \mathbf{D} & \\ & \mathbf{A}_{22} - \mathbf{S} \end{pmatrix} \begin{pmatrix} \mathbf{L}^T & \mathbf{M} \\ & \mathbf{I} \end{pmatrix}, \tag{2.8}$$

where **M** is the matrix of multipliers

$$\mathbf{M} \;=\; \mathbf{D}^{-1}\mathbf{L}^{-1}\mathbf{A}_{12} \tag{2.9}$$

and $\mathbf{A}_{22} - \mathbf{S}$ is the Schur complement, where

$$\mathbf{S} \;=\; \mathbf{M}^T\mathbf{D}\,\mathbf{M} \;=\; \mathbf{A}_{12}^T\mathbf{A}_{11}^{-1}\mathbf{A}_{12} \tag{2.10}$$

is the Schur update. **D** may be perturbed to a positive-definite matrix by examining its (1×1 and 2×2) diagonal blocks and changing the diagonal entries as necessary.

A comparable factorization to (2.7) for a block tile pivot is

$$\begin{pmatrix} \mathbf{0} & \mathbf{A}_1 \\ \mathbf{A}_1^T & \mathbf{A}_2 \end{pmatrix} \;=\; \begin{pmatrix} \mathbf{L}_1 & \\ \mathbf{U}_2^T & \mathbf{U}_1^T \end{pmatrix} \begin{pmatrix} \mathbf{0} & \mathbf{D}_1 \\ \mathbf{D}_1 & \mathbf{D}_2 \end{pmatrix} \begin{pmatrix} \mathbf{L}_1^T & \mathbf{U}_2 \\ & \mathbf{U}_1 \end{pmatrix}, \tag{2.11}$$

where $\mathbf{L}_1$ is unit lower triangular, $\mathbf{U}_1$ is unit upper triangular, $\mathbf{U}_2$ is strictly upper triangular, and $\mathbf{D}_1$ and $\mathbf{D}_2$ are diagonal. The special case $\mathbf{U}_2 = \mathbf{D}_2 = \mathbf{0}$ corresponds to a block oxo pivot. Note that the relationship

$$\mathbf{A}_1 \;=\; \mathbf{L}_1\,\mathbf{D}_1\,\mathbf{U}_1 \tag{2.12}$$

holds, so that a conventional triangular factorization of $\mathbf{A}_1$ is included. That this is a correct factorization can be seen by performing a symmetric permutation to place the rows and columns in the order 1, r+1, 2, r+2, 3, r+3, ..., where 2r is the order of the matrix in equation (2.11). This gives the block tile form of the symmetric matrix

$$\begin{pmatrix} t_{11} & t_{12} & . & . & t_{1r} \\ t_{21} & t_{22} & . & . & t_{2r} \\ . & . & . & . & . \\ . & . & . & . & . \\ t_{r1} & t_{r2} & . & . & t_{rr} \end{pmatrix} \tag{2.13}$$

where each t_{ij} is a 2×2 matrix whose (1,1) element is zero (each t_{ij} is a tile). Using $t_{11}, t_{22}, ..., t_{rr}$ as pivots, this matrix has the factorization

$$\begin{pmatrix} \mathbf{I} & & & & \\ l_{21} & \mathbf{I} & & & \\ . & . & . & & \\ . & . & . & . & \\ l_{r1} & l_{r2} & . & . & \mathbf{I} \end{pmatrix} \begin{pmatrix} d_{11} & & & & \\ & d_{22} & & & \\ & & . & & \\ & & & . & \\ & & & & d_{rr} \end{pmatrix} \begin{pmatrix} \mathbf{I} & l_{21}^T & . & . & l_{r1}^T \\ & \mathbf{I} & . & . & l_{r2}^T \\ & & . & . & . \\ & & & . & . \\ & & & & \mathbf{I} \end{pmatrix}, \qquad (2.14)$$

where each l_{ij} has a zero in position (2,1) and each d_{ii} is a symmetric tile. If we now apply the inverse permutation, we get the form (2.11) with the lower triangular entries of $\mathbf{L}_1$, $\mathbf{U}_2^T$, $\mathbf{U}_1^T$, respectively, being the (1,1), (2,1), (2,2) entries of l_{ij}, and the diagonal entries of $\mathbf{D}_1$ and $\mathbf{D}_2$, respectively, being the (2,1) and (2,2) entries of d_{ii}. An alternative derivation of (2.11) is by application of a sequence of elementary row and column operations that reduce $\begin{pmatrix} \mathbf{0} & \mathbf{A}_1 \\ \mathbf{A}_1^T & \mathbf{A}_2 \end{pmatrix}$ to the form $\begin{pmatrix} \mathbf{0} & \mathbf{D}_1 \\ \mathbf{D}_1 & \mathbf{D}_2 \end{pmatrix}$ in column 1, row 1, column r+1, row r+1, column 2, row 2, column r+2, row r+2, ...

The diagonal entries of $\begin{pmatrix} \mathbf{0} & \mathbf{D}_1 \\ \mathbf{D}_1 & \mathbf{D}_2 \end{pmatrix}$ can be changed to make it nonsingular as easily as those of $\mathbf{D}$ in (2.7). Thus, we may regard (2.7) as applicable to the tile case with

$$\mathbf{L} = \begin{pmatrix} \mathbf{L}_1 & \\ \mathbf{U}_2^T & \mathbf{U}_1^T \end{pmatrix} \quad \text{and} \quad \mathbf{D} = \begin{pmatrix} \mathbf{D}_3 & \mathbf{D}_1 \\ \mathbf{D}_1 & \mathbf{D}_2 \end{pmatrix}. \qquad (2.15)$$

The special case $\mathbf{U}_2 = \mathbf{0}$ corresponds to an oxo pivot.

We therefore store, for each block pivot, **L**, **D**, and **M**. The matrices **L** and **D** may be packed to take advantage of their form. For an oxo pivot, the nonzero columns of **M** are ordered to the form

$$\mathbf{M} = \begin{pmatrix} \mathbf{B}_5 & \mathbf{B}_6 & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{B}_7 & \mathbf{B}_8 & \mathbf{0} \end{pmatrix}. \qquad (2.16)$$

and only the blocks $\mathbf{B}_5$, $\mathbf{B}_6$, $\mathbf{B}_7$, and $\mathbf{B}_8$ are stored. For a tile pivot, the first block column is absent. These forms are discussed further in Section 2.3.

2.2 Analyse phase

In the analyse phase, we simulate the operations of the factorization, representing each generated element with an index list and assuming that every chosen pivot is acceptable numerically. This allows the processing to be efficient and there is no need for any numerical values, but the pivotal sequence chosen has to be regarded as tentative. The analyse phase is faster than it would be if numerical values were taken into account and its storage demands are much more modest.

For pivotal starategy we use the variant of the Markowitz (1957) criterion recommended by Duff, Gould, Reid, Scott, and Turner (1991). The Markowitz cost

$$(r_i - 1)^2 \tag{2.17}$$

for a diagonal entry a_{ii}, with row count r_i (number of entries in the row), is extended to

$$(r_i - 1)\,(r_i + r_j - 3) \tag{2.18}$$

for a tile pivot, and

$$(r_i - 1)\,(r_j - 1) \tag{2.19}$$

for an oxo pivot. We also provide an option to limit the search to a given number of rows with least entries.

We found it convenient to build a tree that represents the grouping of variables into blocks as well as the assemblies. It has a node for every variable, rather than for every block pivot. In the positive-definite case, this bigger tree is the elimination tree (see, for example, Liu 1990), so we will call it the ***elimination tree*** in this more general setting. The variables of a full pivot or of one of the halves of a structured pivot are linked together in a chain whose head we call the ***principal variable.*** The two principal variables of a structured pivot are linked together as father and son. Only these father nodes and the principal variables of full pivots have a node in the assembly tree.

One of the ways to speed the analyse phase is to recognize rows with the same structure, both in the original matrix and the successive reduced matrices. The set of variables that correspond to such a set of rows is called a ***supervariable*** and we represent the matrix pattern in terms of supervariables. We allow a supervariable to consist of a single variable. If the rows do not have entries on the diagonal, we say that the supervariable is ***defective***. Each supervariable is indexed by one of its variables, which is its principal variable.

A simple example of a matrix and its elimination tree is shown in Figure 2.3. Here there are just two block pivots; the first is an oxo pivot of order 6 with variable 4 as its principal variable and the second is a full pivot of order 2 with variable 7 as its principal variable. Variable 1 is the principal variable of the other half of the oxo pivot and node 1 is the son of node 4 in the elimination tree. The other variables are linked in chains to the three principal variables.

$$\begin{array}{cccccccc}
 & & & \times & \times & \times & \times & \times \\
 & & & \times & \times & \times & \times & \times \\
 & & & \times & \times & \times & \times & \times \\
\times & \times & \times & & & & \times & \times \\
\times & \times & \times & & & & \times & \times \\
\times & \times & \times & & & & \times & \times \\
\times & \times & \times & \times & \times & \times & \times & \times \\
\times & \times & \times & \times & \times & \times & \times & \times
\end{array}$$

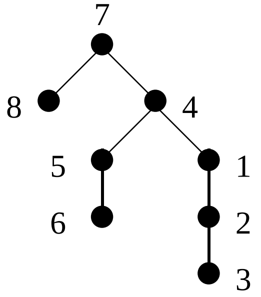

Figure 2.3. A matrix pattern with its elimination tree.

We begin the analyse phase by recognizing rows with identical structure and forming supervarariables. With careful choice of data structures (see Section 3.10), the amount of work performed for each matrix entry is bounded by a constant, so the whole process executes in $O(n+\tau)$ time, where τ is the number of entries. This leaves us with a forest with a node for each variable and the variables of each supervariable in a chain with the principal variable at its head.

This forest is gradually modified until it becomes the elimination tree. At an intermediate stage, each tree of the forest has a root that represents the principal variable of either

(i) a supervariable that has not yet been pivotal, or

(ii) a supervariable that has been pivotal and whose generated element matrix is zero or has not yet contributed to a pivotal row.

The node of a principal variable that has been eliminated may be regarded as also representing the element generated when it was eliminated. When a pivot block is chosen, the node of its principal variable is given as sons the roots of all the trees that contain one or more generated elements that contribute to the pivot rows. If the pivot is a structured pivot, it is also given as a son the principal variable of the second part. Some merging of supervariables may follow the pivotal step, since the fill-ins may cause some rows to have identical structures in the reduced matrix.

We have noted that a generated element matrix need not be absorbed at its father node. Indeed, it may persist for many generations of the tree, contributing to pivotal rows at each stage. This may mean that all that is left is a zero matrix, which is all or part of one of the zero blocks of (2.3). It might be thought that such a zero matrix could be discarded, but to do so risks the loss of an important property of an assembly tree: the block pivotal steps may be reordered provided those at a node follow those at all descendants of the node. We must therefore retain such a zero element matrix and whenever one of its variables becomes pivotal, make the root of its tree a son of the pivot node.

In the irreducible case, this eventually yields the elimination tree, whose root is the only node with a zero generated element matrix. The node represents the final block pivot, which obviously leaves a null reduced matrix. If the original matrix is reducible, it must be block diagonal since it is symmetric. In this case, the problem consists of several independent subproblems and there will be a tree for each. It is not difficult to allow for this and our code does so. For simplicity of description, however, we assume that the matrix is irreducible.

Given an elimination tree, there is considerable freedom in the ordering of the block pivotal steps during an actual matrix factorization. The operations are the same for any ordering such that the pivotal operations at a node follow those at a descendant of the node. Subject to this requirement, the order may be chosen for organizational convenience. We base ours on postordering following a depth-first search of the tree, which allows a stack to be used to store the generated elements awaiting assembly. Depth-first searches of the elimination tree allow the chosen pivotal sequence to be found, including the grouping of variables into blocks that are eliminated together. At the same time, the assembly tree is constructed. We have chosen to index each assembly tree node by its position in the block pivotal sequence, rather than the index of the principal variable. For example, the assembly tree corresponding to the elimination tree of Figure 2.3 has just two nodes; node 1 corresponds to node 4 of the elimination tree and it has as its father node 2 which corresponds to node 7.

2.3 Factorization

The factorization is controlled by the assembly tree created at the end of the analyse phase. For stability, all the pivots are tested numerically. This may mean that some rows that we expected to eliminate during a block pivotal step remain uneliminated at the end of the step. These rows are stored alongside the generated element for treatment at the father node. We call these rows ***fully-summed*** since we know that there are no contributions to be added from elsewhere, unlike the rows of the generated element. At the father node, the possible pivots rows consist of old fully-summed rows coming from this son and perhaps other sons, too, and the new fully-summed rows that were recommended as pivots by the analyse phase.

If a full block pivot was recommended, we choose a simple 1×1 or 2×2 pivot and perform the corresponding elimination operations on the pivot rows before choosing the next simple 1×1 or 2×2 pivot. We know of no other strategy for ensuring that the block pivot as a whole is satisfactory. Note, however, that the calculations for the Schur update can be delayed and performed as a block operation once all pivots are chosen.

If the updated entries of the fully-summed rows in the front are f_{ij}, the test for a 1×1 pivot is

$$|f_{kk}| \geq u \max_{j \neq k} |f_{kj}|, \tag{2.20}$$

where u is a parameter given a default value during initialization (see discussion in Section 6), and the test for a 2×2 pivot is

$$\left| \begin{pmatrix} f_{kk} & f_{k,k+1} \\ f_{k+1,k} & f_{k+1,k+1} \end{pmatrix}^{-1} \right| \begin{pmatrix} \max\limits_{j \neq k,\, k+1} |f_{kj}| \\ \max\limits_{j \neq k,\, k+1} |f_{k+1,j}| \end{pmatrix} \leq \begin{pmatrix} u^{-1} \\ u^{-1} \end{pmatrix}. \tag{2.21}$$

If a structured block was recommended, the analyse phase expects that the pivot rows have the form

$$\begin{pmatrix} \mathbf{A}_2 & \mathbf{A}_1 & \mathbf{A}_3 & \mathbf{0} \\ \mathbf{A}_1^T & \mathbf{0} & \mathbf{A}_4 & \mathbf{0} \end{pmatrix} \quad \text{or} \quad \begin{pmatrix} \mathbf{0} & \mathbf{A}_1 & \mathbf{A}_3 & \mathbf{0} \\ \mathbf{A}_1^T & \mathbf{0} & \mathbf{A}_4 & \mathbf{0} \end{pmatrix} \tag{2.22}$$

after suitable permutations. We need to check that the potential pivots (leading two block columns in 2.22) still have this form since earlier changes to the pivotal sequence may destroy it. Assuming that the form is still there, we again choose simple pivots one at a time and perform the corresponding elimination operations on the pivot rows before choosing the next simple pivot.

We restrict these pivots to 2×2 pivots of the desired form, noting that each is identified by an entry of the $\mathbf{A}_1$ block. We therefore search $\mathbf{A}_1$ or its reduced form, again using the test (2.21) for stability. For a tile pivot, it is possible for this test to fail and yet two 1×1 pivots in the same rows both to satisfy inequality (2.20). Using this pair is mathematically equivalent, but would lead to undesirable fill-in. We therefore accept the tile pivot in this case, too.

If any fully-summed rows remain in the front after completion of this sequence of simple 2×2 pivot operations, we look for simple 1×1 and 2×2 pivots in these rows, exactly as for the case when full pivots were recommended. Note that these rows may be new fully-summed rows in which we failed to find a structured pivot or old fully-summed rows from the sons of the node. For convenience of organization, we order the new fully-summed rows ahead of the old fully-summed rows in the front. If any 1×1 or 2×2 full pivots are chosen, we regard the generated element as full, but by forming the Schur update for the rows of the structured pivots separately, we can at least take some advantage of the zero block or blocks within it.

As well as the relative tests we have just described, we also apply an absolute test on the size of a 1×1 pivot or the off-diagonal entry of a 2×2 pivot. By default, the value of this ***pivot tolerance*** is zero.

Provided enough pivots are selected in the block pivot step, we use Level 3 BLAS (Dongarra, Du Croz, Duff, and Hammarling 1990) for constructing the Schur update. In the case of a full pivot, the Schur update is

$$\mathbf{S} \;=\; \mathbf{M}^T\mathbf{D}\,\mathbf{M} \tag{2.23}$$

where $\mathbf{M}$ is a rectangular matrix and $\mathbf{D}$ is a diagonal matrix with blocks of order 1 and 2 (see equation (2.10)). Within the frontal matrix, we hold a compact representation of $\mathbf{M}$ that excludes zero columns. Therefore, our real concern is the efficient formation of the product (2.23) for a full rectangular matrix $\mathbf{M}$. Unfortunately, there are no BLAS routines for forming a symmetric matrix as the product of two rectangular matrices, so we cannot form $\mathbf{D}\,\mathbf{M}$ and use a BLAS routine for calculating $\mathbf{M}^T(\mathbf{D}\,\mathbf{M})$ without doing about twice as much work as necessary. We therefore choose a block size *b* (with default size 5 set by the initialization routine) and divide the rows of $\mathbf{M}$ and $\mathbf{D}\,\mathbf{M}$ into strips that start in rows 1, *b*+1, 2*b*+1, ... This allows us to compute the block upper triangular part of each corresponding strip of $\mathbf{S}$ in turn, using `SGEMM` for each strip except the last. For the last (undersize) strip, we use simple Fortran code and take full advantage of the symmetry.

For an oxo pivot, the matrix $\mathbf{D}\,\mathbf{M}$ has the form

$$\mathbf{D}\,\mathbf{M} \;=\; \begin{pmatrix} \mathbf{L}_1 & \\ & \mathbf{U}_1^T \end{pmatrix}^{-1} \begin{pmatrix} \mathbf{A}_5 & \mathbf{A}_6 & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{A}_7 & \mathbf{A}_8 & \mathbf{0} \end{pmatrix} \;=\; \begin{pmatrix} \mathbf{B}_5 & \mathbf{B}_6 & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{B}_7 & \mathbf{B}_8 & \mathbf{0} \end{pmatrix} \tag{2.24}$$

and the matrix $\mathbf{M}$ has the form

$$\mathbf{M} \;=\; \begin{pmatrix} & \mathbf{D}_1^{-1} \\ \mathbf{D}_1^{-1} & \end{pmatrix} \begin{pmatrix} \mathbf{B}_5 & \mathbf{B}_6 & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{B}_7 & \mathbf{B}_8 & \mathbf{0} \end{pmatrix} \;=\; \begin{pmatrix} \mathbf{0} & \mathbf{B}_9 & \mathbf{B}_{10} & \mathbf{0} \\ \mathbf{B}_{11} & \mathbf{B}_{12} & \mathbf{0} & \mathbf{0} \end{pmatrix}. \tag{2.25}$$

We may form the Schur update

$$\begin{pmatrix} \mathbf{0} & \mathbf{B}_2 & \mathbf{B}_3 & \mathbf{0} \\ \mathbf{B}_2^T & \mathbf{B}_1 & \mathbf{B}_4 & \mathbf{0} \\ \mathbf{B}_3^T & \mathbf{B}_4^T & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} \end{pmatrix} \;=\; \begin{pmatrix} \mathbf{0} & \mathbf{B}_{11}^T \\ \mathbf{B}_9^T & \mathbf{B}_{12}^T \\ \mathbf{B}_{10}^T & \mathbf{0} \\ \mathbf{0} & \mathbf{0} \end{pmatrix} \begin{pmatrix} \mathbf{B}_5 & \mathbf{B}_6 & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{B}_7 & \mathbf{B}_8 & \mathbf{0} \end{pmatrix} \tag{2.26}$$

by the calculations

$$\begin{aligned} \begin{pmatrix} \mathbf{B}_2 & \mathbf{B}_3 \end{pmatrix} &\;=\; \mathbf{B}_{11}^T \begin{pmatrix} \mathbf{B}_7 & \mathbf{B}_8 \end{pmatrix}. \\ \mathbf{B}_4 &\;=\; \mathbf{B}_{12}^T \mathbf{B}_8, \\ \mathbf{B}_1 &\;=\; \begin{pmatrix} \mathbf{B}_9^T & \mathbf{B}_{12}^T \end{pmatrix} \begin{pmatrix} \mathbf{B}_6 \\ \mathbf{B}_7 \end{pmatrix}. \end{aligned} \tag{2.27}$$

We may use the BLAS 3 routine SGEMM directly for the first two calculations. For the symmetric matrix $\mathbf{B}_1$, we subdivide the computation into strips, as for the full-pivot case.

For a tile pivot, the sparsity in the first block column of the above form is lost:

$$\mathbf{D}\mathbf{M} \;=\; \begin{pmatrix} \mathbf{L}_1 & \\ \mathbf{U}_2^T & \mathbf{U}_1^T \end{pmatrix}^{-1} \begin{pmatrix} \mathbf{A}_5 & \mathbf{A}_6 & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{A}_7 & \mathbf{A}_8 & \mathbf{0} \end{pmatrix}. \tag{2.28}$$

We therefore merge the first two blocks to give

$$\mathbf{D}\mathbf{M} \;=\; \begin{pmatrix} \mathbf{B}_6 & \mathbf{0} & \mathbf{0} \\ \mathbf{B}_7 & \mathbf{B}_8 & \mathbf{0} \end{pmatrix} \tag{2.29}$$

$$\mathbf{M} \;=\; \begin{pmatrix} \mathbf{D}_1^{-1}\mathbf{D}_2\mathbf{D}_1^{-1} & \mathbf{D}_1^{-1} \\ \mathbf{D}_1^{-1} & \mathbf{0} \end{pmatrix} \begin{pmatrix} \mathbf{B}_6 & \mathbf{0} & \mathbf{0} \\ \mathbf{B}_7 & \mathbf{B}_8 & \mathbf{0} \end{pmatrix} \;=\; \begin{pmatrix} \mathbf{B}_9 & \mathbf{B}_{10} & \mathbf{0} \\ \mathbf{B}_{12} & \mathbf{0} & \mathbf{0} \end{pmatrix}. \tag{2.30}$$

The Schur update calculation is therefore as in the oxo case except that the first block row and column is not present and we have only the calculations for $\mathbf{B}_1$ and $\mathbf{B}_4$.

2.4 Solution

The solution is conveniently performed in two stages. The first, forward substitution, consists of solving

$$(\mathbf{P}\,\mathbf{L}\,\mathbf{P}^T)\,\mathbf{y} \;=\; \mathbf{b} \tag{2.31}$$

and the second, back-substitution, consists of solving

$$(\mathbf{P}\,\mathbf{D}\,\mathbf{L}^T\mathbf{P}^T)\,\mathbf{x} \;=\; \mathbf{y}. \tag{2.32}$$

For the first step of the forward substitution, let

$$\mathbf{P}^{-1}\mathbf{b} = \begin{pmatrix} \mathbf{b}_1 \\ \mathbf{b}_2 \\ \mathbf{b}_3 \end{pmatrix} \tag{2.33}$$

where $\mathbf{b}_1$ corresponds to the block pivot and $\mathbf{b}_2$ corresponds to the rest of the first front. We need to perform the update

$$\begin{pmatrix} \mathbf{L} & & \\ \mathbf{M}^T & \mathbf{I} & \\ \mathbf{0} & \mathbf{0} & \mathbf{I} \end{pmatrix} \begin{pmatrix} \mathbf{b}'_1 \\ \mathbf{b}'_2 \\ \mathbf{b}'_3 \end{pmatrix} = \begin{pmatrix} \mathbf{b}_1 \\ \mathbf{b}_2 \\ \mathbf{b}_3 \end{pmatrix}. \tag{2.34}$$

This involves the solution of the equation

$$\mathbf{L}\,\mathbf{b}'_1 = \mathbf{b}_1 \tag{2.35}$$

and the update

$$\mathbf{b}'_2 = \mathbf{b}_2 - \mathbf{M}^T\mathbf{b}'_1 \tag{2.36}$$

In the case of a full pivot, we can employ the Level 2 BLAS routine STPSV in equation (2.35) (we pack the triangular array **L** to save storage) and the Level 2 BLAS routine SGEMV in equation (2.36). For a structured pivot, it is slightly more complicated. Now **L** has the form $\begin{pmatrix} \mathbf{L}_1 & \\ \mathbf{U}_2^T & \mathbf{U}_1^T \end{pmatrix}$, so solving equation (2.35) requires two applications of STRSV (here, we pack the arrays $\mathbf{L}_1$, $\mathbf{D}_1$ and $\mathbf{U}_1$ together in a square array) and one application of STPMV. Also, **M** has the form $\begin{pmatrix} \mathbf{B}_5 & \mathbf{B}_6 & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{B}_7 & \mathbf{B}_8 & \mathbf{0} \end{pmatrix}$ so two applications of SGEMV are needed for equation (2.36).

Similar considerations apply to the other steps and to the back-substitution, too.

2.5 The singular case

Our code is designed primarily for the nonsingular case, but it can also treat singular matrices and returns the computed rank. It seeks as many pivots as possible, but the factorization is regarded as complete if all elements of the reduced matrix are found to be less than the pivot tolerance. No forward substitution or back-substitution operations are performed for the rows and columns that correspond to this submatrix. Thus a solution is obtained of the subproblem consisting of the rows in which pivots were found, and it will be a solution of the whole problem if a consistent set of equations is presented. Note that this is not a robust way to determine the rank in the presence of rounding errors. In particular, the rank may be overestimated which is likely to cause the components of the computed solution to behave wildly. We recommend the use of iterative refinement to check for this.

2.6 The complex case

We also provide a complex version of the code, which we call ME47. It follows very closely to MA47 and we do not need to describe it separately. It handles symmetric complex matrices and we have chosen not to offer a version for Hermitian matrices because more significant changes would have been needed to keep track of which off-diagonal entries move between the two triangular halves as permutations are made. Note that the complex version of MA27 handles only Hermitian matrices.

3 Analysis of the sparsity pattern

In this section, we describe the subroutines that analyse the sparsity pattern.

3.1 MA47A: driver for the analyse phase

MA47A is the driver subroutine called directly by the user, who is required to provide only the order N, the number of entries NE, and the row and column indices of the entries, IRN(*k*) and ICN(*k*), $k = 1, 2, ...,$ NE. The essentials of these data are preserved since they are needed for MA47B (factorize). For efficient execution of MA47B, both indices for an entry are replaced by zero if either is outside the range $[1, n]$ and the indices of an off-diagonal pair are interchanged if they represent the entry in the lower triangular part of the matrix when it is permuted according to the chosen permutation.

The other data needed by MA47B are packed into the array KEEP. The user wishing to provide a tentative pivotal sequence must place it at the front of KEEP. Otherwise, the user need not set KEEP. On return from MA47A, KEEP contains:

(i) the permutation, arranged so that each structured pivot in the permuted matrix has the form (2.1) or (2.2),

(ii) the row lengths of the upper triangular part of the permuted matrix,

(iii) the indices of the assembly tree nodes at which the variables are eliminated, negated for defective variables,

(iv) the Markowitz costs of the pivots,

(v) the fathers of the tree nodes,

(vi) the number of faulty entries, and

(vii) the map that orders the entries so that the permuted matrix is held by rows.

In addition, information about the analyse phase is placed in the arrays RINFO and INFO (see the appendix for details).

MA47A begins with optional printing of the problem data with a level of detail that is chosen by the user. Next, the work array IW is partitioned into subarrays, as is the array KEEP.

If the pivotal sequence is not provided, MA47G is now called to construct the pattern of the whole matrix by rows and MA47H uses this pattern to find the elimination tree (defined in Section 2.2). Efficient execution of MA47H demands ready access to all the entries of a given row.

If the pivotal sequence is provided, MA47J is called to construct the pattern of the upper triangular part of the permuted matrix by rows and MA47K uses this pattern to find the elimination tree.

Apart from minor roundoff caused by the non-associativity of floating-point addition and subtraction, the pivotal order can be changed to any that results in the variables of a pivot preceding the variables at its father node in the assembly tree. When executing on a single processor, it is convenient to use the postordering from a depth-first search of the assembly tree for then a stack may be used for the intermediate results during actual factorization. Such an ordering is performed by calling MA47L, which actually works with the elimination tree because this provides a compact and convenient representation for the sets of variables eliminated at the nodes of the assembly tree. This subroutine also merges father-son pairs of nodes of the assembly tree where this causes no extra fill-in or little extra fill-in.

To avoid an expensive sort for each matrix factorized by MA47B, we construct a mapping vector for permuting the entries to the order corresponding to the upper triangular part of the permuted matrix being stored by rows. This map is created by a call of MA47N.

Finally, MA47M is called to compute the storage and operation counts on the assumption that all the tentative pivots are acceptable numerically.

3.2 MA47F: compress analyse data structure

Index lists for the rows of the reduced matrix and for the generated elements are held in the array IW. Each list is preceded by an entry that holds its length. Since the index used for a generated element is that of the principal eliminated variable, a single array IPE of length N is sufficient to hold all the start positions within IW of the lists. If i is inactive (a variable that is not a principal variable or an element that has been merged into another element), FLAG(i) = –1.

The task performed by MA47F is to move all the index lists forward to become adjacent, so that all the free space is together following the final list.

This routine is called only by MA47H and MA47K, and they only call it when the free space is insufficient for their needs. They both maintain a pointer IWFR to the first free location and present only the subarray IW(1:IWFR-1) for compression. To ease the task, they ensure that no entry of this subarray is negative. MA47F begins by replacing each pointer in IPE to an active list by the length of the list and the entry preceding the list (the entry that held the length) by the negation of the index of the supervariable or element. This allows a simple forward search of IW to recognize each active list. As each is found, MA47F stores the length in the next free location, moves the list forward, and resets the pointer in IPE.

The opportunity is taken also to remove from the generated element lists any supervariables that have been eliminated or merged. These are flagged by negative entries in the array FLAG. The three parts of each element list, corresponding to the blocks of (2.3), must be scanned separately in order that the numbers of supervariables in each part are revised appropriately. There is no need to perform the corresponding task for the row lists, since fresh lists are constructed at each pivotal step for the rows active in the step.

3.3 MA47G: construction of the sparsity pattern by rows

MA47G is given a list of pairs of row and column indices that represent the matrix entries. The first task is to run through them checking for indices outside the range 1 to *n* and counting the numbers of entries in the rows. To avoid having to recheck for out-of-range indices subsequently, both indices of a pair are reset to zero if either is out of range. This has the effect of moving all such entries to the position (0,0), where they are easy to treat by regarding the matrix as having an extra row and column with index zero.

Once the numbers of entries in the rows are known, we accumulate them to find pointers to where in array IW the ends of the rows would lie if lists of column indices for all the rows (including one for row zero) were stored there. A further pass through the index pairs allows these lists to be constructed. As each index is placed in IW, the pointer for the row is decremented, so that it is easy to place each index in the desired place. Including row zero obviates the need for tests and makes the execution faster.

Out-of-range and duplicate entries are now removed in a single pass of the array IW that ‘compacts’ it with the help of the array FLAG, initially set to zero. We scan the rows in turn, i = 1, 2, ..., n, moving non-duplicate entries forward. For each entry a_{ij}, we set FLAG(j) to i unless it already has this value, which indicates a duplicate that can be ignored. Because we start with row 1, the out-of-range entries are ignored too.

3.4 MA47H: Markowitz analysis

`MA47H` is provided with the sparsity pattern of the whole matrix, ordered by rows, and finds a pivotal order and elimination tree using a variant of the Markowitz (1957) criterion (see Section 2.2). We also provide an option to limit the search to a given number of rows with least entries.

To speed the analysis, `MA47H` recognises rows with the same structure, that is, it works with supervariables. We will refer to the set of rows associated with a supervariable as a ***super-row*** and the set of columns associated with a supervariable as a ***super-column***.

3.4.1 The data structures used

The array `IW` is used to hold index lists that represent the sparsity pattern of the reduced matrix. There is a list for each super-row, which consists of indices for generated elements that involve the super-row and super-column indices from the pattern of the original matrix **A**. We have found it convenient to place the indices for generated elements ahead of all the super-column indices (see end of Section 3.4.2). Each list is preceded by a header of length one that holds the number of entries in the list itself. The array `IPE` holds pointers to these headers. Note that working with super-rows and super-columns means that there are fewer lists and each is shorter. For example, if every variable belongs to a supervariable of three nodes, the length of every list is reduced by the factor three and the number of lists is also reduced by the factor three.

For efficiency, it is also necessary to record to which part of each generated element the super-row belongs. Without this, an additional search of the element's index list would be needed on every access. We therefore add n to the element index if the super-row belongs in the part with a trailing zero block and add $2n$ if the super-row belongs in the part with a leading zero block. This requires that $3n$ be representable, which will not be a limitation in practice.

The array `IW` also holds a list of supervariable indices for each generated element. When a supervariable is eliminated, we use its index as the name of the generated element. This permits the array `IPE` to hold pointers to the headers for both super-row lists and generated element lists. Each element list is ordered so that the matrix has the form (2.3) and has a header of length 3 that contains the length, the number of supervariables in the part with leading zeros, and the number of supervariables in the part with trailing zeros. This choice was made so that it is easy to search all the indices or exclude those of one of the zero blocks.

As the analyse phase progresses, supervariables become merged and elements become absorbed into other elements. To avoid significant computation at each such occurrence, we temporarily keep each list that is no longer needed, but set its pointer in `IPE` to zero. Such a list will be removed when the data structure is compressed by `MA47F`. Similarly, we do not remove indices within lists immediately they are no longer needed, but use the value –1 in `FLAG` to indicate a merged variable. When a list for a super-row is revised because of pivotal activity, such dummy indices are removed. They are removed from element lists when the data structure is compressed by MA47F. Thus, they are updated only when the list or its storage is actually needed.

We use the work array FLAG to label the status of the variables and generated elements and to enable the rapid recognition of duplicates when index lists are merged. Associated with this is a scalar key NFLG. Initially, the key value is set to $3n$ and all the components of FLAG that correspond to supervariables are given this value; other components are given the value –1. At the start of the main loop, which chooses a block pivot, FLAG(i) has the value NFLG or more if i is a supervariable, the value -NFLG or less if i is an element not yet absorbed, and the value –1 otherwise. During part of the loop, the FLAG values –1, 0, 1, 2 are used for the supervariables of the pivot and the three parts of the generated element. For each super-row search, we decrement NFLG by one, which will ensure that its value differs from that of any component of FLAG without any need for an $O(n)$ loop to reset FLAG. Once an index i has been encountered, FLAG(i) is set to NFLG, so that subsequent occurrences are immediately recognizable. The key is not permitted to be less than 4; if it reaches the value 4, MA47Z is called to reset it to $3n$ and to adjust FLAG to correspond. The value $3n$ is chosen to ensure that MA47Z is called infrequently and we can be sure that it is always representable.

For speed of execution, it is important to avoid an expensive search for the pivot at any stage of the elimination. We therefore hold doubly-linked chains of supervariables that have the same row count (number of entries in the row). This allows us to look at super-rows in order of increasing row count without an $O(n)$ search for the next one. Any supervariable that is active in a pivotal step and whose row count may therefore change can be removed from its chain without any searching. Once its new row count is known, it can be placed at the head of the corresponding chain. The arrays NEXT and LAST are used to hold forward and backward pointers in the chains and the array IPR points to their starts.

Since this use of the array NEXT is limited to active supervariables (each named after its principal variable), it may also be used to hold son-to-father pointers for the current forest. To distinguish the two roles clearly, we negate the son-to-father pointers. If g is a supervariable that has been eliminated and the corresponding generated element g has not yet contributed to a pivotal row, it is a tree root and NEXT(g) is given the value zero.

Calculating the new row counts for the supervariables that have been active in a pivotal step can be expensive, particularly when the counts are high. Those with high row counts are unlikely to be chosen before their counts change again, so we keep a threshold THRESH and content ourselves with lower bounds for counts above the threshold. Initially, we set the threshold to $\max(3,\sqrt{n})$. If we find ourselves considering a row of count greater than the threshold, we raise the threshold by $n/10$ and call MA47V to recompute the row counts lying between the old and new threshold.

Despite all these precautions, we found that the analysis phase can sometimes be very slow.

We therefore provide the option to limit the search to a given number of rows with least entries.

3.4.2 The execution of the analysis code

We begin by calling MA47T (see Section 3.10) to recognize identical rows in the original matrix and form supervariables. Initially, IW holds a list of column indices for each row of the matrix and it is not altered to reflect the identification of supervariables. Instead, we simply rely on the mechanism described in the previous subsection to remove the dummy indices when the list is revised or the whole structure compressed. MA47T sets IPE(i) to zero for any variable i that is not a principal variable.

Next, the array FLAG is initialized and the doubly linked chains of supervariables of the same row count are established. Apart from the storage of some information in INFO at the end, the rest of the code consists of a long do loop, each cycle of which chooses a block pivot and makes the appropriate revision to the data structure. We now describe the execution of this loop.

The pivot is chosen in a do loop over row counts starting from a lower bound, MINR. Maintaining MINR avoids unnecessary searches for super-rows with impossibly small row counts. For each row count, we first look for a nondefective supervariable. If we find one, we take it as a full pivot, since a structured pivot with a lower Markowitz count would have been found earlier when searching super-rows with lesser counts, see expressions (2.17), (2.18). and (2.19). If all the supervariables with the current count are defective, we scan each corresponding super-row for the off-diagonal entry of a structured pivot.

The index list for the super-row is not available directly. Instead, we have to scan the relevant part of each generated element that involves the super-row as well as the remains of the original super-row list. For quick recognition of duplicates, we use the array FLAG and its key NFLG. For each off-diagonal entry a_{ij} encountered, we compute the Markowitz cost (2.18) or (2.19). Up-to-date row counts for the supervariables are kept in the array COUNT to enable this calculation to be performed efficiently. The best candidate pivot so far found is recorded, and once we can be sure that a better one cannot be found by further searching, this is tentatively accepted. If this tentative pivot does not have a row count above the threshold THRESH, we accept it. Otherwise, we follow our procedure for raising the threshold and repeat the search.

If a full pivot is accepted, we remove its supervariable from the doubly-linked chain and construct the index list for the generated element. This requires a very similar procedure to that just described for the search in a super-row for a structured pivot, and again full use is made of the array FLAG and its key. Son-father links to the new element e must be recorded in NEXT for all the generated elements g encountered. If g is not already linked (that is, NEXT(g) = 0), we have only to set NEXT(g) to –e. However, a structured generated element may be already linked because of being partially absorbed in one or more pivotal steps. In this case, we link the root r of the tree containing g by setting NEXT(r) to –e unless it is already so set. Each supervariable in the index list is active in the pivotal step and so must be removed from its doubly-linked chain. Any element encountered that covers the pivot is absorbed by the newly generated element and is not needed any more. Its component in FLAG is set to –1 to indicate this.

For some problems, we found that running through the links to the root was time-consuming because an element remained active over many generations. We therefore use the array LAST to provide more direct pointers to ancestors. For each element *f* encountered in the search, including *g* and *r*, we set LAST(*f*) to –*e*. To find the root, we use LAST rather than NEXT.

If a structured pivot is accepted, we must check for a mismatch between the sizes of its two supervariables. In this case, the larger supervariable must be split into a part that does match and a part that must be left for later processing. It was the desire to perform this splitting with simple code that led us into ensuring that each subtree that represents a supervariable is a chain. It also means that our tree really is a generalization of the elimination tree.

For any structured pivot, index lists for both pivot super-rows must be constructed, which is done with an outer loop of length 2. We use FLAG to record a supervariable that occurs in the first super-row only (value 1), the second super-row only (value 2), or both (value 0). We scan the second super-row first to ensure that its entries occur first, so that the list for a tile pivot is correctly ordered (see Figure 2.2). As for a full pivot, if the pivot is covered by an element encountered, the element is not needed any more and its component in FLAG is set to –1. The association of the second supervariable with the pivot is recorded in the forest by regarding it as having the first supervariable as its father. For an oxo pivot, a minor reordering of the index list is needed to ensure that entries that occur only in the second super-row precede those in both super-rows (see Figure 2.1); this is done by a simple scan with a swap for each index belonging in the first block.

We now check whether the newly generated element absorbs any of the old generated elements. If such an old element does not already have the new element as its root, its present root (which may be itself) is linked to the new element. The element is not needed any more and its component in FLAG is set to –1 to indicate this. The candidates are found by searching all the elements associated with the supervariables of the new element. To ensure that no element is checked more than once, we once again make use of the array FLAG and its key NFLG. We need to check that the full part of the old element lies within the full part of the new element (FLAG value 2 or 0), that its non-full parts lie in the new element (FLAG value 0, 1, or 2), and that the old element does not have an index from its first block and an index from its third block both in the same non-full block of the new element.

For each active supervariable, we now revise its associated list, recalculate the row count, and merge any that are now duplicates. We treat the supervariables of the full part of the generated element first, then those of the third block, then those of the first. For calculating a row count, we always start from the corresponding count in the new element, noting that its supervariables have already been given FLAG values 0, 1, or 2. We run through the elements associated with the supervariable first (they are at the front of the list); for each element, we see if the part involved contains any supervariables not already counted. If it does, the row count is incremented and the element index is retained in the list. It is also retained if it does not have the new element as its ancestor; here, even if it is not needed to construct the pattern of the row, it will be needed later when the supervariable is eliminated so that the element is linked into the forest appropriately. Otherwise, the element index is deleted. Once all the elements in the old list have been considered, we consider the list of supervariables and need to copy to the new list only those that make a contribution. Finally, we add the new element to the new list. Unfortunately, there may have been no deletions so that the new list may be longer than the old one was. In this case, we make a fresh copy of the list at the start of the free space in IW, after having compressed the data structure if necessary by calling MA47F. The first supervariable index in the list is now moved to the end of the list and the first element index is moved to take its place, allowing the new element index to be added at the front. Having the new element at the front helps the test for identical rows (see next paragraph).

Unless the Markowitz cost is zero, which means that super-rows not previously identical cannot be identical now, we test each super-row of the new element for being identical with a previous one in the list. We look in the chain of supervariables with the same row count until we find one that is not associated with the new element or is associated with a different part of it (no search is needed since the new element is at the front of the list of a relevant variable).

Finally, the list of supervariables for the new element is stored in IW, after having compressed the data structure if necessary by calling MA47F, and the FLAG values for the supervariables of the new element are reset.

3.5 MA47J: construction of the upper-triangular sparsity pattern by rows

Subroutine MA47J is given a list of pairs of row and column indices that represent the matrix entries and a permutation. It constructs the sparsity pattern of the upper triangular part of the permuted matrix by rows. It is very similar to MA47G, see Section 3.3. It, too, replaces any pair by zeros if either is out of range. It, too, excludes out-of-range and duplicate entries from the sorted form. The difference is that it checks that it has been given a valid permutation and that only one of each pair of off-diagonal entries is included unlike MA47G where both must be held for efficiency in pivot selection.

3.6 MA47K: analysis for a given pivotal sequence

MA47K constructs the elimination tree when the pivotal sequence is provided. It is essentially a simplified version of MA47H. It does not work with supervariables because there is not the same scope for efficiency gains. There is no need for chains of variables with the same row count because no pivot choice has to be made. There is no need even to keep up-to-date row counts, so the index lists are used far less intensively.

The index lists for the rows correspond to the upper-triangular part of the permuted matrix and are not revised; in fact, they are only referenced when the row is pivotal. In order to be able to access the correct generated elements when the row is pivotal, each element is classified according to the variable that it contains which is earliest in the remaining pivotal sequence. Those of the same class are chained together. It requires only two integer arrays of length n to hold this information:

IPR(i) is the first element of the class with variable i as its earliest variable, and

NEXTE(e) is the next element of the same class as element e or zero if it is the last of the class.

After each element is used to construct the index list for a pivotal row, it is linked in its new class, corresponding to its next variable in pivotal order.

The user is required to provide the variable indices in pivotal order. For a 2×2 structured pivot, the variables must be next to each other in the sequence with their indices negated. We provide no mechanism for specifying block pivots, since these will be recognized during the depth-first search of MA47L. We allow a tile pivot to be either way round and even permit a full pivot to be presented as a structured pivot. The final depth-first search performed by MA47L (see Section 3.7) ensures that the two halves of a tile pivot are the right way round.

The main loop processes a 1×1 pivot or a 2×2 structured pivot. In much the same way as in MA47H, the list of variables in the generated element is constructed and the tree links are established. The difference lies in using the chains of elements and putting the element in its new chain after use unless it is absorbed because it covers the pivot. For a 2×2 pivot, direct tests on the entries are made to determine if it is full, tile, or oxo. For an oxo pivot or a tile pivot with second diagonal entry zero, the index list of the generated element matrix is sorted to give the form (2.3). No attempt is made to find element absorptions other than those that are a consequence of covering a pivot, since we judge that the lists are not used intensively enough to justify the effort. It remains only to store the generated element list in IW, which may require a call of the compression subroutine MA47F. Since this is the only occasion when data is added to IW, compressions are far less frequent than for MA47H.

3.7 MA47L: depth-first search of the assembly tree

Subroutine MA47L performs depth-first searches of the elimination tree constructed by MA47H or MA47K. The tree itself is held as a vector of son-to-father pointers, which is an economical representation.

The opportunity is taken to look for father-son merges that can take place with no extra fill-in, since most of them can be found by a simple test involving the number of variables eliminated at the son and the row counts before elimination. It is the only mechanism that we have to find block pivots when the user specifies the pivotal order and the tree is constructed by MA47K. It is also useful when the pivotal order and tree are chosen by MA47H. Many block pivots will be found through its use of supervariables, but not all since it would be costly to ensure that every possible supervariable is identified.

We need also to know which nodes represent full, tile, or oxo pivots. This data is coded using the signs of the components of the vectors holding the row counts and elimination counts.

If a father and son are both full pivots and the number of variables eliminated at the son is equal to the difference between the son's row count before elimination and the father's before elimination, there is no extra fill-in if they are merged. The frontal matrix for the father node has the pattern of the element generated by the son and there is no advantage in treating them separately. This is particularly likely to happen near the root because the reduced matrix is often full in the final stages.

Similarly, a father-son pair of tiles can be merged without extra fill-in if the differences in the counts for the rows with nonzero diagonal entries differs by the number of variables eliminated and the difference for the rows with zero diagonal entries differs by half the number of variables eliminated. A simple example is shown in Figure 3.1. Here there is no fill-in and the row counts at the time of elimination are (5,3) and (3,2).

$$\begin{array}{cc|cc|c}
\times & \times & \times & \times & \times \\
\times & & \times & & \times \\
\hline
\times & \times & \times & \times & \times \\
\times & & \times & & \times \\
\hline
\times & \times & \times & \times & \times
\end{array}$$

Figure 3.1. A father-son pair of tile pivots for which a merge is possible.

For a father-son pair of oxos, we test the differences for each of the halves against half the number of variables eliminated. Unfortunately, when the row counts for the two halves are identical, we cannot be sure that the two halves of the son do not need to be interchanged if extra fill-in is to be avoided. A simple case is shown in Figure 3.2. where variables 1 and 2 are eliminated at the son node and variables 3 and 4 at the father node. Checking for such an event requires more information at the nodes than is available to MA47L. We therefore do not merge oxo nodes where the row counts of the two halves are the same.

$$\begin{array}{cc|cc|c}
 & \times & \times & & \times \\
\times & & & \times & \times \\ \hline
\times & & & \times & \times \\
 & \times & \times & & \times \\ \hline
\times & \times & \times & \times & \times
\end{array}$$

Figure 3.2. A father-son pair of oxo pivots for which a merge is possible if the variables of the father are interchanged.

We also perform some merging that does involve extra fill-in because of the advantages of reasonably large block sizes, particularly on a vector or parallel computer. Interfering with a structured pivot may lead to greatly increased fill-in, so we do this only for father-son pairs that are full. Also, we require that all ancestors be full, too, to avoid indirectly affecting a structured pivot. This leads us to using a depth-first search to look for father-son merges. These amalgamations are controlled by a parameter (with default value 5) which is a limit on the number of variables at a node that is merged with another.

Son-to-father pointers provide a compact representation of a tree, but are not suitable for a depth-first search. We use two temporary vectors of length *n*:

`SON(`*i*) is the index of one of the sons of node *i* or zero for a leaf node and

`NA(`*i*) is the index of the next brother of node *i* or the negation of the index of its father if there is no next brother.

These arrays are established with a simple scan of the vector holding son-father pointers, and tests for son-father merges are made as each father is reached from the son in a subsequent depth-first search. Each merge is recorded by adjusting the row count and number of eliminations at the father and setting to zero the number of eliminations at the son.

A second depth-first search is used to adjust the son-to-father pointers so that every node merged with its father is skipped.

A final depth-first search establishes the pivotal sequence and assembly tree with nodes indexed by position in the block pivot order. Here we establish the brother pointers in three separate loops to ensure that the depth-first search encounters the elimination nodes first, then the secondary nodes of structured pivots, then the non-principal variables. This ensures that the postordering of the variables places any associated with eliminations in the subtree rooted at a node ahead of any that are eliminated at the node itself. Furthermore, it ensures that for a structured pivot, the variables of each part are adjacent in the ordering. This is illustrated in the tree of Figure 3.3, which is a reversed image of the elimination tree of Figure 2.3. This reversal, which effectively reorders the brothers, is important because of the way we subsequently perform our depth first search where we choose always to search the brothers from the left. With the brothers reordered in this way, node 1 is a secondary node of a structured pivot and lies ahead of node 5, a non-principal node. Node 4 is an elimination node and lies ahead of node 8, a non-principal node. The postordering yields the order 3, 2, 1, 6, 5, 4, 8, 7.

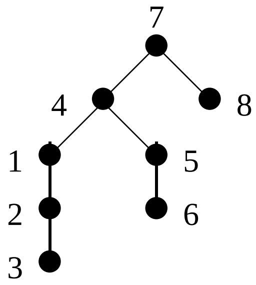

Figure 3.3. The elimination tree of Figure 2.3 with new ordering of brothers.

During this final search, the number of nodes in the assembly tree and the number of sons of each node of this tree are calculated, indexing each node by its position in the block pivot sequence. This provides a representation of the assembly tree from which son-to-father pointers may be found in a simple final loop that visits the nodes in sequence and keeps a stack of nodes whose fathers are not yet known. At a typical node *i* with *k* sons, *k* nodes are popped from the stack and given *i* as their father, then *i* is pushed onto the stack. For the example shown in Figure 3.4, the numbers of sons are 0, 0, 0, 3, 0, 2 and the stack contents are successively (1), $\begin{pmatrix}2\\1\end{pmatrix}$, $\begin{pmatrix}3\\2\\1\end{pmatrix}$, (4), $\begin{pmatrix}5\\4\end{pmatrix}$.

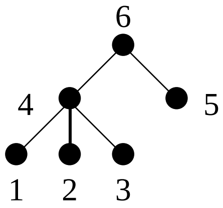

Figure 3.4. An assembly tree.

3.8 MA47M: calculate storage and operation counts

MA47M calculates storage and operation counts by simulating the action of factorization, assuming that every pivot recommended by the analyse phase is acceptable numerically. Its logic is very close to that of MA47O, see Section 4.2.

3.9 MA47N: construct map for sorting the entries

Subroutine MA47N is given a permutation and a list of pairs of row and column indices that represent the entries. Any entry with an out-of-range index will already be represented by a pair of zeros. It constructs a mapping vector that for each entry holds its position when the upper triangular part of the matrix is stored by rows. No attempt is made to order the entries within each row and no check is made for duplicate entries. Any duplicates will be mapped to different locations, that is, the map is one-one. It is quite natural to accommodate duplicates in the MA47B coding since there are anyway duplicates that must be accumulated arising from the generated elements. Having a one-one map means that the sort performed by MA47B, essentially of the form

$$\texttt{B(MAP(K))} = \texttt{A(K)},\ \texttt{K} = \texttt{1,2,...,NE}$$

may be vectorized.

MA47N also interchanges the row and column indices of any entry in the lower triangular part of the permuted matrix.

The algorithm for finding the map is essentially the same as that used by MA47G, which is described in Section 3.3.

3.10 MA47T: recognition of supervariables

MA47T recognizes sets of identical rows in the original matrix and forms supervariables. This is done progressively so that after *j* steps we have the supervariable structure for the submatrix of the first *j* columns. We start with all variables in one supervariable (for the submatrix with no columns), then split it into two according to which rows do or do not have an entry in column 1, then split these according to the entries in column 2, etc. The splitting is done by moving the variables one at a time to the new supervariable. With careful choice of data structures, the whole process can be made to execute in *O*(N+NE) time. We use the following:–

SVAR(*i*) is the index of the supervariable to which variable *i* belongs.

NEXT(*i*) is the next variable to *i* in a circular chain of variables in supervariable SVAR(*i*).

LAST(*i*) is the previous variable to *i* in the same circular chain.

FLAG is a work array of length *n*, initially set to zero.

VAR is a work array of length *n*. If IS is a supervariable already encountered in the current column, VAR(IS) is the first variable to be removed from IS. If IS is not a supervariable, VAR(IS) is the next free supervariable index in a chain of such indices.

We scan column *j* and for each entry *i* use SVAR(*i*) to find its supervariable *s*. If FLAG(s)$<j$, this is the first occurrence of *s* for the column; unless *s* contains *i* alone, we remove *i* from its old supervariable, create a new supervariable with *i* as its only variable, give FLAG(*s*) the value *j*, and give VAR(*s*) the value *i*. If FLAG(s)$=j$, this is a subsequent occurrence of *s* for the column; we use VAR(*s*) to find the new supervariable. We now remove *i* from the old supervariable and add it to its new one. If this makes the old supervariable empty, we add it to the free chain.

Finally, we scan all the linked lists to count their numbers of variables and store them in the way required by the analysis code MA47H:–

if *i* is not a supervariable, NV(*i*) = 0, IPE(*i*) = 0, NEXT(*i*) is next variable of the supervariable, and

if *i* is a supervariable, LEAF(*i*) holds its first variable and abs(NV(*i*)) holds its number of variables; if *i* is defective, NV(*i*) is negative.

3.11 MA47V: raise the row-count threshold

Subroutine MA47V is called by MA47H when it needs to raise the threshold above which lower bounds for the row counts are held instead of exact values. It is provided with the old threshold THRESH and the desired new threshold NEWTHR and recalculates the row counts whenever they lie in the range THRESH ≤ COUNT(i) < NEWTHR. The calculation is done with the help of the array FLAG and its key NFLG, as in MA47H. Each supervariable whose row count is found to be different from that currently held is removed from its doubly-linked chain and linked into the appropriate chain.

3.12 MA47Z: reset the array of flags

Subroutine MA47Z is called by MA47H and MA47V to reset the array FLAG and its key NFLG when the key becomes too small. Every component of FLAG that is greater than 2 is reset to $3n$, every component that is less than -1 is reset to $-3n$ and NFLG is reset to $3n$.

4 Numerical factorization

In this section, we describe the routines that perform the numerical factorization. The pattern must have been analysed by a prior call to MA47A and the numerical factorization uses this information to guide the factorization. Because the numerical values of the entries in the matrix are now considered and pivots are only accepted if they satisfy a numerical test (see Section 4.2.6), it is unlikely that all of the pivots recommended by MA47A are suitable. This causes several added complications and is why the amount of storage and work is usually greater than that forecast by MA47A.

The main data that is input is the assembly tree generated by MA47A (see Section 3.7) together with the numerical values of the matrix entries in the same order as their indices were input to MA47A. The tree is processed in the order provided by MA47A (postordering following a depth-first search). We show the subdivision of our main work arrays at an intermediate stage of the factorization in Figures 4.1 and 4.2. The integer array holds index lists for the corresponding matrices and rows of the real array. The calculated factors are placed directly into their final positions at the front of the arrays. The entries of the original matrix are placed at the end; as each row is assembled, the space it occupied becomes free. The stack is placed ahead of this free space with its bottom just ahead of it. The frontal matrix is placed immediately after the calculated factors. If the top of the stack reaches the end of the front, the stack is moved to take advantage of the free space beyond its bottom and is also compressed since it may contain free space within it. Note that the integer work array is smaller than the real array since less storage is needed for the index lists. Its storage management is handled independently; for example, it is unlikely that both arrays will be compressed in the same block pivot step.

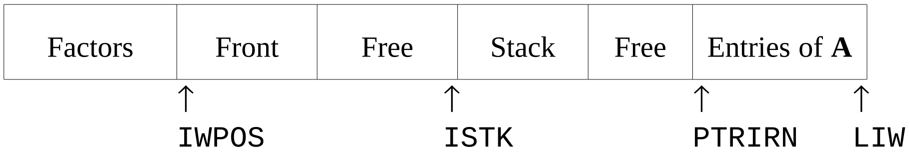

Figure 4.1. Subdivision of integer work array IW during numerical factorization.

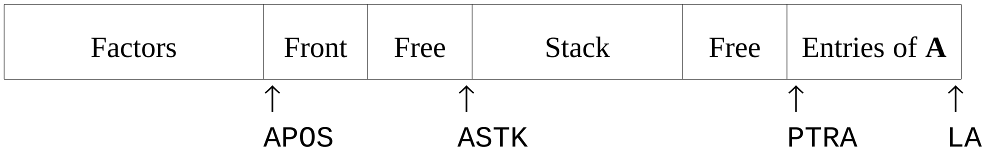

Figure 4.2. Subdivision of real work array A during numerical factorization.

For each node, information is assembled into the frontal matrix from the original matrix and the stack, eliminations are performed, the contribution to the LU factors is stored, and data are placed on the stack. As in the analyse phase, a major complication over the case with no structured pivots is that the contributions from the sons are not necessarily absorbed by the father and so may be left on the stack for processing at ancestors. A further complication in the factorization is that, although we try to follow the pivotal sequence recommended by the analyse phase, some pivots may be numerically unsatisfactory leaving some rows fully summed but uneliminated at the end of the step. If this happens when processing a structured pivot, it is not sensible to try to keep its structure for use at a later step since a pivot will only be chosen from these rows after they have been modified and the modification is likely to destroy the structure. Thus in both the case of full and structured pivots, we stack any fully-summed uneliminated rows for assembly at the father node. We call this a ***fully-summed element*** and store it as a separate generated element in the form of a full rectangular matrix. It is always absorbed at the father, which is our main reason for keeping it separate.

A block pivot step begins with the assembly of the rows recommended by MA47A for elimination, which we call the ***new*** fully-summed rows. These rows are followed by any fully-summed rows from the sons, which we call the ***old*** fully-summed rows. In order to take full advantage of the Level 3 BLAS and to avoid unnecessary work and storage, it is important to assemble only the fully-summed rows and not the whole of the frontal matrix. As each pivot is selected, these rows are updated but other real operations are delayed so that they can be later performed using blocking. Initially, we restrict the search to the recommended pivots in the hope of preserving the structure anticipated by MA47A. When we have exhausted our search for such pivots, we open the search to all candidates. After all the pivots have been selected, the Schur update is formed, possibly using Level 3 BLAS. If any pivot that was not recommended is chosen, we use full storage for the generated element because of the complications of any other choice. This is why we look first for pivots of the kind recommended by MA47A; for example, a 2×2 pivot in one old and one new row would destroy any sparsity that MA47A was hoping to get in the generated element.

All the elements involved at this node are then scanned to see if any can be absorbed into the new element. We do not do any partial absorption here, but note that some elements may have been partially absorbed when the fully-summed rows were being assembled. The resulting generated element is placed on the stack together with the fully-summed element, if present, and the factors are written to the beginning of the storage area. A more detailed sketch of the logic of the numerical factorization routine is given in Section 4.2.1.

4.1 MA47B: driver for the numerical factorization phase

MA47B is just a driver routine. It first performs simple checks on the data, initializes the information array, INFO, and optionally prints information on the input parameters. It then uses the mapping vector provided by MA47A (see MA47N in Section 3.9) to sort the matrix entries by rows in tentative pivotal order and to remove entries with out-of-range indices. We do not perform this sorting in-place because we want it to vectorize. We note that in MA27 the sort was occasionally as expensive as the actual numerical factorization. Here it will normally be very much faster than the factorization. Having performed the sort in a single loop of length the number of entries, we then call MA47O to perform the numerical factorization. After checking for error returns from MA47O (there are only two, corresponding to having insufficient real or integer work space), output data is optionally printed using MA47U and we exit from the MA47B subroutine.

4.2 MA47O: numerical factorization

The work is performed on data in two arrays. There is an integer array IW and a real array A and they are illustrated in Figures 4.1 and 4.2. The sorted entries of the original matrix are held at the end of these arrays. The factors are held from the beginning of the arrays with the intermediate space used as working space to hold generated elements of the form (2.3) and fully-assembled elements as dense rectangular blocks. After an initialization of pointers and various data arrays, the MA47O code performs the numerical factorization according to the assembly tree generated by MA47A. The work is thus done in a loop over the nodes of the assembly tree in the order obtained by MA47A by postordering following a depth-first search. Since the routine is long and complicated, we first present a sketch of the logic before discussing the fine detail.

4.2.1 Basic structure of MA47O

For each tree node in turn the following steps are executed. We will discuss each step in more detail in Sections 4.2.3 to 4.2.15.

Step 1. Construct the index list for the front. The indices of the rows to become fully summed in this step (new rows) followed by those that are already fully summed (old rows) are placed in the leading positions and their number is stored in `NASS`. The total number of indices is stored in `NFRONT`. We call this phase the ***symbolic assembly.***

Step 2. Assemble the fully-summed rows numerically as a full rectangular matrix with `NASS` rows and `NFRONT` columns. Note that the leading `NASS` columns form a symmetric submatrix.

Step 3. Check if the recommended block pivot is of the kind (full, oxo, or tile) predicted by `MA47A` and compute the number of entries in its rows (degree counts).

Step 4. Find the pivots. If a structured block pivot is recommended, choose simple recommended pivots, one at a time, that satisfy the stability and sparsity test. The number found (counting each as 2) is denoted by `NSTRUC`. If this is not equal to the number of fully-assembled rows, `NASS`, choose simple 1×1 or 2×2 full pivots, one at a time, that satisfy the stability and sparsity test. The total number of pivots found at this step (counting each 2×2 pivot as 2) is held in `NPIV`.

Step 5. If step 4 chose any structured pivots, sort the columns of the front so that the resulting Schur update has the form (2.3).

Step 6. Reserve space for the Schur update. This is at the top of the stack if no structured pivots have been chosen at this stage and the requested block size for using Level 3 BLAS is greater than the order of the Schur update. Otherwise, it is in the frontal matrix after the fully-summed rows.

Step 7. Form the remaining multipliers and compute the Schur update. If there are both structured and full pivots, the calculations corresponding to structured pivots are performed first then the contributions from full pivots are accumulated on top of the earlier Schur update.

Step 8. Scan the generated element matrices of the descendants and add them into the Schur update wherever possible, to make the newly generated element.

Step 9. Stack the generated element (unless it is already on the stack).

Step 10. Stack the fully-summed element (if present).

Step 11. Reorganize the data structure of the factors.

4.2.2 Some data structures

Before discussing the fine detail, it will be useful to discuss the principal data structure that is used by the code. We consider the storage of generated and fully-summed element matrices.

The integer data is held in array `IW`, illustrated in Figure 4.1. The upper triangle of the original matrix is held by rows in tentative pivotal order in the last `NE` positions. The rows that have not yet been pivotal are held from position `PTRIRN`. The space corresponding to already assembled rows is free and can be reclaimed by a data compression. The stack is held before the original matrix from position `ISTK+1`, the top of the stack. For each element, the first entry holds the total number of integer entries for the element, the second holds the number of columns in the leading block of (2.3) (or –1 if the element is fully summed), and the third entry holds the number of columns in the middle block (or the number of rows of a fully-summed element). There then follows the list of variables in the first block, the list for the second block and the list for the third block (note that up to two of these lists may be null). The variables are in no particular order within the blocks. An index is zero if the corresponding row has already been absorbed. The final entry for the element is equal to the first and is needed for data compression (see Section 4.3). Although our depth-first search strategy ensures that required elements are always at the top of the stack, there may be free space in the stack, corresponding to previously absorbed elements. Such free space is flagged by the first and last entries being set to the negation of its length.

The corresponding real data is held in the array `A`, illustrated in Figure 4.2. The upper triangle of the original matrix is held by rows in tentative pivotal order in the last `NE` positions. The rows that have not yet been pivotal are held from position `PTRA`. The space corresponding to already assembled rows is free and can be reclaimed by a data compression. The generated and fully-summed elements are held in a stack from position `ASTK+1`, the top of the stack. For the generated elements, the upper-triangular part of the matrix (2.3) is held by rows. Thus the first part corresponds to $\mathbf{B}_2$ and $\mathbf{B}_3$ stored by rows, then the upper trapezoidal matrix corresponding to the upper triangular part of $\mathbf{B}_1$ together with $\mathbf{B}_4$ stored by rows. These numerical values are preceded and succeeded by a real holding the total number of entries (including these counts). Free space is indicated by setting the first and last entry of the free space to the negation of its length, as for the integer storage.

The main output from the numerical factorization consists of integer and real information on the matrix factorization. For a full pivot, the pivot itself is held first in the form

$$\mathbf{LDL}^T, \tag{4.1}$$

where $\mathbf{L}$ is unit lower triangular and $\mathbf{D}$ is block diagonal with blocks of order 1 or 2. $\mathbf{L}$ and the diagonal part of $\mathbf{D}$ are held packed by columns with the off-diagonal entries of the 2×2 blocks held separately. Following the pivot, the remainder of the pivot rows are held as a full rectangular matrix. For structured pivots, the factorized pivot has the form

$$\begin{pmatrix} \mathbf{0} & \mathbf{A}_1 \\ \mathbf{A}_1^T & \mathbf{A}_2 \end{pmatrix} = \begin{pmatrix} \mathbf{L}_1 & \\ \mathbf{U}_2^T & \mathbf{U}_1^T \end{pmatrix} \begin{pmatrix} \mathbf{0} & \mathbf{D}_1 \\ \mathbf{D}_1 & \mathbf{D}_2 \end{pmatrix} \begin{pmatrix} \mathbf{L}_1^T & \mathbf{U}_2 \\ & \mathbf{U}_1 \end{pmatrix}, \tag{4.2}$$

where $\mathbf{L}_1$ is unit lower triangular, $\mathbf{U}_1$ is upper triangular, $\mathbf{U}_2$ is strictly upper triangular, and $\mathbf{D}_1$ and $\mathbf{D}_2$ are diagonal. The pivot is followed by the remainder of the first block row, excluding leading zero columns, held as a rectangular matrix, then the remainder of the second block row, excluding trailing zero columns. We have discussed this way of representing factors in Section 2.1. More details on the storage of the factors can be found in Section 4.2.15.

4.2.3 Step 1. Symbolic assembly

We construct the list of indices in the current front from position IWPOS of IW (see Figure 4.1) and there is a separate pointer array (PPOS) that is used to indicate the position in the list of each variable. For example, PPOS(9) = 2 would indicate that variable 9 was the second index in the list. For a variable not in the front, the value N+1 is stored in PPOS.

First, the indices of new prospective pivot rows are found from the array PERM, which holds the indices of the variables in permuted order, with the help of the array NODE containing flags from MA47A. We hold the number of such variables in NUMORG. The next indices to be added to the list are those of old prospective pivots that for sparsity or numerical reasons were not used as pivots. These are obtained by scanning all the elements that are unabsorbed descendants of the node (they will be at the head of the stack and their number will have been computed in ELEMS). For each such element that is fully-summed, we add the indices of its rows to the index list for the front.

We now add the indices of the other variables of the front. The first to be added are those from the original matrix for the new prospective pivot rows and are found by a simple scan of each such row. Note that PPOS can be used to immediately identify whether a variable is already recorded in the front since it is set to N+1 for variables not in the front. Each new variable is just added to the next position in IW and its PPOS value set accordingly.

We next scan the descendant elements, incorporating variables from generated elements. An added complication here is that it is possible that not all of a descendant element need be involved in the current assembly. We need to find in which of its blocks (see (2.3)) prospective pivot rows lie. If one lies in the middle block or one lies in block 1 and another lies in block 3, then the whole index list is needed. If the pivot rows are only present in the first block, only indices corresponding to the middle and last block are needed; if pivot rows are only present in the last block, only indices corresponding to the first and middle block are needed. It is easy to identify whether a variable corresponds to a potential pivot row through the use of the array PPOS whose entry will be less than or equal to NASS (the number of fully-summed rows) in such a case. One device that we use to avoid an extra IF test is to set the zeroth entry of PPOS to N+1 so that zero indices, which correspond to already assembled rows, do not need a separate test.

The final step of the symbolic assembly is to scan the descendant elements again, incorporating any remaining indices from the fully-summed elements. We do this last so that any new indices encountered will occur last in the front. This leads to the new pivot rows having trailing zeros, which are easy to exploit. At the end of this stage, we have a list of the `NFRONT` variables in the front (with the `NASS` fully-summed variables first). With this ordering, the prospective pivot rows have the form shown in Figure 4.3 in the case of a recommended structured pivot.

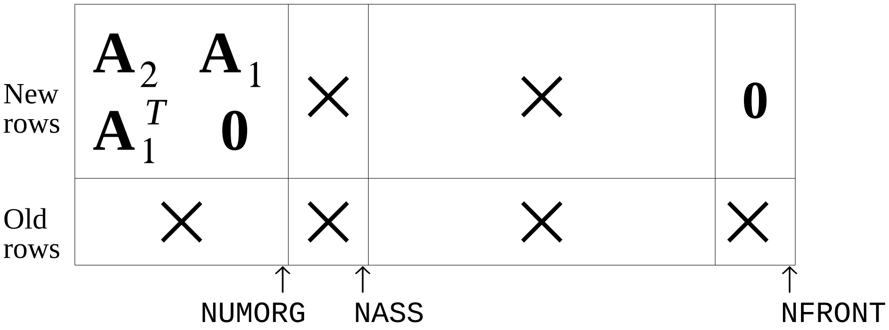

Figure 4.3. The form of the fully-summed rows for a recommended structured pivot.

4.2.4 Step 2. Numerical assembly

The rows of the front corresponding to prospective pivots are assembled as a rectangular array of dimensions `NASS` by `NFRONT` (see Figure 4.3). After checking that there is sufficient space and compressing if necessary, the positions that will be used in the real array `A` (from `APOS`, see Figure 4.2) are initialized to zero. The entries from the prospective pivot rows of the original matrix (all of which will lie in the front) are first assembled. Each row in turn is assembled with entry in row *i* column *j* being placed in position

```
APOS+(PPOS(i)–1)*NFRONT+PPOS(j)–1
```

of `A`. The pointers `PTRIRN` and `PTRA` (see Figures 4.1 and 4.2) are adjusted to show that the rows have been assembled and the space is now free. We now run through the descendant elements, again using the fact that they are at the head of the stack.

A fully-summed descendant element must all be assembled in the current front. We do this by columns to avoid indirect addressing in the inner loop. A generated descendant element must be scanned to assemble only those parts that are present in the prospective pivot rows. We scan the element by rows. If the row is a prospective pivot row (identified by having a `PPOS` value that is less than or equal to `NASS`), it is assembled into the front.

As we perform this assembly, we check to see if all of the descendant element has been assembled. If so, we mark both integer and real storage as free and combine it with any adjacent free space. Marking each free space with (the negation of) its length in its first and last position makes it easy to identify and combine adjacent free space.

We finally make the leading NASS by NASS block symmetric by replacing each off-diagonal entry in the new rows by the sum of itself and the corresponding entry of the transpose. This step is necessary because only one representative of each such pair of entries is stored and it may be in either the upper or lower triangular part.

4.2.5 Step 3. Definition of pivot and degree calculation

If the analyse phase has indicated that we should have a structured pivot at this stage (identified through sign flags on the permutation array NODE), we check to see if the assembled pivot block does in fact have this property since it may have been destroyed by fill-ins caused by a change in pivotal sequence from that recommended by the analyse phase. We check explicitly whether the appropriate part of the front is zero (see Figure 4.3). If it is not, we treat the whole elimination at this stage as unstructured.

4.2.6 Step 4, first part. Selection of structured pivots

If structured pivots are recommended, we choose them one at a time. Each is identified by its off-diagonal entry, which must lie within the block recommended by MA47A. An exhaustive search is made for each one in the recommended block. For the current row being searched, we first find the largest entry in modulus in the candidate block and the largest in the row. Because the front is stored as a full array, the BLAS code ISAMAX can be used for this. If the prospective 2×2 pivot satisfies the pivot tolerance and the calculated Markowitz cost is zero, the candidate is accepted as pivot. If the Markowitz cost is not zero, the potential pivot must now be checked for stability. We first (again using ISAMAX) find the largest entry outside the pivot in its other row so we have now computed the data illustrated in Figure 4.4, where a is zero for an oxo pivot and c and d are the largest entries in their rows outside the pivot. The pivot is then accepted if it satisfies the stability criteria

$$(|a|*|c|+|p|*|d|)*u \le |det|$$

and

$$(|p|*|c|)*u \le |det|$$

where u is the threshold parameter (set by MA47I) and $det = -p*p$, the determinant of the 2×2 pivot, with $|det|/|p|$ greater than the pivot tolerance.

$$\begin{array}{ccccc} 0 & p & & c & \\ p & a & & & d \end{array}$$

Figure 4.4. The candidate pivot and the largest entries of its rows outside the pivot.

If the candidate pivot does not pass this numerical test, we then see if it would be accepted numerically as two 1×1 pivots. The tests for this are that

$$|a| \geq u{*}|d|$$

and that the second 1×1 pivot (det/a) is greater than or equal in modulus to the largest entry in the updated first row of the 2×2 pivot.

If the candidate pivot is acceptable on numerical grounds, we first explicitly permute the rows and columns so that its off-diagonal entry is on the diagonal of the recommended block immediately after the off-diagonal entry of the previous pivot (or in the leading position if it is the first pivot of the block). We then replace the pivot $\begin{pmatrix} 0 & p \\ p & a \end{pmatrix}$ by its inverse $\begin{pmatrix} a/det & -p/det \\ -p/det & 0 \end{pmatrix}$ and update the fully-summed rows using this pivot, calculating and storing the multipliers for the fully-summed rows. Both indices for the pivot are flagged negative for identification by the solve routine. If the candidate pivot does not satisfy the stability criterion, we continue the pivot search to the next row.

If some but not all of the recommended structured pivots are chosen, we perform further permutations to give the form shown in Figure 4.3, where now the leading rows and columns are those of the chosen block pivot.

4.2.7 Step 4, second part. Selection of full pivots

If there are any uneliminated fully-assembled rows, we now look for full pivots. The logic is similar to that for structured pivots although now the candidate block is the whole of the uneliminated part of the leading NASS columns. We search the remaining fully-summed rows for a pivot. For the row being searched, if the diagonal entry is larger than u times the largest entry in the row, it is chosen as pivot. Otherwise, a 2×2 full pivot whose off-diagonal is the largest entry of the row in the candidate block is chosen as the pivot candidate and a similar stability test to that for structured 2×2 pivots is conducted. If a pivot is chosen, it is permuted to the leading position(s) in the uneliminated part of the front, the fully-summed rows are updated, the multipliers for these rows are stored, and the pivot is overwritten by its inverse. If a pivot is not chosen, the search continues to the next row. The pivot search finishes either when all possible pivots have been selected or when a complete search fails to find a pivot.

4.2.8 Step 5. Structured pivot: column sort

If any structured pivots were found, we now sort the non-pivotal part of the pivot rows. First any columns that are zero in the pivot rows are moved to the end and play no part in the operations with the structured pivot. Columns that are zero in the second part of the pivot are moved to the front. For an *oxo* pivot, columns that are zero in the first part of the pivot are moved to the back. The resulting structure is illustrated in Figure 4.5. Note that we are now storing the inverse of the pivot.

$$\begin{array}{|c|c|c|c|c|c|}
\hline
\mathbf{0} & \mathbf{A}_3 & \mathbf{A}_5 & \mathbf{A}_6 & \mathbf{0} & \mathbf{0} \\
\hline
\mathbf{A}_3^T & \mathbf{A}_4 & \mathbf{0} & \mathbf{A}_7 & \mathbf{A}_8 & \mathbf{0} \\
\hline
\multicolumn{1}{c}{} & \multicolumn{1}{c}{} & \multicolumn{1}{c}{\texttt{NUM1}} & \multicolumn{1}{c}{\texttt{NUM2}} & \multicolumn{1}{c}{\texttt{NUM3}\ \uparrow} & \multicolumn{1}{c}{} \\
\multicolumn{1}{c}{} & \multicolumn{1}{c}{} & \multicolumn{1}{c}{} & \multicolumn{1}{c}{} & \multicolumn{1}{c}{} & \multicolumn{1}{c}{\texttt{NCOL}} \\
\end{array}$$

Figure 4.5. The structure of the pivot rows after column sorting.

4.2.9 Step 6. Reserving space for Schur update

If the Schur update has size zero, we immediately proceed to Step 10. Otherwise, we will need to generate appropriate multipliers and form the Schur update. We now reserve the space in our working array to store this matrix.

We first scan the non-pivot columns of the pivot rows to see if any consist of only zeros. This can happen if the maximum possible number of pivots have not been chosen at this stage. This enables us to reduce the amount of work and storage in the following phases. The zero columns are swopped to the end of the front and the number of nonzero columns in the front is recorded in NCOL.

If the pivot is structured, it is possible that, even though there are entries in the pivot rows outside the pivot, the Schur update is zero. This can happen if the form (2.3) consists of a single zero block. If the three groups outside the pivot have NUM1, NUM2 and NUM3 columns (see Figure 4.5), the Schur update is zero if (NUM1+NUM2) or (NUM2+NUM3) is zero. In this case, we must still compute the multipliers in the non-fully-summed part of the front, since they are needed for the solve routine. We then go to Step 10.

At this stage, the size and shape of the Schur update is known but before we reserve space we must know whether Level 3 BLAS will be used in its creation. If the size of the Schur update is less than or equal to the block size (NBLOC) for the Level 3 BLAS and there are no structured pivots, then there is no loss of efficiency in generating only the upper triangle of the Schur update directly and space will be reserved at the top of the stack for this. Otherwise, space will be reserved after the current front. Space is reserved both for a temporary copy of the pivot rows and for the expanded form of the Schur update where the (2,2) block of (2.3) is stored as a full matrix.

In all cases when we reserve space, we check that there is sufficient room in the working arrays and compress if there is not. If there is not enough space after the compress, we reuse the space occupied by the already computed factors. If we can then continue we get an accurate value for the space required although we will not have factorized the matrix. After discarding the factors, if there is still insufficient space, we exit giving the amount of space required to progress beyond the point of failure.

4.2.10 Step 7, part 1. Generation of Schur update for a structured pivot

Our method of calculating the Schur update for a structured pivot was explained in Section 2.3, see equations (2.26) and (2.27). It is the multiplier matrix **M** that we pass to the solve routine, so it is needed anyway. We therefore make a copy of the pivot rows in the space reserved, then overwrite the original copy with the multipliers. Note that $\mathbf{B}_5$ is not needed to calculate $\mathbf{B}_1$, $\mathbf{B}_2$, $\mathbf{B}_3$, and $\mathbf{B}_4$, so need not be copied.

If `NUM1` is greater than or equal to `NBLOC`, we make a call to the Level 3 BLAS routine `SGEMM` to calculate

$$\begin{pmatrix}\mathbf{B}_2 & \mathbf{B}_3\end{pmatrix} = \mathbf{B}_{11}^T \begin{pmatrix}\mathbf{B}_7 & \mathbf{B}_8\end{pmatrix}.$$

If `NUM1` is less than `NBLOC`, we compute the same result by explicit Fortran code. Similarly, if `NUM3` is greater than or equal to `NBLOC`, we make a call to `SGEMM` to calculate

$$\mathbf{B}_4 = \mathbf{B}_{12}^T \mathbf{B}_8,$$

and otherwise use explicit Fortran code.

We next compute

$$\mathbf{B}_1 = \begin{pmatrix}\mathbf{B}_9^T & \mathbf{B}_{12}^T\end{pmatrix} \begin{pmatrix}\mathbf{B}_6 \\ \mathbf{B}_7\end{pmatrix}.$$

There are no Level 3 BLAS that produce a symmetric matrix from the multiplication of two unsymmetric matrices and if we just use `SGEMM` we will needlessly compute the lower triangle $\mathbf{B}_1$. As explained in Section 2.3, we therefore divide the `NUM2` rows into blocks of `NBLOC` rows and a final block of up to `NBLOC` rows. We compute the block upper triangular part of each strip of $\mathbf{B}_1$ in turn, using `SGEMM` for each strip except the last. This limits the amount of unnecessary matrix entries calculated to (`NBLOC*(NBLOC-1)`) times the number of strips (block rows). If no full pivots have been chosen, we can then proceed to Step 8.

4.2.11 Step 7, part 2. Generation of Schur update for a full pivot

Our method of calculating the Schur update for a full pivot was explained in Section 2.3. If we are generating the Schur update directly on the stack, we simply scan the non-fully-summed columns for each pivot in turn (which may be 1×1 or 2×2), compute the multipliers and compute the corresponding additions to the upper triangle of the Schur update in place.

Otherwise, we first compute the multipliers **M** and store a copy of the pivot rows **DM** in the space reserved. We divide the resulting Schur update matrix into strips of `NBLOC` rows and a final block of up to `NBLOC` rows. We call `SGEMM` to calculate the block upper triangular part of each strip in turn using the copy of the pivot row and the newly computed multipliers, except the last, for which explicit Fortran code is employed.

4.2.12 Step 8. Absorption of elements

We scan the descendant elements of the current node to see if any can be fully absorbed into the newly generated Schur update. For each element, we can find out whether it will be absorbed by a test that is linear in the order of the element. Clearly, absorption cannot take place if any index is not in the current element. By setting `PPOS(0)` to zero for this part of the code, we can identify such an index by a `PPOS` value equal to `N+1` and this is used as a cheap test to terminate the absorption process. We scan the index list of the descendant element, starting with the second block. All unabsorbed indices in this block of the descendant element must be in the index list of the second block of the new element if the descendant element can be absorbed. If that requirement is satisfied, we then scan indices from the first block. All these indices must lie entirely within the first two or last two blocks of the new element for absorption. Finally, when checking indices from the last block, if any are in the first block of the new element then none from the first block of the descendant element can be and if any are in the last block of the new element then none from the first block of the descendant element can be.

If the descendant element passes the absorption test, it is added by rows into the current generated element. The only complications with this assembly are to determine the position in the upper trapezoidal part and to find the position in array `A` of the entry, which will also depend on whether the element is on the stack or not. Finally, the integer and real data for the absorbed element must now be flagged as being free space and amalgamated with any adjacent free space.

4.2.13 Step 9. Stacking the generated element

If this step creates a generated element, we increment the number of descendant nodes for the parent node (held in `ELEMS`). If the element is not already on the stack, we must place it there. We stack backwards from the last row and since this is a form of data compression we always know there will be sufficient space in the real array. We also stack the integer data, again in reverse order, although here we must first check there is sufficient room and compress if necessary.

4.2.14 Step 10. Stacking the fully-summed block

If this step creates a fully-summed element, we first scan it by columns and remove any totally zero column. The count of the parent node for the number of descendants (held in `ELEMS`) is increased by 1. Both integer and real arrays are tested to see if there is sufficient space and compressed if necessary. The fully-summed element is then copied by rows onto the stack and the column indices are moved similarly on to the integer stack.

4.2.15 Step 11. Reorganization of the data structure for the solve routine

When we store the factors, we need to reorganize the data from the data structure of the front to that of the factors, as defined in Section 4.2.2. We first check any columns that are fully summed but not in the pivot (columns NPIV+1 to NASS) and remove any that are all zero. The main complication in the storage of the factors occurs if we have chosen both structured and full pivots at this stage. In this case, we store the factors as two sets, first the structured pivots, then the full ones. We now discuss in turn the integer and real storage.

The integer storage for a pivot consists of

(1) The number of columns (flagged negative for a structured pivot).

(2) The number of rows (each simple structured pivot counts as 2). If the pivot is an oxo pivot this is flagged negative.

(3) The number of columns in the third block (block NUM3). (Structured pivots only.)

(4) The number of columns in the first block (block NUM1). (Oxo pivots only.)

(5) The indices of the columns in the pivot.

So, for tile pivots the index list must be moved up by one (space was reserved initially for an oxo pivot) and for full pivots by two. If both exist, the appropriate indices for the full pivot are copied immediately after the structured pivot and the number of pivot blocks in the factors is incremented. A further complication occurs when there is an oxo pivot with some fully assembled columns that are not in the pivot block (NASS > NSTRUC). Then the NUM1 indices of the first block must be moved before those corresponding to the fully-summed non-pivotal columns.

$$\begin{array}{ccccccccccccccc}
d_2 & u_2 & d_1 & u_1 & n_1 & n_1 & 0 & 0 & r_1 & r_1 & r_1 & r_1 & r_1 & r_1 & r_1 \\
x & d_2 & l_1 & d_1 & n_1 & n_1 & 0 & 0 & r_1 & r_1 & r_1 & r_1 & r_1 & r_1 & r_1 \\
x & x & 0 & 0 & n_2 & n_2 & r_2 & r_2 & r_3 & r_3 & r_3 & r_3 & 0 & 0 & 0 \\
x & x & x & 0 & n_2 & n_2 & r_2 & r_2 & r_3 & r_3 & r_3 & r_3 & 0 & 0 & 0
\end{array}$$

Figure 4.6. The storage held in the frontal matrix for a block structured pivot.

The factors of the pivot are stored as shown in Section 4.2.2. For a structured pivot, we first store the $\mathbf{L}_1$, $\mathbf{D}_1$, and $\mathbf{U}_1$ matrices of (4.2). Because of the ordering performed after choosing the structured pivots, these can be found easily. To illustrate this, we show in Figure 4.6 the storage held in the frontal matrix for a block structured pivot consisting of two tile pivots (NSTRUC=4). In Figure 4.6, a lower-case subscripted letter d_1, d_2, l_1, u_1, u_2 indicates an entry in the corresponding upper case matrix of (4.2), a subscripted *n* indicates a fully-summed uneliminated variable, and a subscripted *r* signifies the entries in pivot rows outside the pivot block. We denote by *x* the lower trapezoidal entries which are held but not referenced. Note that the zero blocks are swopped relative to Figure 4.5. This is because we are now considering multipliers rather than original pivot rows.

Thus $\mathbf{L}_1$, $\mathbf{D}_1$, and $\mathbf{U}_1$ are first copied as a full NSTRUC/2 by NSTRUC/2 block by rows. For a tile pivot, we then store the strictly upper triangular matrix $\mathbf{U}_2$ by rows followed by the diagonal matrix $\mathbf{D}_2$. After this we store `NSTRUC/2` zeros (corresponding to the (2,2) entries of the inverse pivots). Although this is not necessary for the initial factorization, we need this storage if the matrix is later to be modified. For a similar reason, we store `NSTRUC` zeros in the case of an oxo pivot. The rectangular block is then stored, respecting the structure illustrated in Figure 4.6. The entries are stored in the sequence n_1, r_1, r_2, n_2, r_3.

The real factors corresponding to any full pivots are then stored. The upper triangle of the pivot is stored first in packed form by rows followed by the rest of the pivot rows in full form by rows. After all assembly tree nodes have been processed, subroutine `MA47W` will postprocess this in preparation for the solve routine `MA47C`.

4.3 MA47P and MA47S: data compression routine

There are several points within `MA47O` where we can find that we do not have enough contiguous free space to accommodate new factors or generated elements. If this happens, we first recover space from absorbed elements or already processed parts of the original matrix and then check to see if there is sufficient space. This is performed by subroutines `MA47P` and `MA47S`, the former for compressing data in array `IW`, the latter for the reals in array `A`. The logic of both is identical so we just discuss `MA47P`.

We scan from the bottom (rightmost entry) of the stack backwards and move data to the right to overwrite free space in the stack and original entries used since the last compress. An integer (`MOVE`) is set to the number of positions which data will be moved. It is initialized (by `MA47O`) to the number of entries in the original rows being overwritten and is incremented each time free space (identified by the negative length flags) is encountered. Any active data encountered is moved (in blocks corresponding to the positive length flags) `MOVE` positions to the right. The scan completes when we reach the top (leftmost end) of the stack.

4.4 MA47U: print the factors

The code for printing the factors in a pleasing format is quite complicated, so we have written it as a separate subroutine.

4.5 MA47W: postprocess the factors from full pivots

At the end of subroutine `MA47O`, when all tree nodes have been processed, we call `MA47W` to postprocess the way the full pivots are held. This is done to enable better performance in `MA47`.

The factors are scanned. We just skip over blocks corresponding to structured pivots (identified by first entry being flagged negative) although the computation of how much to skip is not trivial. For blocks corresponding to full pivots, the off-diagonal entries of two by two pivots are placed directly after the factors and their original position is set to zero. Also the entries of **L** are negated.

4.6 MA47X and MA47Y: auxiliary routines.

Subroutines MA47X and MA47Y are used by MA47O to order the pivot block when some but not all possible structured pivots are found at some tree node. MA47X zeros a triangular array, MA47Y copies a matrix (or its transpose) to another matrix where the matrices are full or triangular.

5 Solution

In this section, we describe the subroutines that solve systems, given the factorization.

5.1 MA47C: driver for the solution phase

MA47C is a short driver routine. It first optionally prints the factorization using MA47U and the right-hand-side vector. Unless the matrix is zero, it calls MA47Q for forward substitution and MA47R for back-substitution. Finally, it optionally prints the computed solution.

5.2 MA47Q: forward substitution

MA47Q performs the forward substitution operations a block pivot at a time. The form of the block pivot is as discussed in Sections 2.1, 4.2.2, and 4.2.15. The diagonal pivot block is given by equations (4.1) or (4.2) and is stored as described near the end of Section 4.2.15 where the storage for rest of the block pivot row/column is also discussed (see Figure 4.6 and following text).

We first determine the type of the block pivot (full, tile, or oxo) and the number of rows (pivots) and columns. If the number of pivots is greater than the controlling parameter ICNTL(7), then the floating-point arithmetic will be performed using direct addressing, and we will use Level 2 BLAS routines. To do this, relevant entries of the (modified) right-hand side vector must first be loaded into a working vector of length the number of columns in the block pivot. At the end of the block pivot step, this vector will be scattered back to the main vector. All the calculations using Level 2 BLAS are performed on the working vector.

If the number of pivots is less than or equal to ICNTL(7) then we perform the same numerical calculations but reference the (modified) right-hand sides in situ using indirect addressing. Since the logic for the indirect addressing is identical to that for using direct addressing (although the coding is much more complicated), we only discuss the use of direct addressing in this section and the next.

For a full pivot, the pivot block is given by (4.1) and the diagonal entries of **D** and the off-diagonal entries of the unit triangular matrix **L** are stored by rows in a packed triangular form, followed by the non-pivotal columns by rows. The off-diagonal entries of full 2×2 pivots are stored separately at the end of the storage for the factors. The forward substitution for such block pivots is then performed by a call to `DTPSV` (to solve for **L** of (4.1)) followed by a `DGEMV` for the off-diagonal rectangular matrix.

For a structured pivot, the pivot block is factorized as in (4.2) and is stored as discussed in Section 4.2.15. Thus $\mathbf{L}_1$, $\mathbf{D}_1$, and $\mathbf{U}_1$ are stored as a square matrix so packed form BLAS are not required, while $\mathbf{U}_2$, if present, is stored in packed form. For a tile pivot, the forward substitution then proceeds by solving for $\mathbf{L}_1$ using `DTRSV` followed by `DGEMV` with the rectangular matrix comprising the n_1 and r_1 of Figure 4.6. Then the working vector is multiplied by $\mathbf{U}_2$ using `DTPMV` and we solve for $\mathbf{U}_1^T$ using `DTRSV.` Finally `DGEMV` is used for the off-diagonal entries.

In the case of an oxo pivot, the calculation simplifies to two calls to `DTRSV` for $\mathbf{L}_1$ and $\mathbf{U}_1^T$ followed by two calls to `DGEMV` for the n_1 r_1 and r_2, n_2, r_3 off-diagonals respectively.

5.3 MA47R: back-substitution

`MA47R` performs the back-substitution operations block pivot by block pivot. As for `MA47Q`, each is performed either with the help of a work array using Level 2 BLAS or by Fortran code using indirect addressing. Which is chosen is again under the control of the parameter `ICNTL(7)`. We assume that the reader has already read Section 5.2 and so is familiar with the factorizations and data structures in use.

For a full pivot, we must first solve for the block diagonal matrix **D**, the only complication being that the off-diagonal entries of 2×2 pivots are stored separately. The non-pivotal columns are then accounted for through a call to `DGEMV` before a call to `DTPSV` effects solution with $\mathbf{L}^T$.

For a structured pivot, the operation of the middle block in (4.2) is the same for both tile and oxo pivots since $\mathbf{D}_2$ is stored for both to permit subsequent matrix modification (it will consist of explicit zeros for an oxo pivot). The off-diagonal entries are then handled as before using two calls to `DGEMV`.

The subsequent computation for a tile pivot then consists of a call to `DTRSV` (to solve for $\mathbf{U}_1$), a call to `DTPMV` (for multiplication by $\mathbf{U}_2$), and finally a call to `DTRSV` (for solution by $\mathbf{L}_1^T$). Again the case for oxo pivots simplifies to two `DTRSV` calls because $\mathbf{U}_2$ is null.

6 Numerical Experience

6.1 Introduction

In this section, we examine the performance of the `MA47` code on a range of test problems on a SUN SPARCstation 10 and a CRAY Y-MP. We study the influence of various parameter settings on the performance of `MA47` and determine the values for the defaults. We also compare the `MA47` code with the code `MA27`.

Case	Identifier	Order *m*	Order *m*+*n*	Number of entries	Description
1	BRITGAS	3102	5802	15282	British gas pipe network distribution problem.
2	BIGBANK	2230	3342	8056	Nonlinear network problem.
3	MINPERM	8347	16551	16863	Minimize the permanent of a doubly stochastic matrix.
4	SVANBERG	14000	21000	91000	Structural optimization.
5	BRATU2D	5184	10084	29684	Finite-difference discretization of nonlinear PDE on unit square.
6	BRATU3D	4913	8288	28538	Finite-difference discretization of nonlinear PDE on unit cube.
7	GRIDNETC	7564	11408	30256	A nonlinear network problem on a square grid.
8	QPCSTAIR	614	970	4617	STAIR LP with additional convex Hessian.
9	KSIP	1021	2022	22023	Semi-infinite QP.
10	AUG3DQP	3873	4873	10419	QP from nine-point formulation of 3-D PDE.

Table 1. The CUTE matrices used for performance testing.

Case	Identifier	Order *m*	Order *m*+*n*	Number of entries	Description
11/17	FFFFF800	1028	1552	7429	LP. Oil industry.
12/18	PILOT	4860	6301	40235	LP. Energy model from Stanford.
13/19	ORSIRR 2	886	1772	6861	Oil reservoir simulation.
14/20	JPWH 991	991	1982	7018	Circuit physics modelling.
15/21	BCSSTK27	1224	2448	29899	Structural engineering. Buckling analysis.
16/22	NNC1374	1374	2748	9980	Nuclear reactor core modelling.

Table 2. Matrices used for augmented systems. Matrices 11 to 16 have **I** as leading block and matrices 17 to 22 have **D**.

As always, the selection of test problems is a compromise between choosing sufficient to obtain meaningful statistics while keeping run times and this section of the report manageable. In Tables 1 and 2, we list the test problems used for the numerical experiments in this section. In choosing this set, many runs were performed on other matrices so we do feel that this selection is broadly representative of a far larger set. Our set of 22 matrices can be divided into three distinct sets. The first (matrices 1 to 10), in Table 1, are obtained from the CUTE collection of nonlinear optimization problems (Bongartz, Conn, Gould, and Toint, 1993). The problems in that set are parameterized, and we have chosen the parameters to obtain a linear problem of order between about 1000 and 20000. In each case, we solve a linear system whose coefficient matrix is the Kuhn-Tucker matrix of the form

$$\begin{pmatrix} \mathbf{H} & \mathbf{A} \\ \mathbf{A}^T & \mathbf{0} \end{pmatrix},$$

where **H** is an $m \times m$ symmetric positive definite matrix and **A** is of dimension $m \times n$. We are grateful to Ali Bouaricha and Nick Gould for extracting these matrices for us from CUTE. The second and third sets are obtained by forming augmented systems of the form

$$\begin{pmatrix} \mathbf{I} & \mathbf{A} \\ \mathbf{A}^T & \mathbf{0} \end{pmatrix} \quad \text{and} \quad \begin{pmatrix} \mathbf{D} & \mathbf{A} \\ \mathbf{A}^T & \mathbf{0} \end{pmatrix}, \quad \text{with} \quad \mathbf{D} = \begin{pmatrix} \mathbf{I} & \\ & \mathbf{0} \end{pmatrix}$$

respectively, where the matrix **A** is from the Harwell-Boeing Collection (Duff, Grimes, and Lewis, 1992) or the netlib LP collection (Gay, 1985). The matrix **D** has m–n unit diagonal entries and n zeros on the diagonal. We use the six matrices from these collections that are shown in Table 2, which gives us another twelve test cases according to the two forms of augmentation shown above. In all cases, the dimensions are given in the tables and are the number of rows in **A** and the total order of the augmented system. The number of entries is the total number in the upper triangular part of the augmented matrix.

Before conducting a systematic study of the parameters, we first experimented to see the effect of using an estimate of the Markowitz costs from the analyse phase to decide whether to allow prospective pivots in the factorize phase. The logic of this scheme, which was proposed by Duff, Reid, Gould, Scott, and Turner (1991), is that changes to the pivotal sequence from the analyse phase may mean that later pivots in their recommended position in the pivotal sequence would be poor choices. There is, however, a counter-balancing effect whereby even if the pivot is very much poorer on sparsity grounds than predicted, it may be worse to delay using it because of further build up of unanticipated fill-in. In effect, it may be better to “bite the bullet”and take the bad pivot early rather than late.

When we tested this option on the examples in Table 1, we found that there was usually little to choose between including the Markowitz test or not. However, in most cases it was slightly worse to use the Markowitz test and, in one case, the factorization time was increased by nearly a factor of 14 and the total storage for the factorization by nearly a factor of 6. Since there were no examples so dramatically favouring the Markowitz test, we have decided to drop it from the code. We have, however, left in the structure for the test and have only commented out the test itself since we believe that possible future changes to how we handle fully summed blocks might make the test useful in a later version of the code.

In our program for running the numerical experiments, we have an option to prescale the matrices using Harwell Subroutine `MC30`. In nearly all the cases, there is very little difference in performance or accuracy whether scaling is used or not. There were, however, two cases where additional pivoting on the unscaled systems caused a significant increase in time and storage for the factorization with only one case significantly the other way. We feel more comfortable assessing the performance on scaled systems so we use this option in all the runs in this paper.

In the following subsections, we examine the relative performance when a single parameter is changed by means of the median, upper-quartile, and lower-quartile ratios over the 22 test problems. We use these values rather than means and variances to give some protection against stray results caused either by the timer or by particular features of the problems. We remind the reader that half the results lie between the quartile values. Full tables of ratios are available by anonymous ftp from numerical.cc.rl.ac.uk (130.246.8.23) in the file pub/reports/ma47.tables. In Section 6.2, we consider the effect of choosing the option to restrict the pivot choice to a small number of columns, and we examine the effect of altering the pivot threshold in Section 6.3. We study the effect of changing the amount of node amalgamation in Section 6.4. In Sections 6.5 and 6.6 we examine the use of higher level BLAS; the Level 3 BLAS in the factorization in the first section and the use of Level 2 BLAS in the solution phase in the second. Finally, in Section 6.7, we show the performance of our code with default parameter values on the test problems and compare it with the Harwell Subroutine Library code `MA27`.

6.2 Effect of restricting pivot selection

If we restrict the number of rows that we search to find a pivot, we might expect to reduce the time for the analyse phase at the cost, perhaps, of more storage and time in the factorization. Clearly, the choice depends on the relative importance of the phases. We show the results of runs varying the search limit in Tables 3 to 5. The times in these tables are in seconds on the SUN SPARCstation 10.

	Total storage		Storage for factors		Time (SUN)			
	Predicted	Actual	Predicted	Actual	Analyse	Factorize	Solve	One-off
lower q.	0.97	1.00	0.98	1.00	0.95	0.98	0.97	0.97
median	1.00	1.00	1.00	1.00	0.99	1.00	1.00	1.00
upper q.	1.00	1.04	1.00	1.02	1.01	1.06	1.03	1.03

Table 3. Results with search limit of 2 rows divided by those with limit of 4.

	Total storage		Storage for factors		Time (SUN)			
	Predicted	Actual	Predicted	Actual	Analyse	Factorize	Solve	One-off
lower q.	1.00	0.97	1.00	0.99	0.98	0.96	0.99	0.97
median	1.00	1.00	1.00	1.00	0.99	1.01	1.00	1.01
upper q.	1.08	1.09	1.02	1.01	1.01	1.10	1.03	1.07

Table 4. Results with search limit of 4 rows divided by those with limit of 10.

	Total storage		Storage for factors		Time (SUN)			
	Predicted	Actual	Predicted	Actual	Analyse	Factorize	Solve	One-off
lower q.	1.00	1.00	0.97	1.00	0.73	0.95	0.96	0.94
median	1.00	1.00	1.00	1.00	0.95	1.00	0.98	0.99
upper q.	1.06	1.76	1.00	1.35	1.00	1.86	1.28	1.43

Table 5. Results with search limit of 10 divided by those with no restriction.

The medians are near 1.0, showing slight gains to the analyse time by restricting the pivot search at a cost of a slightly more expensive factorization. The costs for a “one-off”run of a single analysis followed by one factorization and solution, show that there is really quite a balance in the competing trends. The upper quartile figures markedly support using a Markowitz count so we have decided to keep that as the default and use it for the later runs in this paper.

6.3 Effect of change in pivot threshold

In MA27, a value of 0.1 was chosen for the default value of the threshold parameter, u (see equations (2.20) and (2.21)), since smaller values gave little appreciable benefit to sparsity in the experiments that we conducted at that time. However, the more complicated data structures in MA47 and the greater penalties for not being able to follow the pivotal sequence recommended by the analyse phase penalizes higher values of u to a greater extent than in the earlier code. We investigate this in Tables 6 to 10.

	Storage required		Time (SUN)		
	Total	Factors	Analyse	Factorize	Solve
lower q.	1.00	1.00	1.00	1.08	1.01
median	1.14	1.09	1.00	1.78	1.07
upper q.	2.14	1.53	1.00	3.32	1.42

Table 6. Results with threshold u set to 0.5 divided by those with u = 0.1.

	Storage required		Time (SUN)		
	Total	Factors	Analyse	Factorize	Solve
lower q.	1.00	1.00	0.96	0.99	1.00
median	1.02	1.05	0.98	1.07	1.03
upper q.	1.28	1.23	1.00	1.97	1.22

Table 7. Results with threshold u set to 0.1 divided by those with u = 0.01.

	Storage required		Time (SUN)		
	Total	Factors	Analyse	Factorize	Solve
lower q.	1.00	1.00	0.94	1.00	0.98
median	1.00	1.03	0.99	1.11	1.02
upper q.	1.14	1.16	1.03	1.63	1.12

Table 8. Results with threshold u set to 0.01 divided by those with u = 0.001.

	Storage required		Time (SUN)		
	Total	Factors	Analyse	Factorize	Solve
lower q.	1.00	1.00	1.00	1.03	1.02
median	1.00	1.00	1.03	1.08	1.07
upper q.	1.07	1.05	1.08	1.42	1.10

Table 9. Results with threshold u set to 0.001 divided by those with u = 0.0001.

	Storage required		Time (SUN)		
	Total	Factors	Analyse	Factorize	Solve
lower q.	1.00	1.00	0.99	1.00	0.99
median	1.00	1.00	1.00	1.01	1.00
upper q.	1.01	1.02	1.00	1.09	1.03

Table 10. Results with threshold u set to 0.0001 divided by those with u = 0.00001.

We also, of course, monitored the numerical performance for all these runs. Although the results from using a threshold value of 0.5 were better than 0.1 for a couple of test problems, notably the NNC1374 example, it was substantially more expensive to use such a high value for the threshold. For lower values of the threshold, the scaled residual was remarkably flat for all of the test problems until values of the threshold less than 10^{-6} when poorer results were obtained on three of the examples.

From the performance figures in Tables 6 to 10, the execution times and storage decline almost monotonically but have begun to level off by about 0.0001 (although there are a couple of outliers). However, we are anxious not to compromise stability and recognize that our numerical experience is necessarily limited. We have therefore decided to choose as default a threshold value of 0.001 since, for many problems, much of the sparsity benefit has been realized at this value.

6.4 Effect of node amalgamation

The node amalgamation parameter, discussed in Section 3.7, controls the merging of neighbouring nodes in the assembly tree to obtain larger blocks (and more eliminations within each node) at the cost of extra fill-in and arithmetic. No node is amalgamated with another unless its number of variables is less than the parameter value. This feature was also present in the MA27 code. One intention of performing more amalgamations is that there should be more scope for the use of Level 3 BLAS which might be expected to benefit platforms with efficient Level 3 BLAS kernels.

We show the results of running with various levels of amalgamation in Tables 11 to 13. As expected, there is a difference of performance between the two machines. On the CRAY, a higher amount of amalgamation is beneficial while, on the SUN, there is a slight gain from some amalgamation but the effect is reversed before the amalgamation parameter reaches 10. In the interests of choosing a default value that is satisfactory on several platforms (even if not optimal) we have chosen the value 5. Note that people running extensively on a vector machine, like a CRAY, may wish to increase this (say to 20).

		Total storage		Storage for factors		Time			
		Predicted	Actual	Predicted	Actual	Analyse	Factorize	Solve	One-off
CRAY	lower q.	0.78	0.88	0.77	0.84	1.00	1.00	0.73	1.00
	median	0.95	0.95	0.87	0.92	1.01	1.13	0.83	1.03
	upper q.	1.00	1.00	1.00	1.00	1.03	1.37	1.00	1.10
SUN	lower q.	0.78	0.90	0.77	0.84	1.00	1.00	0.82	1.00
	median	0.95	0.95	0.87	0.92	1.02	1.03	0.92	1.01
	upper q.	1.00	1.00	1.00	1.00	1.03	1.11	1.00	1.07

Table 11. Results with no amalgamation divided by those with parameter 5.

		Total storage		Storage for factors		Time			
		Predicted	Actual	Predicted	Actual	Analyse	Factorize	Solve	One-off
CRAY	lower q.	0.84	0.85	0.80	0.81	1.00	1.00	1.00	1.00
	median	0.91	0.96	0.87	0.92	1.00	1.02	1.11	1.01
	upper q.	1.00	1.00	1.00	1.00	1.01	1.14	1.25	1.05
SUN	lower q.	0.84	0.86	0.80	0.81	0.98	0.95	0.91	0.96
	median	0.91	0.96	0.87	0.92	0.98	0.97	0.96	0.98
	upper q.	1.00	1.00	1.00	1.00	1.00	1.00	1.01	0.99

Table 12. Results with amalgamation parameter set to 5 divided by those with parameter 10.

		Total storage		Storage for factors		Time			
		Predicted	Actual	Predicted	Actual	Analyse	Factorize	Solve	One-off
CRAY	lower q.	0.82	0.83	0.77	0.80	1.00	1.00	1.00	1.00
	median	0.89	0.94	0.86	0.88	1.00	1.00	1.10	1.00
	upper q.	1.00	1.00	1.00	1.00	1.01	1.05	1.22	1.01
SUN	lower q.	0.82	0.83	0.77	0.80	1.00	0.87	0.91	0.98
	median	0.89	0.94	0.86	0.87	1.02	0.98	0.98	1.00
	upper q.	1.00	1.00	1.00	1.00	1.05	1.03	1.00	1.03

Table 13. Results with amalgamation parameter set to 10 divided by those with parameter 20.

6.5 Effect of change of block size for Level 3 BLAS during factorization

A major benefit of multifrontal methods is that the floating-point arithmetic is performed on dense submatrices. In particular, if we perform several pivot steps on a particular frontal matrix, Level 3 BLAS can be used. However, in the present case, we also wish to maintain symmetry and the current Level 3 BLAS suite does not have an appropriate kernel. We thus, as discussed in Section 2.3, need to split the frontal matrix into strips, starting at rows 1, *b*+1, 2*b*+1, ... so that we can use Level 3 BLAS without doubling the arithmetic count. In fact, in a block of size *b* on the diagonal, the extra work is *b*∗(*b*–1) floating-point operations. Clearly this means that while we would like to increase *b* for Level 3 BLAS efficiency, by doing so we increase the amount of arithmetic. In this section, we examine the trade off between these competing trends.

We show results for various values of the block-size parameter, *b*, in Tables 14 to 16. It would seem, from these results, that a modest value is best and we choose 5 as the default value on the basis of these figures.

		Analyse	Factorize	Solve	One-off
CRAY	lower q.	1.00	1.01	0.99	1.00
	median	1.00	1.05	1.00	1.01
	upper q.	1.00	1.14	1.00	1.03
SUN	lower q.	1.00	1.00	0.98	0.99
	median	1.01	1.02	1.00	1.01
	upper q.	1.01	1.06	1.02	1.03

Table 14. Times with block-size parameter *b* set to 1 divided by those with *b* = 5.

		Analyse	Factorize	Solve	One-off
CRAY	lower q.	1.00	0.95	1.00	0.98
	median	1.00	0.98	1.00	0.99
	upper q.	1.00	1.00	1.00	1.00
SUN	lower q.	0.99	0.99	0.99	0.99
	median	1.00	0.99	1.00	0.99
	upper q.	1.01	1.01	1.01	1.00

Table 15. Times with block-size parameter *b* set to 5 divided by those with *b* = 10.

		Analyse	Factorize	Solve	One-off
CRAY	lower q.	1.00	0.73	1.00	0.91
	median	1.00	0.90	1.00	0.97
	upper q.	1.00	1.00	1.00	1.00
SUN	lower q.	0.98	0.92	0.98	0.94
	median	1.00	0.96	0.99	0.98
	upper q.	1.01	0.99	1.00	1.00

Table 16. Results with block-size parameter *b* set to 10 divided by those with *b* = 20.

6.6 Effect of change of block size for level 2 BLAS during solution

In a single block pivot stage of the solution phase, one can either use indirect addressing for every operation or can load the appropriate entries of the right-hand side vector into a small full vector corresponding to the rows in the current front, update this vector with Level 2 BLAS operations, and finally scatter it back to the full vector.

We have experimented with the parameter that determines whether to use indirect or direct addressing in the solution phase. Direct addressing is used (and Level 2 BLAS called) if the number of pivots at a step is more than this parameter. Thus, for high values of the parameter, there will be less use of Level 2 BLAS. We show a summary of our results in Table 17. As can be seen, the results are very flat. On the largest of our problems, there was some gain by using a value of 4 for the Level 2 blocking and so we have chosen that as our default.

Parameter ratio		1:2	2:4	4:8	8:16
CRAY	lower q.	1.00	1.00	1.00	1.00
	median	1.04	1.03	1.03	1.00
	upper q.	1.10	1.06	1.13	1.07
SUN	lower q.	0.97	0.99	0.94	0.94
	median	1.00	1.00	0.98	0.98
	upper q.	1.02	1.03	1.06	1.04

Table 17. Ratios of solve times with different values for the parameter for indirect addressing.

6.7 Performance of MA47 and comparison with MA27

In the past five subsections, we have considered the effect of various controlling parameters on the performance of MA47. We now examine the performance of our code with the default values for the parameters on both the SUN SPARCstation 10 and the CRAY Y-MP. The storage counts and times for the problems of Tables 1 and 2 are shown in Tables 18 and 19. It was our original intention that this new MA47 code would replace MA27 in the Harwell Subroutine Library. However, the added complexity of the new code will penalize it if it is unable to take advantage of the structure. We thus might expect that sometimes MA47 would be better and sometimes MA27. We illustrate this by showing the comparison ratios for the two codes in Tables 20 and 21.

	Storage (millions of words)						
	Total	For factors			Time		
	Predicted	Predicted	Actual	Analyse	Factorize	Solve	One-off
BRITGAS	0.516	0.591	0.283	0.345	12.400	10.960	0.156
BIGBANK	0.091	0.091	0.068	0.068	1.220	0.367	0.044
MINPERM	0.117	0.117	0.095	0.095	0.600	0.447	0.127
SVANBERG	1.065	0.895	0.944	0.774	12.430	3.170	0.363
BRATU2D	1.218	1.218	0.949	0.951	80.350	8.290	0.333
BRATU3D	4.085	3.939	2.324	2.180	18.000	70.530	0.650
GRIDNETC	0.425	0.425	0.358	0.360	1.620	1.450	0.190
QPCSTAIR	0.126	0.139	0.071	0.078	0.443	0.800	0.029
KSIP	0.144	0.696	0.106	0.198	4.680	19.280	0.067
AUG3DQP	0.290	0.290	0.198	0.198	0.700	1.160	0.092
FFFFF800	0.180	0.168	0.121	0.088	1.240	1.080	0.037
PILOT	1.582	1.738	0.796	0.838	9.800	15.070	0.285
ORSIRR 2	0.435	0.439	0.244	0.246	1.480	2.870	0.082
JPWH 991	0.836	0.941	0.428	0.352	1.570	6.990	0.111
BCSSTK27	0.179	0.179	0.096	0.096	0.945	0.210	0.043
NNC1374	0.380	1.317	0.318	0.476	1.880	29.890	0.160
FFFFF800	0.041	0.059	0.029	0.044	0.775	1.290	0.023
PILOT	0.287	0.289	0.164	0.192	4.970	2.760	0.105
ORSIRR 2	0.283	0.929	0.130	0.300	1.210	25.140	0.106
JPWH 991	0.573	0.993	0.135	0.259	2.370	24.080	0.100
BCSSTK27	0.172	0.172	0.097	0.097	0.975	0.216	0.043
NNC1374	0.110	0.245	0.086	0.138	2.470	5.540	0.062

Table 18. Performance of runs of MA47 code on the SUN SPARC-10.

	Storage (millions of words)						
	Total	For factors		Time			
	Predicted	Predicted	Actual	Analyse	Factorize	Solve	One-off
BRITGAS	0.516	0.593	0.283	0.354	9.634	3.460	0.060
BIGBANK	0.091	0.091	0.068	0.068	0.986	0.192	0.034
MINPERM	0.117	0.117	0.095	0.095	16.271	0.348	0.091
SVANBERG	1.065	1.065	0.944	0.953	10.064	1.629	0.267
BRATU2D	1.218	1.218	0.949	0.951	75.635	1.789	0.116
BRATU3D	4.085	3.939	2.324	2.180	16.686	7.169	0.108
GRIDNETC	0.425	0.425	0.358	0.360	1.591	0.706	0.152
QPCSTAIR	0.126	0.139	0.071	0.078	0.390	0.276	0.011
KSIP	0.144	0.696	0.106	0.198	6.245	2.966	0.024
AUG3DQP	0.290	0.290	0.198	0.198	0.584	0.409	0.061
FFFFF800	0.180	0.168	0.121	0.087	1.163	0.346	0.018
PILOT	1.582	1.738	0.796	0.838	9.057	2.775	0.082
ORSIRR 2	0.435	0.439	0.244	0.246	1.385	0.561	0.022
JPWH 991	0.836	0.940	0.428	0.352	1.298	1.253	0.023
BCSSTK27	0.179	0.179	0.096	0.096	0.907	0.155	0.019
NNC1374	0.380	1.317	0.318	0.475	1.512	2.429	0.031
FFFFF800	0.041	0.058	0.029	0.042	0.781	0.675	0.013
PILOT	0.287	0.289	0.164	0.191	3.942	1.628	0.053
ORSIRR 2	0.283	0.933	0.130	0.300	1.166	3.382	0.020
JPWH 991	0.573	0.985	0.135	0.256	2.393	4.917	0.021
BCSSTK27	0.172	0.172	0.097	0.097	0.920	0.173	0.018
NNC1374	0.110	0.229	0.086	0.139	2.000	1.511	0.026

Table 19. Performance of runs of `MA47` code on the CRAY Y-MP.

	Total storage	Storage for factors		Time			
	Predicted	Predicted	Actual	Analyse	Factorize	Solve	One-off
BRITGAS	1.90	1.11	1.25	30.24	7.72	1.49	12.15
BIGBANK	1.49	1.31	1.29	4.52	1.85	1.63	3.30
MINPERM	1.01	1.12	1.12	1.85	1.32	3.02	1.66
SVANBERG	0.96	1.36	1.12	2.78	1.05	1.17	2.05
BRATU2D	1.36	1.15	0.47	82.83	0.23	0.55	2.33
BRATU3D		`MA27` ran out of space					
GRIDNETC	1.56	1.37	1.38	2.66	1.25	1.46	1.72
QPCSTAIR	2.52	1.57	1.64	3.49	3.08	1.45	3.12
KSIP	1.30	0.96	0.44	14.62	0.57	0.42	0.70
AUG3DQP	1.86	1.46	1.46	2.37	1.22	1.37	1.48
FFFFF800	1.80	1.63	1.01	2.74	1.40	1.06	1.87
PILOT	1.55	1.20	1.15	1.26	0.99	1.04	1.08
ORSIRR 2	1.61	1.15	0.42	4.11	0.25	0.51	0.37
JPWH 991	2.04	1.48	0.43	3.18	0.24	0.47	0.29
BCSSTK27	0.80	0.45	0.27	1.41	0.07	0.44	0.31
NNC1374	2.43	2.19	1.10	5.93	3.50	1.13	3.55
FFFFF800	0.41	0.40	0.39	1.88	1.13	0.61	1.31
PILOT	0.28	0.25	0.15	0.66	0.06	0.25	0.16
ORSIRR 2	1.04	0.61	0.50	3.36	2.32	0.71	2.34
JPWH 991	1.40	0.47	0.31	5.01	0.98	0.47	1.05
BCSSTK27	0.76	0.45	0.27	1.44	0.07	0.44	0.32
NNC1374	0.70	0.59	0.81	7.79	6.72	1.11	6.74
lower q.	0.96	0.59	0.42	1.88	0.25	0.47	0.70
median	1.45	1.15	0.91	3.27	1.17	1.05	1.69
upper q.	1.86	1.46	1.25	5.93	2.32	1.45	3.12

Table 20. Ratio of performance of `MA47` code to `MA27` code on the SUN SPARC-10.

	Total storage	Storage for factors			Time		
	Predicted	Predicted	Actual	Analyse	Factorize	Solve	One-off
BRITGAS	1.90	1.11	1.28	23.67	9.51	2.22	16.48
BIGBANK	1.49	1.31	1.29	3.11	1.71	2.00	2.72
MINPERM	1.01	1.12	1.12	58.53	1.43	3.25	30.38
SVANBERG	0.96	1.36	1.37	1.69	1.30	2.32	1.63
BRATU2D	1.36	1.15	0.47	72.24	0.51	1.97	16.79
BRATU3D		MA27 ran out of space					
GRIDNETC	1.56	1.37	1.38	2.72	1.49	2.30	2.18
QPCSTAIR	2.52	1.57	1.64	2.36	3.03	1.83	2.58
KSIP	1.30	0.96	0.44	14.87	0.30	1.20	0.89
AUG3DQP	1.86	1.46	1.46	1.93	1.62	2.18	1.81
FFFFF800	1.80	1.63	1.01	2.03	2.06	2.00	2.04
PILOT	1.55	1.20	1.15	0.87	1.35	1.86	0.95
ORSIRR 2	1.61	1.15	0.42	3.02	0.54	1.83	1.31
JPWH 991	2.04	1.48	0.43	2.31	0.54	1.44	0.88
BCSSTK27	0.80	0.45	0.27	1.01	0.22	1.46	0.67
NNC1374	2.43	2.19	1.10	4.07	1.91	1.72	2.39
FFFFF800	0.41	0.40	0.38	1.37	3.14	1.62	1.85
PILOT	0.28	0.25	0.14	0.38	0.39	1.13	0.38
ORSIRR 2	1.04	0.61	0.50	2.55	3.30	1.67	3.05
JPWH 991	1.40	0.47	0.31	4.27	2.37	1.40	2.76
BCSSTK27	0.76	0.45	0.27	1.03	0.25	1.50	0.69
NNC1374	0.70	0.59	0.81	5.39	5.90	2.00	5.53
lower q.	0.96	0.59	0.42	1.69	0.54	1.50	0.95
median	1.45	1.15	0.91	2.63	1.56	1.85	2.11
upper q.	1.86	1.46	1.29	5.39	3.03	2.18	3.05

Table 21. Ratio of performance of MA47 code to MA27 code on the CRAY Y-MP.

We show the full results in these tables as well as the medians and quartiles since the performance can vary very widely. On the SUN, the new code factorizes matrix PILOT over sixty times faster than MA27 but is nearly 7 times slower than MA27 on BRITGAS. With the exception of PILOT, the analyse phase times are always greater for the new code, once by over a factor of 80 (BRATU2D). The variation is only slightly less dramatic on the CRAY.

We have also run the codes on a set of ten Harwell-Boeing matrices with nonzero diagonal entries (BCSSTK14/15/16/17/18/26/27/28 and BCSSTM26/27) and the results are summarized in Table 22. On the SUN, MA47 just outperforms MA27 for the factorize and solve phases, but otherwise is inferior. We therefore recommend that where the matrix has nonzeros on the diagonal, MA27 should continue to be used. We plan to make a new version of MA27 that incorporates the BLAS and some other minor improvements. We anticipate that the revised MA27 will always outperform MA47 on this kind of matrix.

		Total storage		Storage for factors		Time		
		Predicted	Predicted	Actual	Analyse	Factorize	Solve	One-off
CRAY	lower q.	1.25	0.99	0.99	1.54	1.11	1.75	1.26
	median	1.34	1.03	1.02	1.56	1.15	1.93	1.29
	upper q.	1.38	1.07	1.04	1.80	1.21	2.00	1.30
SUN	lower q.	1.25	0.99	0.99	1.55	0.86	0.87	0.89
	median	1.34	1.03	1.02	1.70	0.94	0.94	0.98
	upper q.	1.38	1.07	1.04	1.98	1.02	0.98	1.06

Table 22. MA47 to MA27 ratios on 10 matrices with nonzero entries on the diagonal.

In these comparisons, we have used the same parameter settings (where applicable) for both codes. In particular, we have run MA27 with a value for the threshold, *u*, of 0.001 which is 100 times less than the default value for MA27. However, it would appear empirically that the stability of MA27 is more sensitive to the threshold value than MA47, so we have also compared MA47 with MA27 using its default threshold. A summary of the results in Table 23 indicates that, in general, MA47 with its default outperforms MA27 with its default on the SUN but the variation in relative performance over different problem classes is still substantial. However, the roles are reversed on the CRAY due to the greater penalty for the extra integer manipulation in MA47.

		Total storage	Storage for factors		Time			
		Predicted	Predicted	Actual	Analyse	Factorize	Solve	One-off
CRAY	lower q.	0.96	0.59	0.31	1.69	0.53	1.44	0.93
	median	1.45	1.15	0.57	2.64	1.38	1.82	1.80
	upper q.	1.86	1.46	1.21	5.41	2.04	2.18	3.13
SUN	lower q.	0.96	0.59	0.31	1.88	0.25	0.45	0.33
	median	1.45	1.15	0.57	3.32	0.96	0.75	1.38
	upper q.	1.86	1.46	1.12	5.58	1.88	1.40	2.53

Table 23. MA47 to MA27 ratios using default threshold value for MA27.

In summary, these results indicate that it is “horses for courses”. MA47 does well on our augmented systems when the (1,2) block is nearly square or when several of the diagonals of the (1,1) block are zero. However, the efficiency of MA47 is very dependent on the details of the assembly tree structure so it is often difficult to judge the relative performance in advance. We find this fragility to be the most disturbing aspect of the present code and hope that further work will improve the performance of MA47 on the “bad”cases.

7 References

Anon (1993). Harwell Subroutine Library Catalogue (Release 11). Theoretical Studies Department, AEA Technology, Harwell.

Bongartz, I., Conn, A. R., Gould, N. I. M., and Toint, Ph. L. (1993). CUTE: Constrained and Unconstrained Testing Environment. Technical Report TR/PA/93/10, CERFACS, Toulouse, France. ACM Trans Math Softw (to appear).

Bunch, J. R. and Parlett, B. N. (1971). Direct methods for solving symmetric indefinite systems of linear equations. *SIAM J. Numer. Anal.* **8**, 639-655.

Dongarra, J. J., Du Croz, J., Duff, I. S., and Hammarling, S. (1990). A set of level 3 basic linear algebra subroutines. *ACM Trans. Math. Softw.* **16**, 1-17.

Dongarra, J. J., Du Croz, J., Hammarling, S., and Hanson, R. J. (1988). An extended set of Fortran Basic Linear Algebra Subprograms. *ACM Trans. Math. Softw.* **14**, 1-17 and 18-32.

Duff, I. S. and Reid, J. K. (1982). MA27 – A set of Fortran subroutines for solving sparse symmetric sets of linear equations. Report AERE R10533, HMSO, London.

Duff, I. S. and Reid, J. K. (1983). The multifrontal solution of indefinite sparse symmetric linear systems. *ACM Trans. Math. Softw.* **9**, 302-325.

Duff, I. S., Erisman, A. M., and Reid, J. K. (1986). Direct methods for sparse matrices. Oxford University Press, London.

Duff, I. S., Gould, N. I. M., Reid, J. K., Scott, J. A. and Turner, K. (1991). The factorization of sparse symmetric indefinite matrices. *IMA J. Numer. Anal.* **11**, 181-204.

Duff, I. S., Grimes, R. G., and Lewis, J. G. (1992). Users' Guide for the Harwell-Boeing Sparse Matrix Collection. Report RAL 92-086, Rutherford Appleton Laboratory, Oxfordshire.

Gay, D. M. (1985). Electronic mail distribution of linear programming test problems. *Mathematical Programming Society COAL Newsletter.*

Lawson, C. L., Hanson, R. J., Kincaid, D. R., and Krogh, F. T. (1979). Basic linear algebra subprograms for Fortran use. *ACM Trans. Math. Softw.* **5**, 308-325.

Liu, J. W. H. (1990). The role of elimination trees in sparse factorization. *SIAM J. Matrix Anal. Appl.* **11**, 134-172.

Markowitz, H. M. (1957). The elimination form of the inverse and its application to linear programming. *Management Sci.* **3**, 255-269.

1 SUMMARY

To **solve a sparse symmetric indefinite system of linear equations.** Given a sparse symmetric matrix $\mathbf{A}=\{a_{ij}\}_{n\times n}$ and an *n*-vector **b**, this subroutine solves the system $\mathbf{Ax}=\mathbf{b}$.

The method used is a direct method using multifrontal sparse Gaussian elimination and is discussed by Duff and Reid (1995), *MA47, A Fortran code for direct solution of indefinite sparse symmetric linear systems,* Report RAL-95-001, Rutherford Appleton Laboratory, Oxfordshire.

ATTRIBUTES — **Versions:** MA47A, MA47AD. **Calls:** _GEMM, _TPSV, _GEMV, _TRSV, _TPMV, I_AMAX. **Origin:** I.S. Duff and J.K. Reid, Rutherford Appleton Laboratory. **Date:** May 1993. **Conditions on external use:** (i), (ii), (iii) and (iv).

2 HOW TO USE THE ROUTINE

Although there are both single and double precision versions of the routine available, the user is **strongly advised** to use the double precision version unless single precision on his or her machine actually means 8-byte arithmetic.

2.1 Argument lists and calling sequences

There are four subroutines that can be called by the user:

(a) MA47I/ID sets default values for the components of the arrays that hold control parameters for the other subroutines.

(b) MA47A/AD accepts the pattern of **A** and chooses pivots for Gaussian elimination to preserve sparsity. It also constructs information for actual factorization by MA47B/BD. The user may provide a pivot sequence, in which case the necessary information for MA47B/BD will be generated.

(c) MA47B/BD factorizes a matrix **A** using the information from a previous call to MA47A/AD. The actual pivot sequence used may differ from that of MA47A/AD.

(d) MA47C/CD uses the factors generated by MA47B/BD to solve a system of equations $\mathbf{Ax}=\mathbf{b}$.

Normally the user would call MA47I/ID prior to any other call of the package. If non-default values for any of the control parameters are required, they should be set immediately after the call to MA47I/ID. A call to MA47C/CD must be preceded by a call to MA47B/BD which in turn must be preceded by a call to MA47A/AD. Since the information passed from one subroutine to the next is not corrupted by the second, several calls to MA47B/BD for matrices with the same sparsity pattern but different values may follow a single call to MA47A/AD, and similarly MA47C/CD can be used repeatedly to solve for different right-hand sides **b**.

2.1.1 To set default values of controlling parameters

The single precision version

```
        CALL MA47I(CNTL,ICNTL)
```

The double precision version

```
        CALL MA47ID(CNTL,ICNTL)
```

CNTL is a REAL (DOUBLE PRECISION in the D version) array of length 2 that need not be set by the user. On return it contains default values. For further information see Section 2.2.

ICNTL is an INTEGER array of length 7 that need not be set by the user. On return it contains default values. For further information see Section 2.2.

2.1.2 To perform symbolic manipulations

The single precision version

```
CALL MA47A(N,NE,IRN,JCN,IW,LIW,KEEP,ICNTL,RINFO,INFO)
```

The double precision version

```
CALL MA47AD(N,NE,IRN,JCN,IW,LIW,KEEP,ICNTL,RINFO,INFO)
```

N is an INTEGER variable that must be set by the user to the order n of the matrix **A**. It is not altered by the subroutine. **Restriction:** $1 \le$ N $\le$ *HUGE*/3, where *HUGE* is the largest possible integer value, equivalent to that defined by Fortran 90 as HUGE(N).

NE is an INTEGER variable that must be set by the user to the number of entries in the matrix **A**. It is not altered by the subroutine. **Restriction:** NE ≥ 1.

IRN and JCN are INTEGER arrays of length NE. The user must set them so that each diagonal entry a_{ii} is represented by IRN(k)=i and JCN(k)=i and each pair of off-diagonal entries a_{ij} and a_{ji} is represented by IRN(k)=i and JCN(k)=j or by IRN(k)=j and JCN(k)=i. Entries (on or off the diagonal) that are known to be zero should be excluded. Multiple entries are permitted. If IRN(k) or JCN(k) are less than 1 or greater than N, IRN(k) and JCN(k) are reset to zero and the entry is ignored. For some off-diagonal entries, the values of JCN(k) and IRN(k) are interchanged by the subroutine. These arrays are not otherwise altered by MA47A/AD. JCN must be preserved between a call to MA47A/AD and subsequent calls to MA47B/BD.

IW is an INTEGER array of length LIW. This is used as workspace by the subroutine. Its length must be at least NE*2+5*N+4 (or NE+5*N+4 if the pivot order is specified in KEEP), but we recommend that it should be at least 20% greater than this. INFO(12) (see Section 2.2) provides a means of checking whether the length is adequate.

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

KEEP is an INTEGER array of length NE+5*N+2. If the user wishes to provide the pivot sequence, the index of the variable in position i of the pivot order must be placed in KEEP(i), i = 1, 2,..., N and ICNTL(4) must be set to 1; the variables of any 2×2 pivot required must be adjacent in the sequence and their indices must be negated within KEEP. The given order is likely to be replaced by one that is equivalent apart from reordering of additions and subtractions. Otherwise, KEEP need not be set by the user. It must be preserved between a call to MA47A/AD and subsequent calls to MA47B/BD.

ICNTL is an INTEGER array of length 7 that contains control parameters and must be set by the user. Default values for the components may be set by a call to MA47I/ID. Details of the control parameters are given in Section 2.2. This array is not altered by the subroutine.

RINFO is a REAL (DOUBLE PRECISION in the D version) array of length 4 that need not be set by the user. On exit from MA47A/AD, RINFO(1) will be set to the number of floating-point operations that would always be required by a subsequent factorization that exactly respects the pivot sequence chosen by MA47A/AD and RINFO(2) will be set to the number of redundant floating-point operations required because of the use of the Level 3 BLAS routine GEMM. RINFO(3) and RINFO(4) are not accessed by MA47A/AD.

INFO is an INTEGER array of length 24 that need not be set by the user. On return from MA47A/AD, a value of zero for INFO(1) indicates that the subroutine has performed successfully. For nonzero values, see Section 2.3. For the meaning of the value of other components of INFO set by MA47A/AD, see Section 2.2.

2.1.3 To factorize a matrix

The single precision version

```
CALL MA47B(N,NE,JCN,A,LA,IW,LIW,KEEP,CNTL,ICNTL,IW1,RINFO,INFO)
```

The double precision version

```
CALL MA47BD(N,NE,JCN,A,LA,IW,LIW,KEEP,CNTL,ICNTL,IW1,RINFO,INFO)
```

N is an INTEGER variable that must be set by the user to the order n of the matrix **A**. It must be unchanged since the call to MA47A/AD and is not altered by the subroutine.

NE is an INTEGER variable that must be set by the user to the number of entries in the matrix **A**. It must be unchanged since the call to MA47A/AD and is not altered by the subroutine.

JCN is an INTEGER array of length NE that must be unchanged since the call to MA47A/AD and is not altered by the subroutine.

A is a REAL (DOUBLE PRECISION in the D version) array of length LA that must be set by the user so that A(k) holds the value of the diagonal entry or pair of off-diagonal entries whose indices were held in IRN(k) and JCN(k) on entry to MA47A/AD, for k = 1, 2,..., NE. Multiple entries are summed and any that correspond to an IRN(k) or JCN(k) value that was out of range are ignored. On return, A will hold the entries of the factors of the matrix **A**. A must be preserved between calls to this subroutine and subsequent calls to MA47C/CD.

LA is an INTEGER variable that must be set by the user to the length of array A. It must be at least as great as INFO(6) as output by MA47A/AD (see Section 2.2). It is advisable to allow a greater value because the use of numerical pivoting may increase storage requirements. It is not altered by the subroutine.

IW is an INTEGER array of length LIW that need not be set by the user. It is used as workspace by MA47B/BD and on return holds integer indexing information on the matrix factors. It must be preserved by the user between calls to this subroutine and subsequent calls to MA47C/CD.

LIW is an INTEGER variable that must be set by the user to the length of array IW. It must be at least as great as INFO(7) as output from MA47A/AD (see Section 2.2). A greater value is recommended because numerical pivoting may increase storage requirements. It is not altered by the subroutine.

KEEP is an INTEGER array of length NE+5*N+2 that must be unchanged since the call to MA47A/AD. It is not altered by the subroutine.

CNTL is a REAL (DOUBLE PRECISION in the D version) array of length 2 that contains control parameters and must be set by the user. Default values for the components may be set by a call to MA47I/ID. Details of the control parameters are given in Section 2.2. This array is not altered by the subroutine.

ICNTL is an INTEGER array of length 7 that contains control parameters and must be set by the user. Default values for the components may be set by a call to MA47I/ID. Details of the control parameters are given in Section 2.2. This array is not altered by the subroutine.

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

RINFO is a REAL (DOUBLE PRECISION in the D version) array of length 4 that need not be set by the user. On exit from MA47B/BD, RINFO(3) will be set to the number of floating-point operations that would always be required for a factorization with the same pivot sequence and RINFO(4) will be set to the number of redundant floating-point operations performed because of the use of the Level 3 BLAS routine GEMM. RINFO(1) and RINFO(2) are not accessed by MA47B/BD.

INFO is an INTEGER array of length 24 that need not be set by the user. On return from MA47A/AD, a value of zero for INFO(1) indicates that the subroutine has performed successfully. For nonzero values of INFO(1), see Section 2.3. For the meaning of the value of other components of INFO set by MA47B/BD see Section 2.2.

2.1.4 To solve equations, given the factorization

The single precision version

```
CALL MA47C(N,A,LA,IW,LIW,W,RHS,IW1,ICNTL)
```

The double precision version

```
CALL MA47CD(N,A,LA,IW,LIW,W,RHS,IW1,ICNTL)
```

N is an INTEGER variable that must be set by the user to the order *n* of the matrix **A**. It must be unchanged since the call to MA47B/BD and is not altered by the subroutine.

A is a REAL (DOUBLE PRECISION in the D version) array of length LA that must be unchanged since the call to MA47B/BD. It is not altered by the subroutine.

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

IW is an INTEGER array of length LIW that must be unchanged since the call to MA47B/BD. It is not altered by the subroutine.

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

W is a REAL (DOUBLE PRECISION in the D version) array of length N that is used as workspace.

RHS is a REAL (DOUBLE PRECISION in the D version) array of length N that must be set by the user so that RHS(i) holds the *i*-th component of the right-hand side of the equations being solved. On return, it will hold the corresponding entry of the solution vector.

IW1 is an INTEGER array of length N that is used as workspace.

ICNTL is an INTEGER array of length 7 that contains control parameters and must be set by the user. Default values for the components may be set by a call to MA47I/ID. Details of the control parameters are given in Section 2.2. This array is not altered by the subroutine.

2.2 Arrays for control and information

The elements of the arrays CNTL and ICNTL control the action of MA47A/AD, MA47B/BD, and MA47C/CD. Default values for the elements are set by MA47I/ID. The elements of the arrays RINFO and INFO provide information on the action of MA47A/AD, MA47B/BD, and MA47C/CD.

CNTL(1) has default value 0.001 and is used for threshold pivoting by MA47B/BD. Values greater than 0.5 are treated as 0.5 and less than zero as zero. Since this parameter is used on the assumption that the matrix is well-scaled, it is advisable that the user scale the matrix by calling MC30 before calling MA47B/BD (see Example 5.2).

CNTL(2) has default value zero. If it is set to a positive value, MA47B/BD will treat any pivot whose modulus is less than CNTL(2) as zero.

ICNTL(1) has default value 6 and holds the unit number to which the error messages are sent. A non-positive value suppresses all messages.

ICNTL(2) has default value 6 and holds the unit number to which warning messages and additional printing is sent. A non-positive value suppresses all such printing.

ICNTL(3) is used by the subroutines to control printing. It has default value 1. Possible values are:

0 No printing.

1 Error messages only.

2 Error and warning messages only.

3 Scalar parameters and a few entries of arrays on entry and exit from subroutines.

4 All parameter values printed on entry and exit from subroutines.

ICNTL(4) has default value 0. It must be set by the user to a value of 1 when calling MA47A/AD if a pivot sequence is being supplied by the user in array KEEP. If the value is 0, the pivot order is chosen automatically by the Markowitz strategy. If greater than 1, the pivot order is chosen automatically, but each search for a structured pivot limited to this number of rows.

ICNTL(5) has default value 5. It is used by MA47A/AD and MA47B/BD as the block size for the use of Level 3 BLAS (see Section 4). A value greater than N will mean that each frontal matrix is treated as a single block.

ICNTL(6) has default value 5. MA47A/AD may amalgamate tree nodes (see Section 4) in order to decrease the amount of indirect addressing, but at the cost of more arithmetic. Two nodes are merged at the risk of more arithmetic only if both involve less than ICNTL(6) eliminations.

ICNTL(7) has default value 4. Direct addressing and Level 2 BLAS are used by the routine MA47C/CD on any block of the factorization having more than ICNTL(7) rows.

RINFO(1:4) are used to record the number of floating-point operations performed (see Sections 2.1.2 and 2.1.3).

INFO(1) has the value zero if the call was successful, a positive value in the case of a warning, and a negative value in the event of an error (see Section 2.3).

INFO(2) gives additional information on some returns:

MA47A/AD or MA47B/BD with INFO(1) = –3: a length for IW sufficient to continue after the point of failure.

MA47B/BD with INFO(1) = –4: a length for A sufficient to continue after the point of failure.

MA47A/AD with INFO(1) = –5 or –6: the index in KEEP of the faulty component.

INFO(3) gives, on return from MA47A/AD, the number of entries with indices that are out of range.

INFO(4) gives, on return from MA47A/AD, the number of duplicate entries.

INFO(5) gives, on return from MA47A/AD, the number of nodes in the assembly tree.

INFO(6) and INFO(7) give, on return from MA47A/AD and MA47B/BD, the lengths of A and IW required for a successful completion of MA47B/BD with the same pivot sequence. The actual lengths required may be greater because of numerical pivoting.

INFO(8) gives, on return from MA47A/AD, the number of zero eigenvalues detected in the structure of **A**.

INFO(9) gives, on return from MA47A/AD, the maximum front size encountered.

INFO(10) and INFO(11) give, on return from MA47A/AD, the number of REAL (or DOUBLE PRECISION for the D version) and INTEGER words required to hold the matrix factors provided the same pivot sequence is employed. Numerical pivoting may change this.

INFO(12) gives, on return from MA47A/AD, the number of compresses of the internal data structure performed by MA47A/AD. If this is high (say more than 10), the performance of MA47A/AD may be improved by increasing the length of array IW.

INFO(13) and INFO(14) give, on return from MA47A/AD, the number of tile and oxo pivots chosen (see Section 4).

INFO(15) is set by MA47B/D to the maximum front size encountered.

INFO(16) and INFO(17) give, on return from MA47B/BD, the amount of REAL (or DOUBLE PRECISION for the D version) and INTEGER words actually used to hold the factorization.

INFO(18) gives, on return from MA47B/BD, the number of compresses performed on the real data. If it is high (say more than 10), the speed of the factorization may be increased by allocating more space to the array A.

INFO(19) gives, on return from MA47B/BD, the number of compresses performed on the integer data. If it is high (say more than 10), the speed of the factorization may be increased by allocating more space to the array IW.

INFO(20), INFO(21), and INFO(22) give, on return from MA47B/BD, the number of tile, oxo, and full 2×2 pivots chosen (see Section 4).

INFO(23) gives, on return from MA47B/BD, the number of negative eigenvalues of **A**.

INFO(24) gives, on return from MA47B/BD, the number of zero eigenvalues detected in **A**.

2.3 Error diagnostics

A successful return from MA47A/AD or MA47B/BD is indicated by a value of INFO(1) equal to zero. Possible nonzero values for INFO(1) are given below. There are no error returns from MA47C/CD.

A negative flag value is associated with an error message that will be output on unit ICNTL(1).

–1 N < 1 or N > *HUGE*/3 where *HUGE* is the largest possible integer value (MA47A/AD and MA47B/BD).

–2 NE < 1 (MA47A/AD and MA47B/BD).

–3 Failure due to insufficient space allocated to array IW. INFO(2) is set to a value needed to progress beyond the current point of failure (MA47A/AD and MA47B/BD).

–4 Failure due to insufficient space allocated to array A. INFO(2) is set to a value needed to progress beyond the current point of failure (MA47B/BD only).

–5 The indices supplied in KEEP(i), i = 1, 2,..., N when ICNTL(4) = 1 do not constitute a permutation; if KEEP(j) was found to be faulty, j is returned in INFO(2) (MA47A/AD only).

–6 A negative index supplied in KEEP(i), i = 1, 2,..., N when ICNTL(4) = 1 is not immediately followed by another negative index; if KEEP(j) was found to be faulty, j is returned in INFO(2) (MA47A/AD only).

A positive flag value is associated with a warning message that will be output on unit ICNTL(2).

+1 Index (in IRN or JCN) out of range. Action taken by subroutine is to set their value to zero and ignore them and continue. INFO(3) is set to the number of faulty entries. Details of the first ten are printed on unit ICNTL(2) (MA47A/AD only).

+2 There are duplicate entries. These will be summed by MA47B/BD. This number is recorded in INFO(4) (MA47A/AD only).

+3 Both warnings +1 and +2 are operative.

+4 The matrix is rank deficient. The number of zero eigenvalues found is given by INFO (8) (MA47A/AD) or INFO(24) (MA47B/BD).

+5 Both warnings +1 and +4 are operative.

+6 Both warnings +2 and +4 are operative.

+7 Warnings +1, +2, and +4 are operative.

2.4 Badly-scaled systems

If the user's input matrix has entries differing widely in magnitude, then an inaccurate solution may be obtained. In such cases, the user is advised to first use MC30A/AD to obtain scaling factors for the matrix and then explicitly scale it prior to calling MA47A/AD. Thereafter, both left and right-hand sides should be scaled as indicated in the code following the example in Section 5.2.

3 GENERAL INFORMATION

Use of common: None.

Other routines called directly: MA47F/FD, MA47G/GD, MA47H/HD, MA47J/JD, MA47K/KD, MA47L/LD, MA47M/MD, MA47N/ND, MA47O/OD, MA47P/PD, MA47Q/QD, MA47R/RD, MA47S/SD, MA47T/TD, MA47U/UD, MA47V/VD, MA47W/WD, MA47X/XD MA47Y/YD, MA47Z/ZD. The package uses the Basic Linear Algebra Subprograms SGEMM/DGEMM, STPSV/DTPSV, SGEMV/DGEMV, STRSV/DTRSV, STPMV/DTPMV, and ISAMAX/IDAMAX.

Input/output: Error messages on unit ICNTL(1). Warning messages and additional printing on unit ICNTL(2). Each has default value 6, and printing is suppressed if the value is non-positive.

Restrictions:

N ≥ 1, N ≤ *HUGE*/3.
NE ≥ 1.

4 METHOD

A version of sparse Gaussian elimination is used. It is implemented using a multifrontal method.

MA47A/AD chooses diagonal pivots of orders 1 and 2 using the Markowitz criterion or accepts a sequence of such pivots supplied by the user. Because of the facility for handling matrices with zeros on the diagonal, the 2×2 pivots can be of the form

$$\begin{pmatrix} 0 & \times \\ \times & \times \end{pmatrix} \quad \text{or} \quad \begin{pmatrix} 0 & \times \\ \times & 0 \end{pmatrix}$$

called tile and oxo pivots respectively. MA47A/AD employs a generalized element model of the elimination, thus avoiding the need to store the filled-in pattern explicitly. The elimination is represented as an assembly and elimination tree with the order of elimination determined by a depth-first search of the tree. Opportunities are sought for grouping the variables into blocks where this would not affect the sparsity. This corresponds to merging nodes of the tree. Merges that do affect the sparsity are also performed under the control of ICNTL(6). For most vector and superscalar machines, the best balance is achieved with ICNTL(6) set to its default value of 8.

MA47B/BD factorizes the matrix by using the assembly and elimination ordering generated by MA47A/AD, but with additional numerical pivoting. The numerical pivoting can yield full 2×2 pivots in addition to the tile and oxo pivots above. An option exists for MA47B/BD to use the (full) matrix-matrix multiplication routine SGEMM/DGEMM from the Level 3 BLAS. Because we maintain symmetry in the factorization this can result in more operations than not using this kernel (given by RINFO(2) and RINFO(4)) but may still perform better on vector or parallel machines. The size of the blocks used by SGEMM/DGEMM is given by ICNTL(5).

MA47C/CD uses the factors from MA47B/BD to solve systems of equations either by loading the appropriate parts of the vectors into an array of the current front size and using full matrix code or by indirect addressing at each stage. This is determined by control parameter ICNTL(7) which has default value set so that any block pivot with more than 4 rows uses direct addressing.

The pivotal strategy is explained by Duff, Gould, Reid, Scott, and Turner, *Factorization of sparse symmetric indefinite matrices* (IMA J. Numer. Anal. **11**, 181-204, 1991). An explanation of the whole algorithm is given by Duff and Reid (1995), *MA47, A Fortran code for direct solution of indefinite sparse symmetric linear systems,* Report RAL-95-001, Rutherford Appleton Laboratory, Oxfordshire.

5 EXAMPLE OF USE

5.1 Solving sparse equations without scaling.

We illustrate the use of the package on the solution of the single set of equations given by:

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

Note that this example does not illustrate all the facilities.

Program

```
C Simple example of use of MA47 package
      INTEGER LA, LIW, NEMAX, NMAX
      PARAMETER (LA=50, LIW=50, NEMAX=10, NMAX=5)
      INTEGER IRN(NEMAX),JCN(NEMAX),IW(LIW),KEEP(NEMAX+5*NMAX+2),
     *        IW1(2*NMAX+2)
      INTEGER I,ICNTL(7),INFO(24),N,NE
      DOUBLE PRECISION A(LA),W(NMAX),RHS(NMAX),CNTL(2),RINFO(3)

C Set default parameters
      CALL MA47ID(CNTL,ICNTL)

C Read matrix and right-hand side
      READ (5,*) N,NE
      IF (N.GT.NMAX .OR. NE.GT.NEMAX) THEN
        WRITE(6,'(A,2I10)') ' Immediate return N or NE too large = ',
     *                   N, NE
        STOP
      ENDIF
      READ (5,*) (IRN(I),JCN(I),A(I),I=1,NE)
      READ (5,*) (RHS(I),I=1,N)

C Analyse sparsity pattern
      CALL MA47AD(N,NE,IRN,JCN,IW,LIW,KEEP,ICNTL,RINFO,INFO)

C Factorize matrix
      CALL MA47BD(N,NE,JCN,A,LA,IW,LIW,KEEP,CNTL,ICNTL,IW1,
     *            RINFO,INFO)

C Solve the equations
      CALL MA47CD(N,A,LA,IW,LIW,W,RHS,IW1,ICNTL)

C     Print out solution vector.
      WRITE(6,'(/A/(1P,5D13.5))') ' The solution vector is:',
     *                       (RHS(I),I=1,N)
      END
```

Data

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

Output

```
 The solution vector is:
  1.00000D+00  2.00000D+00  3.00000D+00  4.00000D+00  5.00000D+00
```

5.2 Solving sparse equations using scaling.

In the example code shown below we scale the matrix and right-hand vector (see Section 2.4) prior to solution of the linear equations.

```
C Example of use of scaling with MA47
      INTEGER  LA, LIW, NEMAX, NMAX
      PARAMETER (LA=200, LIW=200, NMAX=20, NEMAX=100)
      DOUBLE PRECISION CNTL(2),RINFO(3),A(LA),RHS(NMAX),W(4*NMAX)
      INTEGER  I,ICNTL(7),II,INFO(24),IRN(NEMAX),IW(LIW),J,JCN(NEMAX),
     *         KEEP(NEMAX+5*NMAX+2),N,NE,IW1(2*NMAX+2),LP,IFAIL
      DOUBLE PRECISION S(NMAX)

C     Read in input matrix.
      READ(5, * ) N,NE
      IF (N.GT.NMAX .OR. NE.GT.NEMAX) THEN
         WRITE(6,'(A)') ' Error in input data for N and/or NE'
         STOP
      END IF
      READ(5, * ) (IRN(I), JCN(I), A(I), I=1,NE)

C     Scale input matrix
      LP = 6
      CALL MC30AD(N,NE,A,IRN,JCN,S,W,LP,IFAIL)
      IF (IFAIL.LT.0) THEN
        WRITE(6,'(A)') ' Failure in scaling routine'
        STOP
      ENDIF
      DO 340 I=1,N
        S(I)=EXP(S(I))
 340  CONTINUE
      DO 350 II=1,NE
        I=IRN(II)
        J=JCN(II)
        A(II)=A(II)*S(I)*S(J)
 350  CONTINUE

C     Set default controls
      CALL MA47ID(CNTL,ICNTL)

C Analyse sparsity pattern
      CALL MA47AD(N,NE,IRN,JCN,IW,LIW,KEEP,ICNTL,RINFO,INFO)
C
C Factorize matrix
      CALL MA47BD(N,NE,JCN,A,LA,IW,LIW,KEEP,CNTL,ICNTL,IW1,
     *            RINFO,INFO)

C     Read in right hand side.
      READ(5, * ) (RHS(I), I=1,N)

C     Scale the right hand side vector by row weights
      DO 425 I=1,N
        RHS(I)=RHS(I)*S(I)
 425  CONTINUE

C Solve the equations
      CALL MA47CD(N,A,LA,IW,LIW,W,RHS,IW1,ICNTL)

C     Scale the right-hand side vector by column weights
      DO 475 I=1,N
        RHS(I)=RHS(I)*S(I)
 475  CONTINUE

C     Print out solution vector.
      WRITE(6,'(/A/(1P,5D13.5))') ' The solution vector is:',
     *                       (RHS(I),I=1,N)
      END
```

Thus if, in this example we wish to solve:

$$\begin{pmatrix} 3.14E5 & 7.5E1 & \\ 7.5E1 & 3.2E{-}3 & 0.3 \\ & 0.3 & 4.1E2 \end{pmatrix} \mathbf{x} = \begin{pmatrix} 3.1415E5 \\ 7.59064E1 \\ 1.2306E3 \end{pmatrix}$$

we have as input

```
      3     5
      1     1       3.14000D+05
      2     3       0.30
      3     3       4.10000D+02
      1     2       7.50000D+01
      2     2       3.20000D-03
   0.31415D+06  0.759064E2   0.12306E4
```

and output would be

```
  The solution vector is:
   1.00000D+00  2.00000D+00  3.00000D+00
```