RAL-TR-95-039

The design of MA48, a code for the direct solution of sparse unsymmetric linear systems of equations

by

I. S. Duff and J. K. Reid

Abstract

We describe the design of a new code for the direct solution of sparse unsymmetric linear systems of equations. The new code utilizes a novel restructuring of the symbolic and numerical phases, which increases speed and saves storage without sacrifice of numerical stability. Other features include switching to full matrix processing in all phases of the computation enabling the use of all three levels of BLAS, treatment of rectangular or rank-deficient matrices, partial factorization, and integrated facilities for iterative refinement and error estimation.

Categories and subject descriptors: G.1.3 [**Numerical Linear Algebra**]: Linear systems (direct methods), Sparse and very large systems.

General Terms: Algorithms, performance.

Additional Key Words and Phrases: sparse unsymmetric matrices, Gaussian elimination, block triangular form, error estimation, BLAS.

Computer and Information Systems Department,
Rutherford Appleton Laboratory,
Oxon OX11 0QX.

August 1995.

CONTENTS

1 Introduction 1

2 Overview 2

3 MA50 – routines for the numerical processing of a single block 4
3.1 MA50A: analyse 5
3.1.1 Treating a square matrix with the default pivotal strategy. 5
3.1.2 Markowitz pivoting 6
3.1.3 Drop tolerances 7
3.1.4 Singular and rectangular matrices 7
3.2 MA50B: factorize 8
3.2.1 First factorization 8
3.2.2 Drop tolerances 10
3.2.3 Subsequent factorizations 11
3.2.4 Singular and rectangular matrices 11
3.3 MA50C: solve 12
3.4 Solving full sets of linear equations 12

4 The MA48 Package 14
4.1 MA48A: analyse 14
4.2 MA48B: factorization 16
4.3 MA48C: solve 16
4.3.1 Solution without iterative refinement 16
4.3.2 Solution with iterative refinement 17

5 Performance results 18
5.1 Density threshold for the switch to full code 22
5.2 The use of the BLAS 23
5.3 The strategy for pivot selection 25
5.4 The block triangular form 25
5.5 Comparison with calling MA50 directly 28
5.6 Iterative refinement and error estimation 28
5.7 Comparison with MA28 29

6 Acknowledgements 31

Appendix. Solving full sets of linear equations 31
A.1 Factorization using BLAS at Level 1, 2, or 3 33
A.2 Solution using BLAS at Level 1 or 2 33

References 34

1 Introduction

This paper describes the design of MA48, a collection of Fortran 77 subroutines for the direct solution of a sparse unsymmetric set of linear equations

$$\mathbf{A}\mathbf{x} = \mathbf{b}, \tag{1.1}$$

where $\mathbf{A}$ is usually square and nonsingular. The main features of this new software package are:

(i) the algorithm is fast and robust,
(ii) the input format is user-friendly (entries in any order in a real array and corresponding row and column indices in two parallel integer arrays, with duplicates allowed and summed),
(iii) the code switches to full-matrix processing when the reduced matrix is sufficiently dense, using Basic Linear Algebra Subprograms (BLAS) at Levels 1, 2, and 3,
(iv) the pivot sequence is normally chosen automatically from anywhere in the matrix, but the choice may be limited to the diagonal or the pivot sequence may be specified,
(v) in the event of insufficient storage allocation by the user, the package continues with the computation to obtain a good estimate of the amount required,
(vi) the code computes the block triangular form and makes use of it,
(vii) entries smaller than a threshold are dropped from the factorization,
(viii) singular or rectangular matrices are permitted,
(ix) another matrix of the same pattern may be factorized with or without additional row interchanges for stability,
(x) there is an option of specifying that some columns have not changed when factorizing another matrix,
(xi) another problem with the same matrix or its transpose may be solved,
(xii) iterative refinement of the solution is available to improve its accuracy or provide an error estimate.

An overview of the design considerations is provided in Section 2. The heart of MA48 lies in a separate package called MA50, which treats a single block from the diagonal of the block triangular form and assumes that the entries have been sorted by columns and that there are no duplicates. It may be called directly by a user that accepts these restrictions. It does not have facilities for iterative refinement and has a less convenient means for specification of columns that have not changed since factorizing another matrix. We consider this in Section 3 and the remainder in Section 4. To distinguish notation for the two cases, we use Helvetica for variables associated with MA48. Section 5 is devoted to our experience of the actual running of the codes. It is our intention that this code should supersede the code MA28 (Duff 1977, Duff and Reid 1979) and we compare our new code with MA28 in this section.

The MA48 code is available from AEA Technology, Harwell; the contact is John Harding, Harwell Subroutine Library, B 552, AEA Technology, Harwell, Didcot, Oxon OX11 0RA, tel (44) 1235 434573, fax (44) 1235 434340, email john.harding@aeat.co.uk, who will provide details of price and conditions of use. A version also exists for complex matrices. For a more detailed description, including specification sheets for the software, we refer the reader to a separate report (Duff and Reid 1993).

2 Overview

It is common practice in designing code for the solution of sparse equations is to divide the computation into phases. A common division in the case of unsymmetric matrices, as used for example in the MA28 code, is to use the three phases:

Analyse-factorize finds the block triangular form, chooses pivots for good sparsity preservation, and computes the factorization.

Factorize factorizes another matrix with exactly the same sparsity pattern using exactly the same pivots.

Solve uses the factorization to solve an equation.

In applications, there is often a need for many factorizes for each analyse-factorize and many solves for each factorize. Since it is normal for factorize to be substantially faster than analyse-factorize and solve to be substantially faster than solve, this subdivision fits well in such an environment.

The most exciting algorithmic development in recent years is the work of Gilbert and Peierls (1988) for economically generating the patterns of the columns of the factors when factorizing with a given column sequence but allowing for row interchanges. They use a depth-first search of the directed graph of the previous row operations to generate the pattern of the current column. The overall complexity is $O(n)+O(f)$, where n is the number of columns and f is the number of floating-point operations. There are overheads associated with the recomputation of the sparsity patterns of the columns and the row interchanges may cause extra fill-ins, so in MA48 we provide two 'factorizes' which we call ***first*** and ***fast.*** The first factorize must be provided with a column sequence; we have chosen also to require a row sequence to which we adhere unless numerical considerations dictate otherwise, on the assumption that the specified sequence is good for sparsity. Thus, the analyse phase need only provide a recommended pivot sequence; there is no need for this phase to provide the sparsity pattern of the factorized matrix. We have therefore designed the analyse phase to provide the permutations without the actual factors. This saves storage since working storage is then needed only for the active submatrix of the block on the diagonal of the block triangular form that is currently being processed. It may save time since the vectors that hold the active columns are shorter and data compressions are much less likely to be needed. Thus the phases of MA48 are:

Analyse finds the block triangular form and chooses pivots for good sparsity preservation.

First-factorize factorizes a matrix with exactly the same sparsity pattern using a given column sequence and with row interchanges guided by a recommended row sequence.

Fast-factorize factorizes another matrix with exactly the same sparsity pattern using exactly the same pivots.

Solve uses the factorization to solve an equation.

A benefit of this subdivision over the more conventional one given earlier is that if changes to the matrix elements mean that the original pivot sequence chosen by the relatively expensive analyse call is not numerically suitable, we do not need to resort to another call to

analyse but all that is usually needed is the first-factorize call, which is only slightly more expensive than the fast-factorize call (see Tables 2-4). On the other hand, for the first matrix to be factorized with MA48, we need to follow an analyse call with a first-factorize call, whereas in the conventional approach a single analyse-factorize call suffices. This may appear to be a disadvantage, but our experience is that the two MA48 calls are almost always significantly faster than an analyse-factorize call to MA28 (see Section 5.7). There may be an advantage in providing an analyse-factorize phase for the case where only one matrix of a given pattern is to be factorized, and we plan to provide this later.

MA48 seeks to permute a square matrix to the block upper triangular form

$$\mathbf{PAQ} = \begin{pmatrix} \mathbf{A}_{11} & \mathbf{A}_{12} & . & . & . & . \\ & \mathbf{A}_{22} & . & . & . & . \\ & & \mathbf{A}_{33} & . & . & . \\ & & & . & . & . \\ & & & & . & . \\ & & & & & \mathbf{A}_{ll} \end{pmatrix}. \qquad (2.1)$$

The blocks $\mathbf{A}_{ii}, i = 1, 2, ..., l$ are all square. If the matrix is reducible (that is, if $l > 1$), many blocks are often of very small order, particularly one. For efficiency, we merge adjacent blocks of order one and note that the resulting diagonal block is triangular and so does not need factorization. We have an option to merge adjacent blocks of order greater than one until they have a specified minimum size. This latter merging does affect the sparsity of the subsequent factorization, but permits better exploitation of the Level 3 BLAS and reduces procedure-call overheads.

If the matrix is square but structurally singular (there is no set of entries that can be permuted onto the diagonal) or is rectangular, we treat it as a single block. Block triangularization can be extended to these cases (see, for example, Pothen and Fan 1990), but our main goal was to treat the square nonsingular case and we have not included this extension.

MA48 handles a rectangular or singular matrix by continuing with the factorization even if all entries of a row or column of the reduced matrix are zero (or below a threshold ε). We make no claim that this provides a robust method for revealing the rank, but it does permit the sensible treatment of simple cases (surprisingly common) where a row or column of $\mathbf{A}$ is zero or identical to another row or column.

One of the main ways that we achieve high performance, particular on vector or super scalar machines, is to switch to full-matrix processing using Level 3 BLAS (Dongarra, Du Croz, Duff, and Hammarling 1990) once the matrix is sufficiently dense. This has led us to using a column-oriented representation internally because of the column-major ordering used by Fortran. It also means that the inner loops of the solve phase will vectorize more readily because they involve adding a multiple of one vector to another, rather than a dot product.

We have adopted a design philosophy of requiring more storage when this leads to worthwhile performance improvements. For example, we construct a map array when first permuting a matrix of a given pattern to the internal column-oriented form so that subsequent matrices can be permuted by a single vectorizable loop of length the number of entries.

In addition to integrating iterative refinement as an option in the solve subroutine, we also provide options for calculating estimates of the relative backward error and of the error in the solution (Arioli, Demmel, and Duff 1989). This provides a good method for assessing the stability and the accuracy in the solution and obviates the need for any a posteriori determination of growth in the entries as was used by MA28, which was in any case rather pessimistic.

We provide a facility for dropping entries that are smaller than a threshold δ (with default value 0). Normally, we expect iterative refinement to be used when this option is active, but it is quite possible to use this option to obtain a preconditioner for a more powerful iterative method. In this case, a higher value for δ may be possible. Zlatev (1991, chapter 11) points out that conjugate-gradient type methods may be very effective in this context.

In some applications, only a small number of entries differ between successive factorizations. If the changed entries are confined to columns late in the pivot sequence, the factorization operations will be identical until the first changed column is reached. We have decided in MA48 to allow the user the option of specifying which columns may change. During the analyse phase, these columns are restricted to the end of the pivot sequence and, during the factorize phase, operations for the early columns are omitted once the first factorization of a sequence is complete.

We avoid the use of COMMON in MA48 since this feature is not well-matched to the requirements of parallel processing, where several copies of the routines may be executing at once. Instead we use array arguments, with a separate initialization routine to provide default values for controlling parameters.

3 MA50 – routines for the numerical processing of a single block

MA50 accepts an $m \times n$ sparse matrix **A** whose entries are stored by columns. When called from MA48, the matrix **A** of this section is one of the blocks $\mathbf{A}_{ii}$ of the block triangular form (2.1). We do not allow repeated indices within a column, since knowing that there are no duplicates allows us to write more efficient code for handling fill-ins. Any duplicates presented to MA48 are summed by it before MA50 is called.

There are four subroutines that are called directly by the user:

Initialize. MA50I provides default values for the parameters that control the execution of the package. The user may alter one or more values before passing the parameters to the other subroutines.

Analyse. MA50A is given a matrix **A** and finds permutations **P** and **Q** suitable for the triangular factorization **PAQ** = **LU**, where **L** is block lower triangular and **U** is unit upper triangular. In normal LU factorization, all blocks in **L** are of size 1, but since we switch to full-matrix processing the final block of **L** is of order greater than unity. MA50A does no full-matrix processing but merely determines the size of this block and arbitrarily completes the permutation vectors to order the full-matrix block at the end. There is an option for dropping small entries from

the factorization and an option for providing **Q** together with a recommendation for **P**. Estimates for storage and operation counts during a subsequent factorization are provided.

Factorize. `MA50B` accepts a matrix **A** together with recommended permutations and size for the final block. It performs the factorization **PAQ** = **LU** and the factorization of the final block of **L**, including additional row permutations when needed for numerical stability. Options exist for subsequent calls for matrices with the same sparsity pattern to be made faster on the assumption that exactly the same permutations are suitable, that no change has been made to the leading columns of **PAQ**, or both.

Solve. `MA50C` uses the factorization produced by `MA50B` to solve the equation **Ax** = **b** or the equation $\mathbf{A}^{\mathrm{T}}\mathbf{x} = \mathbf{b}$.

3.1 MA50A: analyse

`MA50A` chooses row and column permutations suitable for the factorization

$$\mathbf{PAQ} = \mathbf{LU}. \tag{3.1.1}$$

At each pivotal stage, the reduced matrix is updated and then the pivotal row and column are discarded. Once the density of the reduced matrix (ratio of its number of entries to its total size) reaches a threshold with default value 0.5, the whole reduced matrix is discarded. Any ordering for the remaining rows and columns is acceptable since full-matrix processing with row interchanges is applied in the factorize phase. The pivot sequence and the number of columns in the first part are stored for use in the factorize phase.

3.1.1 Treating a square matrix with the default pivotal strategy.

In this subsection, we describe the most important case, where the matrix is square and nonsingular, the default pivotal strategy is in operation, and the option for dropping small entries is not in operation. We defer the other cases to the following subsections.

The column-oriented storage scheme is suitable for the processing of the matrix provided we do not insist that the columns remain contiguous and supplement it by also holding the pattern by rows. Careful use of sparse-matrix techniques allow this row-oriented pattern to be constructed in $O(m) + O(n) + O(\tau)$ time, where τ is the number of entries:

(i) sweep the column-oriented storage to count the numbers of entries in the rows;

(ii) accumulate the counts to give pointers to just beyond the row ends; and

(iii) sweep the column-oriented storage again, storing the column indices in appropriate positions for each row while decrementing the pointers.

For stability, we require each pivot to satisfy the column threshold test

$$|a_{pj}| \geq u \max_i |a_{ij}| \tag{3.1.2}$$

within the reduced matrix, where u is a threshold, with default value 0.1 (see, for example, Duff, Erisman, and Reid 1986). We also require pivots to be greater in absolute value than a tolerance ε with default value zero.

For sparsity, we follow Zlatev (1980) in searching a small number of columns of the reduced matrix for an entry with least Markowitz cost (product of the number of other entries in the row of the reduced matrix and the number of other entries in the column) among entries that satisfy the stability criterion. With this strategy, we choose the columns with least numbers of entries and limit the search to a given number of columns (the default number is 3).

In order to be able to find quickly which columns to search, we maintain doubly-linked chains of columns with equal numbers of entries. They are constructed so that the chains are in forward order, giving an initial bias towards keeping to the natural ordering.

For systems that are of symmetric structure or nearly symmetric structure, it can be advantageous to restrict pivots to the main diagonal. For example, Duff (1984) found significant savings when factorizing matrices from a five-point discretization of Poisson's equation. We provide an option to restrict pivots to the main diagonal although this restriction may cause significant loss of sparsity. We implement this without modifying the data structures, because we judge that it is sufficient to offer comparable efficiency to that of the ordinary case. If the restriction to diagonal entries and the stability test (3.1.2) together mean that no satisfactory pivot can be found, we switch immediately to full-matrix processing. The full-matrix processing (see Section 3.4) uses row interchanges and does not restrict pivots to the diagonal.

We provide an option for specifying that a given number of columns at the end of **A** are also at the end of **PAQ**. This allows for rapid refactorizations when entries in only these columns change. We refer to them as ***late*** columns. Only allowing changes to columns at the end of **A** makes the code simpler and can be recorded using only one integer for each block of the block triangular form. In `MA48` itself, we provide a general facility in which any set of columns may be labelled as the only ones to change.

We also allow the user to specify the column permutation **Q** together with a recommended row permutation **P**. The entries in the specified column that satisfy the stability test (3.1.2) are candidates for the pivot and we take the one that is earliest in the recommended row sequence.

3.1.2 Markowitz pivoting

For sparsity, we also offer the strategy of Markowitz (1957). The pivot is chosen to minimize the Markowitz cost over all entries that satisfy inequality (3.1.2) and are bigger than the pivot tolerance ε. For speed, we maintain doubly-linked chains of rows and also of columns with equal numbers of entries. It is possible to combine these chains and save one pointer array but the added complication to the code and consequent inefficiency does not justify this small saving. As we did before for the columns, we construct the chains of rows with equal numbers of entries in forward order to give an initial bias towards keeping to the natural ordering. In this case also, we permit the option of restricting pivots to the diagonal.

If a column ordering **Q** has not been specified, a search is made of the columns and rows in order of increasing numbers of entries. Because we store reals by columns, it is easy to combine the threshold and Markowitz tests when searching columns, but it is substantially more costly to perform the stability test when searching by rows. For this reason we search columns before rows with the same number of entries and only make the stability test when searching by rows after determining that the entry has lower Markowitz cost than the current

candidate. We terminate the search as soon as it is clear that the Markowitz cost of the current pivot candidate cannot be bettered; that is, we can terminate when searching columns with l entries if the candidate pivot has cost no more than $(l-1)^2$, and can terminate when searching rows with l entries if the candidate pivot has cost no more than $l(l-1)$. This procedure usually finds the pivot with the lowest Markowitz cost very quickly. However, it occasionally can be very slow (see, for instance, the example at the end of Section 5.3). For this reason, we choose the Zlatev scheme as the default.

3.1.3 Drop tolerances

There is an option for dropping, that is removing from the data structure, any entry of the original matrix or a reduced matrix if its absolute value is less than a tolerance δ (with default value zero). Such small entries are dropped from the original matrix and from the columns when they are updated. Separate loops are used to avoid overheads in the case without drop tolerances, which we expect to be the usual one. Each fill-in value is checked against the drop tolerance and is added to the data structure only if it is sufficiently large.

Fast factorizations are not available following a first factorization that drops any entries since the sparsity pattern may be incorrect. However, there is no problem with first factorizations following analyses that drop entries since only the pivot sequence and size of the full block are needed.

3.1.4 Singular and rectangular matrices

It is straightforward to factorize a singular or rectangular matrix and we decided that `MA50` should do this. If it finds r pivots, the factorization can be written in the form

$$\mathbf{P}\,\mathbf{A}\,\mathbf{Q} = \begin{pmatrix} \mathbf{L}_r & \\ \mathbf{M}_r & \mathbf{E} \end{pmatrix} \begin{pmatrix} \mathbf{U}_r & \mathbf{W}_r \\ & \mathbf{I} \end{pmatrix}, \tag{3.1.3}$$

where $\mathbf{L}_r$ is lower triangular of order r, $\mathbf{U}_r$ is unit upper triangular of order r, and all the elements of $\mathbf{E}$ are less than the pivot tolerance ε or the drop tolerance δ. Replacing $\mathbf{E}$ by $\mathbf{0}$ corresponds to perturbing the elements of $\mathbf{A}$ by at most the pivot or drop tolerance and gives us a rank r matrix. The corresponding set of equations is

$$\begin{pmatrix} \mathbf{L}_r\mathbf{U}_r & \mathbf{L}_r\mathbf{W}_r \\ \mathbf{M}_r\mathbf{U}_r & \mathbf{M}_r\mathbf{W}_r \end{pmatrix} \begin{pmatrix} \mathbf{x}_1 \\ \mathbf{x}_2 \end{pmatrix} = \begin{pmatrix} \mathbf{b}_1 \\ \mathbf{b}_2 \end{pmatrix}, \tag{3.1.4}$$

and we solve this by solving

$$\mathbf{L}_r\mathbf{U}_r\mathbf{x}_1 = \mathbf{b}_1 \tag{3.1.5}$$

and setting $\mathbf{x}_2 = \mathbf{0}$. If the whole system is consistent, this will be a solution. If the whole system is underdetermined, the choice of $\mathbf{0}$ for $\mathbf{x}_2$ means that the solution has a reasonably small norm, though in general it will not be of minimum norm.

A key problem is the identification of the rank r. It can quite easily happen that it is overestimated by this procedure but the solution may be verified by using the iterative refinement option of `MA48`. An overestimate leads to equation (3.1.5) being ill-conditioned and usually having a solution of large norm.

At any stage of the `MA50A` processing, we may encounter a row or column that is either structurally or numerically zero. Such a row or column is ordered immediately without choosing a pivot. The natural place to put it is at the end of the pivot sequence, as in (3.1.3),

and this is done for the rows. It cannot be done for the columns since this may put a column that is not 'late' (see Section 3.1.1) among the late columns. Therefore, MA50A places a column in which it cannot find a pivot in the next pivotal position or the next position among the 'late' columns. In both cases, we effectively continue with a matrix with one less row or column. Note that MA50B needs to be able to tolerate (see Section 3.2.4) such a column in the middle of the pivot sequence since different numerical values may provoke the event in different columns. For reasons explained in Section 3.2.4, zero rows are included in the part that is processed as a full matrix, so the number of zero rows must be taken into account when choosing the point for switching to full-matrix processing.

A row or column is regarded as numerically zero if all its entries are less than the pivot tolerance ε. If ε is less than the drop tolerance δ, the pivot tolerance will never come into play since any small enough entries will already have been dropped. The important case will occur when ε is positive and $\delta = 0$.

3.2 MA50B: factorize

MA50B is given an $m \times n$ sparse matrix **A**, recommended permutations, and the number of columns p to be processed as packed sparse vectors. It calculates the actual factorization

$$\mathbf{PAQ} = \mathbf{LU}, \tag{3.2.1}$$

where **L** is block lower triangular and **U** is unit upper triangular. Only the final rectangular block of **L** is of order greater than unity. The permutations and the value of p may have been calculated by a prior call of MA50A, but any choice is acceptable (note that MA50A uses the density of the reduced matrix to choose p). We provide an option for the special case **Q** = **I**. This option is used by MA48 since the column permutations for the blocks of the block triangular form and the permutations chosen by MA50 within the blocks can be integrated into a single overall permutation, thereby saving storage.

We have chosen for **U** to have unit diagonal rather than **L** because in a column-oriented algorithm it seemed natural to hold the multiples of the previous columns of the reduced matrix that need to be added to the current column. There is the minor advantage that we avoid the overheads of an additional loop to scale the column of **L** once the pivot has been chosen. Note that we are still performing the stability test (3.1.2) by columns, and we in fact store the reciprocal of the pivot to avoid excessive divisions.

3.2.1 First factorization

We begin by considering a first factorization when the rank is n. The operations are performed column by column because the technique of Gilbert and Peierls (1988) then allows row interchanges to be introduced while ensuring that the organizational overheads are proportional to the number of floating-point operations. It also means that the factorization, including fill-ins, can be built progressively by columns with very simple data management. If, however, numerical pivoting causes row interchanges, it is possible that the fill-in will be much higher than predicted by the analyse phase. In such a case, it may be sensible to rerun the analyse phase.

To understand the technique of Gilbert and Peierls, it is convenient to regard the packed representation of **L** as a representation of the product

$$\mathbf{L} = \mathbf{D}_1 \mathbf{L}_1 \mathbf{D}_2 \mathbf{L}_2 \ldots \mathbf{D}_p \mathbf{L}_p \mathbf{D}_n \tag{3.2.2}$$

where each $\mathbf{D}_k$, $k = 1, 2, \ldots, p$, is diagonal and equal to the unit matrix except in position (k,k), each $\mathbf{L}_k$ is lower triangular and equal to the unit matrix except below the diagonal in column k, and $\mathbf{D}_n$ is equal to the unit matrix except in the final block of order $n–p$. To calculate column k of $\mathbf{L}$ and $\mathbf{U}$ requires the premultiplication of column k of $\mathbf{PAQ}$ by

$$\mathbf{L}_l^{-1} \mathbf{D}_l^{-1} \ldots \mathbf{L}_1^{-1} \mathbf{D}_1^{-1}, \; l = \min(k-1, p) \tag{3.2.3}$$

In the sparse case, many of these operations may be omitted since the application of $\mathbf{L}_i^{-1} \mathbf{D}_i^{-1}$ to a vector whose i–th component is zero does not alter the vector. In fact, when processing column k, only those $\mathbf{L}_i^{-1} \mathbf{D}_i^{-1}$ whose index i is the index of an entry of column k of $\mathbf{U}$ are needed. Furthermore, there is freedom to reorder them provided no modification of component i is performed after the application of $\mathbf{L}_i^{-1} \mathbf{D}_i^{-1}$. We choose to generate the index list for column k of $\mathbf{U}$ so that index i precedes index j if $\mathbf{L}_j$ has an entry in row i. We use a backward loop to do the actual floating-point operations later. Any order suffices for the indices of column k of $\mathbf{L}$. Gilbert and Peierls construct these lists using a depth-first search, as in Figure 1a, where U-list is a list of integers in which the indices of the entries of column k of $\mathbf{U}$ (apart from the diagonal) are accumulated and L-list is a similar list for column k of $\mathbf{L}$. Each entry of column k of $\mathbf{A}$ and each entry of each column of $\mathbf{L}$ that is involved in the column k calculation is visited just once, so the overall complexity is that of the number of entries involved. Since Fortran 77 does not permit recursive procedures, our code manages a stack explicitly, as illustrated in Figure 1b.

```
set U-list and L-list to be empty
call search(k)

recursive subroutine search(j)
  do i = each index of column j
    if (i not in U-list or L-list) then
      if (i > k) then
        add i to L-list
      else
        call search(i)
      end if
    end if
  end do
  add j to U-list
end subroutine search
```

Figure 1a. Recursive description

```
set stack, U-list, and L-list to be empty
push k on stack
do until stack empty
  set j = stack top
  do i = each unsearched index of column j
    if (i not in U-list or L-list) then
      if (i > k) then
        add i to L-list
      else
        push i on stack; cycle outer do
      end if
    end if
  end do
  add j to U-list and pop stack
end do
```

Figure 1 b. Stack description

Figure 1. Pseudocode for the Gilbert-Peierls algorithm

For efficient execution of the actual floating-point operations, we load the entries of column k of $\mathbf{A}$ into a real work vector that has previously been set to zero. Appropriate multiples of the active columns of $\mathbf{L}$ are added into this vector in the order given by traversing the U-list backwards. Once this has been done, the entries of the upper-triangular part of the column can be unloaded into the packed vector using the known pattern in the U-list and the entries of the real work vector reset to zero ready for their next use. A search for the pivot is performed for columns 1, 2,..., p. The pivot is the entry that lies earliest in the recommended row order among those that exceed the pivot tolerance ε and satisfy the threshold test (3.1.2). This means that if the matrix entries that were presented to `MA50A` are presented to `MA50B`, the same pivot sequence will normally be taken; however, the floating-point operations may

be performed in a different order so the roundoff errors may differ and may very occasionally lead to a change in the pivot order.

A very worthwhile improvement to the Gilbert-Peierls algorithm has been suggested by Eisenstat and Liu (1993). Suppose that column k is the first column with the properties that it is updated by column j and that column j has an entry in row k. Any entries of column j that lie beyond k in the pivot sequence will also be entries in column k. We place these physically at the end of column j and mark the boundary. When a later column l is updated by column j, it is also updated by column k, so the entries beyond the boundary in column j are not needed to find the pattern of column l. Thus, when the operations for column k have been completed and the pivot chosen, we examine all the columns active in the step, looking for columns not already marked and involving the pivot row. For any such column, the entries are physically reordered and the column is marked.

The columns of the final block of **L** (which need not be square), corresponding to columns p+1,..., n of **A**, need only a single vector of row indices. This is constructed when column p+1 is reached and corresponds to all the rows not so far ordered. We run through the rows in order, i = 1, 2,..., m placing each in turn in the vector if it has not been ordered. This makes the indices monotonic, which allows an in-place sort during the solution (see Section 3.3). For each remaining column, we need to apply the operations of the first p pivotal steps and find the sparsity pattern of the U-part. This is done efficiently by the Gilbert-Peierls algorithm, as for the previous columns. The differences are that no pivot need be chosen and the single vector of row indices of the full block is used to unload the L-part of the column.

If the user provides insufficient storage for the factorization, a serious attempt is made to calculate how much is needed for a successful factorization. This is done by discarding all the factorization except the first p columns of **L** which are required so that the processing of column k can take place as in the successful case.

Once the processing of column n is complete, the full block is factorized by full-matrix processing, see Section 3.4. The resulting factorization has the form

$$\begin{pmatrix} \mathbf{L}_p & \\ \mathbf{M}_p & \mathbf{F} \end{pmatrix} \begin{pmatrix} \mathbf{U}_p & \mathbf{V}_p \\ & \mathbf{I} \end{pmatrix}, \tag{3.2.4}$$

where $\mathbf{L}_p$ is a $p \times p$ lower-triangular matrix, $\mathbf{U}_p$ is a $p \times p$ unit upper-triangular matrix, an **F** is the full block of order $m–p \times n–p$.

3.2.2 Drop tolerances

If the option for dropping small entries is active ($\delta > 0$), checks are made on the entries following the updating of column k. We use an absolute drop tolerance on the assumption that the original matrix is well scaled, which is an assumption underlying the stability test (3.1.2). With a relative tolerance, the code would be reluctant to drop entries in a column of the reduced matrix whose entries are all small because of near singularity.

If the matrix entries that were presented to `MA50A` are presented to `MA50B`, the same entries will normally be dropped; however, the floating-point operations may be performed in a different order so the roundoff errors may differ and may very occasionally lead to a different set of dropped entries.

For efficient execution in the default case, we use separate loops for the default and

non-default cases. In the default case ($\delta = 0$), no entries are dropped, not even those with the value zero. This is in order to ensure that the correct structure is generated for a subsequent fast factorization.

If any entries are dropped from column *k*, it cannot be relied upon to supply any of the pattern of an earlier column *j*, so the technique of Eisenstat and Liu (1993) is not applicable. We do not mark and set boundaries for any columns active in the step. If any column that would have been treated is active in a later pivotal step in which no entries are dropped, the technique may be applied then.

3.2.3 Subsequent factorizations

Following a successful first factorization, if drop tolerances are not in use, if the pattern is unchanged, and if the pivotal sequence is numerically stable for the new values, the fast factorization may be used. It is faster because of not needing to find the pattern and choose the pivots. The algorithm is unchanged from that used for calculating the numerical values during the first factorization. An error return is made if any pivot is smaller than the pivot threshold ε.

The user may specify that only a certain number of late columns have changed values so that processing can be confined to these columns, because the factorization in the leading columns will be exactly as previously. If the pattern is unchanged and the previous pivot sequence is expected to be satisfactory, this processing may be that of a fast factorization. Otherwise, the processing is that of a first factorization, with Gilbert-Peierls calculation of the pattern and pivoting within each column.

3.2.4 Singular and rectangular matrices

If the rank is less than *n*, we may fail to find a pivot for column *k*. The L-part may be null or all its entries may be smaller than the pivot threshold ε or the drop tolerance δ. This is handled by recording the L-part of the column as null and not recording any row in the pivot sequence. The rest of the reduction is effectively treated as if column *k* were omitted.

The column by column processing makes it impossible to recognize a zero row until all columns have been processed. We do not remove such rows from the full matrix before passing it to the full-matrix factorization subroutine, although this would have been possible. We felt that coding this was not justified given that the full-matrix code needs anyway to handle the possibility of zero rows occurring during its processing.

To explain the mathematics, it is convenient to permute each column in which no pivot is found to just ahead of the columns holding the full block, though we emphasize that in the actual code these columns are left in place. This gives us the factorization

$$\begin{pmatrix} \mathbf{L}_q & & \\ \mathbf{M}_q & \mathbf{0} & \mathbf{F} \end{pmatrix} \begin{pmatrix} \mathbf{U}_q & \mathbf{V}_q & \mathbf{W}_q \\ & \mathbf{I} & \\ & & \mathbf{I} \end{pmatrix}. \tag{3.2.5}$$

It is also convenient (see Section 3.3) to regard this as the factorization

$$\begin{pmatrix} \mathbf{L}_q & \\ \mathbf{M}_q & \mathbf{I} \end{pmatrix} \begin{pmatrix} \mathbf{U}_q & \mathbf{V}_q & \mathbf{W}_q \\ & \mathbf{0} & \mathbf{F} \end{pmatrix}. \tag{3.2.6}$$

3.3 MA50C: solve

MA50C uses the factorization produced by MA50B to solve the equation

$$\mathbf{A}\mathbf{x} = \mathbf{b} \tag{3.3.1}$$

or the equation

$$\mathbf{A}^{\mathrm{T}}\mathbf{x} = \mathbf{b}. \tag{3.3.2}$$

In the square nonsingular case, this involves simple forward and back-substitution using the factorization (3.2.4). We use a work vector to avoid altering **b**. MA50H is called for full-matrix processing of the final block. It is helpful that the row indices of the full block are monotonic, as noted in the penultimate paragraph of Section 3.2.1. When solving (3.3.1), this permits an in-place sort for loading the required components of the right-hand side; when solving (3.3.2), it permits an in-place sort for placing the solution in the required positions.

The rectangular or rank-deficient case is not so straightforward. For (3.3.1), we use the form (3.2.6) and begin by solving the system

$$\begin{pmatrix} \mathbf{L}_q & \\ \mathbf{M}_q & \mathbf{I} \end{pmatrix} \begin{pmatrix} \mathbf{y}_1 \\ \mathbf{y}_2 \end{pmatrix} = \begin{pmatrix} \mathbf{b}_1 \\ \mathbf{b}_2 \end{pmatrix} \tag{3.3.3}$$

by forward substitution, followed by solving the system

$$\begin{pmatrix} \mathbf{U}_q & \mathbf{V}_q & \mathbf{W}_q \\ & \mathbf{0} & \mathbf{F} \end{pmatrix} \begin{pmatrix} \mathbf{x}_1 \\ \mathbf{x}_2 \\ \mathbf{x}_3 \end{pmatrix} = \begin{pmatrix} \mathbf{y}_1 \\ \mathbf{y}_2 \end{pmatrix} \tag{3.3.4}$$

by back-substitution. Mathematically, we solve $\mathbf{F}\mathbf{x}_3 = \mathbf{y}_2$, set $\mathbf{x}_2 = \mathbf{0}$, and then solve $\mathbf{U}_q\mathbf{x}_1 = \mathbf{y}_1 - \mathbf{W}_q\mathbf{x}_3$, but the two final steps are merged in the coding since the columns are interspersed. We traverse the columns backwards either calculating a component of **x** and doing the corresponding back-substitution updating, or setting a component of **x** to zero.

For (3.3.2), we use the form (3.2.5) and begin by solving the system

$$\begin{pmatrix} \mathbf{U}_q^{\mathrm{T}} & & \\ \mathbf{V}_q^{\mathrm{T}} & \mathbf{I} & \\ \mathbf{W}_q^{\mathrm{T}} & & \mathbf{I} \end{pmatrix} \begin{pmatrix} \mathbf{y}_1 \\ \mathbf{y}_2 \\ \mathbf{y}_3 \end{pmatrix} = \begin{pmatrix} \mathbf{b}_1 \\ \mathbf{b}_2 \\ \mathbf{b}_3 \end{pmatrix} \tag{3.3.5}$$

by forward substitution, thereafter solving the system

$$\begin{pmatrix} \mathbf{L}_q^{\mathrm{T}} & \mathbf{M}_q^{\mathrm{T}} \\ & \mathbf{0} \\ & \mathbf{F}^{\mathrm{T}} \end{pmatrix} \begin{pmatrix} \mathbf{x}_1 \\ \mathbf{x}_2 \end{pmatrix} = \begin{pmatrix} \mathbf{y}_1 \\ \mathbf{y}_2 \\ \mathbf{y}_3 \end{pmatrix} \tag{3.3.6}$$

by back-substitution. Here, we ignore the middle block row.

3.4 Solving full sets of linear equations

For sufficiently dense matrices, it is more efficient to use full-matrix processing and we therefore switch to this towards the end of the factorization. We use notation in this section that is independent of that of the rest of this paper and refers only to the full matrix. We had hoped to use the routines SGETRF and SGETRS from LAPACK (Anderson *et al.,* 1992), but their treatment of the rank-deficient case is unsatisfactory since no column interchanges are included. For example, the matrix

$$\mathbf{A} = \begin{pmatrix} 0 & 1 & 1 \\ & 0 & 1 \\ & & 0 \end{pmatrix}$$

will be factorized as **A** = **LU** with **L** = **I** and **U** = **A**, which is of no help for solving the consistent set of equations

$$\begin{pmatrix} 0 & 1 & 1 \\ & 0 & 1 \\ & & 0 \end{pmatrix} \mathbf{x} = \begin{pmatrix} 2 \\ 1 \\ 0 \end{pmatrix}.$$

On the other hand, interchanging columns 1 and 3 gives

$$\mathbf{AQ} = \begin{pmatrix} 1 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix} = \begin{pmatrix} 1 & & \\ 1 & 1 & \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 1 & 0 \\ & -1 & 0 \\ & & 0 \end{pmatrix}$$

and the reduced set of equations

$$\begin{pmatrix} 1 & 1 & 0 \\ & -1 & 0 \\ & & 0 \end{pmatrix} \begin{pmatrix} x_3 \\ x_2 \\ x_1 \end{pmatrix} = \begin{pmatrix} 2 \\ -1 \\ 0 \end{pmatrix}.$$

The value of $\mathbf{x}_1$ is arbitrary and we may choose 0. By back-substitution, we then get the solution

$$\mathbf{x} = \begin{pmatrix} 0 \\ 1 \\ 1 \end{pmatrix}.$$

Another reason for rejecting SGETRF is that it tests only for exact zeros. We test for exact zeros by default, but wish to offer the option of a test against a threshold. The final factorization will be as if we had started with a matrix whose entries differ from those of **A** by at most the threshold.

In our early tests, we found that factorization routines using Basic Linear Algebra Subroutines (BLAS) at Level 1 (Lawson *et al.* 1979) and Level 2 (Dongarra *et al.* 1988) sometimes performed better than those at Level 3 (Dongarra *et al.* 1990), and have therefore included them all. They are, respectively, MA50E, MA50F, and MA50G. A parameter controls which of them is called. In the tests reported in Section 5, we found that the Level 3 versions performed best on all three of our test platforms, so the default parameter value chooses them.

MA50H solves a set of equations using the factorization produced by MA50E, MA50F, or MA50G, whose output data are identical. Each actual forward or back-substitution operation associated with **L** or **U** is performed either with the Level 2 BLAS STRSV or by a loop involving calls to SAXPY or SDOT. An argument controls which of these happens. Unlike the case for factorization, the logic is very similar for the two cases, so there is no need for separate subroutines.

We defer a more detailed description of our modifications of the LAPACK subroutines to the Appendix.

4 The MA48 Package

We anticipate that most users will access MA48 itself. The data interface is much simpler than that of MA50. MA48 accepts an $m \times n$ sparse matrix whose entries are stored in any order. Multiple entries are permitted and are summed. Any entry with an out-of-range index is ignored.

Four subroutines are called directly by the user:

Initialize. MA48I provides default values for the parameters that control the execution of the package.

Analyse. MA48A prepares data structures for factorization and chooses permutations **P** and **Q** that provide a suitable pivot sequence and optionally permute the matrix **A** to block upper triangular form. There is an option for dropping small entries from the factorization, an option for limiting pivoting to the diagonal, and an option for providing **Q** together with a recommendation for **P**. Any set of columns may be specified as sometimes being unchanged when refactorizing.

Factorize. MA48B factorizes a matrix **A**, given data provided by MA48A. On an initial call, it performs additional row permutations when needed for numerical stability. Options exist for subsequent calls for matrices with the same sparsity pattern to be made faster on the assumption that exactly the same permutations are suitable, that no change has been made to certain columns of **PAQ**, or both.

Solve. MA48C uses the factorization produced by MA48B to solve the equation $\mathbf{Ax} = \mathbf{b}$ or the equation $\mathbf{A}^T\mathbf{x} = \mathbf{b}$ with the option of using iterative refinement. Estimates of both backward and forward error can also be provided.

The data structure is arranged so that the user with a single problem to solve can provide the matrix to MA48A, pass the MA48A output data on to MA48B, and finally pass the MA48B output data and the vector **b** to MA48C. Further calls to MA48C can then be made for other vectors **b**. The first of a sequence whose matrices have the same pattern is treated similarly, and for subsequent matrices MA48B can be called with just the array of reals having a different value. For efficient performance of the sorting needed for the later factorizations, we use a map array so that a single vectorizable loop is all that is needed. Note also that a representation of both the original matrix and its factorization is needed by MA48C since it performs iterative refinement. Storage for this can be saved when the matrix has a non-trivial block triangular form since the off-diagonal blocks and triangular blocks on the diagonal are only stored with the original matrix as they are unchanged by the factorization.

4.1 MA48A: analyse

The action of MA48A is controlled by the argument JOB, which must have one of the values:

1 Unrestricted pivot choice.

2 Column permutation provided by the user, together with a recommended row permutation.

3 Pivots to be restricted to the diagonal.

An attempt is made to order the matrix to block triangular form (2.1) as long as the matrix

is square, the minimum block size (default value 1) is less than *n*, and JOB=1. It is conventional (see, for example, Chapter 6 of Duff, Erisman, and Reid 1986) to do this in two stages: first find a column permutation such that the permuted matrix has entries on its diagonal and then find a symmetric permutation that permutes the resulting matrix to block triangular form. We use the Harwell Subroutine Library subroutines MC21A and MC13D for these two stages. MC21A uses a depth-first search algorithm with look ahead and is described by Duff (1981a, 1981b). If it fails to permute entries onto the whole of the diagonal, the matrix must be structurally singular and the block triangularization is abandoned, for the reasons given in Section 2.

If the matrix is structurally nonsingular, MC13D is used to symmetrically permute the resulting matrix to block triangular form. It employs the algorithm of Tarjan (1972) and is described by Duff and Reid (1978a, 1978b). Adjacent blocks of size one are amalgamated into triangular blocks in a single pass that combines the current block with the previous one if the current block is 1×1 and the previous block is either 1×1 or is itself an amalgamation of 1×1 blocks. The triangular case is indicated by negating the block size. A second pass through the blocks is made to merge the current block with its predecessor (which may itself be a merged block) if the predecessor is of size less than the minimum block size.

Since permutations for the block triangular form may conflict with the user's permutations or may move diagonal entries away from the diagonal, we do not perform block triangularization if JOB has the value 2 or 3.

The user may specify that, for some refactorizations, changes are confined to a given set of columns. These columns must be placed at the end of any non-triangular block in order that MA50 handles them appropriately as 'late' columns. If the column sequence has been specified (JOB=2), all we can do is scan for the first of the set of columns and treat all subsequent columns as if they too were columns that change. The appropriate value is recorded for MA50A. If no column sequence is specified (JOB=1 or JOB=3), each block is checked in turn and the columns of the set are moved to the end of the block. The permutation arrays are adjusted accordingly and the number of late columns in each block recorded for subsequent use by MA50A. Note that the late-column convention in MA50 not only leads to simplifications in MA50 but also limits the MA48 storage overhead for this feature to one integer per block.

The analysis proceeds by calling MA50A for each block. After completing all calls to MA50A, the row indices are revised to those of the permuted matrix and are reordered to the new column order. Also the map array is revised to correspond. This is done for the sake of simplicity in MA48B and MA48C. MA48B does not have to be concerned with the permutations since it works entirely with the permuted matrix and MA48C has only to apply one permutation to the incoming vector and another to the outgoing solution.

The original matrix is preserved unaltered so that it can be passed to MA48B and so that MA48B can treat it in exactly the same way as a matrix with the same pattern but changed numerical values.

4.2 MA48B: factorization

MA48B factorizes a sparse matrix, given data from MA48A and possibly changed numerical values for the entries. The action of the subroutine is controlled by the argument JOB that must have one of the values:

1 Initial call, with pivoting.

2 Faster subsequent call for changed numerical values, using exactly the same pivot sequence.

3 Faster subsequent call for changed numerical values only in certain columns, with fresh pivoting in those columns.

MA48B first uses the map array constructed by MA48A to place the real input array immediately in the correct order for the factorization. Separate code is executed according to whether or not duplicates were found by MA48A. With duplicates, an array is initialized to zero and used to accumulate the result. Without duplicates, no initialization is needed and the values can be placed directly in position.

Having reordered the data in this very easy way, it is now a simple matter to work through the block triangular structure, calling the factorize routine MA50B for each non-triangular diagonal block. We also call the factorize routine MA50B for any triangular diagonal block that has a diagonal entry smaller than the pivot threshold ε since MA50B has facilities for including row interchanges and can handle singular matrices.

4.3 MA48C: solve

MA48C solves a system of equations, given data from MA48B. The action of the subroutine is controlled by the argument JOB that must have one of the values:

1 No iterative refinement or error estimation.

2 No iterative refinement but with estimation of relative backward errors.

3 With iterative refinement and estimation of relative backward errors.

4 With iterative refinement and estimation of relative backward errors and relative error in the solution.

We separate the tasks of solution using the block triangular factorization from permutation of the incoming vector, iterative refinement, error estimation, and permutation of the solution. The former task is performed by a separate routine MA48D. For the special case where there is only one block and it is not triangular, we save procedure call overheads by calling MA50C directly rather than calling MA48D.

4.3.1 Solution without iterative refinement

MA48D solves a system of equations using the block structure and calls to MA50C for each non-triangular diagonal block.

For the solution when the matrix **A** is not transposed, the block form is block upper-triangular and the blocks are solved in reverse order. For each block, either MA50C is used or a triangular system is solved, and the new values are then substituted in earlier

equations using the off-diagonal parts of the columns in the current block. Because of the column-oriented storage, the inner loop of the back-substitution for the triangular diagonal blocks and for the off-diagonal blocks involves the addition of a multiple of one vector to another with indirect addressing for the vector being accumulated.

For the transposed problem, the system is block lower-triangular and the solution starts with the (1,1) block and goes forward through the block form. Now the forward substitution loops are dot products with indirect addressing of one of the vectors, which are less likely to vectorize well (see Table 2 in Section 5).

4.3.2 Solution with iterative refinement

Iterative refinement and error estimation is performed on the permuted system so the code is uncluttered by permutations. The initial solution is set to zero and the permuted right-hand side stored to enable the residual calculation. In the iterative refinement loop, the residual equations

$$\mathbf{A}\mathbf{x} = \mathbf{r}^{(k)} = \mathbf{b} - \mathbf{A}\mathbf{x}^{(k)}$$

or (4.3.1)

$$\mathbf{A}^{\mathrm{T}}\mathbf{x} = \mathbf{r}^{(k)} = \mathbf{b} - \mathbf{A}^{\mathrm{T}}\mathbf{x}^{(k)}$$

where $\mathbf{x}^{(k)}$ is the current estimate of the solution, are solved using `MA48D` or `MA50C` as appropriate, and the solution to these residual equations is used to correct the current estimate. We then use the theory developed by Arioli, Demmel, and Duff (1989) to decide whether to stop the iterative refinement. In the following discussion, modulus signs round a matrix or vector indicate the matrix or vector, respectively, obtained by setting all entries equal to the modulus of the corresponding entry of the matrix or vector.

In Arioli *et al.* (1989), the scaled residual

$$\omega_1 = \max_i \frac{|\mathbf{r}^{(k)}|_i}{\left[|\mathbf{A}|\,|\mathbf{x}^{(k)}| + |\mathbf{b}|\right]_i} \tag{4.3.2}$$

is used as a measure of the backward error, in the sense that the estimated solution $\mathbf{x}^{(k)}$ can be shown to be the exact solution of a set of equations

$$(\mathbf{A} + \delta\mathbf{A})\mathbf{x} = \mathbf{b} + \delta\mathbf{b}$$

where the perturbations $\delta\mathbf{A}$ and $\delta\mathbf{b}$ are bounded according to

$$\delta\mathbf{A} \le \omega_1 |\mathbf{A}| \qquad \text{and} \qquad \delta\mathbf{b} \le \omega_1 |\mathbf{b}|.$$

This follows directly from the work of Oettli and Prager (1964) and Skeel (1980). Sparsity, however, can cause an added complication since it is possible for the denominator in (4.3.2) to be zero or very small. We follow the theory developed by Arioli *et al.* (1989) by monitoring the denominator. Let *nvar* be the number of variables in the equation, σ be machine precision times 1000, and $\mathbf{A}_{i.}$ be row i of $\mathbf{A}$. Then, for all i for which the denominator is less than $nvar\,\sigma(|\mathbf{b}|_i + \|\mathbf{A}_{i.}\|_\infty \|\mathbf{x}^{(k)}\|_\infty)$, we replace the denominator by $|\mathbf{A}|\,|\mathbf{x}^{(k)}|_i + \|\mathbf{A}_{i.}\|_\infty \|\mathbf{x}^{(k)}\|_\infty$. We define ω_1 as before for the equations with large denominators, and define ω_2 as

$$\omega_2 = \max_i \frac{|\mathbf{r}^{(k)}|_i}{|\mathbf{A}|\,|\mathbf{x}^{(k)}|_i + \|\mathbf{A}_{i.}\|_\infty \|\mathbf{x}^{(k)}\|_\infty}$$

for these other equations. The calculated backward error is then the sum of ω_1 and ω_2 and the iterative refinement is terminated if this is at roundoff level or has not decreased sufficiently from the previous iteration step. The amount of decrease required is given by a user controllable parameter. If the refinement is being terminated, the solution is set to either the current or previous iterate, depending on which had the lower value for $\omega_1 + \omega_2$; otherwise, the current estimate is saved and we proceed to the next step of iterative refinement.

MA48C now optionally proceeds to estimate the error in the solution, using the backward errors just calculated and an estimate of the condition number obtained by using the Harwell Subroutine Library norm estimation routine MC41, which uses a method based on that developed by Hager (1984), incorporating the modifications suggested by Higham (1988). Condition numbers are estimated corresponding to the two ωs. That corresponding to ω_1 is given by

$$\kappa_{\omega_1} = \frac{\left\| |\mathbf{A}^{-1}| \begin{pmatrix} |\mathbf{A}^{(1)}|\,|\mathbf{x}^{(k)}| + |\mathbf{b}^{(1)}| \\ \mathbf{0} \end{pmatrix} \right\|_\infty}{\|\mathbf{x}^{(k)}\|_\infty},$$

where $|\mathbf{b}^{(1)}|$ are the components of $\mathbf{b}$ corresponding to the equations determining ω_1, and that corresponding to ω_2 by

$$\kappa_{\omega_2} = \frac{\left\| |\mathbf{A}^{-1}| \begin{pmatrix} \mathbf{0} \\ |\mathbf{A}^{(2)}|\,|\mathbf{x}^{(k)}| + \mathbf{f}^{(2)} \end{pmatrix} \right\|_\infty}{\|\mathbf{x}^{(k)}\|_\infty},$$

where $\mathbf{f}^{(2)} = |\mathbf{A}^{(2)}|\mathbf{e}\|\mathbf{x}^{(k)}\|_\infty$, with $\mathbf{e}$ the vector of all 1s. In each case, the norm in the numerator is of the form $\||\mathbf{A}^{-1}|\mathbf{g}\|_\infty$ which is equivalent to $\|\mathbf{A}^{-1}\mathbf{G}\|_\infty$, with $\mathbf{G} = diag\{g_1, g_2, ...\}$, whence the subroutine MC41 can be applied directly.

The bound for the error in the solution, $\frac{\|\delta\mathbf{x}\|_\infty}{\|\mathbf{x}\|_\infty}$, is then given by

$$\omega_1\ \kappa_{\omega_1} + \omega_2\ \kappa_{\omega_2}.$$

5 Performance results

For performance testing, we have taken two subsets of the problems in the Harwell-Boeing collection (see Duff, Grimes, and Lewis 1989 and 1992). The first subset is summarized in Table 1 and was chosen to be representative of the kinds of problems likely to be solved by MA48. We discuss the second subset in Section 5.4.

Case	Identifier	Order	Number of entries	Description
1	SHL 400	663	1712	Basis matrix obtained after 400 steps of the simplex method to a linear programming problem. This matrix is a permutation of a triangular matrix.
2	FS 541 1	541	4285	A matrix that arose in FACSIMILE (a stiff ODE package) in solving an atmospheric pollution problem involving chemical kinetics and two-dimensional transport.
3	FS 680 3	680	2646	Mixed kinetics diffusion problem from radiation chemistry. 17 chemical species and one space dimension with 40 mesh points.
4	MCFE	765	24382	Radiative transfer and statistical equilibrium in astrophysics.
5	BCSSTK19	817	6853	Part of a suspension bridge.
6	ORSIRR 2	886	5970	Oil reservoir simulation.
7	WEST0989	989	3537	Chemical engineering plant model.
8	JPWH 991	991	6027	Circuit physics model.
9	GRE 1107	1107	5664	Matrix produced by the package QNAP written by CII-HB for simulation modelling of computer systems.
10	ERIS1176	1176	18552	Large electrical network.
11	PORES 2	1224	9613	Oil reservoir simulation. Matrix pattern is symmetric.
12	BCSSTK27	1224	56126	Buckling analysis, symmetric half of an engine inlet from a modern Boeing jetliner.
13	NNC1374	1374	8606	Model of an advanced gas-cooled nuclear reactor core.
14	BP 1600	822	4841	Basis matrix obtained after the application of 1600 steps of the simplex method to a linear programming problem.
15	WATT 1	1856	11360	Petroleum engineering problem.
16	WEST2021	2021	7353	Chemical engineering plant model.
17	ORSREG 1	2205	14133	Oil reservoir simulation.
18	ORANI678	2529	90158	Economic model of Australasia.
19	GEMAT11	4929	33185	Initial basis of an optimal power flow problem with 2400 buses.
20	BCSPWR10	5300	21842	Eastern US Power Network – 5300 Bus.

Table 1. The matrices used for performance testing.

We have used the Table 1 matrices to choose default values for parameters and to judge the performance on

(i) one processor of a CRAY YMP-8I/8128 using Release 5.0 of the cf77 compiling system with the option –Zv (maximum vectorization) and vendor-supplied BLAS,

(ii) a SUN SPARCstation 1 using Release 4.1 of the f77 compiler with the option –O (optimization) and Fortran 77 BLAS, and

(iii) an IBM RS/6000 model 550 using Release 2.3 of the xlf compiler with the option –O (optimization) and vendor-supplied BLAS.

We believe that these are representative of the likely runtime environments, but it must be stressed that other platforms, other compilers, or other implementations of the BLAS may require different parameter values for good performance. Also, tuning for particular

requirements may be worthwhile; for example, the choice of density threshold for the switch to full code is affected by whether a single problem is to be solved or many problems with the same pattern are to be solved.

We have been hampered somewhat by the variability of the cpu timers on the IBM RS/6000 and the SUN. To alleviate this, we have embedded each call to MA48 in a loop of length 1000 that is left as soon as the accumulated time exceeds one second and the average time is then calculated. We can judge the repeatability of the timings by the variation of the analyse time when variations of the block size used for the BLAS are made since this does not affect the analyse phase. Occasional individual variations could be as high as 25% on the IBM RS/6000 and 20% on the SUN. The median change over the twenty problems could be as high as 3% on the IBM RS/6000 and 8% on the SUN. The CRAY is much better with all times within 1%. The IBM RS/6000 and the SUN figures presented here were obtained with runs on lightly loaded machines to avoid such extreme variations, but we rely mainly on the CRAY times for our conclusions.

Case	Array size reqd	Analyse	First Fact.	Analyse + Fact.	Fast Fact.	Solve $\mathbf{Ax}=\mathbf{b}$	Solve $\mathbf{A}^T\mathbf{x}=\mathbf{b}$
1	3424	0.012	0.000	0.012	0.000	0.0007	0.0014
2	20229	0.123	0.051	0.175	0.023	0.0013	0.0021
3	7120	0.044	0.017	0.061	0.006	0.0010	0.0017
4	111853	0.762	0.281	1.043	0.175	0.0039	0.0052
5	35507	0.247	0.104	0.351	0.044	0.0021	0.0034
6	61014	0.383	0.139	0.522	0.082	0.0023	0.0036
7	8992	0.069	0.026	0.095	0.008	0.0021	0.0037
8	70973	0.331	0.128	0.458	0.097	0.0029	0.0047
9	72140	0.382	0.170	0.552	0.102	0.0037	0.0054
10	49920	0.176	0.086	0.263	0.042	0.0026	0.0044
11	63840	0.382	0.154	0.536	0.084	0.0030	0.0046
12	216228	1.499	0.561	2.060	0.329	0.0059	0.0085
13	78056	0.483	0.208	0.690	0.108	0.0048	0.0075
14	9682	0.059	0.018	0.076	0.008	0.0022	0.0036
15	167763	1.315	0.504	1.820	0.344	0.0071	0.0100
16	19317	0.150	0.056	0.206	0.017	0.0043	0.0076
17	298348	1.753	0.886	2.639	0.703	0.0092	0.0157
18	182012	0.901	0.263	1.163	0.148	0.0083	0.0130
19	89295	0.595	0.236	0.831	0.073	0.0121	0.0208
20	100810	0.742	0.305	1.047	0.114	0.0115	0.0191

Table 2. Performance on CRAY with default settings.

Case	Array size reqd	Analyse	First Fact.	Analyse + Fact.	Fast Fact.	Solve $\mathbf{Ax}=\mathbf{b}$	Solve $\mathbf{A}^T\mathbf{x}=\mathbf{b}$
1	3424	0.06	0.01	0.07	0.01	0.009	0.010
2	20364	0.99	0.56	1.55	0.38	0.040	0.030
3	6823	0.27	0.12	0.39	0.06	0.017	0.016
4	111875	10.56	7.34	17.90	6.42	0.175	0.131
5	37518	2.61	1.20	3.81	0.85	0.072	0.053
6	65625	3.59	4.06	7.65	3.68	0.121	0.090
7	8986	0.35	0.15	0.50	0.06	0.022	0.021
8	69726	2.34	5.13	7.47	4.81	0.120	0.095
9	72140	4.19	4.83	9.02	4.40	0.138	0.103
10	49920	1.27	1.23	2.50	0.98	0.071	0.056
11	65984	3.68	3.01	6.69	2.56	0.126	0.091
12	218066	23.76	12.32	36.08	10.67	0.353	0.240
13	76941	4.68	3.89	8.57	3.26	0.151	0.113
14	9682	0.28	0.11	0.39	0.06	0.025	0.023
15	169546	20.19	13.69	33.88	12.40	0.330	0.234
16	19314	0.79	0.34	1.12	0.15	0.051	0.046
17	285175	27.23	38.36	65.59	36.47	0.530	0.383
18	182012	10.58	7.10	17.68	6.39	0.292	0.250
19	89329	3.70	1.76	5.46	0.92	0.180	0.153
20	102676	5.60	3.40	9.00	2.36	0.228	0.182

Table 3. Performance on SUN with default settings.

Case	Array size reqd	Analyse	First Fact.	Analyse + Fact.	Fast Fact.	Solve $\mathbf{Ax}=\mathbf{b}$	Solve $\mathbf{A}^T\mathbf{x}=\mathbf{b}$
1	3424	0.013	0.001	0.013	0.000	0.0007	0.0008
2	20364	0.173	0.057	0.231	0.032	0.0033	0.0034
3	6823	0.052	0.016	0.068	0.007	0.0016	0.0017
4	111875	1.900	0.483	2.383	0.353	0.0111	0.0114
5	37518	0.453	0.132	0.586	0.071	0.0066	0.0058
6	65625	0.590	0.236	0.826	0.180	0.0075	0.0074
7	8986	0.071	0.023	0.093	0.008	0.0024	0.0022
8	69726	0.427	0.236	0.663	0.208	0.0067	0.0066
9	72140	0.635	0.308	0.942	0.202	0.0089	0.0088
10	49920	0.232	0.118	0.350	0.082	0.0050	0.0049
11	65984	0.680	0.204	0.884	0.172	0.0078	0.0076
12	218066	3.800	0.945	4.745	0.730	0.0220	0.0249
13	76129	0.975	0.307	1.283	0.187	0.0103	0.0107
14	9682	0.060	0.019	0.079	0.012	0.0028	0.0029
15	169546	3.120	1.030	4.150	0.800	0.0229	0.0217
16	19314	0.160	0.050	0.210	0.020	0.0051	0.0056
17	285175	4.500	1.780	6.280	1.550	0.0312	0.0300
18	182012	2.190	0.410	2.600	0.312	0.0191	0.0171
19	89329	0.685	0.224	0.909	0.100	0.0177	0.0177
20	102676	1.080	0.393	1.473	0.193	0.0220	0.0213

Table 4. Performance on IBM RS/6000 with default settings.

For all three environments, we have chosen the value 0.5 for the density threshold for the switch to full code and Level 3 BLAS with block size 32. We are able to use the same defaults because the performance is very flat around the optimum values, as the results later in this section demonstrate. Tables 2, 3, and 4 summarize the performance of the code with these default values. Note that the different arithmetic on the CRAY sometimes leads to a different pivot sequence and hence to differences in the array sizes required.

The effect of our use of Level 3 BLAS in the full code is most apparent in the solve phase. Since we have chosen a column orientation for the storage of numerical values of the matrix

and factors, the solution of the equations $\mathbf{Ax}=\mathbf{b}$ will be performed using a SAXPY kernel in the innermost loop while the solution of $\mathbf{A}^{\mathrm{T}}\mathbf{x}=\mathbf{b}$ uses an SDOT operation. On the CRAY, the former is more efficient than the latter and this is clearly reflected in the fact that the times for solving the system are up to 50% less than for the solution of the transposed equations. This was one of the reasons why we chose column orientation in the first place. On the IBM, the different relative performance of the two Level 1 BLAS means that times for solution of the system and its transpose are about the same while on the SUN the position is reversed with the faster SDOT routine giving a faster solution time for the transposed equations.

We examine the relative performance when a single parameter is changed by means of the median, upper-quartile and lower-quartile ratios over the 20 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/ma48.tables.

5.1 Density threshold for the switch to full code

Which value is best for the density threshold for the switch to full code depends on the relative importance of analyse time as opposed to factorize time and to the importance of storage. Any reduction will save time in the analyse phase since no further factorization is performed once the threshold is reached. Usually a storage penalty is incurred. Too low a threshold leads to such an increase in factorize time that we lose even if only a single problem is to be solved. We have also been influenced in our choice of default value by the convenience of a single value on all platforms. Our value of 0.5 is based on slightly different priorities on the three platforms.

Table 5 shows the effect of decreasing the value of the density threshold for the switch to full code to 0.4. The factorize times are increased, though only slightly for the first factorize on the CRAY. A smaller value may be preferred if a single problem is to be solved, as may be judged from the sum of the analyse and factorize times (see Table 5). Table 6 shows similar effects from the further reduction to 0.3. For the SUN, this is too low even if only a single problem is to be solved.

		Array size reqd	Analyse	First Fact.	Analyse + Fact.	Fast Fact.	Solve $\mathbf{Ax}=\mathbf{b}$	Solve $\mathbf{A}^{\mathrm{T}}\mathbf{x}=\mathbf{b}$
CRAY	lower q.	1.03	0.90	0.99	0.94	1.02	1.00	0.99
	median	1.09	0.96	1.01	0.97	1.09	1.00	1.00
	upper q.	1.11	0.98	1.05	0.99	1.13	1.02	1.03
SUN	lower q.	1.02	0.85	1.05	0.98	1.08	1.03	1.02
	median	1.09	0.91	1.12	1.00	1.18	1.06	1.07
	upper q.	1.11	0.98	1.24	1.02	1.28	1.09	1.10
IBM	lower q.	1.02	0.80	1.02	0.85	1.04	0.93	0.97
	median	1.09	0.89	1.04	0.92	1.08	0.98	0.99
	upper q.	1.11	0.94	1.07	0.97	1.15	1.03	1.03

Table 5. Results with density threshold value 0.4 divided by those with value 0.5.

		Array size reqd	Analyse	First Fact.	Analyse + Fact.	Fast Fact.	Solve $\mathbf{Ax}=\mathbf{b}$	Solve $\mathbf{A}^T\mathbf{x}=\mathbf{b}$
CRAY	lower q.	1.03	0.89	0.99	0.94	1.02	0.99	0.99
	median	1.09	0.96	1.01	0.97	1.08	1.00	1.00
	upper q.	1.11	0.97	1.05	0.99	1.15	1.01	1.02
SUN	lower q.	1.09	0.68	1.20	1.00	1.32	1.10	1.07
	median	1.23	0.76	1.39	1.04	1.59	1.18	1.18
	upper q.	1.26	0.93	1.56	1.07	1.65	1.24	1.22
IBM	lower q.	1.09	0.66	1.09	0.79	1.23	0.97	1.00
	median	1.23	0.78	1.18	0.89	1.35	1.01	1.02
	upper q.	1.26	0.89	1.27	0.97	1.39	1.06	1.07

Table 6. Results with density threshold value 0.3 divided by those with value 0.5.

		Array size reqd	Analyse	First Fact.	Analyse + Fact.	Fast Fact.	Solve $\mathbf{Ax}=\mathbf{b}$	Solve $\mathbf{A}^T\mathbf{x}=\mathbf{b}$
CRAY	lower q.	0.93	1.02	1.00	1.01	0.97	1.00	1.00
	median	0.96	1.04	1.02	1.04	0.99	1.00	1.00
	upper q.	0.99	1.10	1.03	1.08	0.99	1.01	1.01
SUN	lower q.	0.92	1.03	0.86	1.00	0.82	0.93	0.94
	median	0.96	1.11	0.92	1.01	0.90	0.97	0.96
	upper q.	0.99	1.19	0.98	1.04	0.95	0.99	0.99
IBM	lower q.	0.92	1.01	0.94	0.99	0.91	0.92	0.97
	median	0.96	1.05	0.98	1.03	0.95	0.98	1.00
	upper q.	0.99	1.14	1.02	1.08	0.97	1.01	1.01

Table 7. Results with density threshold value 0.6 divided by those with value 0.5.

		Array size reqd	Analyse	First Fact.	Analyse + Fact.	Fast Fact.	Solve $\mathbf{Ax}=\mathbf{b}$	Solve $\mathbf{A}^T\mathbf{x}=\mathbf{b}$
CRAY	lower q.	0.89	1.04	1.01	1.03	0.95	1.00	0.93
	median	0.91	1.08	1.03	1.07	0.98	1.00	1.01
	upper q.	0.99	1.18	1.06	1.14	0.99	1.01	1.01
SUN	lower q.	0.88	1.04	0.83	1.01	0.76	0.90	0.91
	median	0.93	1.18	0.87	1.06	0.83	0.94	0.95
	upper q.	0.98	1.36	0.96	1.09	0.89	0.98	0.96
IBM	lower q.	0.88	1.01	0.90	1.00	0.87	0.94	0.97
	median	0.93	1.09	0.96	1.06	0.92	0.96	1.00
	upper q.	0.98	1.31	1.01	1.19	0.94	1.01	1.01

Table 8. Results with density threshold value 0.7 divided by those with value 0.5.

Tables 7 and 8 show the effect of increasing the value of the density threshold for the switch to full code. On the CRAY, the performance is very flat, a credit to its success nowadays in vectorizing loops with indirect addressing. The IBM RS/6000 performance is also rather flat. For the SUN and the IBM, there is a loss of performance for the single problem, but some reduction in factorize time. It is unlikely that there would be such a reduction in factorize time with optimized versions of the BLAS, not currently available to us.

5.2 The use of the BLAS

We have designed our codes so that the full code can use Level 1 BLAS, Level 2 BLAS, or Level 3 BLAS with a choice of block size. Tables 9 and 10 demonstrate that there is some advantage in using Level 3 BLAS on all three platforms.

		First Fact.	Analyse + Fact.	Fast Fact.	Solve $\mathbf{Ax}=\mathbf{b}$	Solve $\mathbf{A}^T\mathbf{x}=\mathbf{b}$
CRAY	lower q.	1.03	1.01	1.09	1.07	1.07
	median	1.13	1.04	1.24	1.18	1.21
	upper q.	1.26	1.06	1.43	1.30	1.40
SUN	lower q.	1.01	1.00	1.06	1.02	1.00
	median	1.19	1.08	1.22	1.04	1.00
	upper q.	1.33	1.15	1.37	1.07	1.01
IBM	lower q.	1.00	0.99	1.08	0.98	0.96
	median	1.22	1.04	1.33	1.09	1.05
	upper q.	1.53	1.09	1.76	1.18	1.07

Table 9. Results with Level 1 BLAS divided by those with Level 3 BLAS and block size 32.

		First Fact.	Analyse + Fact.	Fast Fact.	Solve $\mathbf{Ax}=\mathbf{b}$	Solve $\mathbf{A}^T\mathbf{x}=\mathbf{b}$
CRAY	lower q.	1.00	1.00	0.99	1.00	1.00
	median	1.15	1.04	1.17	1.00	1.00
	upper q.	1.22	1.06	1.33	1.01	1.00
SUN	lower q.	0.99	1.00	1.01	1.00	1.00
	median	1.04	1.02	1.05	1.00	1.00
	upper q.	1.12	1.05	1.13	1.01	1.01
IBM	lower q.	1.08	1.04	1.03	0.98	1.02
	median	1.21	1.07	1.23	1.03	1.06
	upper q.	1.33	1.11	1.41	1.08	1.10

Table 10. Results with Level 2 BLAS divided by those with Level 3 BLAS and block size 32.

		First Fact.	Analyse + Fact.	Fast Fact.	Solve $\mathbf{Ax}=\mathbf{b}$	Solve $\mathbf{A}^T\mathbf{x}=\mathbf{b}$
CRAY	lower q.	1.01	1.01	1.02	1.00	1.00
	median	1.02	1.01	1.03	1.01	1.00
	upper q.	1.04	1.02	1.05	1.01	1.01
SUN	lower q.	0.98	0.99	0.98	1.00	1.00
	median	0.99	1.00	0.99	1.00	1.00
	upper q.	1.00	1.00	1.00	1.01	1.01
IBM	lower q.	1.01	0.98	0.99	0.95	0.98
	median	1.04	1.00	1.03	0.99	1.00
	upper q.	1.08	1.01	1.08	1.04	1.05

Table 11. Results with Level 3 BLAS with block size 16 divided by those with block size 32.

		First Fact.	Analyse + Fact.	Fast Fact.	Solve $\mathbf{Ax}=\mathbf{b}$	Solve $\mathbf{A}^T\mathbf{x}=\mathbf{b}$
CRAY	lower q.	1.00	1.00	1.00	1.00	1.00
	median	1.00	1.00	1.00	1.00	1.00
	upper q.	1.01	1.01	1.02	1.01	1.00
SUN	lower q.	1.00	1.00	1.00	0.99	1.00
	median	1.03	1.01	1.03	1.00	1.00
	upper q.	1.07	1.03	1.08	1.01	1.01
IBM	lower q.	0.98	0.99	0.97	0.94	1.00
	median	1.00	1.00	1.01	1.00	1.04
	upper q.	1.05	1.03	1.06	1.04	1.07

Table 12. Results with Level 3 BLAS with block size 64 divided by those with block size 32.

The performance is very flat as the block size is varied around the size 32. Table 11 shows that with block size 16 we get slightly worse performance on the CRAY and on the IBM and unchanged performance on the SUN. Table 12 shows that with block size 64 we get slightly worse performance on the SUN and IBM and unchanged performance on the CRAY. We have chosen 32 for the default block size because it appears to be near optimal in all three cases and because of the convenience of having the same value on the different machines.

5.3 The strategy for pivot selection

We have followed the recommendation of Zlatev (1980) that the search for pivots be limited to three columns. We have found that, compared with the Markowitz strategy, this does give a worthwhile saving in analyse time without significant loss of sparsity in the factors, as illustrated in Table 13. To check the sensitivity of the choice of number of columns, we have also tried two- and four-column searches. The results in Table 13 show that there is little sensitivity.

		Array size reqd	Analyse	First Fact.	Analyse + Fact.	Fast Fact.	Solve $\mathbf{Ax}=\mathbf{b}$	Solve $\mathbf{A}^T\mathbf{x}=\mathbf{b}$
Markowitz	lower q.	0.93	1.51	0.89	1.38	0.86	0.97	0.96
	median	0.99	2.10	0.97	1.76	0.96	0.99	0.99
	upper q.	1.04	4.18	1.02	3.20	1.02	1.00	1.00
Zlatev 2-col.	lower q.	1.00	0.95	0.99	0.96	0.99	1.00	0.99
	median	1.00	0.97	1.01	0.98	1.01	1.00	1.00
	upper q.	1.03	1.00	1.02	1.01	1.04	1.00	1.00
Zlatev 4-col.	lower q.	0.99	1.00	0.98	1.00	0.99	0.99	1.00
	median	1.00	1.03	1.00	1.02	1.00	1.00	1.00
	upper q.	1.04	1.06	1.01	1.04	1.04	1.01	1.01

Table 13. CRAY results with different pivot strategies divided by those with Zlatev's 3-column search.

An example from circuit simulation provided by Norm Schryer of Bell Laboratories (private communication) illustrates the possibility of very slow Markowitz processing. The problem is of order 138409 and has 434918 entries. The largest block of the block triangular form has order 63554 and dominates the analyse time. Compiling as usual on the SPARCstation 1, but running on a SPARCstation 10 in order to get a feasible run, we found that the analyse time increased from 82 seconds to 8901 seconds when we switched to the Markowitz option. The factorize time was 15 seconds and the solve time was 0.02 seconds.

5.4 The block triangular form

Table 14 shows the statistics produced by `MA48` on the block triangular form, namely

(i) the order of the largest non-triangular block on the diagonal of the block triangular form,

(ii) the sum of the orders of all the non-triangular blocks on the diagonal of the block triangular form, and

(iii) the total number of entries in all the non-triangular blocks on the diagonal of the block triangular form (these are the entries that are passed to `MA50` for analyse).

Case	Order	Order of largest block	Sum of block orders	Number of entries	Number of entries in non-triangular diagonal blocks
1	663	0	0	1712	0
2	541	540	540	4285	3744
3	680	235	235	2646	1434
4	765	697	765	24382	24342
5	817	817	817	6853	6853
6	886	886	886	5970	5970
7	989	720	720	3537	2622
8	991	846	846	6027	5562
9	1107	1107	1107	5664	5664
10	1176	1174	1176	18552	18552
11	1224	1224	1224	9613	9613
12	1224	1224	1224	56126	56126
13	1374	1318	1318	8606	8350
14	822	217	392	4841	1997
15	1856	1728	1728	11360	11104
16	2021	1500	1500	7353	5495
17	2205	2205	2205	14133	14133
18	2529	1830	1830	90158	47823
19	4929	4578	4578	33185	31500
20	5300	5300	5300	21842	21842

Table 14. Statistics on the block triangularization.

Case	Identifier	Order	Number of entries	Description
21	FS 680 1	680	2646	Mixed kinetics diffusion problem from radiation chemistry. 17 chemical species and one space dimension with 40 mesh points.
22	SHL 400	663	1712	Basis matrix from a linear programming problem. (Also case 1.)
23	BP 1600	822	4841	Basis matrix from a linear programming problem. (Also case 14.)
24	IMPCOL D	425	1339	Matrix extracted from a run of the chemical engineering package SPEED UP modelling a nitric acid plant.
25	IMPCOL E	225	1308	Matrix extracted from a run of the chemical engineering package SPEED UP modelling a hydrocarbon separation problem.
26	WEST0497	497	1727	Chemical engineering plant model.
27	WEST2021	2021	7353	Chemical engineering plant model. (Also case 16.)
28	WEST0989	989	3537	Chemical engineering plant model. (Also case 7.)
29	MAHINDAS	1258	7682	Economic model of Victoria, Australia.
30	ORANI678	2529	90158	Economic model of Australasia. (Also case 18.)

Table 15. The matrices used for testing the block triangular form.

For comparison, we also show the matrix order and the number of entries in the matrix. It may be seen that there are only six (cases 1, 3, 7, 14, 16, and 18) where less than 75% of the entries lie in the diagonal blocks. Seven of the matrices (cases 5, 6, 9, 11, 12, 17, and 20) are irreducible, while several more are nearly so. We felt that more than six very reducible cases would be needed to judge the block triangularization, so we chose another set of matrices from the Harwell-Boeing collection, summarized in Table 15. The block triangularization statistics for this collection are shown in Table 16.

Case	Order	Order of largest block	Sum of block orders	Number of entries	Number of entries in non-triangular diagonal blocks
21	680	235	235	2646	1434
22	663	0	0	1712	0
23	822	217	392	4841	1997
24	425	199	199	1339	562
25	225	47	78	1308	403
26	497	92	206	1727	769
27	2021	1500	1500	7353	5495
28	989	720	720	3537	2622
29	1258	589	589	7682	4744
30	2529	1830	1830	90158	47823

Table 16. Statistics on block triangularization for the second collection.

Table 17 shows that worthwhile gains are available from block triangularization, though only in one case is the gain dramatic and this is because the matrix is a permutation of a triangular matrix. There are also some worthwhile storage gains.

Case	Array size reqd	Analyse	First Fact.	Analyse + Fact.	Fast Fact.	Solve $\mathbf{Ax}=\mathbf{b}$	Solve $\mathbf{A}^{\mathrm{T}}\mathbf{x}=\mathbf{b}$
21	1.19	1.13	1.59	1.25	1.33	1.16	0.86
22	1.20	2.18	161.51	3.30	40.96	1.70	1.12
23	1.36	1.28	2.31	1.52	1.51	0.77	0.77
24	1.33	0.94	2.00	1.15	1.68	1.07	0.86
25	1.23	0.91	2.87	1.24	1.65	0.83	0.73
26	1.39	1.24	2.19	1.45	1.70	1.03	1.00
27	1.14	0.93	1.28	1.02	1.23	0.97	0.96
28	1.15	0.95	1.27	1.04	1.22	0.97	0.97
29	1.23	1.08	1.54	1.21	1.33	1.03	0.91
30	1.31	1.20	1.96	1.37	1.68	0.98	0.90
lower q.	1.19	0.94	1.54	1.15	1.33	0.97	0.86
median	1.23	1.10	1.98	1.24	1.58	1.00	0.91
upper q.	1.33	1.24	2.31	1.45	1.68	1.07	0.97

Table 17. CRAY results without block triangularization divided by those with it.

We found that the technique of merging blocks smaller than a threshold size came into operation in only two cases when the threshold value was 10, our tentative default. In both cases, there was a loss of performance, see Table 18. In the absence of further data, we have changed our default value to 1. The runs reported in Sections 5.1, 5.2, and 5.3 used the value 10. This is not likely to affect the conclusions about other aspects of the code, particularly since only one case is affected (case 14, which is also case 23).

	Case	Array size reqd	Analyse	First Fact.	Analyse + Fact.	Fast Fact.	Solve $\mathbf{Ax}=\mathbf{b}$	Solve $\mathbf{A}^{\mathrm{T}}\mathbf{x}=\mathbf{b}$
CRAY	23	1.00	1.08	1.19	1.10	1.16	1.00	0.98
	25	1.00	1.14	1.47	1.19	1.21	1.08	1.03
IBM	23	1.00	1.14	1.16	1.15	1.18	0.96	1.01
	25	1.00	1.08	1.33	1.14	1.09	1.16	1.06

Table 18. Results with amalgamation threshold 10 divided by those without block amalgamation.

5.5 Comparison with calling MA50 directly

A user with a matrix that is irreducible or only slightly reducible may wish to consider calling `MA50` directly, provided the less convenient interface is acceptable and the additional facilities of iterative refinement and error estimation are not required. Comparisons are shown in Table 19. The cases that we noted as being significantly reducible (cases 1, 3, 7, 14, 16, 18) constitute the upper quartile of factorize ratios. Overall, judging by the median ratios, the direct use of `MA50` required less storage and reduced the analyse time, but increased both factorize times.

Case	Array size reqd	Analyse	First Fact.	Analyse + Fact.	Fast Fact.	Solve $\mathbf{Ax}=\mathbf{b}$	Solve $\mathbf{A}^{\mathrm{T}}\mathbf{x}=\mathbf{b}$
1	0.70	1.72	164.21	2.85	40.79	1.67	1.10
2	0.79	0.92	1.01	0.95	0.92	0.99	0.99
3	0.94	0.97	1.57	1.14	1.44	1.15	0.84
4	0.92	0.72	1.10	0.82	1.19	0.91	0.95
5	1.01	0.97	1.16	1.03	1.24	1.03	1.01
6	0.81	0.68	0.87	0.73	0.81	0.95	0.93
7	0.76	0.81	1.27	0.94	1.21	0.96	0.96
8	1.14	0.67	1.24	0.83	1.24	1.09	1.02
9	0.91	0.96	1.06	0.99	1.04	0.98	0.98
10	0.63	0.54	1.00	0.69	1.02	0.98	0.98
11	0.76	0.82	1.01	0.87	0.93	1.00	1.00
12	0.75	0.72	1.00	0.80	0.98	0.98	1.01
13	0.87	0.79	0.95	0.84	0.97	0.89	0.92
14	0.86	1.10	2.32	1.38	1.50	0.77	0.76
15	0.98	0.79	1.04	0.86	1.06	0.99	0.97
16	0.76	0.80	1.28	0.93	1.21	0.96	0.95
17	0.67	0.74	0.73	0.74	0.66	0.92	0.73
18	0.81	1.04	1.94	1.24	1.70	0.98	0.90
19	0.65	0.72	1.03	0.81	0.99	0.87	0.87
20	0.81	0.82	1.03	0.88	1.04	1.00	0.99
lower q.	0.75	0.72	1.01	0.82	0.98	0.93	0.91
median	0.81	0.80	1.05	0.88	1.05	0.98	0.96
upper q.	0.92	0.97	1.28	1.01	1.24	1.00	1.00

Table 19. CRAY results with `MA50` divided by those with `MA48`.

5.6 Iterative refinement and error estimation

The possible improvement in solution accuracy through using iterative refinement is well documented. Arioli *et al.* (1989) discuss this and give extensive numerical experiments to illustrate the effectiveness of the backward and forward error estimation discussed in Section 4.3.2. We do not repeat these here but rather examine the overhead in execution time caused by invoking the various options within the solve phase of the code.

Table 20 summarizes the performance when the following options for the solve subroutine MA48C are invoked:

(i) Calculate the solution without iterative refinement but with the calculation of the relative backward errors.

(ii) Calculate the solution with iterative refinement and calculation of the relative backward errors.

(iii) Calculate the solution with iterative refinement, calculation of the relative backward errors and estimation of the ∞–norm of the relative error in the solution.

	Relative backward errors		Also iterative refinement		Also relative error in solution	
	$\mathbf{Ax=b}$	$\mathbf{A^Tx=b}$	$\mathbf{Ax=b}$	$\mathbf{A^Tx=b}$	$\mathbf{Ax=b}$	$\mathbf{A^Tx=b}$
lower q.	1.77	2.18	3.74	4.69	10.93	8.71
median	1.88	2.39	5.32	5.21	12.08	9.13
upper q.	1.98	2.53	5.83	6.68	13.03	10.53

Table 20. CRAY results with iterative refinement and error estimation divided by those without.

At first glance, the increase in time for the solve options seems rather high. However, the overall times are still far smaller than factorize or analyse times and there can be substantially more work because of these options. The amount of extra work will depend on the ratio of the number of entries in the factors to the number in the original matrix since the latter corresponds to the work in calculating a residual. Indeed, to obtain the backward error estimate, three "residuals" are calculated which is clearly itself as expensive as a solution if the factors are three times denser than the original. Option (i) involves one of these "residual" calculations so it could well be two to three times the cost of the straightforward solution, as we see in Table 20. The higher ratio for the transpose option is caused by the use of a dot product rather than a vector addition in the residual calculation loops. Option (ii) will depend on the number of iterations of iterative refinement. This is usually very low (around 2) but there are several simple loops of length N in addition to the extra solutions and "residual" calculations. Thus a factor of around 5 over straight solution is quite expected. The increase to a factor of around 10 for option (iii) can also be predicted since there are usually about six solutions required to calculate the appropriate matrix norms.

Our advice is to use JOB = 1 if some other means is available for checking the solution and JOB = 2 if not. Only in cases when the user is anxious about the accuracy of the solution need JOB = 3 or 4 be required.

5.7 Comparison with MA28

Since the Harwell Subroutine Library code MA28 is a benchmark standard in the solution of sparse unsymmetric equations and because we plan that it will be replaced by MA48, we compare MA48 with MA28 on the three computing environments. MA28 always produces a factorization when it performs an analysis and its only form of factorization is without pivoting. The MA28 analyse time is therefore strictly comparable with the sum of the analyse and factorize times of MA48, and this comparison is shown in column "Analyse + Fact.".

However, analyse alone or factorize with pivoting may also be needed by the user, so we also use the MA28 analyse time to compare separately with the analyse (column "Analyse") and first factorize (column "First Fact.") times of MA48.

Case	Array size reqd	Analyse	First Fact.	Analyse + Fact.	Fast Fact.	Solve
1	0.50	1.38	196.73	1.37	154.50	3.90
2	0.69	2.64	6.36	1.87	2.04	3.14
3	0.71	2.35	6.20	1.70	3.43	3.22
4	0.62	6.15	16.67	4.49	2.02	1.42
5	0.66	3.92	9.29	2.76	1.81	2.27
6	0.68	4.30	11.81	3.15	1.82	2.18
7	0.54	3.64	9.46	2.63	3.06	2.33
8	0.91	16.51	42.68	11.90	3.54	2.13
9	0.65	7.81	17.50	5.40	1.64	1.74
10	0.42	3.98	8.13	2.67	3.29	2.48
11	0.68	8.68	21.52	6.18	1.82	2.32
12	0.63	4.64	12.40	3.38	2.23	1.59
13	0.98	21.39	49.73	14.95	2.49	1.84
14	0.59	1.65	5.49	1.27	3.81	1.93
15	1.53	29.64	77.33	21.43	2.74	1.96
16	0.54	6.62	17.61	4.81	3.03	2.35
17	0.51	5.40	10.68	3.59	0.96	1.44
18	0.58	6.22	21.31	4.81	5.53	1.75
19	0.58	1.84	4.63	1.31	3.38	2.22
20	0.60	3.10	7.55	2.20	2.03	2.36
lower q.	0.50	2.87	7.84	2.03	1.92	1.79
median	0.69	4.47	12.10	3.26	2.61	2.20
upper q.	0.71	7.21	21.41	5.11	3.41	2.35

Table 21. CRAY results with MA28 divided by those with MA48.

Case	Array size reqd	Analyse	First Fact.	Analyse + Fact.	Fast Fact.	Solve
1	0.50	2.06	21.67	1.88	18.33	1.48
2	0.68	2.14	3.77	1.37	1.59	0.80
3	0.74	2.31	5.17	1.60	2.22	1.91
4	0.62	3.06	4.40	1.80	1.31	0.67
5	0.63	3.29	7.16	2.25	1.27	0.73
6	0.63	2.96	2.61	1.39	1.07	0.70
7	0.54	3.66	8.30	2.54	2.00	1.15
8	0.93	21.82	9.95	6.84	2.60	1.04
9	0.65	5.99	5.19	2.78	1.06	0.66
10	0.42	3.37	3.48	1.71	1.74	0.75
11	0.66	7.74	9.46	4.26	1.27	0.71
12	0.62	1.90	3.66	1.25	1.67	0.64
13	1.00	22.81	27.44	12.46	2.20	0.95
14	0.59	1.68	4.36	1.22	2.81	0.94
15	1.51	21.71	32.01	12.94	6.63	1.26
16	0.54	6.79	15.68	4.74	1.84	1.03
17	0.53	2.26	1.61	0.94	0.65	0.52
18	0.58	3.33	4.96	1.99	2.20	0.70
19	0.58	1.90	3.99	1.29	2.02	1.01
20	0.59	3.04	5.01	1.89	1.07	0.86
lower q.	0.50	2.20	3.88	1.38	1.27	0.70
median	0.68	3.17	5.09	1.89	1.79	0.83
upper q.	0.74	6.39	9.71	3.52	2.21	1.03

Table 22. SUN results with MA28 divided by those with MA48.

Case	Array size reqd	Analyse	First Fact.	Analyse + Fact.	Fast Fact.	Solve
1	0.50	1.58	35.71	1.52	61.22	2.74
2	0.68	2.65	8.04	2.00	1.86	1.11
3	0.74	2.50	7.98	1.90	4.16	1.40
4	0.62	3.72	14.61	2.96	2.55	1.21
5	0.63	3.62	12.38	2.80	2.12	0.85
6	0.63	4.14	10.34	2.95	1.94	1.23
7	0.54	3.54	11.00	2.68	2.44	1.17
8	0.93	24.59	44.45	15.83	5.77	1.94
9	0.65	7.45	15.38	5.02	2.18	1.02
10	0.42	3.71	7.30	2.46	2.67	1.16
11	0.66	8.12	27.06	6.24	2.33	1.32
12	0.62	2.69	10.84	2.16	2.82	1.20
13	1.01	19.56	62.02	14.87	4.18	1.62
14	0.59	1.33	4.12	1.01	1.68	0.99
15	1.51	26.07	78.97	19.60	9.21	2.14
16	0.54	6.69	21.61	5.11	3.50	1.14
17	0.53	2.90	7.33	2.08	1.37	0.89
18	0.58	3.08	16.46	2.60	7.23	1.09
19	0.58	2.00	6.12	1.51	3.30	1.04
20	0.59	2.92	8.01	2.14	1.81	0.85
lower q.	0.50	2.67	7.99	2.04	2.03	1.03
median	0.68	3.58	11.69	2.64	2.61	1.16
upper q.	0.74	7.07	24.33	5.06	4.17	1.36

Table 23. IBM results with `MA28` divided by those with `MA48`.

6 Acknowledgements

We wish to thank our colleagues Nick Gould and Jennifer Scott, John Gilbert of Xerox, the Associate Editor, Michael Saunders, and the two non-anonymous referees, John Lewis and Zahari Zlatev, for their many helpful suggestions for improving the presentation.

Appendix. Solving full sets of linear equations

For the full-matrix processing we use towards the end of the factorization, we need to consider the solution of dense systems

$$\mathbf{A}\mathbf{x} = \mathbf{b}, \tag{A.1}$$

where **A** is of order *m* by *n*. The mathematical notation used in this appendix is independent of that of the main part of the paper. The matrix **A** here is the matrix **F** of equation (3.3.4) and the vector **b** is the vector $\mathbf{y}_2$. We feel that it is easier to understand the ideas using uncluttered 'local' notation. For consistency with LAPACK (Anderson *et al.*, 1992), we work here with a factorization in which the lower-triangular matrix has a unit diagonal.

We had hoped to use the LAPACK routines `SGETRF` and `SGETRS` for this purpose, but their treatment of the rank-deficient case is unsatisfactory since no column interchanges are included.

Our factorization algorithm proceeds as follows. We first describe the case where the pivot threshold is zero. At a typical stage, we look in the next column for a pivot. We either find one and perform the pivotal operations or interchange the column with the last column that has not already been interchanged. At the start of step k, we have the factorization

$$\mathbf{P}_k \mathbf{A} \mathbf{Q}_k = \begin{pmatrix} \mathbf{L}_k & \\ \mathbf{M}_k & \mathbf{I} \end{pmatrix} \begin{pmatrix} \mathbf{U}_k & \mathbf{V}_k & \mathbf{W}_k \\ & \mathbf{S}_k & \mathbf{0} \end{pmatrix}, \tag{A.2}$$

where $\mathbf{P}_k$ and $\mathbf{Q}_k$ are permutation matrices, $\mathbf{L}_k$ is unit lower triangular and of order $j_k - 1$, $\mathbf{U}_k$ is upper triangular of order $j_k - 1$, and $\mathbf{W}_k$ has $k - j_k$ columns. Initially, $k = 1$; $j_1 = 1$; $\mathbf{P}_1$ and $\mathbf{Q}_1$ are identity matrices; $\mathbf{L}_1$, $\mathbf{M}_1$, and $\mathbf{W}_1$ have no columns; $\mathbf{U}_1$ and $\mathbf{V}_1$ have no rows; and $\mathbf{S}_1 = \mathbf{A}$. We find the largest entry of the first column of $\mathbf{S}_k$. If this is nonzero, we interchange rows to make it the leading entry of $\mathbf{S}_k$ and perform the pivotal operations; otherwise, we interchange the first and last columns of $\mathbf{S}_k$. In the former case, j_{k+1} has the value $j_k + 1$; in the latter case, j_{k+1} has the value j_k. The row interchange is also performed in $\mathbf{M}_k$ and the column interchange is also performed in $\mathbf{V}_k$. The final factorization is

$$\mathbf{P} \mathbf{A} \mathbf{Q} = \begin{pmatrix} \mathbf{L} & \\ \mathbf{M} & \mathbf{I} \end{pmatrix} \begin{pmatrix} \mathbf{U} & \mathbf{W} \\ & \mathbf{0} \end{pmatrix}. \tag{A.3}$$

Solving $\mathbf{A}\mathbf{x} = \mathbf{b}$ consists of the steps

$$\mathbf{c} = \mathbf{P}\mathbf{b} \tag{A.4}$$

$$\begin{pmatrix} \mathbf{L} & \\ \mathbf{M} & \mathbf{I} \end{pmatrix} \begin{pmatrix} \mathbf{d}_1 \\ \mathbf{d}_2 \end{pmatrix} = \begin{pmatrix} \mathbf{c}_1 \\ \mathbf{c}_2 \end{pmatrix} \tag{A.5}$$

$$\begin{pmatrix} \mathbf{U} & \mathbf{W} \\ & \mathbf{0} \end{pmatrix} \begin{pmatrix} \mathbf{e}_1 \\ \mathbf{e}_2 \end{pmatrix} = \begin{pmatrix} \mathbf{d}_1 \\ \mathbf{d}_2 \end{pmatrix} \tag{A.6}$$

$$\mathbf{x} = \mathbf{Q}\mathbf{e}. \tag{A.7}$$

We solve (A.6) by setting $\mathbf{e}_2 = \mathbf{0}$ and finding $\mathbf{e}_1$ by back-substitution through $\mathbf{U}$. This means that $\mathbf{d}_2$ is not required so that in (A.5) we need only forward substitute through $\mathbf{L}$ to find $\mathbf{d}_1$.

Similarly, solving $\mathbf{A}^{\mathrm{T}}\mathbf{x} = \mathbf{b}$ consists of the steps

$$\mathbf{c} = \mathbf{Q}^{\mathrm{T}}\mathbf{b} \tag{A.8}$$

$$\begin{pmatrix} \mathbf{U}^{\mathrm{T}} & \\ \mathbf{W}^{\mathrm{T}} & \mathbf{0} \end{pmatrix} \begin{pmatrix} \mathbf{d}_1 \\ \mathbf{d}_2 \end{pmatrix} = \begin{pmatrix} \mathbf{c}_1 \\ \mathbf{c}_2 \end{pmatrix} \tag{A.9}$$

$$\begin{pmatrix} \mathbf{L}^{\mathrm{T}} & \mathbf{M}^{\mathrm{T}} \\ & \mathbf{I} \end{pmatrix} \begin{pmatrix} \mathbf{e}_1 \\ \mathbf{e}_2 \end{pmatrix} = \begin{pmatrix} \mathbf{d}_1 \\ \mathbf{d}_2 \end{pmatrix} \tag{A.10}$$

$$\mathbf{x} = \mathbf{P}^{\mathrm{T}}\mathbf{e}. \tag{A.11}$$

Here, $\mathbf{d}_1$ is calculated by forward substitution through $\mathbf{U}^{\mathrm{T}}$ and $\mathbf{d}_2$ is set to zero. In turn, this means that $\mathbf{e}_2$ is zero and $\mathbf{e}_1$ is calculated by back-substitution through $\mathbf{L}^{\mathrm{T}}$.

Note that neither $\mathbf{M}$ nor $\mathbf{W}$ is used in either case.

There is no real loss of generality in setting the undetermined coefficients of the solution to zero. If other values are required, say those of the vector $\mathbf{y}$, we may solve the equation

$$\mathbf{A}\mathbf{x} = \mathbf{b} - \mathbf{A}\mathbf{y} \text{ or , } \mathbf{A}^{\mathrm{T}}\mathbf{x} = \mathbf{b} - \mathbf{A}^{\mathrm{T}}\mathbf{y}$$

to yield a solution $\mathbf{x} + \mathbf{y}$ with the desired components.

Another reason for rejecting SGETRF is that it tests only for exact zeros. We test for exact

zeros by default, but wish to offer the option of a test against a threshold. When this option is active, if the largest entry of the first column of $\mathbf{S}_k$ is below the threshold, we set the nonzero entries of this column to zero. The final factorization will be as if we had commenced with a matrix whose entries differ from those of $\mathbf{A}$ by at most the threshold.

A.1 Factorization using BLAS at Level 1, 2, or 3

MA50E performs the factorization using Level 1 BLAS. Step k begins with the form shown in equation (A.2) with each submatrix overwriting the corresponding submatrix of $\mathbf{A}$ in the obvious way except that $\mathbf{V}_k$ and $\mathbf{S}_k$ have not been calculated. The first column of $\begin{pmatrix}\mathbf{V}_k\\ \mathbf{S}_k\end{pmatrix}$ is calculated from the corresponding column of $\mathbf{A}$ by j_k-1 calls of the BLAS routine SAXPY, each of which adds a multiple of a column of $\begin{pmatrix}\mathbf{L}_k\\ \mathbf{M}_k\end{pmatrix}$. If the largest entry of the first column of $\mathbf{S}_k$ is greater than the pivot threshold ε, it is chosen as pivot, j_{k+1} is set equal to j_k+1 and the last column of $\begin{pmatrix}\mathbf{L}_{k+1}\\ \mathbf{M}_{k+1}\end{pmatrix}$ is constructed. Otherwise, j_{k+1} is set equal to j_k and the first column $\mathbf{W}_{k+1}$ is constructed.

MA50F performs the factorization using Level 2 BLAS and is based on the LAPACK subroutine SGETF2. Step k begins with the form shown in equation (A.2), but now only the submatrix $\mathbf{S}_k$ has not been calculated. The first column of $\mathbf{S}_k$ is calculated with a call of the Level 2 BLAS routine SGEMV to multiply $\mathbf{M}_k$ by the first column of $\mathbf{V}_k$. If a pivot is found, the Level 2 BLAS SGEMV is used to form the last row of $\mathbf{V}_{k+1}$. It does this by subtracting the first row of $\mathbf{M}_k$ times $\mathbf{V}_k$, excluding its first column, from the first row of $\mathbf{S}_k$, excluding its first entry. A simple calculation shows that the number of operations performed within a step outside the Level 2 BLAS is $O(m+n)$.

MA50G performs the factorization using Level 3 BLAS and is based on the LAPACK subroutine SGETRF. The matrix is processed in blocks of NB columns, apart from the final block which may have less columns. The processing of a block begins with the form shown in equation (A.2), now with all submatrices calculated. The leading NB columns of $\mathbf{S}_k$ are treated in just the same way as $\mathbf{A}$ itself is treated by MA50F, except that when no entry of a column is big enough to be a pivot, an interchange is made with the last column of $\mathbf{S}_k$ rather than the last column of the block. At the completion of the block (that is, when NB pivots have been found), the row interchanges generated within it are applied to the other columns of $\mathbf{M}_k$ and $\mathbf{S}_k$ (column by column to avoid data movement), and the operations of the block are applied to the remaining columns of $\mathbf{V}_k$ and $\mathbf{S}_k$ using the Level 3 BLAS STRSM and SGEMM. A simple calculation shows that the total number of operations performed outside the Level 3 BLAS is $O(\,n(\mathtt{NB}\,m+n)\,)$. Note that the column interchange that follows a failure to find a pivot does not usually lead to a reduction of the block size, since a column from outside the block is brought in. However, this is not the case for the final block, where the block size is reduced by one for each such column interchange.

A.2 Solution using BLAS at Level 1 or 2

MA50H solves a set of equations $\mathbf{A}\,\mathbf{x}=\mathbf{b}$ or $\mathbf{A}^{\mathrm{T}}\mathbf{x}=\mathbf{b}$ using the factorization produced by MA50E, MA50F, or MA50G, whose output data are identical. Each actual forward or back-substitution operation associated with $\mathbf{L}$ or $\mathbf{U}$ is performed either with the Level 2 BLAS STRSV or by a loop involving calls to SAXPY or SDOT. An argument controls which of these

happens. Unlike the case for factorization, the logic is very similar for the two cases, so there is no need for separate subroutines.

MA50H begins by finding the calculated rank r by searching IPIV from the back for a positive value. We expect this search to be short on the assumption that a rank much less than min(m,n) is unusual.

If $\mathbf{A}\,\mathbf{x}=\mathbf{b}$ is to be solved, we first apply r interchanges to the incoming vector to produce the vector $\mathbf{c}$ of equation (A.4). The row operations encoded in $\mathbf{L}$ are applied to calculate $\mathbf{d}_1$ from $\mathbf{c}_1$, see equation (A.5). Back-substitution through $\mathbf{U}$ is used to calculate $\mathbf{e}_1$ and $\mathbf{e}_2$ is set to zero, see equation (A.6). Finally, the column interchanges, if any, are applied, see equation (A.7).

A similar sequence of steps is applied when $\mathbf{A}^{\mathrm{T}}\,\mathbf{x}=\mathbf{b}$ is to be solved.

For the Level 1 BLAS code, we have followed the lead of LAPACK in accessing $\mathbf{L}$ and $\mathbf{U}$ by columns with SAXPY inner loops such as

```
b(1:k-1) = b(1:k-1) – A(1:k-1,k)*b(k)
```

when solving $\mathbf{A}\,\mathbf{x}=\mathbf{b}$, and accessing $\mathbf{L}^{\mathrm{T}}$ and $\mathbf{U}^{\mathrm{T}}$ by rows with SDOT inner loops such as

```
DOT_PRODUCT(A(1:k-1,k),b(1:k-1))
```

in order to access contiguous elements of the array A.

References

Anderson, E., Bai, Z., Bischof, C., Demmel, J., Dongarra, J., Du Croz, J., Greenbaum, A., Hammarling, S., McKenney, A., Ostrouchov, S., and Sorensen, D. (1992). LAPACK users' guide. SIAM, Philadelphia.

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

Arioli, M. Demmel, J. W., and Duff, I. S. (1989). Solving sparse linear systems with sparse backward error. *SIAM J. Matrix Anal. Appl.* **10**, 165-190.

Dongarra, J. J., Du Croz, J., Duff, I. S., and Hammarling, S. (1990). A set of Level 3 Basic Linear Algebra Subprograms. *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. (1977). MA28 – a set of Fortran subroutines for sparse unsymmetric linear equations. Report AERE R8730, HMSO, London.

Duff, I. S. (1981a). On algorithms for obtaining a maximum transversal. *ACM Trans. Math. Softw.* **7**, 315-330.

Duff, I. S. (1981b). Algorithm 575. Permutations for a zero-free diagonal. *ACM Trans. Math. Softw.* **7**, 387-390.

Duff, I. S. (1984). The solution of nearly symmetric sparse linear systems. *Computing methods in applied sciences and engineering, VI.* Edited by R. Glowinski and J.–L. Lions. North-Holland, Amsterdam, New York, and London, 57-74.

Duff, I. S. and Reid, J. K. (1978a). An implementation of Tarjan's algorithm for the block triangularization of a matrix. *ACM Trans. Math. Softw.* **4**, 137-147.

Duff, I. S. and Reid, J. K. (1978b). Algorithm 529. Permutations to block triangular form. *ACM Trans. Math. Softw.* **4**, 189-192.

Duff, I. S. and Reid, J. K. (1979). Some design features of a sparse matrix code. *ACM Trans. Math. Softw.* **5**, 18-35.

Duff, I. S. and Reid, J. K. (1993). MA48, a Fortran code for direct solution of sparse unsymmetric linear systems of equations. Report RAL-93-072, Rutherford Appleton Laboratory, Oxfordshire.

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

Duff, I. S., Grimes, R. G., and Lewis, J. G. (1989). Sparse matrix test problems. *ACM Trans. Math. Softw.* **15** 1-14.

Duff, I. S., Grimes, R. G., and Lewis, J. G. (1992). Users' guide for the Harwell-Boeing sparse matrix collection (Release 1). Report RAL-92-086, Rutherford Appleton Laboratory, Oxfordshire.

Eisenstat, S. C. and Liu, J. W. H. (1993). Exploiting structural symmetry in a sparse partial pivoting code. *SIAM J. Sci. Stat. Comput.* **14**, 253-257.

Gilbert, J. R. and Peierls, T. (1988). Sparse partial pivoting in time proportional to arithmetic operations. *SIAM J. Sci. Stat. Comput.* **9**, 862-874.

Hager, W. W. (1984). Condition estimators, *SIAM J. Sci. Stat. Comput.* **5**, 311-316

Higham, N. J. (1988). Fortran codes for estimating the one-norm of a real or complex matrix, with applications to condition estimation. *ACM Trans. Math. Softw.* **14**, 381-396.

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.

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

Oettli, W. and Prager, W. (1964). Compatibility of approximate solution of linear equations with given error bounds for coefficients and right-hand sides. *Numerische Math.* **6**, 405-409.

Pothen, A. and Fan, C-J. (1990). Computing the block triangular form of a sparse matrix. *ACM Trans. Math. Softw.* **16**, 303-324.

Skeel, R. D. (1980). Iterative refinement implies numerical stability for Gaussian elimination. *Math. Comp.* **35**, 817-832.

Tarjan, R. E. (1972). Depth-first search and linear graph algorithms. *SIAM J. Computing* **1**, 146-160.

Zlatev, Z. (1980). On some pivotal strategies in Gaussian elimination by sparse technique. *SIAM J. Numer. Anal.* **17**, 18-30.

Zlatev, Z. (1991). Computational methods for general sparse matrices. Kluwer Academic Publishers, Dordrecht, Boston, and London.