RAL-TR-2002-024

MA57 -A new code for the solution of sparse symmetric definite and indefinite systems[1]

Iain S. Duff[2]

ABSTRACT

We introduce a new code for the direct solution of sparse symmetric linear equations that solves indefinite systems with 2×2 pivoting for stability. This code, called MA57, is in HSL 2002 and supersedes the well used HSL code MA27. We describe the user interface in some detail and emphasize some of the novel features of MA57. These include restart facilities, matrix modification, partial solution for matrix factors, solution of multiple right-hand sides, and iterative refinement and error analysis. There are additional facilities within a Fortran 90 implementation that include the ability to identify and change pivots. Several of these facilities have been developed particularly to support optimization applications and the performance of the code on problems arising therefrom will be presented.

Keywords: sparse indefinite systems, augmented systems, direct sparse factorization, multifrontal method, numerical optimization.

AMS(MOS) subject classifications: 65F05, 65F50.

---

[1]Current reports available by anonymous ftp to `ftp.numerical.rl.ac.uk` in directory pub/reports. This report is in file duRAL2002024.ps.gz. The report also available through the URL `www.numerical.rl.ac.uk/reports/reports.html`.
[2]i.s.duff@rl.ac.uk. This work was supported by the EPSRC Grant GR/R46441.

Computational Science and Engineering Department
Atlas Centre
Rutherford Appleton Laboratory
Oxon OX11 0QX

September 26, 2002

Contents

1 Introduction

In this report, we discuss the design and use of a code for the solution of the linear systems of equations

$$\mathbf{AX} = \mathbf{B} \tag{1.1}$$

where the $n \times n$ matrix $\mathbf{A}$ is large, sparse, and symmetric but need not be definite. $\mathbf{B}$ is an $n \times nrhs$ ($nrhs \geq 1$) matrix of right-hand sides and $\mathbf{X}$ is the $n \times nrhs$ solution matrix. Our algorithm for the solution of (1.1) uses an $\mathbf{LDL^T}$ factorization of the matrix implemented using a multifrontal approach. We very briefly discuss multifrontal methods in Section 2 and the $\mathbf{LDL^T}$ factorization in Section 3.

The resulting code is called `MA57` (Fortran 77 version) or `HSL_MA57` (Fortran 90 version) in HSL (2002) and supersedes the earlier code `MA27` (Duff and Reid 1982, Duff and Reid 1983). There were several reasons for deciding to embark on this enterprise. `MA27` was one of the most popular codes in HSL and, in particular, was heavily used by people working in optimization. It was, however, written in the early 1980's and did not use the BLAS (Basic Linear Algebra Subprograms), in particular the Level 2 and Level 3 BLAS (Dongarra, Du Croz, Hammarling and Hanson 1988, Dongarra, Du Croz, Duff and Hammarling 1990). Recent experience has shown that these building blocks lead to efficient and portable code on a wide range of modern computers. In addition, there were several features that were being frequently requested by some of the major users of `MA27`. We list the new features that we have added to address these concerns in Section 4. An overview of our new code is presented in Section 5 with the structure of the code given in Section 5.4.

We discuss the performance of our code in Section 6. There are many user controllable parameters that can affect performance including numerical pivoting controls and blocking for the Level 2 and Level 3 BLAS, and we discuss their effect and sensitivity in Sections 6.2 to 6.4 before we compare the `MA57` code with other HSL codes.

Finally, we have been fortunate in having several experts in optimization who were willing to test our code within their software and include comments related to this experience in Section 7. Concluding comments are presented in Section 8.

2 Frontal and multifrontal methods

Our algorithm for the solution of (1.1) uses a multifrontal approach. We give only a brief outline of the multifrontal method here and refer the reader to our earlier papers (Duff and Reid 1982, Duff and Reid 1983) for further details. A more didactic presentation of the multifrontal method can be found in the book by Duff, Erisman and Reid (1986).

Before we describe the essential features of a multifrontal approach, we first discuss a frontal scheme. These have their origins in the solution of finite-element problems (Irons 1970) and we introduce them in this context, although they are applicable also when the matrix is assembled.

If we thus assume that $\mathbf{A}$ is of the form

$$\mathbf{A} = \sum_{l=1}^{m} \mathbf{A}^{[l]}$$

where each element matrix $\mathbf{A}^{[l]}$ has nonzeros in only a few rows and columns and is normally held as a small dense matrix representing contributions to $\mathbf{A}$ from element l. If a_{ij} and $a_{ij}^{[l]}$ denote the (i,j)th entry of $\mathbf{A}$ and $\mathbf{A}^{[l]}$, respectively, the basic assembly operation when forming $\mathbf{A}$ is of the form

$$a_{ij} \Leftarrow a_{ij} + a_{ij}^{[l]}, \tag{2.1}$$

and it is evident that the basic operation in Gaussian elimination

$$a_{ij} \Leftarrow a_{ij} - a_{ik}[a_{kk}]^{-1}a_{kj}, \tag{2.2}$$

with the product $a_{ik}[a_{kk}]^{-1}$ called the *multiplier*, may be performed as soon as all the terms of the triple product in (2.2) are *fully summed* (that is, will be involved in no more sums of the form (2.1)). The assembly and Gaussian elimination processes can therefore be interleaved and the matrix $\mathbf{A}$ is never assembled explicitly. Variables that are internal to a single element can be immediately eliminated (called *static condensation*) and this can be extended to a submatrix that is a sum of several element matrices. In this scheme, all intermediate working can be performed within a dense matrix, termed the *frontal matrix*, whose rows and columns correspond to variables that have not yet been eliminated but occur in at least one of the elements that have been assembled. We can partition the frontal matrix, $\mathbf{F}$, as:

$$\mathbf{F} = \begin{bmatrix} \mathbf{F_{11}} & \mathbf{F_{12}} \\ \mathbf{F_{12}^T} & \mathbf{F_{22}} \end{bmatrix} \tag{2.3}$$

where the fully summed variables correspond to the rows and columns of the block $\mathbf{F_{11}}$. Pivots can be chosen from $\mathbf{F_{11}}$ to perform the factorization $\mathbf{F_{11}} = \mathbf{L_1 D_1 L_1^T}$, where $\mathbf{L_1}$ is a triangular matrix and $\mathbf{D_1}$ is block diagonal with blocks of order 1 or 2 (this factorization is discussed in Section 3). The kernel computation in a frontal scheme then continues as

$$\mathbf{F'_{22}} \leftarrow \mathbf{F_{22}} - \mathbf{F_{12}^T L_1^{T^{-1}} D_1^{-1} L_1^{-1} F_{12}}, \tag{2.4}$$

and can be performed using Level 3 BLAS.

The frontal method can be easily extended to non-element problems because any set of rows of a sparse matrix can be held in a rectangular array whose number of columns is equal to the number of columns with nonzero entries in the selected rows. A variable is regarded as fully summed whenever the equation in which it last appears is assembled. These frontal matrices can often be quite sparse but are suitable for computations involving Level 3 dense BLAS. A full discussion of the equation input can be found in Duff (1984).

The computations involved in the frontal scheme can be represented by a computational tree where the nodes correspond to computations of the form (2.4) and the edges represent the transfer of the $\mathbf{F}'_{22}$ matrix to the next step of the elimination. Of course, for the frontal method as just described, this computational tree will be a chain.

If the frontal scheme is combined with an ordering to preserve sparsity and reduce the number of floating-point operations and if a new frontal matrix can be formed independently of already existing frontal matrices, we can develop a scheme that combines the benefits of using Level 3 BLAS with the gains from using sparsity orderings. This is developed in *multifrontal schemes* where, as before, the computation can be viewed as a tree, whose nodes represent computations of the form (2.4), but now most of the nodes have more than one child. The edges represent the transfer of data from a child to its parent node. This data transfer corresponds to the Schur complement matrix $\mathbf{F}'_{22}$ being sent to the parent node, where it is summed with contributions from the original matrix and the other children to form another dense frontal matrix on which similar operations are performed. If the matrix is irreducible, this tree will have only one root. We use the term *assembly tree* to describe this structure.

We note that, at each node of the tree, a block of the factors will be generated. This will correspond to the matrices $\mathbf{L_1}$ and $\mathbf{D_1}$ and the matrix $\mathbf{F_{12}}$ or, if the multipliers are held, the matrix $\mathbf{D_1^{-1}L_1^{-1}F_{12}}$.

Before we discuss the finer details of our implementation it is useful to first define the steps in our solution scheme. These are:

1. Analysis. Generate a pivot ordering and an assembly tree.
2. Factorization. Use the assembly tree to guide the numerical factorization, performing numerical pivoting at this stage.
3. Solve. Use the assembly tree and the factors just generated to solve the linear system(s) by forward and backward substitution.

3 Factorization of sparse indefinite matrices

The analysis step of our multifrontal method proceeds on the assumption that all diagonal entries are acceptable as pivots, which will be the case if the matrix is definite. However, in the case of symmetric indefinite systems this will no longer be the case as the simple example

$$\begin{pmatrix} 0 & \times \\ \times & 0 \end{pmatrix}$$

testifies. Thus some additional form of numerical pivoting is required.

The numerical pivoting takes place entirely within the frontal matrix and uses a modified version of the algorithm by Bunch, Kaufman and Parlett (1976) to effect a stable decomposition when the matrix is indefinite. The factorization in this case is of the form

$$\mathbf{PAP^T} = \mathbf{LDL^T} \tag{3.1}$$

where $\mathbf{P}$ is a permutation matrix, the matrix $\mathbf{D}$ is block diagonal with blocks of order 1 and 2, and $\mathbf{L}$ is block lower triangular, with blocking compatible to that of $\mathbf{D}$. The 1×1 blocks correspond to single pivots, a_{kk}, that satisfy the threshold test

$$|a_{kk}| \geq u \ . \ \max |a_{kj}| \, , \ j = k+1, n. \tag{3.2}$$

Each 2×2 block corresponds to a 2×2 pivot that satisfies the threshold test

$$\left| \begin{pmatrix} a_{kk} & a_{k\ k+1} \\ a_{k+1\ k} & a_{k+1\ k+1} \end{pmatrix}^{-1} \right| \begin{pmatrix} \max_{j \geq k+2} \ |a_{kj}| \\ \max_{j \geq k+2} \ |a_{k+1\ j}| \end{pmatrix} \leq \begin{pmatrix} u^{-1} \\ u^{-1} \end{pmatrix}, \tag{3.3}$$

where $0 < u \leq 0.5$ to ensure that a pivot can always be chosen when the matrix is fully summed and the modulus signs around the matrix mean that the matrix entries are replaced by their absolute values. Note that this test is less severe than the test for 2×2 pivots in `MA27`, the precursor to `MA57`, which uses the infinity norm of the potential pivot. Our new test (3.3) means that it is slightly more likely that a 2×2 pivot will be chosen by `MA57` than by `MA27`. Of course, the use of these tests presupposes that matrix entries are of similar magnitude or that some automatic scaling routine like `MC30` (Curtis and Reid 1972) from HSL is used. We discuss scaling further in Sections 6.6 and 7.4.1.

We note that the entire 2×2 pivot must lie within the $\mathbf{F_{11}}$ block of the frontal matrix and it is possible that large entries in the $\mathbf{F_{12}}$ block will prevent some (maybe many) potential pivots from satisfying either (3.2) or (3.3). In this case, the factorization can still proceed but the Schur complement matrix passed to the parent node will be increased in dimension to include the failed candidates for pivot. We say that some pivots have been *delayed*. This will normally increase both storage and work for the factorization above that required if no pivots are delayed. At the root node of the tree, all rows and columns of the frontal matrix are fully summed (that is, there is no (2,2) block in (2.3)) and so, if the matrix is nonsingular, the factorization can complete stably.

4 Novel features of the code

We first give a brief history of some of the sparse symmetric codes within HSL.

The code MA17 was written by Reid (1972) and used a totally dynamic approach implemented using linked lists. Only 1×1 pivots were used so that the factorization could fail on indefinite systems. It would continue if the pivots changed sign or were very small and would only terminate with failure if a zero pivot were encountered.

Duff and Reid (1982) used pivots of order 1 and 2 in their MA27 code that was the first code in a software library to use the multifrontal technique. MA27 was widely used by the numerical optimization community and, for constrained problems, was often employed on the augmented matrix of the form

$$\begin{pmatrix} \mathbf{H} & \mathbf{A} \\ \mathbf{A}^{\mathbf{T}} & \mathbf{0} \end{pmatrix} \tag{4.1}$$

so, in 1993, Duff and Reid developed another multifrontal code MA47 (Duff and Reid 1995) that used structured pivots of the form $\begin{pmatrix} \times & \times \\ \times & 0 \end{pmatrix}$ or $\begin{pmatrix} 0 & \times \\ \times & 0 \end{pmatrix}$ and used data structures to preserve the block structure of (4.1) during the factorization. At the same time, several improvements were made to the implementation including the use of Level 3 BLAS and the avoidance of COMMON blocks. However, although there were some notable successes with this code (Duff and Reid 1996), its complexity meant that it sometimes performed worse than MA27, sometimes even on matrices of the form (4.1) for which it was designed. Thus it never did replace or supersede MA27 in HSL. More recently, John Reid has developed a new HSL routine, MA67, to replace MA47. It combines the analysis and factorization phases. Although it does not use a multifrontal approach, it does make use of the Level 2 and Level 3 BLAS. We compare our new code with these specialized codes in Section 6.6.

One of the main reasons for the decision to develop MA57 was to keep to the relative simplicity of the MA27 algorithm but provide a code with a better user interface, that does not use COMMON blocks, and that uses higher level BLAS in both factorization and solve phases. In addition, there were several new features that had been often requested by users of MA27 that we wished to add.

These new features of MA57 include:

In the analysis phase:

1. Use of an approximate minimum degree ordering (Amestoy, Davis and Duff 1996) as well as minimum degree, with a version that is efficient even if the input matrix has some dense rows.

In the factorization phase:

1. A range of pivoting options.
   - Numerical pivoting using the Bunch, Kaufman, Parlett decomposition with threshold pivoting, as described in Section 3.

- No numerical pivoting and error return if the matrix is discovered non-definite.
- No numerical pivoting and only exit if a zero pivot is found.
- No numerical pivoting but, if the matrix is not definite, then modify pivots dynamically using a variant of the Schnabel-Eskow scheme (Eskow and Schnabel 1991) to obtain a factorization $\mathbf{L'D_2L'^T}$ of the modified matrix $\mathbf{A}+\mathbf{D_1}$ where $\mathbf{D_1}$ and $\mathbf{D_2}$ are diagonal matrices and entries of $\mathbf{D_2}$ are all of the same sign.

2. The ability to restart the computation from where it stops if it runs out of storage. It is also possible to discard the factors to provide more space so that the factorization can continue and will provide accurate information on the space required for a subsequent factorization of the same matrix.

3. The option to return the pivots to the user and to alter them if desired.

In the solve phase:

1. A range of entries for error analysis and iterative refinement. This can be automatic, using the strategy of Arioli, Demmel and Duff (1989), with either

   - no estimate of the solution provided by the user,

     or
   - an estimate of the solution provided by the user,

   or can be left more to the control of the user when only one step of iterative refinement is performed on each call. There are five possible options.

   - Solve and return the residual.
   - Solve, return the residual, and perform one iterative correction.
   - Estimate of the solution provided by the user. Compute the residual and perform one iterative correction.
   - Estimate of the solution and the residual provided by the user. Perform one iterative correction.
   - Estimate of the solution and the residual provided by the user. Perform one iterative correction and return correction and new residual.

2. The solution of multiple right-hand sides using Level 3 BLAS.

3. If the factorization is written as

$$\mathbf{A} = \mathbf{PLDL^TP^T}$$

   then a partial solution facility offering the solution of equations with coefficient matrix:

- $\mathbf{A}$
- $\mathbf{PLP^T}$
- $\mathbf{PDP^T}$ or
- $\mathbf{PL^TP^T}$.

5 The software package MA57

In this section, we describe the user interface to `MA57`, its internal data structures, and its use of BLAS kernels.

5.1 The user interface

An initial decision when designing `MA57` was that it should have a user interface which was similar to that of the previous indefinite solver `MA27`. This was because we wanted to make it straightforward for a user who was familiar with `MA27` to use `MA57`. In addition we have also provided a Fortran 90 interface with additional functionality. We describe the version in Fortran 77, making appropriate reference to the added features of the Fortran 90 version as we proceed. The `MA57` package has six entries which may be called directly by the user. Each of the subroutines are named according to the naming convention of HSL with the single precision version having names commencing with `MA57` plus one more letter, and double precision versions with the additional sixth letter `D`. For simplicity, we will use the single precision names throughout this report. For similar reasons, if a BLAS routine is referenced explicitly, we will use the single precision name. The user-callable entries are:

Initialization : `MA57I` initializes the parameters which control the execution of the package. A single call should be made to `MA57I` before any other routines in the `MA57` package are called.

Analysis and symbolic factorization : `MA57A` computes an ordering using a variant of the minimum degree algorithm and then generates an assembly tree and calculates the work and storage required for subsequent numerical factorization if there are no delayed pivots because of numerical pivoting.

Numerical factorization : `MA57B` performs the numerical factorization of the matrix using the information generated by `MA57A`.

Solve : `MA57C` solves for one or more right-hand sides using the factors produced by `MA57B`. A single call to `MA57C` will solve for the number of right-hand sides specified by the user. If the iterative refinement and error analysis of Arioli et al. (1989) is required, then `MA57D` should be called, in which case only one set of equations (that is, one right-hand side) can be solved on each call.

Copying data : `MA57E` copies the data into larger arrays so that the `MA57B` routine can be recalled to continue the factorization if it had earlier stopped because of running out of space. The default for the Fortran 90 code `HSL_MA57` is to automatically increase the storage by a user-supplied factor, to use `MA57E` and then to restart the factorization from the point at which it ran out of space.

We briefly discuss each of these subroutines. Full details of their argument lists and their calling sequences are given in the Specification Sheets (see Section 9).

5.1.1 Initialization

The user should make a single call to `MA57I` prior to calling any of the other routines in the `MA57` package. `MA57I` assigns default values to the control parameters held in the arrays `ICNTL` and `CNTL`. These parameters control the action of the subroutines within the `MA57` package. They include parameters to control the level of diagnostic printing. Should the user want a control parameter to have a value other than its default, the appropriate parameter should be reset after the call to `MA57I`.

5.1.2 Analysis phase

`MA57` allows the user to specify (using the control parameter `ICNTL(12)`) the minimum number of pivots that will be selected at any stage. Delaying performing eliminations until the number of fully summed variables is at least `ICNTL(12)` increases the Level 3 BLAS component of the factorization albeit at the cost of more floating-point operations and an increased number of reals in the factor. We show the influence of this *node amalgamation* parameter on our analysis phase in Table 6.4 and the effect of the changed assembly tree on the numerical factorization in Table 6.6.

The default ordering in the analysis is the approximate minimum degree ordering (AMD) of Amestoy et al. (1996). This is almost always much faster than a regular minimum degree ordering, as for example implemented in `MA27`, but often gives as good if not a better ordering for the subsequent factorization. The current implementation of AMD in HSL is `MC47` and this does not take any special action if there are dense or nearly dense rows in the matrix. We thus offer the option to use a research version of a routine, `MC50`, that does this and have made this the default ordering in the Fortran 90 version. We thus include `MC47`, `MC50`, and the minimum degree ordering routine from `MA27` in our analysis phase and we compare these orderings on some test examples in Table 6.3.

5.1.3 Numerical factorization

In the indefinite case, the space required by the numerical factorization cannot be determined by the symbolic factorization, so it is possible for the factorization to run out of storage. The main control for this is the threshold parameter, u, see equations (3.2) and (3.3). This is set by the user in CNTL(1) and we study the effect of varying this in Section 6.3. In MA57, each pivot candidate (that is, each fully summed variable) is checked to see that it is of absolute value at least as large as the control parameter CNTL(2) (with default value zero). If a pivot is found to be too small, the action taken depends on the setting of the parameter ICNTL(7). If a pivot is found to be negative, the matrix is not positive definite, and an immediate failure occurs if ICNTL(7)=2 but, if ICNTL(7)=3, the computation continues provided the pivot is of absolute value at least CNTL(2). In this case, and in the case when numerical pivoting is performed (ICNTL(7)=1), the number of negative pivots is returned to the user in the information array INFO.

As mentioned earlier, a major difference between MA57 and the earlier code MA27 is that extensive use of the Level 3 BLAS are made in MA57. The block size for the Level 3 BLAS is determined by the user (with a default value set in MA57I).

A major benefit of multifrontal methods is that the floating-point arithmetic is performed on dense submatrices. In particular, if we perform several pivot steps on a single frontal matrix, the Level 3 BLAS can be used. However, in the present case, we also wish to maintain symmetry and the current Level 3 BLAS suite does not have an appropriate kernel. Provided at least NBLOC pivots are selected in the block pivot step, we use Level 3 BLAS for constructing the Schur complement. The computation is of the form (2.4) and, if we represent the matrix $\mathbf{F_{22}}$ by $\mathbf{S}$, $\mathbf{D_1^{-1}}$ by $\mathbf{D}$, and $\mathbf{F_{12}}$ by $\mathbf{M}$, this can be expressed in the form

$$\mathbf{S} = \mathbf{S} - \mathbf{M^T D M}, \tag{5.1}$$

where $\mathbf{M}$ is a rectangular matrix and $\mathbf{D}$ is a diagonal matrix with blocks of order 1 and 2. Our concern is the efficient formation of (5.1) for a full rectangular matrix $\mathbf{M}$. Unfortunately, there are no BLAS routines for forming a symmetric matrix as the product of two rectangular matrices, so we cannot form $\mathbf{DM}$ and use a BLAS routine for calculating $\mathbf{M^T}(\mathbf{DM})$ without ignoring symmetry and doing about twice as much work as necessary (although the subroutine SYMM in the new BLAS recently announced by the BLAS Technical Forum (2002) will hopefully be available from vendors in the not too distant future). We therefore choose a block size NBLOC (with default size 16 held in ICNTL(11)) and divide the rows of $\mathbf{M}$ and $\mathbf{DM}$ into strips that start in rows 1, NBLOC+1, 2 NBLOC+1, ... This allows us to compute the block upper triangular part of each corresponding strip of $\mathbf{S}$ in turn, using SGEMM, the Level 3 BLAS matrix-matrix multiplication kernel, for each strip except the last where, for this undersize strip, we use simple Fortran code and take full advantage of the symmetry. In a block of size NBLOC on the diagonal, the extra work is NBLOC*(NBLOC-1) floating-point operations. Clearly this means that while we would like to increase NBLOC for Level 3 BLAS efficiency, by doing so we increase the amount of arithmetic. In Table 6.8 we examine the trade off between these competing trends.

5.1.4 Solve phase

When $nrhs > 1$, both the forward elimination and back substitution steps use SGEMM. When there is only one right-hand side ($nrhs = 1$), Level 2 BLAS are used. It should be noted that $\mathbf{B}$ is involved in the matrix-matrix product and that SGEMM becomes more efficient with an increased number of columns in $\mathbf{B}$ (see results in Table 6.9). The time for this phase is significantly less than for the factorization although, as we will comment when we discuss the solution of optimization problems, there are circumstances where it is very important to keep the cost of the solution as low as possible. In an early version of the MA57 code, we were alarmed to find that the overhead of a call to SGEMM was noticeable when there was only one right-hand side and so we use SGEMV for this case.

The factors (see equation 2.4), are held by rows as a packed triangular matrix $\mathbf{D_1 L_1^T}$ (held as $\mathbf{D_1^{-1}}$ and $\mathbf{L_1}$) and a dense rectangular matrix $\mathbf{F_{12}}$ corresponding to the rows and columns of the pivot block and the out of pivot columns at that elimination stage. When using the block pivot to update the appropriate components of the modified right-hand side, the components can be identified in the main right-hand side vector using indirect addressing or they can first be mapped into a temporary vector of length equal to the size of the current frontal matrix. Thence the operations at this block stage can be effected using direct addressing and Level 2 or 3 BLAS. If there are very few rows in the block pivot, then it may not pay to load the appropriate entries of the modified right-hand side into a dense vector, and it may be more efficient to use indirect addressing on the main vector. We have a switch (ICNTL(13)) to control this. To be more precise, mapping of vectors and direct addressing is used if there are more than 4 rows and the number of columns exceeds ICNTL(13). We show the influence of this parameter in Table 6.10.

5.2 Copying data

One of the weaknesses of sparse direct methods that use numerical pivoting is that it is not known when commencing the factorization how much storage will be required. An estimate is given by the analysis but this will only be accurate if no pivots are delayed and all eliminations take place at the expected node of the assembly tree. In our earlier codes, if insufficient space was allocated, the factorization terminated with an error message, and the user had to recall the code after allocating more space to the arrays used in the factorization. This could be particularly annoying if the termination occurred near the end of the factorization, perhaps after an already costly run.

In our new code, we allow the factorization routine to exit, keeping sufficient data to enable a reentry to continue the factorization from the point of termination. This is permitted if ICNTL(8) is left at its default value of 1. The user can allocate bigger arrays, copy the current data to these and recall the factorization routine. In the Fortran 90 code, this is done automatically and we include a parameter to indicate how much larger the arrays should be on reentry (the default is to double the size).

5.3 Internal data structures

The internal data structures used by `MA57` are similar to those of our earlier multifrontal codes. The user must supply both a real array and an integer array, both of which are subdivided within the package as follows:

`Computed factors`	`Current frontal matrix`	`Free space`	`Stacked elements`

The user's input data is held in separate arrays and is not changed during the calls to `MA57`.

We now discuss this subdivision of workspace in a little more detail.

`Computed factors.` This part of the arrays holds the factors that have already been computed and thus will grow monotonically to the right during the factorization. The real values are held block pivot by block pivot and comprise the triangular factor $\mathbf{D_1 L_1^T}$ (held as $\mathbf{D_1^{-1}}$ and $\mathbf{L_1}$) in packed form followed by the rest of the block pivot rows as a dense rectangular matrix $\mathbf{F_{12}}$ held by rows. The integer storage comprises a list of column indices headed by the number of rows and number of columns in the block.

`Current frontal matrix.` This part is formed afresh for each tree node and is used to assemble the current frontal matrix from original rows and stacked elements. It is held in full form to expedite assembly and subsequent elimination operations.

`Free space.` At the beginning, this is the only partition present in the arrays and subsequently free space is eroded from the front by the factors (and temporarily the frontal matrix) and at the end by the stack.

`Stacked elements.` Any rows and columns remaining after the eliminations within the frontal matrix must be stacked as a dense triangular matrix for later assembly when the parent node is being factorized. We store these matrices in packed form.

5.4 Structure of the package

5.4.1 Analysis

The driver subroutine `MA57A` first does simple checks on the user's input data and then executes code depending on the choice of ordering option determined by the parameter `ICNTL(6)`. The options are:

`ICNTL(6)=1`. The user is supplying an ordering. First `MA57J` is called to sort the input matrix into the working arrays and then the assembly tree is generated using `MA57K`.

`ICNTL(6)=0 or 3`. The approximate minimum degree ordering is being used. The matrix is first sorted using `MA57G` and either `MC47B` is called if the standard AMD is used (`ICNTL(6)=0`) or the experimental code `MC50` that should be faster if the input matrix has some dense rows (`ICNTL(6)=3`).

`ICNTL(6)=2`. The minimum degree ordering as used in the old `MA27` code is requested. The sort is performed using `MA27G` followed by the ordering and tree generation routine `MA27H`.

In all cases, the tree generated is then processed by `MA57L` that performs a depth first search, generates a permutation, orders the children so that the eldest child (the last to be processed) has the largest frontal matrix, and performs node amalgamation if requested. A mapping vector to map the user's input directly to the reordered form for efficient factorization is then constructed using `MA57M` to avoid any expensive sorting within the factorization and, finally, subroutine `MA57N` evaluates storage and operation counts for the case when subsequent numerical pivoting does not alter the tree structure.

5.4.2 Factorize

The driver subroutine `MA57B` first does simple checks on the user's input data and then copies the input data directly to the working arrays using the mapping vector generated by the analysis phase. The factorization is then effected by a call to `MA57O`, which we now describe in more detail. One feature of this code is that there is only one part where problems with insufficient storage can appear.

For each tree node in turn the following steps are executed by `MA57O`.

Step_1. Construct the index list for the frontal matrix. The indices of the rows that are fully summed at this step (new rows), followed by those from the child nodes that were previously fully summed (old rows from which pivots were not earlier chosen because of numerical considerations), are placed in the leading positions and their number is stored in `NASS`. The total number of indices is stored in `NFRONT`. We call this phase the *symbolic assembly.*

Step_2. Assemble the fully summed rows numerically as a full square matrix of dimension `NFRONT` although only the upper triangular part holds the correct data. If there is insufficient space available for this assembly, the garbage collection routine `MA57P` is first called and, if sufficient space is still not obtained, the subroutine will either:

- exit, after having copied the values of key variables to a subroutine parameter to enable a subsequent restart from the same point if `MA57B` is recalled with larger working arrays (`ICNTL(8)` $\neq$ `0`),
- or will destroy previously constructed factors and try to continue (`ICNTL(8) = 0`). In this case, if the factorization completes, exact values for storage and work for the factorization will have been computed although no factors are generated.

Step_3. We then search for pivots. Unless the matrix is assumed definite or matrix modification is used (`ICNTL(7)>1`), numerical threshold pivoting is performed within the frontal matrix. To make use of Level 3 BLAS within the pivot block, the fully summed block is divided into blocks of length `NBLOC` depending on the input parameter `ICNTL(11)`. Of course, the last block might have fewer than `NBLOC` rows. In a cyclic search of the current `NBLOC` rows of the fully summed part of the frontal matrix, first the diagonal is tested to see if it passes the test (3.2) and, if so, is accepted. Otherwise, the 2×2 pivot comprising the diagonal and the largest off-diagonal in its row that is in the current block is checked for stability using the test (3.3) and is accepted if the test is passed. If a pivot is not accepted the search moves cyclically to the next diagonal. If one is, then the multipliers are computed and the part of the fully summed block within the current block of `NBLOC` rows is updated. If it proves impossible to find pivots in the block, the block size is doubled and we try again. At the end of the search or when `NBLOC` pivots are found, Level 3 BLAS are used to update the rest of the fully summed part of the frontal matrix. In the case that more pivots can be found, we then move to the next set of `NBLOC` fully summed rows/columns. At the end of this step, all of the fully summed rows are updated but no operations have been performed on the Schur complement.

Step_4. If the Schur complement part of the frontal matrix has order larger than `NBLOC`, this is updated in place using the Level 3 BLAS by block columns to avoid computing all of the symmetric matrix (see Section 5.1.3). The upper triangle of the Schur complement is then stacked and the copying is done so that no extra space is required.

If the Schur complement has order less than `NBLOC`, the upper triangle is first copied to the stack (again no extra space is required) and then the update operations are performed directly on this copy.

Step_5. The data structure of the factors is reorganized into a packed triangular matrix followed by a rectangular block held by rows, and is stored at the end of the already computed factors. Again no extra space is required for this.

5.4.3 Solve

There are two user-callable solve routines, `MA57C` and `MA57D`. `MA57C` is the primary solution routine and is itself called by `MA57D`.

`MA57C` first checks the user's input data and then calls separate routines for the forward and back substitution. For one right-hand side, `MA57X` is called for the forward substitution and `MA57Y` for the back substitution. The corresponding subroutines for the multiple right-hand side case are `MA57Q` and `MA57R`, respectively. The entries for the partial solutions mentioned at the end of Section 4 call `MA57X`/`MA57Q` for the solution with $\mathbf{L}$ and `MA57S` for the solution with $\mathbf{D}$. However, the solution with $\mathbf{L}^{\mathbf{T}}$ requires requires slightly different logic and is performed by `MA57T`. In all cases, these inner routines scan the factors in the appropriate order and either first load the appropriate components into a dense vector and use Level 2 or 3 BLAS routines or operate directly on the vector using indirect addressing, depending on the value of `ICNTL(13)`. If the number of rows in the block pivot is greater than 4 (a value that we found to be appropriate in some earlier tests) and the number of columns greater than `ICNTL(13)` then the BLAS routines are used.

`MA57D` performs iterative refinement and computes the measures required by the error analysis of Arioli et al. (1989). It calls `MA57C` to perform the solutions. This entry only allows the solution of one right-hand side.

6 Numerical results

6.1 Test problems

In this section, we describe the problems that we use for testing the performance of `MA57`. In all cases, they arise in real engineering, industrial or commercial applications. We record the fact that we performed many experiments on a much extended set but have chosen this subset as a manageable and representative sample. A brief description of the principal set of test problems is given in Table 6.1. The problems are grouped into three categories. In each category, we have ordered the test matrices by increasing dimension. A brief description of the source of each is given and all matrices are available from the author on request and will soon be available through the GRID-TLSE Project in Toulouse (`www.enseeiht.fr/lima/tlse`). The Rutherford-Boeing test set that will be available from this site has been developed from the original Harwell-Boeing Collection (Duff, Grimes and Lewis, 1992) and is available in prototype form from `ftp.cerfacs.fr/pub/algo/matrices/rutherford_boeing/`.

Identifier	Order	Number of entries	Description/source
Indefinite matrices ... nonzero diagonal			
VIBROBOX	12328	177578	Vibroacoustic problem
HELM3D01	32226	230335	3-D Helmholtz
DAWSON5	51537	531157	FLAP, part of actuator system on airplane
COPTER2	55476	407714	Adapted CFD grid for helicopter rotor blade
BMW3_2	227362	5757996	Car crankshaft
HELM2D03	392257	1567096	2-D Helmholtz
Augmented system matrices			
SAWPATH1	1359	4066	CUTEr problem
EXDATA	6001	1137751	Misc problem from Wright
BRATU3D	8288	28583	CUTEr problem
NCVXQP1	12111	47648	CUTEr problem
AUG3DCQP	35543	105372	CUTEr problem
TURON_M	189924	912345	Modelling Uranium mine
DARCY003	389874	1167685	Mixed finite elements
Positive definite matrices			
NASASRB	54870	1366097	Shuttle Rocket Booster
CFD1	70656	949510	CFD problem
FINAN512	74752	335872	Portfolio optimization

Table 6.1: The test problems.

The first category are indefinite systems with a nonzero diagonal. Problems VIBROBOX, DAWSON5, and COPTER2 were all obtained originally from Tim Davis (`www.cise.ufl.edu/~davis/sparse/`); the first one supplied to him by André Cote from Quebec, and the second two supplied by Ed Rothberg (formerly at SGI), the last of which originated from R. Strawn of NASA Ames. HELM2D03 and HELM3D01 were generated for the author by Mario Arioli at the Rutherford Appleton Laboratory and represent a discretization of the Helmholtz equation in two and three dimensions, respectively. BMW3_2 is from the PARASOL test set (`www.parallab.uib.no/parasol/`).

The second group are augmented matrices of the form shown in equation (4.1). Four are from the CUTEr collection (Gould, Orban and Toint 2002), EXDATA is from Mike Gertz (Argonne National Laboratory), and the remaining matrices TURON_M and DARCY003 are from Mario Arioli (Rutherford Appleton Laboratory), both are augmented systems from a mixed finite-element model. The first is from a model of a Uranium mine in the Czech Republic, courtesy of the Department of Mathematical

Modelling in DIAMO, s.e., Stráž pod Ralskem, and the second is a discretization of Darcy's equation by Arioli and Gianmarco Manzini of CNR Pavia, Italy.

Although MA57 is specifically designed for indefinite systems, it should be efficient also when the matrix is positive definite. We thus include in our test set three positive definite matrices: NASARB from Alex Pothen of ICASE, and CFD1 and FINAN512 from Ed Rothberg. FINAN512 was originally generated by John Mulvey at Princeton.

The experimental results given in this paper were obtained using 64-bit floating-point arithmetic on a single processor of a Compaq/HP Alpha DS 20 workstation using the Fortran `f90` compiler with the compiler options `-O5 -arch ev6 -tune ev6 -math_library fast -assume bigarrays -assume nozsize`. For all the runs, we allocate a working array of length 110 million 64-bit words (this is the maximum we could allocate on the DS 20), and we use the default settings for the code parameters, unless explicitly stated otherwise. The default value that we use for all codes for the threshold u is 10^{-2}. The vendor-supplied BLAS were used (from the DXML Library). All times are CPU times in seconds. In all the tables in which the number of floating-point operations ("ops") are quoted, we count all operations (+, -, *, /) equally.

In the following subsections, we examine aspects of the three main phases of solution and then compare the new MA57 code with MA27 on the full set of test matrices. We then compare our code on the set of augmented systems with two other codes from HSL, MA47 and MA67, that are specifically designed for such systems.

6.2 The analysis phase

In the analysis phase, we allow (through the parameter ICNTL(7)) a choice of three ordering routines. The other parameter that directly affects the performance of the analysis, and in particular the tree that is generated, is ICNTL(12) that controls node amalgamation. We examine the effect of these two parameters in this section.

When we ran the three ordering algorithms on our set of test problems in Section 6.1, in all cases the analysis times were almost identical for the MC47 and MC50 orderings while that for the full minimum degree algorithm was expectedly higher, usually by a factor of about two. The quality of the ordering produced in all cases was similar although the AMD algorithms almost always produced slightly less fill-in. The fact that this occurred is not particularly surprising, but we were a little surprised about the consistency with which AMD beat the MA27 implementation of minimum degree, sometimes by quite a large margin. For the purposes of this experiment, we thus introduced some new test problems that we describe in Table 6.2. In this table, GUPTA1 was from Anshul Gupta of Minneapolis (now at IBM), and BRAINPC2 and A0NSDSIL are from the CUTEr test set.

We show the results in Table 6.3. As in the main test set, the MC47 (AMD) times are usually about half that of MA27H (minimum degree) with the exception

Identifier	Order	Number of entries	Description/source
BRAINPC2	27607	117384	CUTEr problem
GUPTA1	31802	1098006	Linear programming matrix of form $\mathbf{AA^T}$
A0NSDSIL	80016	200021	CUTEr problem

Table 6.2: The additional test problems for comparing orderings.

of GUPTA1 where it is about thirteen times faster. However, we see that in this example and in the others, the routine MC50 is much faster than either of these orderings by up to a factor of 100 over MC47. We studied the structure of these matrices and found that there were a significant number of very dense rows (for example, for BRAINPC2 with an average number of entries per row of around 5, there are two rows with over 20,000 entries, one of 14,000, and one of 7,000). Thus, as is evident from our results, a code that respects this will perform much better than one which does not. Although these dense rows are not chosen early in the elimination they are always accessed during the degree update unless special action is taken to prevent this. There is a primitive form of such prevention in the MA27H code, but it clearly does not work all the time, spectacularly so in the case of GUPTA1. The quality of the ordering is very similar in all cases, although occasionally there is a wide variation in factorization times. We investigated thoroughly the SAWPATH1 example and found that quite a different assembly tree was produced after the different orderings. The MA27H and MC50 trees have much shorter paths from the root to the leaves than that generated by MC47. This means that a delayed pivot is often passed up a long chain in the MC47 case, and the frontal matrices in this chain increase in size as the factorization progresses. However, in the other cases, the Schur complement matrix containing the delayed pivot is stacked and not assembled until the root or close to the root. The number of delayed pivots for the ordering produced by MC47, MA27H, and MC50 are 125720, 35594, and 776 respectively, where the count for delayed pivots is augmented each time a pivot is delayed even if it had already been delayed earlier in a previous step. This is reflected in the different factorization times although the final storage for the factors was 170 Kwords in each case.

The influence of the node amalgamation (ICNTL(12)) is shown in Table 6.4 where, as expected, the number of tree nodes and assembly operations decrease monotonically with increasing value of ICNTL(12) while the number of elimination operations increases. The behaviour over the test problems is similar but not identical and we will examine this parameter again in the next section when we discuss the numerical factorization. The analysis times are, of course, hardly altered by changing ICNTL(12) since this is only used during the final post-processing of the assembly tree.

Identifier	Ordering	Analyse time (seconds)	Forecast for factorization Storage (Kwords)	Ops ($*10^6$)	Factorize time (seconds)
SAWPATH1	`MC47`	0.005	5	.023	5.49
	`MA27H`	0.007	5	.023	0.83
	`MC50`	0.004	5	.023	0.14
BRAINPC2	`MC47`	8.85	193	1.42	0.29
	`MA27H`	14.68	193	1.42	0.74
	`MC50`	0.09	193	1.42	0.64
GUPTA1	`MC47`	113.62	2056	302	8.42
	`MA27H`	1452.04	2021	265	8.71
	`MC50`	4.93	2056	302	4.93
A0NSDSIL	`MC47`	10.82	340	1.72	0.23
	`MA27H`	23.98	340	1.72	0.23
	`MC50`	0.28	350	1.90	0.24

Table 6.3: A comparison of different orderings.

6.3 The factorization phase

In this section, we first study the effect of varying the pivot threshold parameter, u. Of course, the fill-in and time may be very influenced by the value set for this in `CNTL(1)`. We show the influence of this through the results in Table 6.5. We then study the influence of the parameters that control the minimum pivot block size (Tables 6.6 and 6.7) and the size of the blocks used by the Level 3 BLAS for updating the frontal matrix (Table 6.8). The control parameters for these are `ICNTL(12)` and `ICNTL(11)`, respectively. In Tables 6.6 to 6.8 results are given for a subset of our test problems for a range of values of the parameters.

The results in Table 6.5 are really quite dramatic and show clearly that vastly different performance can be obtained by reducing the value of the threshold parameter. The effect is usually most dramatic when u is reduced from 10^{-1} to 10^{-2} and in most of our test runs the residual or error did not increase significantly by this reduction. However, we should be wary of reducing the threshold too much since we can potentially get greater growth in the factors and hence greater instability in the factorization. We show the scaled residual in the last column of Table 6.5 and note that there are just two cases when the value is not at rounding level for 64-bit arithmetic. In the case BRATU3D, the residual is unsatisfactory for threshold values smaller than 10^{-2}. If iterative refinement is used, the residual can be reduced to 10^{-15}, but it does not happen with threshold values smaller than 10^{-5}. Although there are sometimes significant gains by reducing the threshold from 10^{-2} to 10^{-5}, we feel that we want to err on the side of caution and choose $u = 10^{-2}$ as the default value. Another reason for this is that, although iterative refinement can sometimes enable a stable solution for lower thresholds, it is quite an expensive option relative to the solution cost. However, it may be very appropriate to set a lower threshold

Identifier	Amalgamation level ICNTL(12)	Number of tree nodes	Forecast for factorization Ops ($*10^6$) Assembly	Ops ($*10^9$) Elimination
DAWSON5	1	18214	14.0	1.22
	5	9519	12.0	1.23
	10	8158	11.1	1.24
	50	6940	8.4	1.40
	100	6840	7.5	1.64
BMW3_2	1	28981	157	54.5
	5	24883	155	54.5
	10	17332	143	54.6
	50	10081	117	56.2
	100	8376	105	58.8
HELM2D03	1	231446	67.0	13.7
	5	121090	61.4	13.7
	10	109865	58.4	13.8
	50	102833	48.9	14.7
	100	102353	43.9	16.2
SAWPATH1	1	1331	.0131	.0000232
	5	479	.0077	.0000590
	10	414	.0070	.0000740
	50	396	.0068	.0002870
	100	394	.0067	.0008420
BRATU3D	1	5941	4.74	.369
	5	3770	3.60	.371
	10	3672	3.31	.374
	50	3582	2.27	.404
	100	3571	1.94	.447
TURON_M	1	112216	50.8	17.0
	5	62791	49.1	17.0
	10	58858	47.9	17.1
	50	56409	42.8	17.7
	100	56228	39.2	18.7
DARCY003	1	312409	12.7	.764
	5	172969	10.9	.777
	10	160176	10.1	.802
	50	155952	8.3	1.02
	100	155801	7.5	1.28
NASASRB	1	7891	33.8	5.58
	5	5953	32.2	4.59
	10	4297	29.7	4.61
	50	3091	23.7	5.00
	100	2940	20.7	5.72

Table 6.4: Effect of node amalgamation.

Identifier	Threshold parameter u	Number of delayed pivots	Factorization Time (Seconds)	Factorization Number reals (Kwords)	Residual
VIBROBOX	10^{-1}	30855	40.58	7301	1. 10^{-15}
	10^{-2}	3509	3.23	2318	.8 10^{-15}
	10^{-5}	0	2.58	2087	.8 10^{-15}
	10^{-10}	0	2.58	2087	.8 10^{-15}
	10^{-12}	0	2.58	2087	.8 10^{-15}
BMW3_2	10^{-1}	$>$ 110 million words of storage required			
	10^{-2}	13800	129.98	55606	2. 10^{-15}
	10^{-5}	0	115.98	52963	3. 10^{-15}
	10^{-10}	0	116.08	52963	3. 10^{-15}
	10^{-12}	0	116.24	52963	3. 10^{-15}
SAWPATH1	10^{-1}	126109	5.38	172	.7 10^{-16}
	10^{-2}	125720	5.24	172	.8 10^{-15}
	10^{-5}	117045	4.00	157	.2 10^{-15}
	10^{-10}	2782	0.01	8	.2 10^{-6}
	10^{-12}	571	0.00	6	.4 10^{-5}
EXDATA	10^{-1}	2965	96.58	5370	.2 10^{-15}
	10^{-2}	2471	47.33	4176	.2 10^{-15}
	10^{-5}	0	2.37	1139	.2 10^{-15}
	10^{-10}	0	2.37	1139	.2 10^{-15}
	10^{-12}	0	2.37	1139	.2 10^{-15}
BRATU3D	10^{-1}	18104	5.59	2220	.6 10^{-13}
	10^{-2}	17312	5.03	2127	.1 10^{-10}
	10^{-5}	13540	3.52	1787	.2 10^{-5}
	10^{-10}	12683	2.74	1647	.3 10^{0}
	10^{-12}	12604	2.67	1632	.4 10^{0}
NCVXQP1	10^{-1}	123115	351.22	22486	.4 10^{-17}
	10^{-2}	122079	208.59	22000	.8 10^{-17}
	10^{-5}	83253	89.20	11936	.1 10^{-16}
	10^{-10}	56036	53.19	8438	.3 10^{-16}
	10^{-12}	23132	23.23	4729	.5 10^{-16}

Table 6.5: Effect of threshold parameter.

value if you are in an environment where there are other controls against unstable factorizations. This is often the case in optimization applications when a threshold of 10^{-9} or less is used by some codes (for example, the HSL code VE12).

Identifier	ICNTL(12)				
	1	5	10	50	100
DAWSON5	3.85	3.76	3.74	3.87	4.21
BMW3_2	131.63	129.61	128.62	129.45	133.07
HELM2D03	33.85	33.63	33.71	34.93	33.27
SAWPATH1	5.44	1.77	1.14	0.71	0.68
BRATU3D	5.09	4.80	4.75	4.55	4.83
TURON_M	39.10	39.00	39.06	40.20	42.16
DARCY003	4.10	3.91	4.04	4.71	5.24
NASASRB	12.59	11.86	11.76	11.99	12.97

Table 6.6: The time in seconds for the numerical factorization for different values of the node amalgamation parameter `ICNTL(12)`.

The factorization times in Table 6.6 indicate that, in general, modest savings can be achieved by allowing `ICNTL(12)` to be greater than 1 but the precise choice of the minimum pivot block size parameter does not appear crucial. This is important from a practical point of view since it is possible to get good performance without having to optimize the parameter from run to run. If we further look, in Table 6.7, at the influence of `ICNTL(12)` on the number of entries in the factors, and hence on the solution time, we see that these increase monotonically as `ICNTL(12)` increases. Because the factorization times are quite flat and the solution time is least when `ICNTL(12) = 1`, we choose this as the default but note that for small examples, such as SAWPATH1, a larger value may help.

Identifier	ICNTL(12)				
	1	5	10	50	100
DAWSON5	4627	4765	4888	5595	6106
BMW3_2	55606	55673	56404	61960	66321
HELM2D03	28016	29064	30046	34558	37694
SAWPATH1	172	247	247	247	247
BRATU3D	2127	2161	2185	2330	2472
TURON_M	23258	23854	24486	25427	27046
DARCY003	6941	7848	8927	11084	11922
NASASRB	11801	11865	12033	13220	14386

Table 6.7: The number of entries in the factors for different values of the node amalgamation parameter `ICNTL(12)`.

Identifier	ICNTL(11)						
	1	5	10	20	50	100	$> n$
HELM3D01	81.78	29.79	23.40	18.80	19.21	20.49	47.40
COPTER2	115.57	43.02	35.06	31.04	31.18	34.30	61.39
BMW3_2	795.45	210.83	158.96	134.46	129.88	137.50	358.48
HELM2D03	163.61	47.86	39.08	34.97	35.81	39.97	66.48
BRATU3D	19.51	6.55	5.61	5.24	5.07	5.29	6.53
NCVXQP1	1098.97	390.62	262.95	215.36	197.81	193.41	576.70
TURON_M	201.68	57.43	45.63	38.02	40.21	44.40	84.41
NASASRB	49.96	15.33	13.39	11.93	13.97	15.92	19.93

Table 6.8: The time in seconds for the numerical factorization for different values of the blocking parameter `ICNTL(11)`.

The effect of using blocking and the Level 3 BLAS is very dramatic, as we see in the results in Table 6.8. The factorization when using even a low value for the blocking is typically three times as fast as when using blocks of size one. We notice that the performance is relatively flat over the range 10-50 for the blocking parameter, `ICNTL(11)`, making it fairly easy to choose a default, which we have set at 16. We note that, for some examples, an even bigger block value can be beneficial although in every case switching off blocking (`ICNTL(11)`$> n$) does worse, sometimes by a factor of two or more.

6.4 The solve phase

In this section, we examine the influence of the number of right-hand sides and the switch between direct and indirect addressing on the solution phase. Although this phase is very much less costly than the factorization, typically about one hundred times faster, it is nevertheless important to tune this code well since there are many environments, for example in some optimization applications, where the solve times can still dominate.

In Table 6.9, we show the solve times for some of our test problems for the number of right-hand sides ranging from 1 to 50 and notice that it is typically over twice as fast to use a block of vectors **B** than to solve for a set of single right-hand sides. In some cases, however, the cost increases when many right-hand sides are being solved, notably for HELM2D03 and DARCY003. These are the largest systems that we solve with order just under 400,000 so it would appear that we are constrained by the size of the Level 2 cache on the DS 20. We show the cost for solving one right-hand side when using `SGEMM` (column 2 in Table 6.9), to indicate why we have chosen to use `SGEMV` for the case of one right-hand side. In further experiments, performed by John Reid at RAL, the preliminary indications are that it is better to use `SGEMV` if less than four right-hand sides are being solved simultaneously.

Identifier	Number of right-hand sides					
	GEMV	GEMM				
	1	1	5	10	20	50
VIBROBOX	0.06	0.08	0.03	0.03	0.03	0.03
HELM3D01	0.19	0.26	0.11	0.11	0.11	0.12
DAWSON5	0.15	0.21	0.08	0.07	0.06	0.07
COPTER2	0.37	0.51	0.20	0.18	0.17	0.19
BMW3_2	1.49	2.16	0.80	0.68	0.58	0.59
HELM2D03	0.97	1.33	0.54	0.49	0.48	0.71
BRATU3D	0.05	0.07	0.03	0.03	0.02	0.03
AUG3DCQP	0.04	0.05	0.02	0.02	0.02	0.02
TURON_M	0.67	0.95	0.37	0.32	0.28	0.34
DARCY003	0.40	0.54	0.28	0.28	0.39	0.73
NASASRB	0.33	0.46	0.17	0.14	0.11	0.12
CFD1	0.91	1.29	0.48	0.41	0.35	0.36

Table 6.9: The time for the solve phase in seconds per right-hand side.

The impact of changing the point at which we switch between direct and indirect addressing is less powerful. However, we see from the results in Table 6.10 that modest gains can be made in some cases by switching when the block size is around 10 (this is our default) and that it is almost always slower if only indirect addressing is used (`ICNTL(13)`$> n$), as we can see from the last column in Table 6.10.

Identifier	ICNTL(13)					
	1	5	10	50	100	$> n$
VIBROBOX	0.06	0.06	0.06	0.06	0.07	0.06
HELM3D01	0.20	0.20	0.21	0.20	0.20	0.24
HELM2D03	0.96	0.96	0.96	0.94	0.92	0.98
EXDATA	0.07	0.06	0.06	0.07	0.07	0.17
BRATU3D	0.05	0.05	0.05	0.05	0.05	0.05
NCVXQP1	0.38	0.38	0.37	0.37	0.40	0.74
NASASRB	0.32	0.32	0.32	0.33	0.32	0.28
FINAN512	0.14	0.15	0.15	0.15	0.14	0.20

Table 6.10: The time in seconds for the solve phase on a single right-hand side for changing the switch to indirect addressing.

6.5 The performance of MA57 compared with MA27

Our aim in designing and developing `MA57` was to produce a code that would be noticeably more efficient than the HSL symmetric indefinite multifrontal solver `MA27`. To assess how successful we have been, in Table 6.11 we compare runs of our new code with `MA27`.

Identifier	Code	Execution time (Seconds) Analyse	Factorize	Solve	Storage (Kwords) Real	Integer
Indefinite						
VIBROBOX	MA57	0.18	3.38	0.06	2318	212
	MA27	0.31	8.94	0.07	2961	210
HELM3D01	MA57	0.52	19.28	0.19	7631	481
	MA27	0.95	48.45	0.18	8148	480
DAWSON5	MA57	0.50	3.83	0.14	4627	562
	MA27	0.72	6.25	0.14	5214	568
COPTER2	MA57	0.76	30.05	0.36	14143	838
	MA27	1.27	101.66	0.36	17237	842
BMW3_2	MA57	2.87	129.02	1.45	55606	2091
	MA27	3.15	469.97	1.38	63692	2147
HELM2D03	MA57	3.16	33.70	0.95	28016	3744
	MA27	3.51	80.47	0.88	31687	3623
Augmented						
SAWPATH1	MA57	0.01	5.49	0.00	172	5
	MA27	0.01	0.53	0.00	172	4
EXDATA	MA57	0.63	47.39	0.07	4176	14
	MA27	0.54	36.27	0.07	4104	15
BRATU3D	MA57	0.06	5.03	0.05	2127	102
	MA27	0.12	6.00	0.05	2342	96
NCVXQP1	MA57	0.18	208.62	0.37	22000	297
	MA27	0.67	621.42	0.47	23330	344
AUG3DCQP	MA57	0.13	1.16	0.04	958	237
	MA27	0.17	1.42	0.03	1003	204
TURON_M	MA57	1.11	39.15	0.65	23258	1339
	MA27	1.13	158.41	0.70	31249	1394
DARCY003	MA57	2.59	4.07	0.38	6941	1937
	MA27	2.55	5.90	0.33	7629	1962
Pos Def						
NASASRB	MA57	0.73	12.53	0.32	11801	603
	MA27	0.83	24.86	0.30	13743	627
CFD1	MA57	1.10	98.06	0.89	37162	1412
	MA27	2.65	386.12	0.94	46747	1459
FINAN512	MA57	0.40	7.46	0.15	4318	664
	MA27	0.86	31.91	0.17	6323	863

Table 6.11: A comparison of the time and storage requirements for `MA57` and `MA27` on the set of test matrices.

The analysis phase is nearly always faster, sometimes by a factor of more than two. For the factorization phase, we see from these tables that MA57 is almost always significantly faster than MA27 and, for the larger problems, MA57 can be more than four times as fast as MA27. The solve times for MA57 are roughly comparable with those of MA27 although of course significant gains can be made with MA57 if several right-hand sides are solved simultaneously. The accuracy obtained by both codes was roughly comparable.

6.6 Comparison with other HSL codes on augmented systems

Augmented systems of the form (4.1) are ubiquitous in numerical computation (Duff 1994) so much so that we have designed codes in HSL to specifically deal with this case. We compare our new code with these codes, MA47 and MA67, on our set of augmented system examples in Table 6.12.

Identifier	Code	Execution time (Seconds)			Storage (Kwords)	
		Analyse	Factorize	Solve	Real	Integer
SAWPATH1	MA57	0.01	5.49	0.00	172	5
	MA47	0.01	4.69	0.00	170	3
	MA67		2.56	0.00	169	4
EXDATA	MA57	0.63	47.39	0.07	4176	14
	MA47	0.84	46.14	0.06	4108	11
	MA67		2092.45	0.42	1630	638
BRATU3D	MA57	0.06	5.03	0.05	2127	102
	MA47	0.83	1.42	0.03	1015	15
	MA67		2.65	0.25	973	108
NCVXQP1	MA57	0.18	208.62	0.37	22000	297
	MA47	17.30	37175.92	0.72	26645	7961
	MA67		614.79	0.35	16033	1016
AUG3DCQP	MA57	0.13	1.16	0.04	958	237
	MA47	0.33	1.50	0.05	1173	143
	MA67		3.20	0.04	1164	241
TURON_M	MA57	1.11	39.15	0.65	23258	1339
	MA47	1363.28	167.09	1.00	37061	1519
	MA67		219.52	1.28	48682	2395
DARCY003	MA57	2.59	4.07	0.38	6941	1937
	MA47	13217.81	6.55	0.42	7629	2264
	MA67		17.94	0.58	8491	2837

Table 6.12: A comparison of the time and storage requirements for MA57 and MA47 and MA67 on the set of augmented matrices.

The results in Table 6.12 show that MA57 often outperforms the other two codes both in terms of execution time and storage and, when it is not the best code, it is not much more than twice as bad as the best one. We note that, for SAWPATH1, the best code for the factorization time was in fact `MA27` (see Table 6.11) although, as we remarked in Section 6.2, this problem has somewhat unusual characteristics.

To check the effect of scaling the system using the HSL code `MC30`, we ran all of our test problems and all the codes after an explicit scaling of the system. In most cases, there was very little difference but, in three cases, the performance improved and, in three cases, it was noticeably worse. All codes were affected similarly. There were two cases that were particularly dramatic. For NCVXQP1, the factorize times were reduced to 90.01, 227.70, 15900.57, and 186.723 seconds for `MA57`, `MA27`, `MA47`, and `MA67` respectively. On the other hand, for BMW3_2, the codes are unable to factorize the scaled version within the memory allocated (which is the maximum we could obtain on the DS 20). This reinforces our belief that scaling is an inexact science and that it must be performed very much in the context of the application or environment in which the system is being solved.

7 The use of `MA57` in optimization packages

In interior point methods for the solution of nonlinear programming problems, a vital subproblem is

$$\min \frac{1}{2}\mathbf{x}^{\mathbf{T}}\mathbf{H}\mathbf{x} + \mathbf{g}^{\mathbf{T}}\mathbf{x}, \text{ subject to } \mathbf{A}\mathbf{x} = \mathbf{0} \tag{7.1}$$

for which the coefficient matrix is of the form

$$\begin{pmatrix} \mathbf{D}+\mathbf{H} & \mathbf{A}^{\mathbf{T}} \\ \mathbf{A} & \mathbf{0} \end{pmatrix}, \tag{7.2}$$

where $\mathbf{D}$ is a positive semi-definite matrix used to penalize inequality constraints.

A common solution scheme is to perform conjugate gradient iterations respecting the constraint $\mathbf{A}\mathbf{x} = \mathbf{0}$ and to use

$$\begin{pmatrix} \mathbf{B} & \mathbf{A}^{\mathbf{T}} \\ \mathbf{A} & \mathbf{0} \end{pmatrix}. \tag{7.3}$$

as a preconditioner that is factorized by a sparse direct code. We are currently investigating a range of preconditioners together with Dominique Orban of Northwestern.

Nick Gould of Rutherford Appleton Laboratory uses this approach and our software in both his interior point code `VE12`, and in `VE19`, an active set code with a Schur complement update, and both Jorge Nocedal and Steve Wright have used `MA57` as a linear solver in their optimization packages, KNITRO (Byrd, Hribar and Nocedal 1999) and OOQP (Gertz and Wright 2001), respectively.

This application of a sparse direct solver is a particularly rich one for testing the linear code. Obviously any matrix of the form (7.3), with **B** symmetric, is symmetric and indefinite. In addition, the matrix **B** is often not positive-definite and may even be (close to or actually) singular, so that really quite evil linear systems can result. A solution to the linear system is sometimes only possible because the right-hand side lies in a subspace for which the projection of the augmented matrix is nonsingular. Additionally, the matrices **A** and **B** can have a very wide range of sparsity patterns depending on the original source of the optimization problem. Such matrices, and the sophistication of the optimization codes which call the linear solver, result in a varied and severe test for all three phases of the direct solution. In the following sections, we discuss the use of `MA57` within these optimization codes, showing the influence that our early experience in this application area has had on the tuning and design of our code.

7.1 Analysis

For the analysis phase, we already remarked in Section 6.2 that the AMD algorithm as implemented in `MC47` can be rather slow when the matrix has one or more very dense rows. Indeed, this is fairly common in our optimization problems. For example, in a matrix of order 30007 supplied by Nick Gould, the analysis phase for `MC47` required 214 seconds on a DS 20 workstation, contrasting with 2.47 and 0.26 seconds for factorization and solve phases, respectively. It was this behaviour that prompted us to include the research code `MC50`. When we used this code, the analysis time dropped to 1.4 seconds and the quality of the ordering was essentially unaffected. Of course, one of the main issues in the case of augmented systems is that, as we saw in Section 6, the forecast from the analysis may not be a good guide to what is actually needed in the factorization.

7.2 Solve

Often, in the solution of sparse linear systems, less attention is placed on the solve phase because, as we saw in Section 6, it is usually up to two orders of magnitude faster than the factorization. However, in the context of these optimization codes, where the direct factorization of (7.3) is used as a preconditioner for an iterative method, there are many more calls to the solve routine than to the factorization routine (for example, in a typical run of KNITRO, there were 34,174 calls to the solve routines but only 125 matrix factorizations). Thus the overall time can be significantly affected by the solve times. Indeed, on early runs using the new `MA57` code, we were dismayed to find that, although the factorization times were less than half that of the earlier `MA27` code, the overall times for many of the optimization runs were much longer when `MA57` was used in place of `MA27`. This early problem was traced to a slower solve time for `MA57` which, on further investigation, was found to be caused by the overhead of SGEMM when there was only one right-hand side (see columns 2 and 3 of Table 6.9). This discovery caused us to write separate code for the case of one right-hand side as we discussed in Section 6.4. In most cases, the optimization code with MA57 included then outperformed the version with MA27.

7.3 Solution within optimization code

In Table 7.1, we show the results of a typical run of MA27 and MA57 on a system of linear equations obtained from test problem STCQP1 from the CUTE collection (Bongartz, Conn, Gould and Toint 1995). We see that MA57 is significantly superior to MA27 in all phases. This is true for all the linear systems that we have tested, and we have observed larger cases on which the MA57 factorize is ten times faster than that of MA27, largely because of the use of the Level 3 BLAS.

	Analyse	Factorize	Solve
MA27	0.59	0.54	0.08
MA57	0.29	0.39	0.06

Table 7.1: Times in seconds for all phases of MA27 and MA57 on example STCQP1 from CUTE. Matrix of order 5036 with 38045 entries.

Identifier	# vars	# inequalities	MA27	MA57
AUG3DCQP	27543	8000	95.8	54.9
AUG3DQP	27543	8000	95.8	56.9
BLOWEYA	20002	10002	18.3	35.9
DEGENQP	50	125025	49.1	19.2
CVXQP1	15000	7500	906.5	2.9
STCQP1	8193	4095	107.8	161.3
STCQP2	8193	4095	108.1	48.3

Table 7.2: Total times to solution of CUTE problems in seconds using VE12.

When we look, in Table 7.2, at times for complete runs of a prototype primal-dual interior point method of Gould, using both the codes, we would naturally expect the performance to mirror that in Table 7.1. In nearly all cases, runs of the optimization package with the new code are faster, sometimes significantly so. There are, however, a few cases that cause us concern (for example BLOWEYA and STCQP1 in Table 7.2). Of course, the times for the solution of the nonlinear problem will be affected strongly by convergence rates and the convergence path taken by the algorithm and even a small change in the solution provided by the linear solver may influence this. One important issue is that the matrices (7.2) become very ill conditioned, particularly near the solution of the optimization problem which is exactly when an accurate solution to the linear problem is required. Additionally, if we look at the results shown in Table 7.3, we see it is important from an efficiency point of view to keep the threshold very low but then there is significant risk of obtaining a poor solution, as happens when the threshold is reduced to 10^{-10} in this example. Because of this risk and because the solution must satisfy the $\mathbf{Ax} = \mathbf{0}$ constraints to high accuracy, the optimization codes have built in a facility for iterative refinement. Because of differences in pivoting strategies of `MA27` and `MA57`, it would appear that on some examples `MA57` required far more iterative refinements causing the overall time for the optimization to increase.

7.3.1 Iterative refinement

We conducted a more detailed investigation into the apparent paradox between the relative times for problem STCQP1 in Tables 7.1 and 7.2. The two runs of the optimization code required about the same number of function evaluations and conjugate gradient iterations but the run with `MA57` required far more iterative refinements. On further examination, we found that, for the same linear equations with the same threshold value of 10^{-10}, `MA57` returned a scaled residual of 10^{-6} while the scaled residual using `MA27` was 10^{-10}. The `MA57` value triggered the iterative refinement option. We then studied both solvers more closely to see why the residuals were so different and found that the `MA57` code had inherited a pivoting strategy from `MA47` whereby, if the candidate 2×2 pivot fails the threshold test (3.3), the (2,2) entry is immediately tested as a possible 1×1 pivot. If it satisfies the test, then the modified (1,1) entry is then tested and if it passes the test, the 2×2 pivot is accepted. This was a reasonable strategy for `MA47` since there is a high premium on being able to choose pivots of the form $\begin{pmatrix} 0 & \times \\ \times & \times \end{pmatrix}$. However, the resulting 2×2 pivot is potentially less stable than the normal test would allow. In fact, although the pivot has been accepted, the bound on the growth for the 2×2 pivot is now $1/u^2$ rather than $1/u$. This was certainly not intended for `MA57` and when this was removed, the resulting scaled residual was the same as for `MA27` and the optimization code ran faster with `MA57` than with `MA27`. Now, although this was all caused effectively by a bug in an early version of `MA57`, we have discussed this in some detail to illustrate the sensitivity of the optimization code to seemingly small changes to the linear equation pivoting strategy.

7.4 Factorize

Although Gould reports that `MA57` efficiently and effectively solves 99% of his problems from the CUTEr test set, he has found a few where `MA57` fails by running out of space. In one of the most dramatic examples, a problem from discrete optimized control called DTOC, the factorization of a matrix of order 24993 with only 69972 entries required more than 110 million words for the factorization. In this case, the

(1,1) block was zero so that no 1×1 pivot could be chosen and the ordering and assembly tree produced by the analysis were totally unable to guide the factorization. The strategies of MA47 and MA67 did work much better on this matrix but Duff and Gilbert are developing an ordering to choose block pivots and a prototype version of this approach has been tried with reasonable success on matrices of this form.

7.4.1 Scaling

Threshold	Factorization	
u	Reals	Time
Analysis	5273	
10^{-8}	44529	.406
10^{-9}	13620	.049
10^{-10}	8355	.013
With scaling		
10^{-8}	5915	.004

Table 7.3: Effect of changing threshold and scaling. Matrix from SAWPATH1 problem. Times in seconds for MA57.

Of course, some of the sensitivity to the threshold parameter that we see most dramatically in Table 7.3, might be significantly affected by scaling the matrix prior to numerical pivoting. When we used the HSL routine MC30 to scale the matrix in this table, we were able to factorize it stably with a threshold of 10^{-8} in a fraction of the time without scaling, and faster than the unstable factorization with threshold 10^{-10}, with the number of entries in the factors close to that predicted by the analysis. Although scaling works extremely well in this case, there are cases, for example the CUTEr problem A5ESINDL, on which the reverse is true so that scaling is as big an issue as always and we are investigating this further with Nick Gould and Jorge Nocedal of Northwestern.

8 Concluding remarks

We have designed and developed a new multifrontal code for solving systems of symmetric indefinite equations. The code makes full use of Level 3 BLAS in its innermost loop and in the solution phase. We have shown that the code can be significantly faster than MA27 and is competitive with specially designed HSL codes on structured matrices of the form (4.1).

9 Availability of the code

`MA57` is written in standard Fortran 77 and `HSL_MA57` in standard Fortran 90. There is also a version for complex symmetric matrices called `ME57`. Documentation on the codes can be obtained from the author. The codes are included in HSL 2002. Further information on HSL 2002 can be obtained from the Web page `www.cse.clrc.ac.uk/nag/hsl`.

Acknowledgments

I am very grateful to Michael Friedlander, Mike Gertz, Nick Gould, Jorge Nocedal, Michael Saunders, Richard Waltz, and Steve Wright for testing various versions of `MA57` within their optimization codes, in some cases generating test cases that I have used in this report. I am also grateful to Mario Arioli for generating and providing some of the test matrices, and to Nick Gould, John Reid, and Jennifer Scott of the Rutherford Appleton Laboratory for their remarks on an earlier draft of this paper.

References

Amestoy, P. R., Davis, T. A. and Duff, I. S. (1996), 'An approximate minimum degree ordering algorithm', *SIAM J. Matrix Analysis and Applications* **17**(4), 886–905.

Arioli, M., Demmel, J. W. and Duff, I. S. (1989), 'Solving sparse linear systems with sparse backward error', *SIAM J. Matrix Analysis and Applications* **10**, 165–190.

BLAS Technical Forum (2002), 'Special Issue: On Basic Linear Algebra Subprograms Technical BLAST Forum Standard -I and II', *The International Journal of High Performance Computing Applications.*

Bongartz, I., Conn, A. R., Gould, N. I. M. and Toint, P. L. (1995), 'CUTE: Constrained and Unconstrained Testing Environment', *ACM Trans. Math. Softw.* **21**(1), 123–160.

Bunch, J. R., Kaufman, L. and Parlett, B. N. (1976), 'Decomposition of a symmetric matrix', *Numerische Mathematik* **27**, 95–110.

Byrd, R., Hribar, M. and Nocedal, J. (1999), 'An interior point algorithm for large scale nonlinear programming', *SIAM J. Optimization* **9**(4), 877–900.

Curtis, A. R. and Reid, J. K. (1972), 'On the automatic scaling of matrices for Gaussian elimination', *J. Inst. Maths. Applics.* **10**, 118–124.

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. J., Hammarling, S. and Hanson, R. J. (1988), 'An extented set of Fortran Basic Linear Algebra Subprograms', *ACM Trans. Math. Softw.* **14**, 1–17.

Duff, I. S. (1984), 'Design features of a frontal code for solving sparse unsymmetric linear systems out-of-core', *SIAM J. Scientific and Statistical Computing* **5**, 270–280.

Duff, I. S. (1994), The solution of augmented systems, *in* D. F. Griffiths and G. A. Watson, eds, 'Numerical Analysis 1993, Proceedings of the 15th Dundee Conference, June-July 1993', Pitman Research Notes in Mathematics Series. **303**, Longman Scientific & Technical, Harlow, England, pp. 40–55.

Duff, I. S. and Reid, J. K. (1982), MA27 – A set of Fortran subroutines for solving sparse symmetric sets of linear equations, Technical Report AERE R10533, Her Majesty's Stationery Office, London.

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

Duff, I. S. and Reid, J. K. (1995), MA47, a Fortran code for direct solution of indefinite sparse symmetric linear systems, Technical Report RAL 95-001, Rutherford Appleton Laboratory, Oxfordshire, England.

Duff, I. S. and Reid, J. K. (1996), 'Exploiting zeros on the diagonal in the direct solution of indefinite sparse symmetric linear systems', *ACM Trans. Math. Softw.* **22**(2), 227–257.

Duff, I. S., Erisman, A. M. and Reid, J. K. (1986), *Direct Methods for Sparse Matrices*, Oxford University Press, Oxford, England.

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

Eskow, E. and Schnabel, R. B. (1991), 'Algorithm 695: Software for a new modified Cholesky factorization', *ACM Trans. Math. Softw.* **17**, 306–312.

Gertz, E. M. and Wright, S. J. (2001), *OOQP User Guide: Object-Oriented Software for Quadratic Programming*, Argonne National Laboratory.

Gould, N. I. M., Orban, D. and Toint, P. L. (2002), CUTEr (and SifDec), a constrained and unconstrained testing environment, revisited, Technical Report RAL-TR-2002-009, Rutherford Appleton Laboratory.

HSL (2002), 'HSL 2002: A collection of Fortran codes for large scale scientific computation'. `www.cse.clrc.ac.uk/nag/hsl`.

Irons, B. M. (1970), 'A frontal solution program for finite-element analysis', *Int. J. Numerical Methods in Engineering* **2**, 5–32.

Reid, J. K. (1972), Two Fortran subroutines for direct solution of linear equations whose matrix is sparse, symmetric and positive-definite, Technical Report AERE R 7119, Her Majesty's Stationery Office, London.