RAL-TR-2003-019

A numerical evaluation of HSL packages for the direct-solution of large sparse, symmetric linear systems of equations

Nicholas I. M. Gould and Jennifer A. Scott[1,2,3]

ABSTRACT

In recent years a number of new direct solvers for the solution of large sparse, symmetric linear systems of equations have been added to the mathematical software library HSL. These include solvers that are designed for the solution of positive-definite systems as well as solvers that are principally intended for solving indefinite problems. The available choice can make it difficult for users to know which solver is the most appropriate for their use. In this study, we use performance profiles as a tool for evaluating and comparing the performance of the HSL solvers on an extensive set of test problems taken from a range of practical applications.

[1] Computational Science and Engineering Department, Rutherford Appleton Laboratory,
Chilton, Oxfordshire, OX11 0QX, England, UK.
Email: n.gould@rl.ac.uk & j.a.scott@rl.ac.uk

[2] Current reports available from "http://www.numerical.rl.ac.uk/reports/reports.shtml".

[3] This work was supported by the EPSRC grant GR/R46641

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

September 22, 2003.

1 Introduction

The solution of linear systems of equations $Ax = b$ (or systems with multiple right-hand sides $AX = B$) is one of the cornerstones of scientific computation. In many cases, particularly when discretizing continuous problems, the system is large and the associated matrix A is sparse. Furthermore, for many applications, the matrix is symmetric; sometimes, such as in finite-element applications, A is positive definite, while in other cases, for example for constrained optimization, it may be indefinite.

HSL 2002 is an ISO Fortran library of packages for many areas in scientific computation (HSL, 2002). It is probably best known for its codes for the direct solution of sparse linear systems, there being a number of different packages for both the symmetric and unsymmetric cases. Since a potential user may be bewildered by such choice, our intention in this paper is to compare the alternatives on a significant set of large test examples from many different application areas, and, as far as is possible, to make recommendations concerning the efficacy of the various packages. We have chosen to concentrate on systems with symmetric A since there has recently been a comparison of codes for the unsymmetric case by Gupta (2002). This study forms part of a wider comparison of both HSL and non-HSL codes for the direct solution of symmetric linear systems, which will be reported on shortly.

For ease of reference, all the HSL codes that are used in this study are listed with a very brief description in Table 1.1. The linear equation solvers given in the top half of the table will be discussed in more detail in Section 2. The interested reader is also referred to the webpage `www.cse.clrc.ac.uk/nag/hsl/contents.shtml`; a complete catalogue for HSL is available from this site.

Code	Description
`MA27`	Sparse symmetric linear solver. Multifrontal algorithm. Minimum degree ordering.
`MA47`	Sparse symmetric indefinite linear solver.
`MA55`	Variable band symmetric positive-definite linear solver.
`MA57`	Sparse symmetric linear solver. Multifrontal algorithm. Approximate minimum degree ordering.
`MA62`	Sparse symmetric positive-definite linear solver for equations in elemental form. (Uni)frontal algorithm.
`MA67`	Sparse symmetric indefinite linear solver. Analyse-factorize code.
`FA14`	Generates pseudo-random numbers.
`MC30`	Matrix scaling routine.
`MC37`	Given a symmetric sparse matrix, computes a set of element matrices that, if assembled, would yield the same matrix.
`MC60`	Orders a matrix with symmetric sparsity pattern for small bandwidth or profile.
`MC63`	Orders element matrices for a frontal solver.
`MC73`	Computes spectral ordering.

Table 1.1: HSL codes used in our numerical experiments.

2 An introduction to HSL sparse symmetric solvers

2.1 Phases of a sparse direct solver

Sparse direct methods solve systems of linear equations by factorizing the coefficient matrix A, employing graph models to control the storage and work required. Many sparse direct solvers have three distinct computational phases: analyse, factorize, and solve. The analyse phase (which is sometimes referred to as the symbolic factorization or ordering step) determines a pivotal sequence. During the factorization phase, this sequence is used to compute the matrix factors. Forward elimination followed by back substitution is performed during the solve phase using the stored factors. Of the three phases, the factorization is usually the most time-consuming. An introduction to sparse direct solvers is given in the book by Duff, Erisman and Reid (1986).

For symmetric matrices that are positive definite, the pivotal sequence may be chosen using the sparsity pattern alone and so the analyse phase involves no computation on real numbers and the factorization phase can use the chosen sequence without modification. For symmetric indefinite problems, many codes again select a tentative pivot sequence based upon the sparsity pattern and then modify this sequence if necessary during the factorization to maintain numerical stability. However, some codes, including MA67 (see Section 2.7 below) work on the actual numbers so that a factorization results from the analysis. In such cases, the code is sometimes described as an analyse-factorize code.

Once the factors have been computed, they may be used to solve repeatedly for different right-hand sides b. Some packages offer the option of solving for more than one right-hand side at once because this enables them to take advantage of higher level Basic Linear Algebra Subprograms (BLAS) (Dongarra, DuCroz, Duff and Hammarling, 1990) for greater efficiency.

A number of codes offer an option of automatically performing iterative refinement to improve the quality of the computed solution and to help assess its accuracy.

2.2 MA27

The code MA27 has been an important routine within HSL since the early 1980s (although, since the release of the code MA57 in 2000, it has been part of the HSL Archive). MA27 uses the multifrontal algorithm of Duff and Reid (1983) (see also Liu, 1992 for a useful overview of the multifrontal method). During the analyse phase, pivots are selected using the sparsity pattern alone, assuming that the matrix is definite so that all the diagonal entries are nonzero and suitable as 1×1 pivots. By default, a minimum degree algorithm is used for selecting a pivotal sequence that aims to preserve sparsity. The implementation used is able to avoid the worst slow-downs that can result from dense rows. The code also allows the user to specify the pivotal sequence.

During the factorization, the pivot order may be modified to maintain numerical stability by delaying the use of a pivot if it is too small (that is, if it does not satisfy the threshold stability criterion) or by replacing two pivots by a 2×2 block pivot. By this means, MA27 can stably factorize symmetric indefinite problems. For efficiency, if the matrix is known to be positive definite (and thus the pivot order need not be altered), the user can set a parameter in the calling sequence so that a logically simpler path in the code is followed. In our tests using MA27 to solve positive-definite systems, this option is used.

Having computed the factors, a separate solve routine may be called repeatedly for solving for different right-hand sides b. The solve routine uses the factors to solve the linear systems either by loading the appropriate parts of the vectors into a local array and using full matrix code or by indirect addressing at each stage, whichever performs better.

2.3 MA47

For problems with a significant number of nonzeros on the diagonal (such as those that arise from equality constrained least-squares problems and quadratic programming problems), the fill-in during the factorization phase of MA27 can be significantly greater than was predicted during the analyse phase. In an attempt to overcome this problem, MA47 (Duff and Reid, 1996) was designed to use the multifrontal principle but also to follow the sparsity structure of the matrix more closely in the case when some of the diagonal entries are zero. MA47 is thus primarily designed for solving symmetric indefinite systems and, in particular, augmented systems where A is of the form

$$\begin{pmatrix} H & C^T \\ C & 0 \end{pmatrix}. \tag{2.1}$$

The analyse phase chooses diagonal pivots of orders 1 and 2 using a generalization of the Markowitz criterion. The search for pivots can be restricted to a user-specified number of rows. An option also exists to accept a sequence of such pivots supplied by the user. Because of the facility for handling matrices with zeros on the diagonal, the 2×2 pivots can have one or both diagonal entries of value zero. These are termed *tile* and *oxo* pivots, respectively.

The factorization uses the assembly and elimination ordering generated by the analyse phase, but with additional numerical pivoting. This can yield full 2×2 pivots in addition to tile and oxo pivots. Level 3 BLAS are used during the eliminations.

As in MA27, the solve phase uses the to solve systems of equations either by loading the appropriate parts of the vectors into a local array and using full matrix code or by indirect addressing at each stage. This is determined by a parameter that may be selected by the user.

The pivoting strategy is discussed by Duff, Gould, Reid, Scott and Turner (1991). An explanation of the whole algorithm is given by Duff and Reid (1995, 1996).

2.4 MA55

MA55 is a variable-band solver for systems of linear equations whose matrix is symmetric and positive definite. It has advantages for systems that can be preordered so that the bandwidth and profile (the total number of entries between the first entries of the rows and the diagonals) are small. MA55 performs no interchanges and takes advantage of variation in the bandwidth. To minimise in-core storage requirements, the code optionally uses a direct-access file to store the real values of the matrix factor. Use of this option can significantly increase the overall solution time because of the added i/o cost. MA55 uses a reverse communication interface to pass the rows of the matrix to the factorization routine; the user can choose to input any number of rows at once.

For efficiency, MA55 blocks the rows together to allow the use of Level 3 BLAS. The size of the blocks is control by a parameter BSIZE that may be set by the user (the default value is 4). The best choice for the blocking parameter varies with the matrix structure and the relative efficiency of the BLAS on the computer in use. As BSIZE increases, more floating-point operations and storage are needed, since the matrix is held as a set of rectangular blocks, each corresponding to the smallest rectangular submatrix that contains all the lower-triangular entries of the block of rows, including the diagonal. Given BSIZE, the analyse routine computes the number of floating-point operations (flops) that will be needed to factorize the matrix and the real storage required for the matrix factor. Since this routine is inexpensive compared with the total factorization cost, the user may choose to call it for several values of BSIZE and look at the different flop counts and storage requirements. A modest increase in the flop count is likely to be justified by the increase of speed caused by giving larger blocks to the Level 3 BLAS.

MA55 offers an option for solving for one or more right-hand sides b or B at the same time as the numerical factorization. There is also a separate solve routine for solving for further right-hand sides and an additional routine that computes residuals.

Since the efficiency of MA55 depends upon the equations being ordered for a small profile, the user is advised to preorder the matrix prior to calling MA55. HSL offers routine MC60 (Reid and Scott, 1999) for doing this. In our numerical experiments, the time taken for MC60 is included in the "analyse" time for MA55. We remark that, although the reverse communication interface avoids the need to hold the whole matrix, in our experiments it was necessary to store A for the reordering phase. Thus the memory statistics reported on in Section 4 include storage for A.

2.5 MA57

The multifrontal code MA57 was designed by Duff (2002) to supersede MA27 for the solution of symmetric indefinite systems. In addition to being more efficient (partly through its use of the Level 3 BLAS), MA57 has a number of added features. These include: a fast mapping of data prior to a numerical factorization, the ability to modify pivots if the matrix is not definite, the efficient solution of several right-hand sides, a routine implementing iterative refinement, and the possibility of restarting the factorization should it run out of space while retaining the part of the factors computed so far.

By default, the analyse phase chooses pivots from the diagonal using an approximate minimum degree (AMD) algorithm (Amestoy, Davis and Duff, 1996). A version of AMD that avoids some of the problems caused by (nearly) dense rows is also offered (see Duff, 2002 for details); this version is called MC50. Alternatively, the pivot sequence may be chosen as in MA27 using the minimum degree algorithm. An option also exists for the user to supply a pivotal sequence.

The ordering generated by the analyse phase is passed to the factorization. At each stage in the multifrontal approach, pivoting and elimination are performed on full submatrices and, when diagonal 1×1 pivots would be numerically unstable, 2×2 diagonal blocks are used. Thus the code can be used to factor indefinite systems.

As for MA27 and MA47, systems of equations are solved either by loading the appropriate parts of the vectors into a local array and using full matrix code or by indirect addressing at each stage. The user may solve for either a single or multiple right-hand sides. A control parameter (with default value 10) controls the number of columns in a block pivot above which, provided the number of rows in the block is at least 4, Level 2 and Level 3 BLAS are used in the solve phase. An additional routine may be called to perform iterative refinement, using the strategy of Arioli, Demmel and Duff (1989).

There are Fortran 77 and Fortran 90 versions of MA57. HSL_MA57 is a Fortran 90 encapsulation of MA57 but also offers some additional facilities. In our numerical experiments, the Fortran 77 version is used.

2.6 MA62

MA62 (Duff and Scott, 1999) is designed for solving systems of symmetric positive-definite unassembled finite-element equations. The code uses a frontal algorithm (Irons, 1970, Hood, 1976, Duff, 1984) and optionally holds the matrix factor in direct-access files (one file for real data and one for the integer data). Use of direct-access files substantially reduces the main memory requirements but the extra i/o cost can impose a time penalty.

MA62 uses reverse communication to obtain information from the user. This adds to the complexity of the interface but reduces the amount of data that must be held by the user; in particular, the user can choose to generate the data for a single element at a time as it is required. The structure of the problem is first provided by calling a subroutine for each element. The order in which the elements are entered is termed the *assembly* order. These calls establish when variables are *fully summed* (that is, are not involved in any elements still to be assembled) and hence are candidates for use as pivots. A further set of calls to another subroutine computes the size of the files required to hold the factors and the maximum order of the frontal matrix (at each stage, the frontal matrix holds the variables involved in one or more of the elements assembled so far that have yet been eliminated).

In the innermost loop of the numerical factorization, MA62 uses Level 3 BLAS routines. In particular, if at some stage the number of available pivots is K, the Level 3 kernel _GEMM with internal dimension K is used. MA62 allows the user to specify the minimum number of pivots that will be selected at any stage (the default value is 16). Delaying performing eliminations until a block of pivots is available increases the BLAS component of the factorization, albeit at the cost of more floating-point operations, increased storage for the reals in the factor, and, in general, an increase in the in-core storage required. Since the symbolic factorization is cheap to perform, the user may want to examine the effects of varying the minimum pivot block size before starting the numerical factorization.

MA62 offers an option for solving for one or more right-hands side b or B at the same time as the numerical factorization. In this case, the right-hand sides must be entered in element form. There is also a separate solve routine for solving for further right-hand sides. In this case, the user must enter assembled right-hand side vectors. When solving for multiple right-hand sides, Level 3 BLAS are used for the forward eliminations and back substitutions; otherwise, Level 2 BLAS are employed.

As the elements are assembled, the size of the frontal matrix increases whenever a variable appears for the first time and decreases whenever it is eliminated. Thus for the efficiency of the code, the user's element assembly order is critical. The elements need to be ordered to keep the size of the frontal matrices as small as possible. In HSL, we offer a separate element ordering routine MC63 (Scott, 1999) that the user is advised to call prior to using MA62. For large problems, the cost of ordering the elements is generally small compared with the factorization cost. Thus in our experiments we run both the direct and indirect element ordering algorithms offered by MC63, and we run each of these with and without supplying a spectral ordering computed using MC73 (that is, we compute four different element orderings and choose the one with the smallest root-mean-squared wavefront as the assembly order for MA62). The "analyse" time for MA62 reported in our numerical experiments is the total time taken to order the elements plus the time for the symbolic factorization phases described above.

2.7 MA67

Like MA47, MA67 is primarily designed for symmetric indefinite problems and, in particular, for augmented systems. The code uses 1 × 1 and 2 × 2 pivots. Advantage is taken of the extra sparsity available within 2 × 2 pivots when either one or both the diagonal entries are of value zero (tile and oxo pivots). Unlike the other codes discussed in this section, MA67 does not have separate analyse and factorize phases. Instead, the numerical values of the entries are taken into account during the selection of pivots and thus there is a single analyse-factorize routine. The aim is to avoid the problems that have been observed with MA47 when pivots chosen during the analyse phase are subsequently found to be numerically unsuitable during the factorize phase. This can lead to the MA47 factorization being significantly more costly than was anticipated by the analyse phase.

MA67 chooses its pivot sequence using the Markowitz criterion to preserve sparsity. For efficiency, supervariables are exploited and use is made of Level 3 BLAS. MA67 is not a multifrontal code. Instead it follows the sparsity pattern of the matrix at each stage of the elimination. It bases its blocking on rows with identical sparsity patterns and merges blocks whenever fill-ins make the sparsity patterns of two block rows identical. Each block is held separately, with links to allow rapid access to the blocks of a block row. Only one copy of each pair of corresponding off-diagonal blocks is held.

The solve phase uses the factors from the analyse-factorize routine to solve a system of equations either by loading the appropriate parts of the vectors into a local array and using full matrix code or by indirect addressing at each stage. By default, block pivots with more than 4 rows use direct-addressing.

2.8 Summary of the key features of the HSL symmetric solvers

Table 2.1 summarises the key features of the HSL symmetric sparse solvers discussed above. The date that each code was first included within HSL is also given.

Although each of the codes may be used to solve positive-definite problems, some have an option that allows the user to indicate that the matrix is positive definite and, in this case, the code follows a logically simpler path. A '$\surd$' in the column headed 'Positive definite' indicates that the code either has such an option or is designed specifically for

Code	Fortran version	Positive definite	Indefinite	Element entry	Out-of -core option	Separate analyse+ factorize	Multiple rhs
MA27 (1982)∗	77	√	√	×	×	√	×
MA47 (1993)∗	77	×	√	×	×	√	×
MA55 (1999)	90	√	×	×	√	√	√
MA57 (2000)	77/90	√	√	×	×	√	√
MA62 (1997)	77	√	×	√	√	√	√
MA67 (2001)	77	×	√	×	×	×	×

Table 2.1: Summary of the key features of the HSL symmetric sparse direct-solvers. A ∗ indicates the code is part of HSL Archive; the remaining codes are in HSL 2002.

positive-definite systems. Note that MA55 and MA62 perform no numerical pivoting and so are generally unsuitable for the solution of indefinite problems.

We observe that only the positive-definite solvers MA55 and MA62 offer the option of holding the matrix factor out-of-core. We anticipate that this facility will allow the solution of problems that are too large for the other codes to successfully solve with the memory available in our test environment. Note also that MA62 is only designed for the solution of finite-element problems in unassembled form. The HSL routine MC37 may be used to split a sparse symmetric matrix in assembled form into a (non-unique) set of element matrices that, if assembled, would yield the same matrix. We use MC37 to enable MA62 to be run on the complete positive-definite test set.

3 The test environment

3.1 The test set

Our aim in this study is to test the solvers on a wide range of test problems from as many different application areas as possible. In collecting test data we imposed only two conditions:

- The matrix must be of order greater than 10, 000.
- The data must be available to other users.

The first condition was imposed because our interest is in large problems. The second condition was to ensure that our tests could be repeated by other users and, furthermore, it enables other software developers to test their codes on the same set of examples and thus to make comparisons with HSL solvers. Provided the above conditions are satisfied, we have included all square real symmetric matrices of order exceeding 10, 000 from Matrix Market (math.nist.gov/MatrixMarket/), the Harwell-Boeing Collection (Duff, Grimes and Lewis, 1989), and the University of Florida Sparse Matrix Collection (www.cise.ufl.edu/∼davis/sparse/). The test set comprises 88 positive-definite problems and 61 numerically indefinite problems. Of these matrices, those of order 50,000 or more are further classed as being in the *subset* of larger examples (there are 43 positive-definite and 30 indefinite examples in this category). Any matrix for which we only have the sparsity pattern available is included in the positive-definite set, and appropriate numerical values have been generated (see Section 3.6). Application areas represented by our test set include linear programming, structural engineering, computational fluid dynamics, acoustics, and financial modelling. A full list of the test problems together with a brief description of each is given in Gould and Scott (2003).

3.2 The performance profile

Benchmark results are generated by running a solver on a set $\mathcal{T}$ of problems and recording information of interest such as the computing time and memory used. In this study, we use a performance profile as a means to evaluate and compare the performance of our HSL solvers on our test set $\mathcal{T}$.

Let $\mathcal{A}$ represent the set of algorithms that we wish to compare. Suppose that a given algorithm $i \in \mathcal{A}$ reports a statistic $s_{ij} \geq 0$ when run on example j from the test set $\mathcal{T}$, and that the smaller this statistic the better the algorithm is considered to be. For example, s_{ij} might be the CPU time required to solve problem j using algorithm i. For all problems $j \in \mathcal{T}$, we want to compare the performance of algorithm i with the performance of the best algorithm in the set $\mathcal{A}$.

For $j \in \mathcal{T}$, let $\hat{s}_j = \min\{s_{ij}; i \in \mathcal{A}\}$. Then for $\alpha \geq 1$ and each $i \in \mathcal{A}$ we define

$$k(s_{ij}, \hat{s}_j, \alpha) = \begin{cases} 1 & \text{if } s_{ij} \leq \alpha \hat{s}_j \\ 0 & \text{otherwise.} \end{cases}$$

The *performance profile* (see Dolan and Moré, 2002) of algorithm i is then given by the function

$$p_i(\alpha) = \frac{\sum_{j \in \mathcal{T}} k(s_{ij}, \hat{s}_j, \alpha)}{|\mathcal{T}|}, \quad \alpha \geq 1.$$

Thus $p_i(1)$ gives the fraction of the examples for which algorithm i is the most effective (according to the statistic s_{ij}), $p_i(2)$ gives the fraction for which algorithm i is within a factor of 2 of the best, and $\lim_{\alpha \longrightarrow \infty} p_i(\alpha)$ gives the fraction for which the algorithm succeeded.

In this study, the statistics used are:

- The CPU times required to perform the analyse, factorize, and solve phases.
- The number of nonzero entries in the matrix factor.
- The total memory used by the solver.

3.3 Computing platform

The numerical results were obtained on a Compaq DS20 Alpha server with a pair of EV6 CPUs; in our experiments only a single processor with 3.6 GBytes of RAM was used. We compiled the Fortran 77 and Fortran 90 codes with full optimisation; the vendor-supplied BLAS were used. All CPU reported times are in seconds and, where appropriate, include all i/o costs involved in holding the factors in direct-access files. A CPU limit of 2 hours was imposed for each code on each problem; any code that had not completed after this time was recorded as having failed.

In all the experiments, double precision reals were used. Thus storage for a real was 8 bytes and for an integer was 4 bytes. The reported statistics for memory are all in Mbytes.

3.4 Control parameters

Each of the HSL sparse solvers used in our numerical experiments has a number of control parameters that are assigned default values through a call to an initialisation subroutine. Unless otherwise stated, we use these defaults in each case, even if different codes sometimes choose a different value for essentially the same parameter. The exceptions are the blocking parameters that control the use of high level BLAS (note that MA27 is the only code that does not employ Level 3 BLAS) and the stability threshold parameter u.

We performed a number of preliminary experiments to determine a suitable choice for the blocking parameters in our test environment and, based on our findings, in our reported numerical experiments these parameters are always set to 16.

When testing the solvers on positive-definite problems the threshold parameter u is set to zero. This results in no numerical pivoting being performed. For our tests on numerically indefinite problems, we run both with the code's default u value and with u set to 10^{-10}. Such a value is frequently used in optimization applications (Saunders, 1994, Gould and Toint, 2002), where speed is of the essence, and any instability is countered either by iterative refinement or ultimately by refactorization with a larger value of u.

The codes MA27 and MA67, as well as the unsymmetric solver MA48, use the default $u = 0.1$ while MA57 and MA47 use $u = 0.01$ and $u = 0.001$, respectively.

We remark that MA47 has a control parameter ICNTL(4) which, if set to a value greater than one, limits the search for a pivot to ICNTL(4) rows. If ICNTL(4) is set to zero, the number of rows searched is unrestricted (the Markowitz criterion is used). Previously, the default value was zero. However, experimentation has shown that restricting the pivot search to a small number of rows can substantially reduce the analyse time (possibly at the cost of a modest increase in the number of entries in the factors). As a result, the default value is now 10, and this value was used in our tests.

3.5 Use of direct-access files

As already mentioned, the option of holding the matrix factor using direct-access files available in MA55 and MA62 can add a significant i/o overhead, particularly to the solve phase. We illustrate this with test problem tsyl201. For this example, the MA62 factorization and solve times for a single right-hand side without using direct-access files are 11.7 and 0.27 seconds, respectively. If files are used, the corresponding times are 12.5 and 1.1 seconds. Thus in our tests, we only use direct-access files if there is insufficient memory for the code to run in memory.

For MA55, the user can optionally specify the number of real values to be output in each direct-access record. In our tests, we use the default value of $2 * n$ (where n is the order of A). MA62 requires the user to specify the number of reals and integers to be output in each direct access record. We use values of 500,000 (or if the number of reals and integers required to hold the matrix factor are found by the analyse phase to be less than this, the values from the analyse phase are used). MA62 requires integer and real workspaces that are at least the length of a single record in the direct-access file. Thus memory requirements can be reduced by reducing the record length.

3.6 Numerical values and scaling

Some of our test examples are not supplied with numerical values (only the sparsity pattern is available). For these cases, appropriate numerical values are generated. Reproducible pseudo-random off-diagonal entries in the range $(0, 1)$ are generated using the HSL routine FA14, while the i-th diagonal entry is set to $\max(100, 10\rho_i)$, where ρ_i is the number of off-diagonal entries in row i of the matrix, thus ensuring that the generated matrix is numerically positive definite.

In all our tests, right-hand side vectors b are computed so that the exact solution x (of the unscaled system) is $x = (1, 1, ..., 1)^T$.

If the input matrix has entries differing widely in magnitude, then an inaccurate solution may be obtained in the indefinite case and the accuracy may be difficult to assess in all cases. We advise HSL users to first employ MC30 to obtain scaling factors for the matrix and then explicitly scale the matrix prior to calling their chosen solver. To examine the effects of scaling on our test examples, for each value of the threshold parameter u used in the tests, we run both with and without scaling of the matrix A and the corresponding right-hand side b. For our positive-definite problems, scaling was found to make an insignificant difference and hence we report on the effects of scaling only for the indefinite examples.

3.7 Residuals and iterative refinement

A number of HSL solvers (including MA57) include routines for automatically performing iterative refinement. We have not used these routines in this study. Instead, once we have computed the approximate solution x, we perform one step of iterative refinement by computing the residual $r = Ax - b$ and then recalling the solve routine to solve $A\delta x = r$ for the correction δx.

For each right-hand side b and corresponding solution x, we compute the 2-norm of scaled residual

$$\|b - Ax\|/(\|A\|\|x\| + \|b\|)$$

A check is made that this residual is sufficiently small. If not, the solver is regarded as having failed to solve the problem correctly. Note that the residual of the unscaled system is computed.

4 Results

The full results are available as an internal technical report (Gould and Scott, 2003).

Since some of the solvers we are examining are specifically designed for positive-definite problems (and may be unreliable, or even fail, on indefinite ones), we will discuss the positive-definite and indefinite cases separately.

4.1 Positive-definite examples

4.1.1 General considerations

The reliability of all six solvers in the positive-definite case was generally high. Only audikw was not solved by any code, this example being one of the two largest—it is of order roughly 950 thousand, and involves some 39 million nonzeros; no solver was able to allocate sufficient memory. Additionally, three of the other large problems, in_line, ldoor and TROLL, were not solved by MA55, either because the CPU time limit was exceeded or there was a memory allocation error (which could not be avoided through the use of files for the factors). MA62 exceeded the CPU time limit for the three gupta examples and for problems audikw and in_line there was insufficient memory to preprocess the matrix (that is, to split the problem into elements using MC37. The CPU time limit was exceeded by MA27, MA55, and MA67 for the gupta2 problem.

We present the performance profile for the CPU time for the complete solution (that is, the CPU time for analysing, factorising and solving for a single right-hand side) for the six solvers in Figure 4.1.

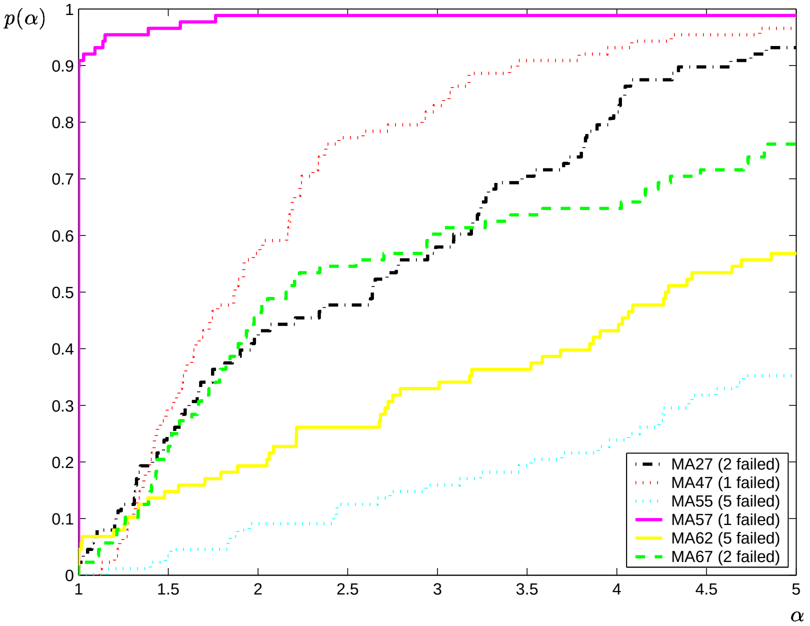

Figure 4.1: Performance profile, $p(\alpha)$: CPU time (seconds) for the complete solution (positive-definite problems).

It is immediately apparent how much better MA57 is than its rivals, being best in 90% of the cases, and within a factor of 2 of the best in every case. Of the other solvers, MA47 is within a factor of 2 of the best in roughly 60% of the cases, while MA67 and MA27 achieve the same level in around 50% of the cases. The out-of-core solvers perform less well but, as already noted, these codes are designed to solve special classes of problems (MA55 is for banded systems and MA62 for finite-element applications); they compare unfavourably with the other solvers in terms of CPU time when run on our general test set.

MA62 performed poorly on problems that were artificially split into elements—it appears that using MC37 generates a large number of very small elements. If we assemble FE problems and then split into elements with MC37, MA62 performs less well than if run on element data. Furthermore, for the largest problems the memory available was insufficient to run MC37. The number of problems for which the out-of-core solvers MA55 and MA62 used files to store their factors was relatively small (one and eight, respectively).

For the subset of large problems, Figure 4.2 reinforces the apparent superiority of MA57. Notice now that MA47 and MA67 outperform MA27, this being almost entirely due to the former using high-level BLAS.

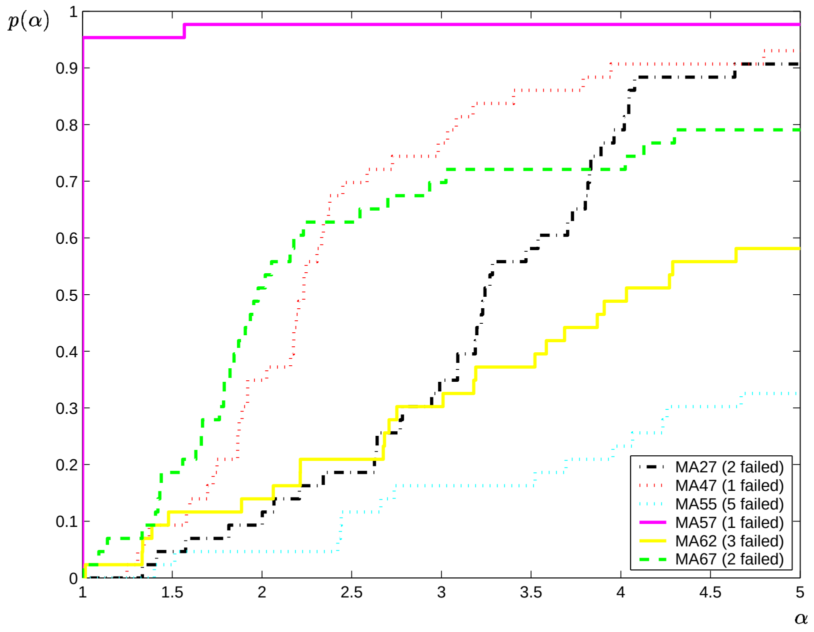

Figure 4.2: Performance profile, $p(\alpha)$: CPU time (seconds) for the complete solution (large positive-definite subset problems).

A close examination of the separate analyse, factorize and solve times for the different solvers (see Gould and Scott, 2003) shows that the potentially less-expensive AMD analysis ordering used by MA57 is indeed faster than the analyse phases of MA27 and MA47. Perhaps more surprisingly, the factorization time gains are even more pronounced. This might be attributable to the use of Level 3 BLAS, although the same advantages should then be shared by MA47 and MA67—the latter codes do actually perform slightly better than the BLAS-free MA27, but not significantly so. Thus one can only surmize that MA57 must be performing less work, and that this is likely because the MA57 factors are actually sparser. This is confirmed in Figure 4.3. Given the factorizations, the solution times for the solvers is less significantly different, although MA57 and MA27 are slightly faster than their competitors. When it comes to overall memory use, Figure 4.4 indicates that MA27 is generally the most frugal, although MA57 and MA47 are not far behind. However, note that the memory used by MA55 and MA62 is dependent upon the choice of the length of the in-core work arrays; reducing these so that more problems are solved using files for

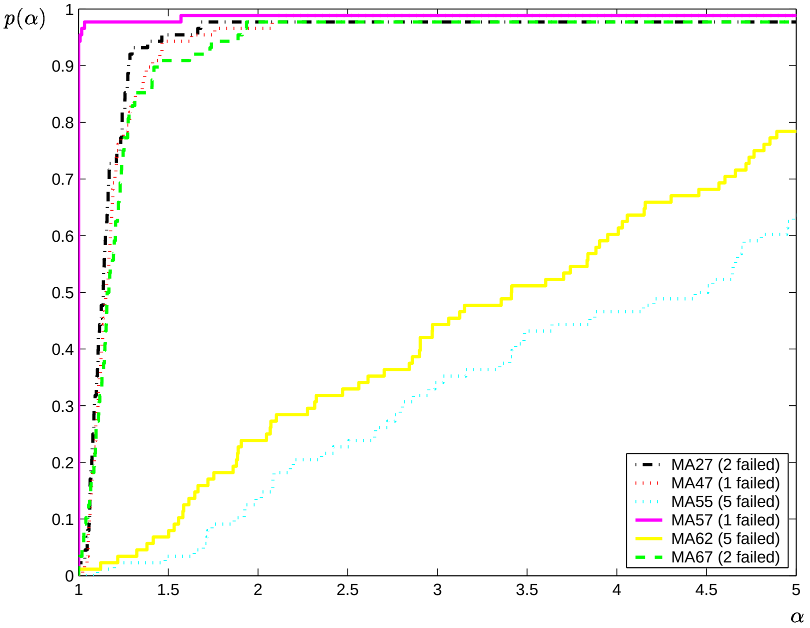

Figure 4.3: Performance profile, $p(\alpha)$: Number of entries in the factors (positive-definite problems).

the factors would make MA55 and MA62 much more competitive in terms of memory. This

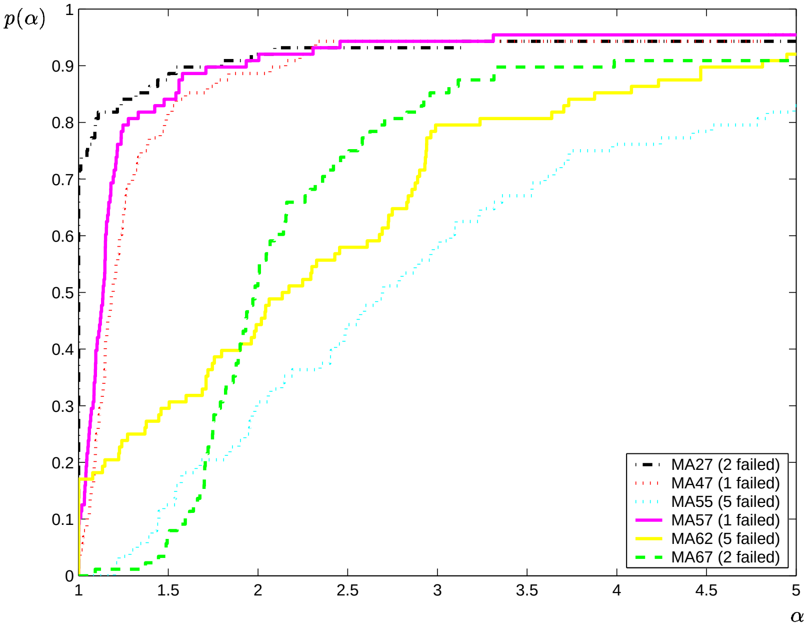

Figure 4.4: Performance profile, $p(\alpha)$: Memory used (positive-definite problems).

is seen by considering the *minimum* memory used. If minimum memory is the overriding concern, Figure 4.5 shows that out-of-core solvers and, in particular MA62, are preferable—

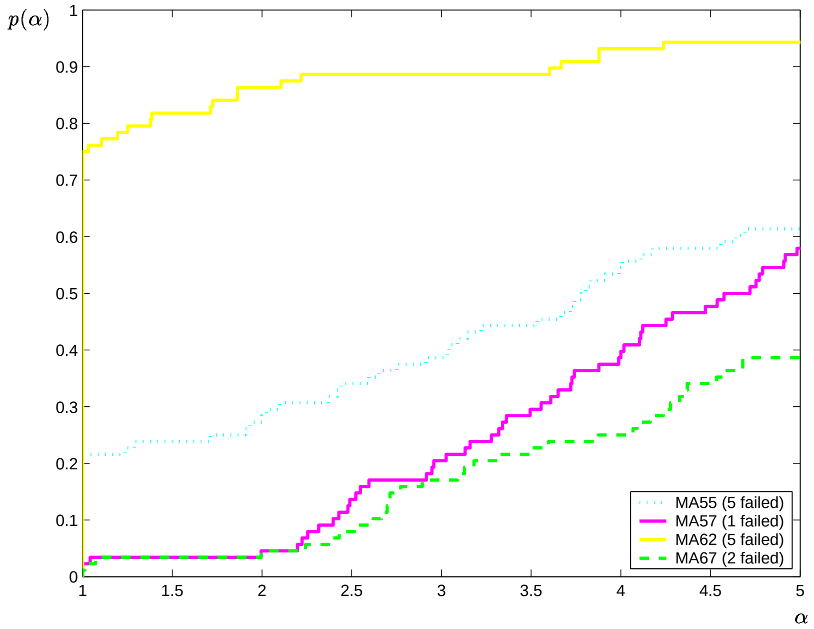

Figure 4.5: Performance profile, $p(\alpha)$: Minimum memory required (positive-definite problems).

note here that data on the minimum memory needed is not available for the older solvers (MA27 and MA47).

4.1.2 Improved orderings for MA57

Since the figures presented in the previous section strongly suggest that MA57 is the best general-purpose HSL solver for positive-definite problems (at least in terms of CPU times required), it is of interest to see if we can further improve its performance. One way to do this might be to replace the AMD ordering generated in the analysis phase. With this in mind, we have considered three alternatives.

The first possibility is to use the multi-level graph partitioning ordering generated by the METIS package (Karypis and Kumar, 1999). In particular, we use routine METIS_NodeND, which aims to compute fill-reducing orderings for sparse matrices using a multilevel nested dissection algorithm. The package is written in C; we compiled it with full optimization and used the facility offered by MA57 which allows the code to accept externally-supplied pivot orderings. We denote this option MA57_METIS.

Secondly, we consider the minimum degree ordering computed by the analyse phase of MA27. This is provided as an option within MA57 itself. We denote this option MA57_MA27.

Finally, we have tried a special version of the AMD ordering (soon to be released in HSL as MC50, see Duff, 2002), in which precautions are taken to ensure that (close to) dense rows do not dominate the local search for the approximate minimum degree. We refer to this as MA57_MC50. Currently, MC50 is supplied as a private sub-package of MA57, and called directly as an option by the latter.

We show the performance profiles for MA57 resulting from these four different ordering strategies in Figures 4.6 and 4.7.

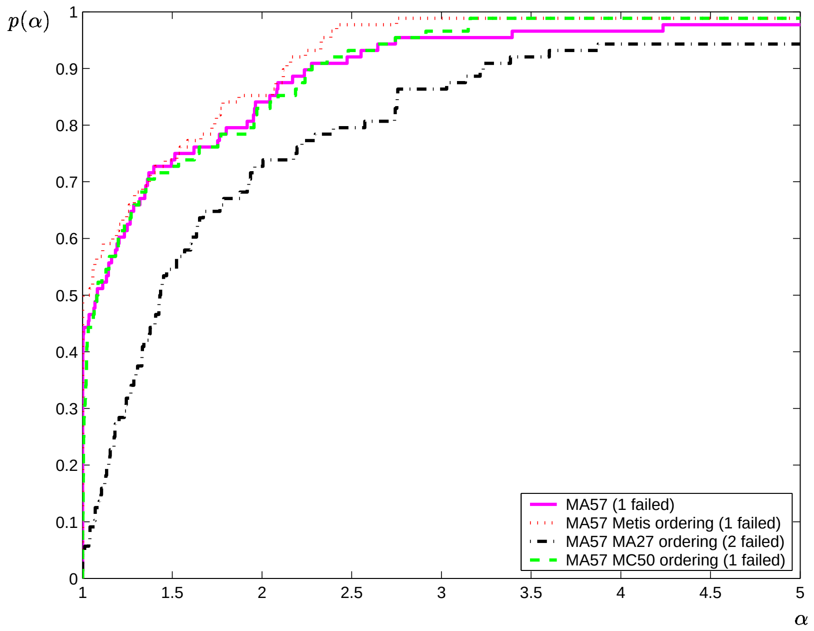

Figure 4.6: Performance profile, $p(\alpha)$: CPU time (seconds) for the complete solution for MA57 with different analysis orderings (positive-definite problems).

For the complete test set, there is little to choose between the default MA57 ordering and the MA57_METIS and MA57_MC50 alternatives. Only MA57_MA27 seems to perform poorly. This seems odd since one naively might expect that the minimum degree ordering should produce a better ordering than the cheaper, but less thorough, AMD alternative. However this trend has already been reported on by, for example, Amestoy et al. (1996). Certainly, detailed examination of the data in Gould and Scott (2003) reveals that both the number of nonzeros in the factors and the total memory required for the solution are significantly lower with the AMD ordering.

For the subset of large problems the nested-dissection METIS ordering is superior. There is still little to choose between the default MA57 and MA57_MC50 orderings, with MA57_MA27 running a poor last. This strongly suggests that a nest-dissection ordering should be provided as an option within any future evolution of the MA57 package.

4.2 Indefinite examples

4.2.1 General considerations

We now turn to the indefinite test suite. Of course, pivoting must now be performed for stability as well as for sparsity reasons. Since MA55 and MA62 were not designed for indefinite problems, they were omitted from these tests.

The overall reliability in the indefinite case was not as high as for the positive-definite one. All of the codes either exceeded the available memory or the two-hour CPU time

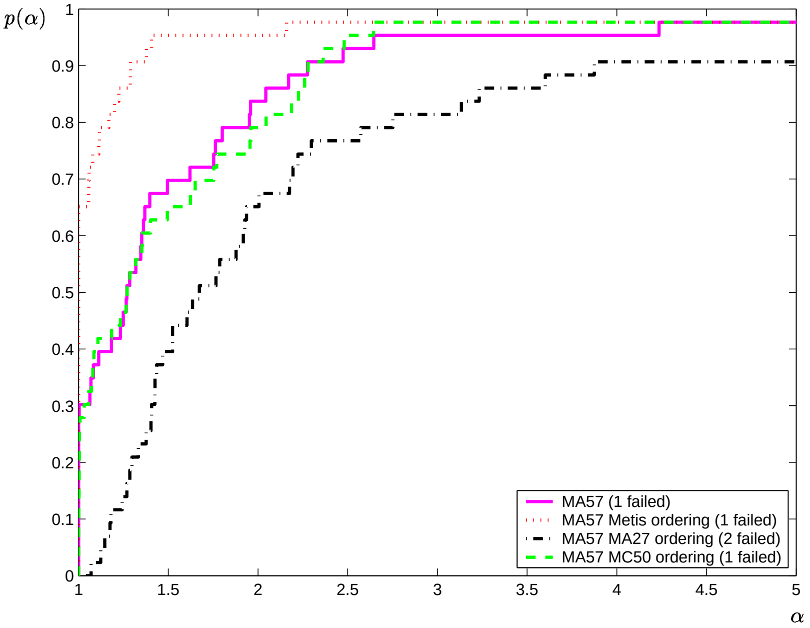

Figure 4.7: Performance profile, $p(\alpha)$: CPU time (seconds) for the complete solution for MA57 with different analysis orderings (large positive-definite subset problems).

limit for problems BOYD1, BOYD2, NCVXQP3, NCVXQP5, NCVXQP7 and SPARSINE; further experimentation indicated that, had the time limit not been exceeded, the memory would ultimately have been exhausted. Although BOYD2 is the largest indefinite matrix in our set, the remainder are not particularly large, and it appears that the difficulty is simply that numerical pivoting during the factorization leads to significant fill-in, even when a reasonable ordering on sparsity grounds has been found during the analyse phase. Problem BLOWEYA was solved only by MA67, and even here the CPU time was large. MA57 appeared to struggle with the singular examples AUG2D and AUG2DC, which were easily solved by the remaining packages, particularly MA47 and MA67. The other singular example, DTOC defeated both MA27 and MA57. Other problems that were not solved by individual packages were BRATU3D, CONT-201 and NCVXQP1 (MA47), CONT-300 (MA47 and MA67, the latter failing to produce an accurate answer for the unscaled problem without iterative refinement), and c-62, c-68 and c-71 (MA67).

We present the performance profile for the overall CPU times for the full test set and the subset of large problems in Figures 4.8 and 4.9. These are for the default settings and without scaling. Once again MA57 appears to be the most effective solver when it succeeds but, as suggested above, the package is far from foolproof. Many of the examples in the test set are from (constrained) optimization or partial differential equation applications, so have the “augmented matrix” structure. Thus one would hope that MA47 and MA67 would be particularly appropriate here. Unfortunately, this really does not appear to be the case. It appears that once again, the good sparsity ordering found by the analyse phase of MA47 is severely compromised by stability needs during the factorization, the resulting factors being generally denser than for MA57 (see Figure 4.10). Likewise, the attempt to combine the analyse and factorize phases to overcome this deficiency in MA67 does not seem to pay off, this package being the least effective in general, especially for the largest problems.

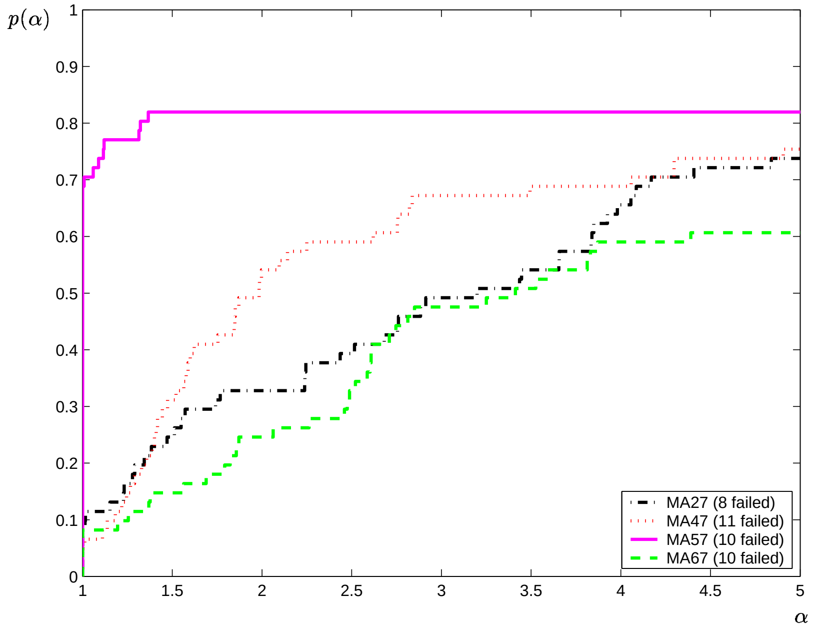

Figure 4.8: Performance profile, $p(\alpha)$: CPU time (seconds) for the complete solution (indefinite problems).

The total memory used by the solvers is illustrated in Figure 4.11. MA27, MA47 and MA57 appear to have broadly similar memory requirements, but MA67 seems to require significantly more memory.

4.2.2 Improved orderings for MA57

Once again, we believe that MA57 is the best general-purpose HSL solver in the indefinite-definite case (at least in terms of CPU times required), and so we consider the effects of the alternative ordering strategies described in Section 4.1.2. We illustrate the effects of the four alternatives employed in Figures 4.12 and 4.13.

By contrast to the positive-definite case, the METIS nested dissection ordering does not seem to be the best in the indefinite case—it actually gives the lowest complete solution times overall. We conjecture that this is yet another example of a good sparsity ordering being overruled by stability considerations. The default ordering is often the fastest, but there is little to choose between it and the MA57_MC50 option.

4.2.3 Other improvements

As we have seen, stability considerations are paramount in the indefinite case. We now examine if we are being unduly cautious with our pivoting strategy.

We consider four variants of threshold pivoting (see, Duff et al., 1986, §5.4). The first (default) is with the threshold parameter, u, set by the solver (this is $u = 0.1$ for MA27 and

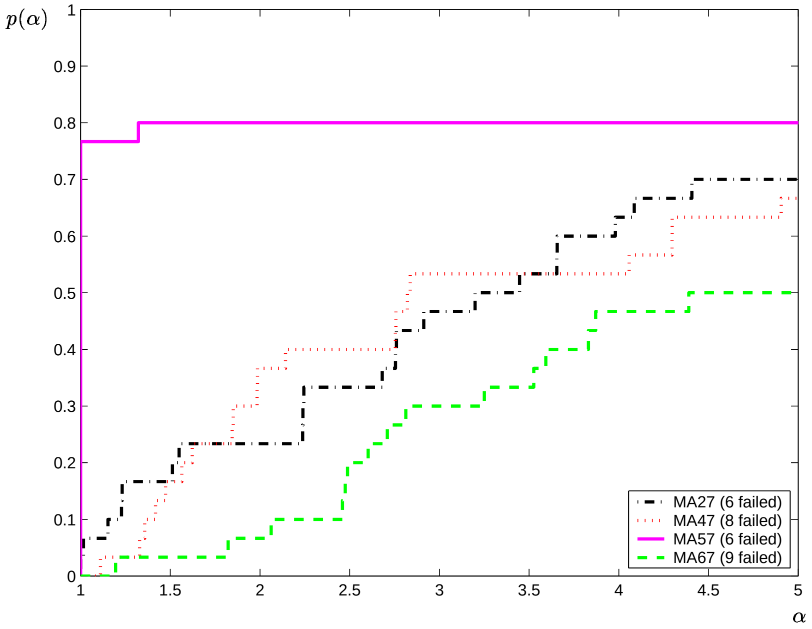

Figure 4.9: Performance profile, $p(\alpha)$: CPU time (seconds) for the complete solution (large indefinite subset problems).

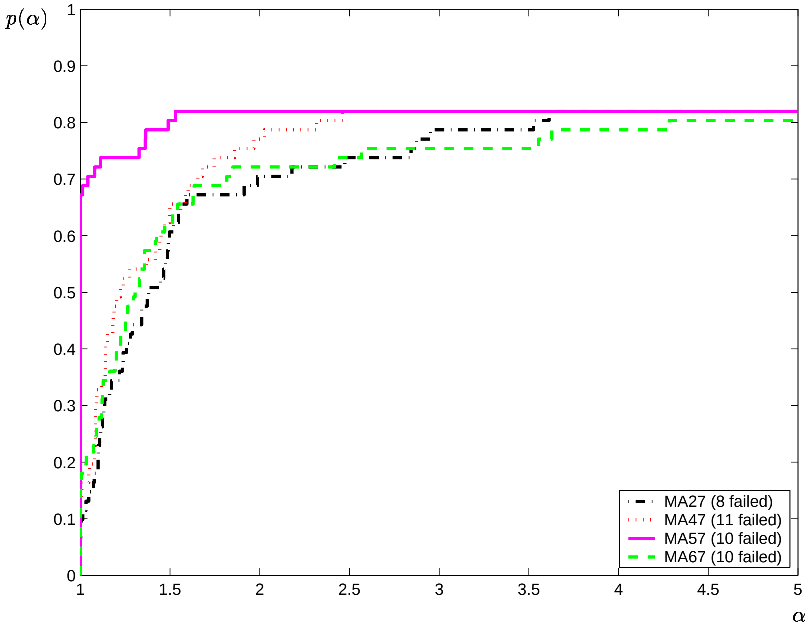

Figure 4.10: Performance profile, $p(\alpha)$: Number of entries in the factors (indefinite problems).

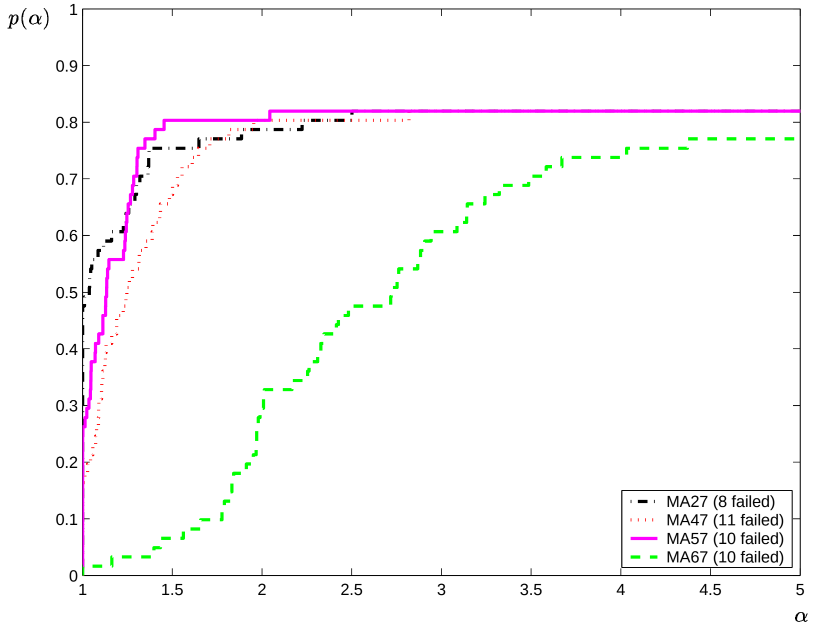

Figure 4.11: Performance profile, $p(\alpha)$: Memory used (indefinite problems).

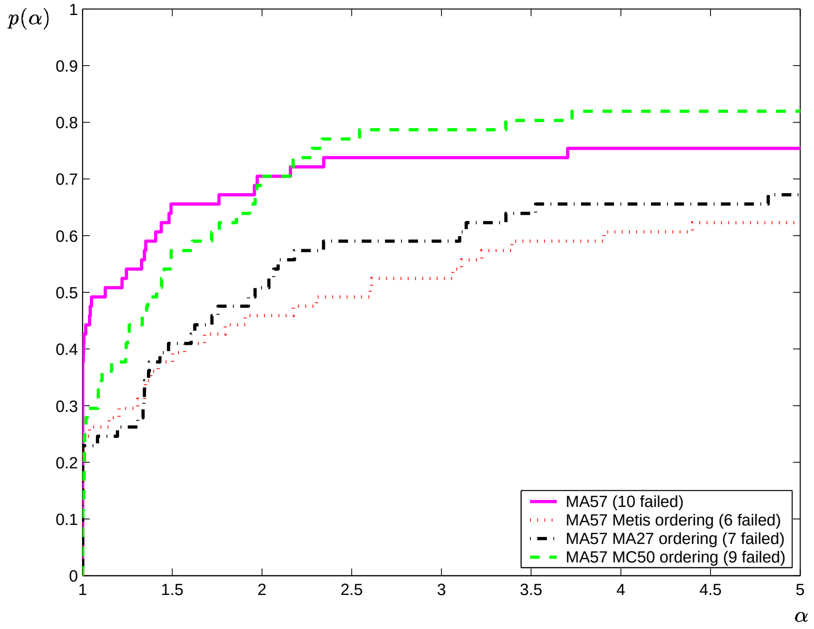

Figure 4.12: Performance profile, $p(\alpha)$: CPU time (seconds) for the complete solution for `MA57` with different analysis orderings (indefinite problems).

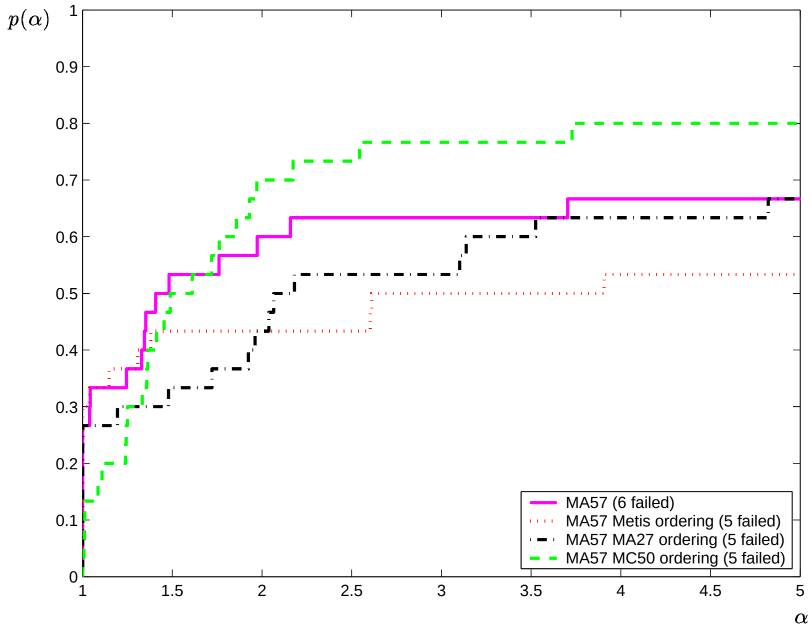

Figure 4.13: Performance profile, $p(\alpha)$: CPU time (seconds) for the complete solution for MA57 with different analysis orderings (large indefinite subset problems).

MA67, $u = 0.01$ for MA57, and $u = 0.001$ for MA47). The second is to use a tiny threshold parameter $u = 10^{-10}$. This has the potential to allow the code to adhere more closely to the ordering suggested by the analyse phase, but also allows the possibility of large growth. The third and fourth strategies are to pre-scale the matrix (using MC30) to try to equilibrate the entries prior to factorization, and then to use threshold pivoting (with default u and $u = 10^{-10}$, respectively) on the resulting scaled system. In Figure 4.14–4.16 we illustrate the effect of the last three of these strategies on our four solvers.

Comparing the profile Figure 4.14 with that in Figure 4.8, we see that (with no scaling) using a tiny pivot tolerance reduces the total number of failures. However, closer examination reveals that for a few problems, the solvers fail with a tiny u because the residuals are too large. For tiny u, MA57 is again the fastest code overall (albeit with the largest number of failures) but there is little to differentiate the other solvers. For given u, scaling the problems also reduces the number of failures (and there are now no failures resulting from the residuals being too large). When scaling is combined with a tiny pivot tolerance the most reliable code (that is, the code with the least number of failures) is MA67.

Finally, in Figures 4.17 and 4.18 we compare all four pivoting options using our leading solver MA57.

We expressed concern in Section 3.7 over the potential for instability when using small pivot tolerances. Our runs with small u show that this is an issue for a small number of our test examples, although reliability is improved by scaling. This reinforces the findings of Duff (2002). We note also that practitioners in optimization frequently find iterative refinement both beneficial and occasionally essential (see, for example, Gould and Toint, 2002). As a result, we believe that, at the very least, residuals should always be computed.

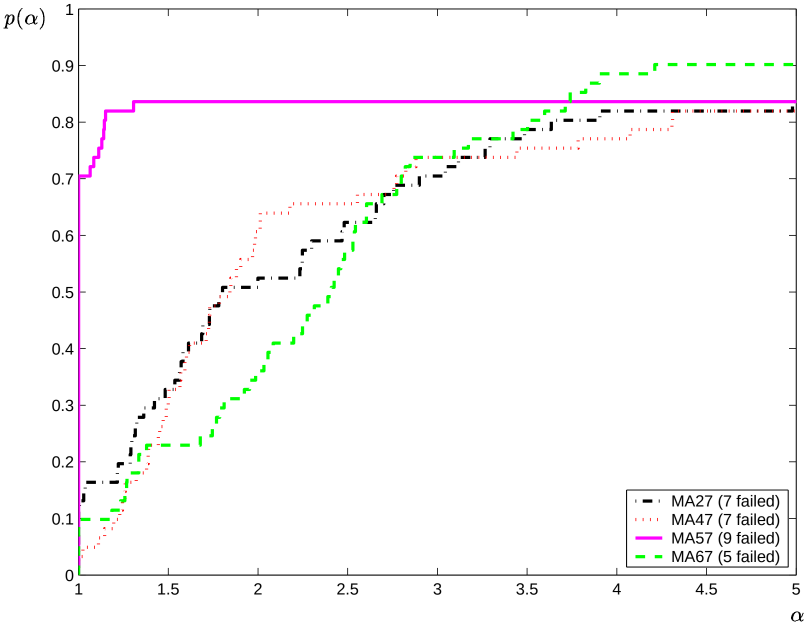

Figure 4.14: Performance profile, $p(\alpha)$: CPU time (seconds) for the complete solution using the tiny threshold pivoting parameter $u = 10^{-10}$ (indefinite problems).

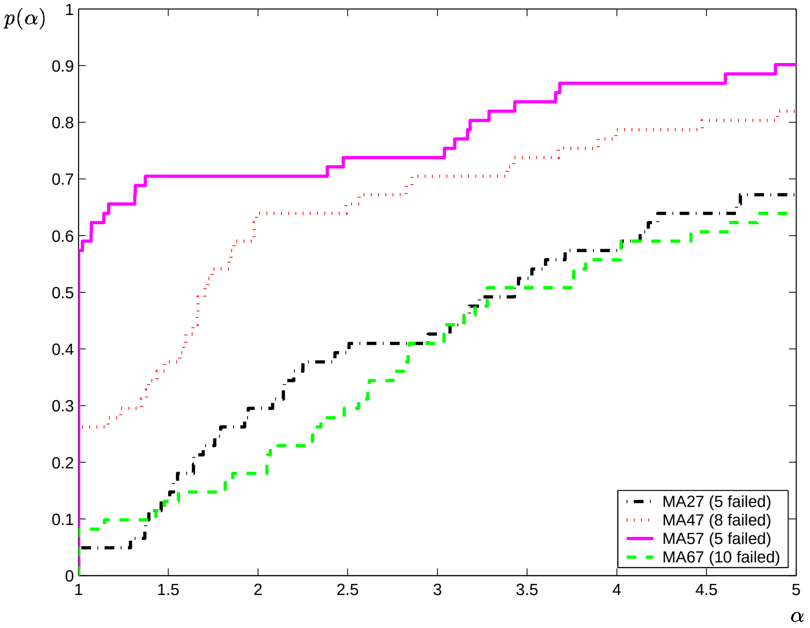

Figure 4.15: Performance profile, $p(\alpha)$: CPU time (seconds) for the complete solution after MC30 scaling (indefinite problems).

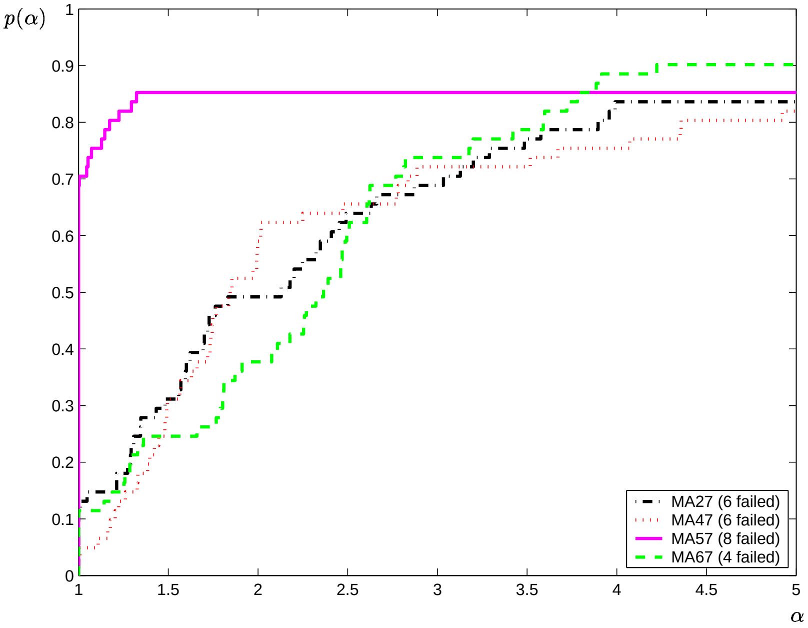

Figure 4.16: Performance profile, $p(\alpha)$: CPU time (seconds) for the complete solution after MC30 scaling and using the tiny threshold pivoting parameter $u = 10^{-10}$ (indefinite problems).

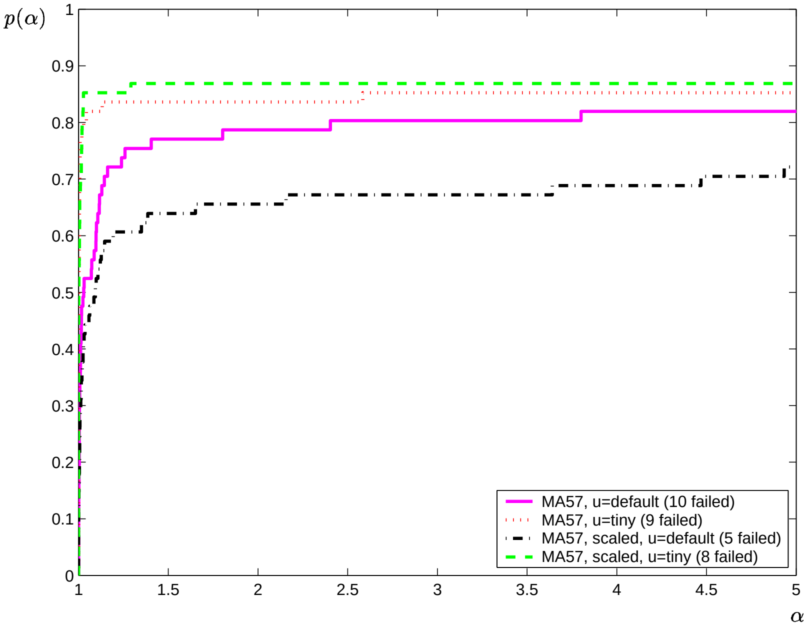

Figure 4.17: Performance profile, $p(\alpha)$: CPU time (seconds) for the complete solution for MA57 with different scaling and pivoting options (indefinite problems).

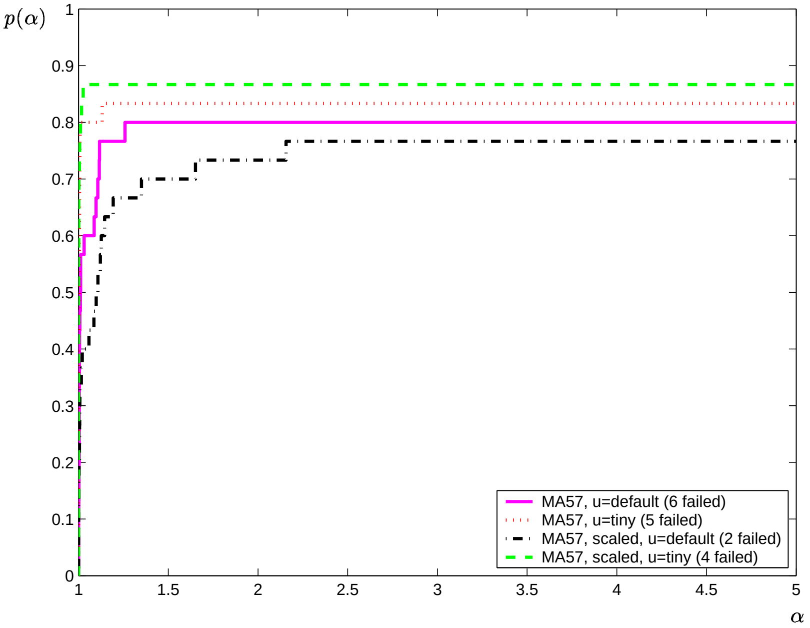

Figure 4.18: Performance profile, $p(\alpha)$: CPU time (seconds) for the complete solution for MA57 with different scaling and pivoting options (large indefinite subset problems).

If large residuals cannot be cured simply through refinement, remedial action should be taken, such as increasing the pivot tolerance and refactorizing the matrix,

5 Comments and conclusions

We summarize our conclusions as follows.

- MA57 is usually the fastest HSL package for solving a system of symmetric linear equations in both the positive-definite and indefinite cases. This conclusion is pronounced for matrices of very large order. The use of Level 3 BLAS clearly pays dividends, and the approximate minimum-degree (AMD) ordering in preference to the traditional minimum-degree one proves to be more effective.

- For large, positive-definite problems, it is advantageous to use MA57 with the nested dissection METIS ordering, rather than a minimum-degree ordering. We believe that this should be an option in any future evolution of the package.

- For indefinite problems, the METIS ordering is not as effective as the default AMD ordering, nor as the MC50 variant of AMD. The MC50 variant offers more protection against significantly dense rows.

- The out-of-core solver MA62 uses the least memory, but the current out-of-core solvers are not competitive with the in-core ones with respect to CPU time. Given the success of the in-core version of MA57, we believe that the development of an out-of-core version of this solver should be a high priority, especially for very large (possibly indefinite) systems.

- For indefinite problems, a tiny pivot tolerance is often better than the default. In addition, scaling using MC30 offers significant improvements. We suggest that the default pivot tolerance should be significantly smaller than it currently is in MA57.
- The robustness of solvers in the large, indefinite case is still a concern. Four of our 61 test problems were not solved by any HSL code.

Our next task will be to compare MA57 with direct solvers from other sources. Tests in this direction are currently underway, and this will be the subject of a future report.

Acknowledgements

We would like to thank our colleagues Iain Duff and John Reid at the Rutherford Appleton Laboratory for answering our queries on their codes and for commenting on a draft of this paper. We are also grateful to Yifan Hu of Wolfram Research for providing us with a C code to measure the memory used by the solvers. Our thanks also to those who supplied test problems, including Mario Arioli, Christian Damhaug, Tim Davis, Anshul Gupta, Alison Ramage, Olaf Schenk, Miroslav Tuma, and Andy Wathen.

References

P. R. Amestoy, T. A. Davis, and I. S. Duff. An approximate minimum degree ordering algorithm. *SIAM J. Matrix Analysis and Applications*, **17**, 886–905, 1996.

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

E. D. Dolan and J. J. Moré. Benchmarking optimization software with performance profiles. *Mathematical Programming*, **91**(2), 201–213, 2002.

J. J. Dongarra, J. DuCroz, I. S. Duff, and S. Hammarling. A set of Level 3 Basic Linear Algebra Subprograms. *ACM Trans. Mathematical Software*, **16**(1), 1–17, 1990.

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

I. S. Duff. MA57– a new code for the solution of sparse symmetric definite and indefinite systems. Technical Report RAL-TR-2002-024, Rutherford Appleton Laboratory, 2002.

I. S. Duff and J. K. Reid. The multifrontal solution of indefinite sparse symmetric linear systems. *ACM Transactions on Mathematical Software*, **9**, 302–325, 1983.

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

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

I. S. Duff and J. A. Scott. A frontal code for the solution of sparse positive-definite symmetric systems arising from finite-element applications. *ACM Trans. Mathematical Software*, **25**, 404–424, 1999.

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

I. S. Duff, N. I. M. Gould, J. R. Reid, J. A. Scott, and K. Turner. Factorization of sparse symmetric indefinite matrices. *IMA Journal of Numerical Analysis*, **11**, 181–2044, 1991.

I. S. Duff, R. G. Grimes, and J. G. Lewis. Sparse matrix test problems. *ACM Trans. Mathematical Software*, **15**, 1–14, 1989.

N. I. M. Gould and J. A. Scott. Complete results from a numerical evaluation of hsl packages for the direct-solution of large sparse, symmetric linear systems of equation. Numerical Analysis Group Internal Report 2003-2, Rutherford Appleton Laboratory, Chilton, Oxfordshire, England, 2003.

N. I. M. Gould and Ph. L. Toint. An iterative working-set method for large-scale non-convex quadratic programming. *Applied Numerical Mathematics*, **43**(1–2), 109–128, 2002.

A. Gupta. Recent advances in direct methods or solving unsymmetric sparse systems of linear equations. *ACM Transactions on Mathematical Software*, **28**(3), 301–324, 2002.

P. Hood. Frontal solution program for unsymmetric matrices. *Inter. Journal on Numerical Methods in Engineering*, **10**, 379–400, 1976.

HSL. A collection of Fortran codes for large-scale scientific computation, 2002. See http://hsl.rl.ac.uk/.

B. M. Irons. A frontal solution program for finite-element analysis. *Inter. Journal on Numerical Methods in Engineering*, **2**, 5–32, 1970.

G. Karypis and V. Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM Journal on Scientific Computing*, **20**(2), 359–392, 1999.

J. W. H. Liu. The multifrontal method for sparse matrix solution: theory and practice. *SIAM Review*, **34**, 82–109, 1992.

J.K. Reid and J.A. Scott. Ordering symmetric sparse matrices for small profile and wavefront. *Inter. Journal on Numerical Methods in Engineering*, **45**, 1737–1755, 1999.

M. A. Saunders. Sparse matrices in optimization, 1994. Presented at Sparse Days at St Girons, International meeting on Sparse Matrix Methods, St Girons, France. See http://www.stanford.edu/group/SOL/talks/saunders-stgirons.ps.

J. A. Scott. On ordering elements for a frontal solver. *Communications in Numerical Methods in Engineering*, **15**, 309–323, 1999.