Facilities Council

Scaling and pivoting in an out-of-core sparse direct solver

J. A. Scott

May 2008

RAL-TR-2008-016

Library and Information Services
SFTC Rutherford Appleton Laboratory
Harwell Science and Innovation Campus
Didcot
OX11 0QX
UK
Tel: +44 (0)1235 445384
Fax: +44(0)1235 446403
Email: library@rl.ac.uk

ISSN 1358-6254

Scaling and pivoting in an out-of-core sparse direct solver[1,2]

by

J. A. Scott

Abstract

Out-of-core sparse direct solvers reduce the amount of main memory needed to factorize and solve large sparse linear systems of equations by holding the matrix data, the computed factors and some of the work arrays in files on disk. The efficiency of the factorization and solution phases is dependent upon the number of entries in the factors. For a given pivot sequence, the level of fill in the factors beyond that predicted on the basis of the sparsity pattern alone depends on the number of pivots that are delayed (that is, the number of pivots that are used later than expected because of numerical stability considerations). Our aim is to limit the number of delayed pivots, while maintaining robustness and accuracy. In this paper, we consider a new out-of-core multifrontal solver that is designed to solve efficiently the systems of linear equations that arise from finite element applications. We consider how equilibration can be built into the solver without requiring the system matrix to be held in main memory. We also examine the effects of different pivoting strategies, including threshold partial pivoting, threshold rook pivoting and static pivoting. Numerical experiments are reported for problems arising from a range of practical applications.

Keywords: large sparse unsymmetric linear systems, element problems, out-of-core solver, multifrontal, rook pivoting, partial pivoting, scaling.

---

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

[2] This work was supported by the EPSRC grant EP/E053351/1.

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

1 Introduction

Most direct methods for solving large sparse linear systems of equations $AX = B$ are variants of Gaussian elimination, involving a factorization $PAQ = LDU$ of the system matrix A, where L is unit lower triangular, U is unit upper triangular, D is diagonal, and P and Q are permutation matrices. The solution process is completed by performing forward and then back substitutions (that is, by first solving a lower triangular system and then an upper triangular system). Direct methods are popular because, when properly implemented, they are generally robust and so can be used as general-purpose black-box solvers for a wide range of problems. Their main limitation is that the memory they require normally increases rapidly with problem size. In recent years, this has led to an interest in the development of out-of-core solvers, that is, solvers that hold the system matrix A and its factors (and possibly some of the work arrays used by the solver) in files (see, for example, [16, 17, 18, 19]). This allows much larger problems to be solved than would otherwise be possible. The main disadvantage is the overhead involved with reading from and writing to files held on disk and it is essential that this be done efficiently; how we achieve this within our recent out-of-core solvers is described in detail in [15].

As well as efficiently handling the input/output operations, it is important to limit the need for such operations by minimising the number of entries in the factors. The purpose of this paper is to examine the effects of scaling and of different pivoting strategies on the fill-in of the factors computed using an unsymmetric out-of-core solver `HSL_MA78` [17] that we have developed for the mathematical software library HSL [9].

`HSL_MA78` is a Fortran 95 package that is designed to solve efficiently the large sparse unsymmetric systems that arise from finite-element problems. In common with other direct solvers, `HSL_MA78` has a number of distinct phases. A pivot sequence must first be chosen, that is, the order in which the eliminations are to be carried out. For the chosen pivot sequence, the analyse phase (`MA78_analyse`) predicts the non-zero pattern of the factors using the sparsity pattern of A. `MA78_analyse` determines lower bounds on the number of entries in the matrix factors, the memory required by the factorization, and the number of operations needed to compute the factors. The factorize phase (`MA78_factor`) computes the numerical factorization using the data structures set up by the analyse phase. The forward and back substitutions are performed by the solve phase (`MA78_solve`), which may be called repeatedly for different right-hand sides B. There is also an option to solve transpose systems $A^T X = B$. In general, to maintain numerical stability, it is necessary to modify the pivot sequence during the factorize phase, delaying small pivots until alternatives are available or they are safer to use. These delayed pivots cause additional fill-in of the factors beyond the lower bound computed by the analyse phase and lead to extra work in both the factorize and solve phases. In the case of an out-of-core solver, the extra work is not just an increase in the flop count, but also an increase in the number of input/output operations. We are therefore interested in trying to limit the number of delayed pivots, while maintaining the robustness and accuracy of our solver.

The first technique we will use to try and limit delayed pivots is scaling. How to find a good scaling is an open question, but a number of scalings have been proposed and are widely used. In particular, there are a number of scaling routines available within the HSL library. Unfortunately, these require that the matrix A is held in main memory. For very large problems, A may not be in main memory and so we want to consider how we can scale A while it is held out-of-core. In particular, we will implement an out-of-core equilibration algorithm. Equilibration [20] is a particular form of scaling, where the rows and columns of a matrix are modified so that they have approximately the same norm. In addition to equilibration, we will look at the effects of different pivoting strategies. Within `HSL_MA78` we have included options for threshold partial pivoting, threshold rook pivoting, and static pivoting.

The remainder of this paper is organised as follows. In Section 2, we give a brief introduction to the out-of-core multifrontal method. We then discuss, in Section 3, how we can incorporate an equilibration algorithm within our multifrontal solver, avoiding the need to assemble the system matrix A. In Section 4, we consider the dense linear algebra kernels that lie at the heart of `HSL_MA78` and explain the pivoting options that are offered. Numerical results for a range of practical problems are presented in Section 5.

2 Introduction to the out-of-core multifrontal method

Our unsymmetric out-of-core solver `HSL_MA78` is designed for solving problems coming from finite-element analysis in which the $n \times n$ matrix A can be expressed in the form

$$A = \sum_{k=1}^{nelt} A^{(k)}. \tag{2.1}$$

Here *nelt* is the number of elements in the model and $A^{(k)}$ corresponds to the contribution from element k and has nonzeros in only a small number of rows and columns. In practice, each $A^{(k)}$ is held as a small square dense matrix, called an element matrix. A list of the global indices of the variables associated with element k, which identifies where the entries in $A^{(k)}$ belong in A, must also be held. Each $A^{(k)}$ is symmetrically structured (the list of indices is both a list of column indices and a list of row indices) but, in the general case, is numerically unsymmetric. The advantage of holding the matrix A as a sum of element matrices is that it is possible to avoid assembling (and thus the storage for) A. Instead, the assembly and elimination operations are interleaved and the main work is performed using a dense *frontal matrix* that is significantly smaller than A: this is key idea behind the frontal method [10] and, more generally, the multifrontal method.

At each stage of the computation, the frontal matrix is a square matrix of order m that may be expressed in the form

$$F = \begin{pmatrix} F_1 & F_2 \\ F_3 & F_4 \end{pmatrix}, \tag{2.2}$$

where $m \ll n$ and the p rows and columns of F_1 are *fully summed*, that is, all the entries in these rows and columns of A have already been assembled, while the rows and columns of F_4 are not yet fully summed. Provided p pivots can be chosen stably from F_1, the *partial factorization* of F takes the form

$$F = \begin{pmatrix} P_1 & 0 \\ 0 & I \end{pmatrix} \begin{pmatrix} L_1 & 0 \\ L_2 & I \end{pmatrix} \begin{pmatrix} D_1 & 0 \\ 0 & F_S \end{pmatrix} \begin{pmatrix} U_1 & U_2 \\ 0 & I \end{pmatrix} \begin{pmatrix} Q_1 & 0 \\ 0 & I \end{pmatrix}, \tag{2.3}$$

where P_1 and Q_1 are permutation matrices, L_1 and U_1 are unit lower and unit upper triangular matrices, and D_1 is a diagonal matrix, all of order p. The Schur complement F_S is given by

$$F_S = F_4 - L_2 D_1 U_2.$$

At the next stage of the frontal method, the contributions from another element are assembled with the Schur complement to form a new frontal matrix; the process continues until all element matrices have been assembled and the final elimination operations are performed. The matrices L_i, U_i and D_i $(i = 1, 2)$, and the permutation matrices P_1 and Q_1, are part of the factorization and are not needed again until the forward and back substitutions are performed. Thus, as they are generated, they can be transfered to a file. The data in these files is read into main memory as it is required (one record at a time) during the forward and back substitution phases. The element data may also be held in files. In Fortran, it is convenient to use separate files for the real data and for the integer data.

In the frontal method, there is a single front (that is, there is a single set of variables that have not yet been eliminated but are involved in one or more of the elements that have been assembled). Duff and Reid [5] extended the frontal concept to use more than one front and, reflecting the use of several fronts, their generalisation is called the *multifrontal* method. For each pivot in turn, the multifrontal method first assembles all the elements that contain the pivot into the frontal matrix and performs a partial factorization. The computed entries of the factors are stored and the Schur complement matrix F_S is treated as a new element, called a *generated element* (the term *contribution block* is also used in the literature). The generated element is added to the set of unassembled elements and the next uneliminated pivot then considered. The basic algorithm is summarised in Figure 2.1. In the unsymmetric case, the key

```
Basic Multifrontal Factorization
do for each pivot in the given pivot sequence
    if the pivot has not yet been eliminated
        assemble all unassembled elements and generated elements that contain
            the pivot into a frontal matrix;
        perform a partial factorization of the frontal matrix;
        add the generated element to the set of elements
    end if
end do
```

Figure 2.1: Basic multifrontal factorization

difference between the original elements and the generated elements is that, for the latter, it is necessary to hold an integer list of both the row indices and the column indices of the variables in the front. This is because the original symmetry is lost by choosing off-diagonal pivots during the partial factorizations of the frontal matrices.

The assemblies can be recorded as a tree, called an *assembly* tree. Each leaf node represents an original element and each non-leaf node represents a set of eliminations and the corresponding generated element. The children of a non-leaf node represent the original elements and generated elements that contain the pivot. If A is structurally irreducible there will be a single *root* node, that is, a node with no parent. Otherwise, the matrix may be permuted to block triangular form and thereis one root for each block on the diagonal.

The partial factorization of the frontal matrix at a node v in the tree can be performed once the partial factorizations at all the nodes belonging to the subtree rooted at v are complete. If the nodes of the tree are ordered using a depth-first search, the generated elements required at each stage are the most recently generated ones of those so far unused. This makes it convenient to use a stack for temporary storage during the factorization. This, of course, alters the pivot sequence, but the arithmetic is identical apart from the round-off effects of reordering the assemblies and the effects that this can have on later computations. The stack can be held out of core in files.

In general, to obtain a numerically stable factorization, it is necessary to incorporate numerical pivoting. This may mean that $q \leq p$ pivots are chosen at non-root nodes and the matrices L_1, U_1 and D_1 in (2.3) are then of order q, while the permutation matrices P_1 and Q_1 remain of order p. In this case, $p-q$ pivots must be *delayed* (passed up the tree), until they are safer to use. The generated element will thus be larger than anticipated on the basis of the sparsity pattern alone and the computed L and U factors will contain more fill-in. Our interest lies in reducing the number of delayed pivots by scaling (Section 3) and by the choice of pivoting algorithm (Section 4) while maintaining accuracy.

3 Scaling within a multifrontal algorithm

There has been much work on the effects of scaling and equilibration on the stability and accuracy of LU factorizations of general matrices. For example, Skeel [21] concluded that no systematic scaling can be derived for general matrices that is always successful; others have also suggested that the benefits of scaling are limited so that it is often the practice to scale matrices on a case-by-case basis. To enable users to experiment with different scalings, the HSL mathematical software library [9] offers a number of packages that are designed to compute scalings of large sparse matrices. From our point of view, the main limitation of these packages is that they all require the user to supply the matrix in assembled form, that is, the non-zero entries of A must be entered on a single call using either coordinate format or compressed column format; there are no facilities for working out-of-core or for matrices that are held in the form (2.1). Thus, as we wish to avoid assembly of A, the existing packages are not suitable for use with our solver `HSL_MA78`. In this section, we describe how we can incorporate an equilibration option into `HSL_MA78`,

using only minimal additional memory.

The matrix

$$D_R^{-1} A D_C^{-1},$$

where D_R and D_C are diagonal matrices, is an equilibration of A if the norms of its rows and columns have approximately the same magnitude. One possibility is to define the diagonal matrices to be

$$D_R = \text{diag}\left(\sqrt{\max_j |A_{ij}|}\right) \quad \text{and} \quad D_C = \text{diag}\left(\sqrt{\max_i |A_{ij}}\right).$$

More generally, the process can be applied iteratively, as summarised in Figure 3.1.

Equilibration algorithm

$A^{(1)} = A, \quad D_1^{(1)} = I, \quad \text{and } D_2^{(1)} = I$

for $k = 2, 3, \ldots,$ until termination do :

$$D_R = \text{diag}\left(\sqrt{\max_j |A_{ij}^{(k)}|}\right), \quad \text{and } D_C = \text{diag}\left(\sqrt{\max_i |A_{ij}^{(k)}|}\right)$$

$$A^{(k)} = D_R^{-1} A^{(k-1)} D_C^{-1}$$

$$D_1^{(k)} = D_1^{(k-1)} D_R \quad \text{and} \quad D_2^{(k)} = D_2^{(k-1)} D_C. \tag{3.1}$$

Figure 3.1: Equilibration algorithm

This algorithm computes the scaling diagonal matrices $D_1^{(k)}$ and $D_2^{(k)}$ such that the infinity norm of each row and column of $A^{(k)} = (D_1^{(k)})^{-1} A (D_2^{(k)})^{-1}$ tends to 1 as $k \to +\infty$. The iteration is terminated when

$$\eta = \max_l \left(1 - \max_j |A_{lj}^{(k)}|, \;\; 1 - \max_i |A_{il}^{(k)}|\right) \leq \epsilon \tag{3.2}$$

for a given value of the tolerance $\epsilon \geq 0$.

The properties of this algorithm (for the infinity norm and for other norms) are discussed by Ruiz [20]. The particular case when the infinity norm of each row and column of A is equal to one is clearly a fixed point for the equilibration algorithm. Furthermore, if for each i, $|A_{ii}| \geq \max(\max_l |A_{il}|, \; \max_l |A_{li}|)$, then the algorithm converges in one iteration and the resulting scaled matrix has ones on the diagonal.

The equilibration algorithm is implemented for dense matrices and for sparse (assembled) matrices within the HSL package `MC77`. In `MC77`, the tolerance ϵ and the maximum number of iterations may be controlled by the user (with default settings of zero and 10, respectively).

We now look at how we can implement equilibration within `HSL_MA78`, avoiding the need to assemble A explicitly. We hold D_R and D_C as rank-1 arrays of length n that we initialise to zero. Consider again the $m \times m$ frontal matrix $F = \{f_{ij}\}$ given by (2.2), in which the p rows of F_1 and F_2 and the p columns of F_1 and F_3 are fully summed. Together with the reals in F, an integer list ind of the global row and column indices of the variables in the front is held. Each row i of F is considered in turn. Let

$$D_R(ind_i) \leftarrow \max\left(D_R(ind_i), \;\; \max_{j \leq k} |f_{ij}| \;\right)$$

$$D_C(ind_j) \leftarrow \max\left(D_C(ind_j), \;\; \max_{i \leq k} |f_{ij}| \;\right)$$

where

$$k = \begin{cases} m & \text{if } i \leq p \\ p & \text{otherwise.} \end{cases}$$

We will refer to this searching and setting of the entries of the scaling matrices as *updating* D_R and D_C

Once the search of the fully summed part of each row and each column is complete, F_1, F_2 and F_3 can be removed from the frontal matrix and what remains (F_4) can be stored in the same way as the generated element (2.4) is stored during the multifrontal algorithm, that is, it can be placed on a stack. We will call the $(m-p) \times (m-p)$ matrix that remains after the removal of the fully summed rows and columns an F_4-element.

Provided the elements (the original elements and the F_4-elements) are assembled in the order determined by the chosen pivot sequence, we can mimic what happens in the factorization phase of the multifrontal algorithm except that, at each stage, we perform no permutations and no actual elimination operations, instead we find the maximum entries in the fully summed part of each row and column. Note that since no permutations are performed, the row and column indices of the variables in the front are the same and so a single list is needed. The first iteration of a multifrontal equilibration algorithm is outlined in Figure 3.2.

Basic multifrontal equilibration algorithm
flag all pivots as uneliminated and initialise D_R and D_C to zero
do for each pivot in the chosen pivot sequence
  if the pivot is flagged as uneliminated
    assemble all unassembled elements and F_4-elements that contain
      the pivot into the frontal matrix F
      and let p be the number of fully summed rows and columns;
    update D_R and D_C with the largest entries in the fully summed part of
      each row and column of F;
    flag the variables corresponding to the first p rows and columns of F as eliminated
      and add the F_4-element to the set of elements
  end if
end do

Figure 3.2: First iteration of an equilibration algorithm within a multifrontal algorithm

As in Figure 3.1, we can apply the algorithm iteratively and terminate when either we have satisfied the requested tolerance (3.2) or the maximum number of iterations has been reached. On the second and subsequent iterations, before they are searched for their largest entries, the entries in the fully summed part of F are scaled with the scaling factors computed so far, and the accumulated scaling factors are stored (see (3.1)). Note that although we have described the construction of the equilibration factors for the infinity norm, it is straightforward to extend the algorithm to other norms.

For an assembled matrix, the time taken to run an equilibration package such as `MC77` is, in general, small compared with the subsequent time needed for the numerical factorization. The main cost associated with implementing the equilibration algorithm within our multifrontal algorithm is that of accessing the element data. By default, the original element matrices supplied by the user are held in files and the F_4-elements are written to a temporary stack that is also held in a file; all this data must be read for each iteration. It is important, therefore, to limit the number of iterations and to perform only the minimum needed to obtain a sufficiently good equilibration. There is a tradeoff between the number of iterations and the quality of the equilibration, with the best choice being problem dependent. In our experiments (Section 5), we include results that illustrate this.

3.1 Equilibration within `HSL_MA78`

The use of equilibration is optional within `HSL_MA78`. In Version 2.0.0, we have added an extra entry, `MA78_scale`, that may be called by the user after the analyse phase and before the factorize phase. The user can control the maximum number of iterations performed (the default is 1), the norm used (the one-norm or the infinity norm are offered with the default being the infinity norm), and the tolerance

parameter ϵ. Based on our numerical experiments, we set the default value of ϵ to 0.5 (see Section 5.2). On exit from `MA78_scale`, the scaling factors are held in a real rank-1 array `scale` of length `2n`. The first n entries (`scale(1:n)`) contain the diagonal entries of the row-scaling matrix D_R and `scale(n+1:2n)` contains the diagonal entries of the column-scaling matrix D_C. The array `scale` may be passed as an optional argument to the factorization subroutine `MA78_factor`. Note that, since `scale` is an optional argument, the user may choose to compute scaling factors using an alternative approach and then pass his or her scaling directly to `MA78_factor`.

`MA78_scale` includes an option to compute the infinity norm of the (unscaled) matrix A. At each stage of the first iteration of the equilibration algorithm, we accumulate the sum of the absolute values of the fully summed part of each row of F in a real rank-1 work array of length n. Once the first iteration is complete (that is, all pivots are flagged as eliminated), $\|A\|_\infty$ is computed to be the maximum norm of this work array.

4 Partial factorization of the frontal matrices

The efficiency of `MA78_factor` is dependent upon the partial factorization of the frontal matrices. Since the frontal matrices are held as full matrices, dense linear algebra kernels may be used. We have developed a separate package, `HSL_MA74`, that is used by `MA78_factor` to perform the partial factorizations of the frontal matrices and by `MA78_solve` to perform the partial forward and back substitutions. In this section, we describe `HSL_MA74` and discuss the pivoting options that it offers and that are available to users of the multifrontal solver `HSL_MA78`.

4.1 Overview of `HSL_MA74`

Given a dense unsymmetric $m \times m$ matrix F, `HSL_MA74` performs a partial factorization, limiting eliminations to the leading $p \leq m$ rows and columns. Stability considerations may lead to $q \leq p$ eliminations being performed (that is, fewer than p pivots are chosen). The factorization takes the form (2.3) where the matrices L_1, U_1 and D_1 are of order q and the permutation matrices P_1 and Q_1 are of order p. Subroutines are provided for partial solutions, that is, solving systems of the form

$$\begin{pmatrix} L_1 & 0 \\ L_2 & I \end{pmatrix} X = B, \quad \begin{pmatrix} D_1 & 0 \\ 0 & I \end{pmatrix} X = B,$$

$$\begin{pmatrix} D_1 & 0 \\ 0 & I \end{pmatrix} \begin{pmatrix} U_1 & U_2 \\ 0 & I \end{pmatrix} X = B, \quad \text{and} \quad \begin{pmatrix} U_1 & U_2 \\ 0 & I \end{pmatrix} X = B,$$

and the corresponding equations for a single right-hand side b and solution x. Subroutines are also provided for partial solutions to transposed systems.

The user inputs the matrix F in a rank-2 (two dimensional) array which, on exit, is overwritten by the factorized matrix. Each diagonal entry holds either the inverse of a pivot or, if a zero pivot is chosen (because the matrix singular), the corresponding diagonal entry is set to zero. A rank-1 integer array `pperm` of length p is used to hold the row permutations P_1 so that, on exit, `pperm`(i) holds the index of the row of F that is permuted to row i, $i = 1, ..., p$. Similarly, a rank-1 integer array `qperm` is used to hold the column permutations Q_1.

`HSL_MA74` uses a block algorithm. If the factorization were to proceed by choosing a single pivot at a time, the updates to the rest of F could only be performed using Level 2 BLAS. To take advantage of the more efficient Level 3 BLAS, the partial factorization is programmed as a sequence of block steps, with the block size nb under the user's control. If q is the number of pivots chosen so far, the code searches columns $q + 1$ to p of F in turn for a pivot. If the column to be searched has had $k < q$ updates, it is first updated with the $q - k$ most recently chosen pivots. Since a single column is being updated, this is performed using the Level 2 BLAS kernels `_trsv` and `_gemv`. Each time a pivot is chosen, q is incremented

by one and the pivotal column is swapped with column q. The position m_1 of the right-most column of F that has been searched for a pivot is held and, whenever a pivot is chosen, columns $q+1$ to m_1 are updated (using the Level 2 rank-1 update routine `_ger`) so that all the columns that have been tested and rejected are fully updated. This avoids the need to hold an array of updates. Once nb pivots have been chosen or $q = p$, columns $m_1 + 1$ to n are updated using the Level 3 BLAS kernels `_trsm` and `_gemm`. If $q < p$, m_1 is reset to $q + 1$ and the column search restarts from column $q + 1$. The remaining columns are searched cyclically to avoid repeatedly searching a previously rejected column.

4.2 Pivoting options

For numerical stability, it is generally necessary to incorporate pivoting within Gaussian elimination. A balance needs to be achieved between stability and efficiency: we want to solve the sparse system fast, with as little fill in as possible in the factors, but we also want the computed solution to be of the required accuracy. There are a number of pivoting strategies available, each of which places a different emphasis on stability and efficiency. The basic options available are threshold partial pivoting, threshold rook pivoting and threshold complete pivoting.

Consider first the case where the matrix to be factorized is dense and set $V = DU$. At the kth stage of Gaussian elimination, partial pivoting selects as pivot the largest entry below the diagonal in the kth column. A traditional quantity used to describe the backward stability of Gaussian elimination is the growth factor ρ, which for partial pivoting can be defined by

$$\rho = \max |V_{ij}| / \max |A_{ij}|.$$

It can be shown that $\rho \leq 2^{n-1}$ and, although ρ usually behaves like n or less, examples can be found for which partial pivoting is unstable (see, for example, [8]). Complete pivoting searches for the largest entry in the remaining matrix of order $n - k$. In this case, the growth factor satisfies $\rho \leq 2\sqrt{n}n^{ln(n)/4}$ but this cannot be attained for $n \geq 3$ [22] and, in practice, complete pivoting is considered to be numerically stable. The disadvantage of complete pivoting is the cost since it requires approximately $n^3/3$ comparisons, beyond the work required by Gaussian elimination with no pivoting, whereas partial pivoting requires only $n^2/2$ comparisons. Gaussian elimination with rook pivoting [12] offers a strategy that is intermediate between partial and complete pivoting in terms of both efficiency and stability. To locate the kth pivot, rook pivoting performs a sequential search (column, row, column, etc) of the remaining matrix until an entry is located whose absolute value is not exceeded by the absolute value of any other entry in the row or column in which it lies. In general, rook pivoting is observed to be more accurate than partial pivoting [2]. Furthermore, it has been shown that the growth factor satisfies $\rho \leq 1.5n^{3ln(n)/4}$ [6] and Poole and Neal [14] report in their numerical experiments that the expected cost of rook pivoting is about three times that of partial pivoting.

Our interest is in Gaussian elimination for sparse matrices. In this case, pivoting strategies that require the largest entry in a column and, possibly, a row are too restrictive. Instead, a threshold is introduced. `HSL_MA74` offers a number of threshold pivoting options that are controlled using the parameters `pivoting`, `small`, `static`, and `u`. We now discuss these options.

4.2.1 Threshold partial pivoting

The default strategy within `HSL_MA74` is *threshold partial pivoting* (`pivoting = 1`). In this case, an entry f_{ij} of the reduced matrix after q pivots have been selected is normally only chosen as a pivot if $i \leq p$ and $j \leq p$ and it satisfies

$$|f_{ij}| \geq \max(\mathtt{u} * \max_{l>q} |f_{lj}|, \mathtt{small}). \tag{4.1}$$

Here `u` is the pivoting *threshold* parameter. Values of `u` close to zero will generally result in a faster factorization with fewer entries in the factors (for `u` sufficiently small, no pivots will will be delayed and

so the number of entries in the factors will be equal to the number predicted by the analyse phase) but values close to 1 are more likely to result in a stable factorization; the default of 0.01 is a compromise between stability and sparsity and is recommended in the user documentation for other direct solvers. In HSL_MA74, values of u that are less than 0.0 are treated as 0.0 and values greater than 1.0 are treated as 1.0. small controls the size of the smallest pivot that is acceptable. The default value is tiny(small), where tiny() is the Fortran numeric inquiry function that returns the smallest positive number that is stored in full precision. When searching column j of the reduced matrix for a suitable pivot, the row index r corresponding to the largest entry in rows $q+1$ to m is sought using the BLAS kernel i_amax. If $r \leq p$, the pivot has been found, q is incremented by 1 and rows q and r are swapped. If $r > p$, the largest entry in rows q to p is found (again using i_amax) and, if this satisfies (4.1), it is chosen as the next pivot. Note that, if u = 0.0, the pivot is still chosen to be the largest entry in rows q to p (even though any entry in the column larger than small satisfies (4.1)).

4.2.2 Diagonal pivoting

In some applications, it may be known that the pivots can be chosen stably from the diagonal (for example, if A is close to a symmetric positive-definite matrix). For threshold diagonal pivoting (pivoting = 2) with threshold u > 0.0, pivots are initially chosen from the diagonal and must satisfy the threshold criteria (4.1) with $j = i$. If $p = n$ and only $q < n$ pivots can be chosen from the diagonal that satisfy (4.1), the code switches to choosing off-diagonal pivots (so that the final $n - q$ pivots may be off-diagonal entries). The number of pivots chosen from the diagonal is returned to the user. If the threshold parameter u is equal to zero, no search is made for the largest entry in the column of the candidate pivot. In this case, there is no pivoting and (4.1) simplifies to checking the candidate pivot is at least small.

4.2.3 Threshold rook pivoting

Threshold rook pivoting may be selected in HSL_MA74 by setting pivoting = 3. A candidate f_{ij} may be chosen as a pivot if $i \leq p$ and $j \leq p$ and it satisfies (4.1) and, additionally, with the same u,

$$|f_{ij}| \geq \mathtt{u} * \max_{l>q} |f_{il}|. \tag{4.2}$$

In other words, for rook pivoting the pivot candidate must satisfy the threshold test in both its column and its row. Suppose q pivots have been chosen. Having found a candidate with row index i in the column that is currently being searched, row i is swapped with row $q+1$. It must then be updated so that all q pivots have been applied to it, before it can be searched for its largest entry and then tested. Thus, rook pivoting involves more Level 2 BLAS updates (and hence fewer Level 3 BLAS operations) and the additional overheads of row swaps and row searches. Because of this extra cost, it is not the default pivoting strategy within HSL_MA74.

If column j is searched but rejected because it fails the test (4.1), provided $j < p$, we next update and search column $j+1$, and continue to search the columns cyclically. However, if the largest entry in column j is in row $i \leq p$ and (4.1) is satisfied, then we update and search row i. Suppose the largest entry in row i lies in column l and that f_{ij} does not satisfy (4.2). If $l > p$ we update and search column $j+1$ but if $l \leq p$ (that is, column l is fully summed), we swap columns $j+1$ and l so that we next update and search column l, and continue in a like manner until a pivot is found. Our experience has been that, compared with searching in a strictly cyclic fashion, this reduces the total number of rows and columns searched during the factorization. Note that a count of the number of rows and columns searched is returned to the user.

Gill, Murray and Saunders [7] report that, provided u is chosen to be sufficiently close to 1, the rank-revealing properties of rook pivoting are essentially as good as for threshold complete pivoting (see also [13]) and they include rook pivoting as an option within the sparse direct solver LUSOL. Since threshold rook pivoting with a sufficiently large u appears to offer the same advantages as threshold complete pivoting but at less expense, we have chosen not to offer an option for threshold complete pivoting within HSL_MA74.

4.2.4 Static pivoting

In some applications, using a value of `u` equal to 0.1 or 0.01 can lead to a large number of delayed (rejected) pivots. In this case, the size of the frontal matrices as the factorization moves up the tree can grow significantly beyond that which was anticipated by the analyse phase. This results in a more expensive factorization, both in terms of the number of flops required to perform the factorization and the number of entries in the matrix factors; this, in turn, leads to a more expensive solve phase. Furthermore, more storage will be required for the frontal matrix (which is held in main memory) and more data has to be written to and read from a stack during the factorization. In recent years, this has led to a number of direct solvers offering options for *static* pivoting (see, for example, [4], [11]). Here, the essential idea is to allow pivots that do not satisfy condition (4.1), thus ensuring the pivot selection closely follows that provided by the user to the analyse phase. The danger is that there will be a potential loss of accuracy in the factorization and it may be necessary to perform refinement steps after the solve phase to try to recover the required accuracy, that is, to use the computed factorization as a preconditioner for an iterative method. This is still a subject of research (see, for example, [1]).

Different variants of static pivoting have been proposed: the strategy we have adopted aims to use the best available pivot and to modify pivots only when they become very small. Within `HSL_MA74` (and `HSL_MA78`), static pivoting is controlled by the parameter `static`. If `static` is positive and fewer than p pivots can be selected that satisfy (4.1), the pivot that came closest to satisfying this condition is chosen, that is, the pivot for which the ratio

$$\max_{q<i\leq p} |f_{ij}| \;/\; \max_{q<l\leq m} |f_{lj}|, \quad q \leq j \leq p, \tag{4.3}$$

is the largest. If its absolute value is greater than `static`, the information parameter `usmall` (which is initialised to the user-supplied threshold `u`) is set to the minimum of `usmall` and (4.3). The computation continues using the reduced threshold `u` $\leftarrow$ `usmall`. If the absolute value of (4.3) is less than `static`, the pivot is given the value that has the same sign but absolute value `static` and `u` is unchanged. On exit, `usmall` holds the threshold parameter that was used or is set to zero if any pivots have been replaced by `static`, `num_thresh` holds the number of pivots that did not satisfy the threshold criteria based on the user-supplied value of `u`, and `num_perturbed` holds the number of pivots that were replaced by `static`. Note that `u` may be reduced a number of times during a single call to `HSL_MA74` and, within `HSL_MA78`, once `u` has been reduced during a call to `HSL_MA74`, the factorization continues using this smaller threshold.

5 Numerical experiments

In this section, we report on the effects of the choice of pivot strategy and scaling when using are multifrontal solver `HSL_MA78` to solve a number of problems from practical applications. The test problems are listed in Table 5.1 in order of the predicted number of entries in the factors (that is, the number of entries if no pivots are delayed) when the analysis phase of the HSL solver `MA57` [3] is used to compute the pivot order. The predicted maximum frontsize is also given (that is, the maximum size of the frontal matrix if no pivots are delayed). With the exception of the last problem (which came from a user of `hsl_ma78`), the problems are all taken from the website `http://www.parallab.uib.no/projects/parasol/data`. The right-hand side for each problem is selected so that the required solution is the vector of ones. We note that, when storing the partial factorization (2.3), the lower triangular part of L_1 and the upper triangular part of U_1, and the rectangular matrices L_2 and U_2, are stored as dense matrices (explicit zeros within the front are ignored). Thus the number of entries $nz(L)$ in the L factor is equal to the number $nz(U)$ in the U factor.

The numerical results were obtained using `double precision` (64-bit) reals on a 3.6 GHz Intel Xeon dual processor Dell Precision 670 with 4 Gbytes of RAM. The Nag Fortran f95 compiler with the optimization flag `-O` was used together with the ATLAS BLAS and LAPACK (`math-atlas.sourceforge.net`). The reported times are elapsed (wall clock) times in seconds and, unless

Table 5.1: The test problems. n and $nelt$ denote the number of variables and elements, respectively, $nz(L)$ is the predicted number entries in L, in millions and max_front is the predicted maximum frontsize.

Identifier	n	$nelt$	$\|A\|_\infty$	$nz(L)$	max_front	Description/discipline
1. `ship_001`	34920	3431	$2.18*10^{12}$	15.6	1596	Ship structure - predesign
2. `thread`	29736	2176	$1.80*10^{19}$	24.7	3099	Threaded connector
3. `x104`	108384	26019	$4.32*10^{5}$	27.1	2076	Beam joint
4. `mt1`	97578	5328	$1.83*10^{12}$	32.7	1941	Tubular joint
5. `shipsec8`	114919	32580	$3.19*10^{12}$	36.3	2624	Section of a ship
6. `shipsec1`	140874	41037	$3.15*10^{13}$	38.7	2142	Section of a ship
7. `shipsec5`	179860	52272	$4.89*10^{12}$	54.2	3243	Section of a ship
8. `ship_003`	121728	45464	$3.48*10^{18}$	60.3	3204	Ship structure - production
9. `raju_001`	151656	46980	$4.34*10^{8}$	139.7	5232	Laminar flow inside pump casing

stated otherwise, are the total solution times (that is, the time for the analyse, factorize and solve phases with a single right-hand side and, where used, for scaling). We compute the scaled residual

$$\frac{\|Ax - b\|_\infty}{\|A\|_\infty \|x\|_\infty + \|b\|_\infty} \tag{5.1}$$

In our experiments, we also monitor $\max_i |1 - x_i|$. Note that, if equilibration is used, the original unscaled matrix A is used when computing the scaled residual.

5.1 Comparison of partial and rook pivoting

We first compare the performance of partial threshold pivoting and rook threshold pivoting (without scaling). Using the default threshold parameter $u = 0.01$, in Table 5.2 we report the total solution time, the number of flops required to compute the factors, the number $nz(L)$ of entries in the L factor, the number `delay` of delayed eliminations, and the number of rows and columns searched during the factorization. If p_i and q_i are, respectively, the numbers of candidate and actual pivots chosen at node i then

$$\texttt{delay} = \sum_i (p_i - q_i).$$

Note that a pivot may be delayed (and hence counted) more than once. The number of columns searched is at least n and, for rook pivoting, the minimum number of rows searched is n. For partial pivoting, no rows are searched and so no row search count is included in this case.

A standard technique to recover from inaccuracies in the factorization is to use the computed matrix factorization as a preconditioner for an iterative method, such as iterative refinement. In Table 5.3, the residuals are given, both after `MA78_solve` and after a single step of iterative refinement.

If there are only a small number of delayed eliminations, we see that rook pivoting adds an overhead; this is because both rows and columns must be searched. But this overhead is small compared with the total solution time (see problems 1 and 2). Furthermore, the more stringent test used by rook pivoting can result in less growth and smaller residuals (see columns 2 and 3 of Table 5.3). Smaller growth can also mean that, although rook pivoting initially rejects more pivots than partial pivoting, eventually rook pivoting rejects fewer pivots, leading to the total number of delayed eliminations being less for rook pivoting. This in turn gives sparser factors that are computed using fewer flops. Because looking for each pivot is more expensive than for partial pivoting, the time can still increase (as illustrated by problem 8) but in some cases, the total time using rook pivoting is significantly less than for partial pivoting (notably for problems 4, 6 and 7). This observation is somewhat unexpected. Problem 9 illustrates what we would

Table 5.2: Comparison between rook and partial threshold pivoting with $u = 0.01$. For the first four pairs of results, one is in bold if it is significantly better than the other.

Problem	Time		flops$*10^9$		$nz(L)*10^6$		`delay`$*10^3$		Searched$*10^3$	
									Rows/Cols	Cols
	rook	partial	rook	partial	rook	partial	rook	partial	rook	partial
1. `ship_001`	15.0	**13.4**	22	22	15.6	15.6	0	0	35/35	35
2. `thread`	37.8	**35.4**	72	72	24.7	24.7	0	0	30/30	30
3. `x104`	**34.0**	37.8	**36**	59	**30.3**	34.5	**20**	33	137/200	202
4. `m_t1`	**55.7**	94.9	**69**	159	**40.2**	56.2	**34**	67	117/231	358
5. `shipsec8`	91.6	92.8	**130**	175	**49.0**	55.6	**61**	78	139/324	329
6. `shipsec1`	**110**	174	**150**	305	**58.6**	78.2	**97**	135	175/467	554
7. `shipsec5`	**175**	275	**246**	492	**80.4**	105	**121**	169	225/589	687
8. `ship_003`	146	**118**	**206**	228	**70.8**	74.0	**50**	610	138/285	281
9. `raju_001`	335	**226**	646	**579**	168	**147**	110	**79**	414/512	345

Table 5.3: Comparison between the scaled residuals for factorization with rook and partial threshold pivoting before and after one step of iterative refinement ($u = 0.01$). For each pair of results, one is in bold if it is significantly better than the other.

Problem	Before		After	
	rook	partial	rook	partial
1. `ship_001`	$5.7 * 10^{-16}$	$3.1 * 10^{-16}$	$5.6 * 10^{-17}$	$9.5 * 10^{-17}$
2. `thread`	$3.1 * 10^{-16}$	$4.0 * 10^{-16}$	$7.4 * 10^{-17}$	$6.8 * 10^{-17}$
3. `x104`	$\mathbf{6.2 * 10^{-16}}$	$9.9 * 10^{-14}$	$5.4 * 10^{-17}$	$7.1 * 10^{-17}$
4. `m_t1`	$\mathbf{4.7 * 10^{-16}}$	$8.5 * 10^{-14}$	$3.7 * 10^{-16}$	$2.7 * 10^{-16}$
5. `shipsec8`	$\mathbf{5.3 * 10^{-16}}$	$2.4 * 10^{-14}$	$7.9 * 10^{-17}$	$9.1 * 10^{-17}$
6. `shipsec1`	$\mathbf{4.0 * 10^{-16}}$	$7.6 * 10^{-14}$	$9.0 * 10^{-17}$	$1.4 * 10^{-16}$
7. `shipsec5`	$\mathbf{1.8 * 10^{-15}}$	$6.8 * 10^{-13}$	$1.5 * 10^{-16}$	$1.9 * 10^{-16}$
8. `ship_003`	$\mathbf{7.9 * 10^{-16}}$	$1.5 * 10^{-13}$	$8.7 * 10^{-17}$	$7.9 * 10^{-17}$
9. `raju_001`	$1.5 * 10^{-15}$	$5.8 * 10^{-15}$	$3.3 * 10^{-16}$	$4.1 * 10^{-16}$

perhaps anticipate happening: rook pivoting is more cautious and here it leads to more delayed pivots, more searching, an increase in the fill-in and in the flop count and hence a greater computational time. We note that, in our tests, after one step of iterative refinement, there was no appreciable difference in the quality of the residuals for partial and rook pivoting.

We have also run the same tests with a larger threshold parameter of 0.1. Our results are in Table 5.4. For some of our test problems, the difference between the performance of rook and partial pivoting is more extreme. For example, for problems 4 and 7, it is much better to use rook pivoting while for problem 9, rook pivoting leads to so many delayed pivots that the code is unable to allocate a sufficently large frontal matrix on our 32-bit test machine. We do not give full details of the residuals but, again, rook pivoting results in smaller residuals and, as expected, the scaled residuals using $u = 0.1$ are generally smaller before iterative refinement than when using $u = 0.01$ (but following one step of refinement they are comparable).

Table 5.4: Comparison between rook and partial threshold pivoting with $u = 0.1$. NS indicates not solved. For the first four pairs of results, one is in bold if it is significantly better than the other.

Problem	Time		flops$*10^9$		$nz(L)*10^6$		`delay`$*10^3$		Searched$*10^3$	
									Rows/Cols	Cols
	rook	partial	rook	partial	rook	partial	rook	partial	rook	partial
1. `ship_001`	14.9	**12.9**	22	22	15.7	15.7	0	0	36/37	36
2. `thread`	35.9	**30.7**	73	73	24.7	24.7	0	0	30/30	30
3. `x104`	**46.3**	90.8	**65**	191	**38.5**	56.1	**55**	87	135/319	451
4. `m_t1`	**74.6**	273	**113**	539	**50.9**	96.6	**68**	137	114/348	695
5. `shipsec8`	**161**	251	**267**	507	**71.0**	92.9	**134**	165	136/550	615
6. `shipsec1`	**264**	776	**413**	1410	**97.5**	163	**203**	259	174/809	1056
7. `shipsec5`	**502**	1629	**768**	3187	**143**	261	**295**	395	218/1161	1569
8. `ship_003`	**227**	303	**395**	623	**96.6**	116	**140**	170	137/551	604
9. `raju_001`	NS	**300**	NS	**718**	NS	**176**	NS	**96**	NS	428

5.2 Effects of equilibration

We now present results for `HSL_MA78` run with both rook and partial pivoting following a single iteration of the equilibration algorithm (using the infinity norm). The reported times in Table 5.5 include the time taken by one iteration of the equilibration algorithm; the threshold parameter used by `HSL_MA78` was 0.01. Comparing these results with those in Table 5.2 we see that, with the exception of problems 1, 2 and 9, equilibration significantly enhances the performance of the solver. In particular, there are now (almost) no delayed pivots for the `ship` problems and this results in substantial reductions in the total time (the savings in the factorization times more than offset the costs of the equilibration). The scaled residuals for rook pivoting are $\mathcal{O}(10^{-16})$; they are typically an order of magnitude larger for partial pivoting.

Problems 1 and 2, which did not suffer from delayed pivots when not scaled, do not benefit from one iteration of the equilibration algorithm. In fact, for problem `thread`, the performance of `HSL_MA78` is significantly worse. Similar results were observed when using the one-norm. This illustrates that scaling will not benefit all problems and the user is advised of the need to experiment with using it.

After one iteration of equilibration, only problems 2, 4 and 9 have a significant number of delayed pivots. These problems may benefit from using more that one iteration. We have experimented with up to 10 iterations, setting ϵ in the stopping criteria (3.2) to 0.0 so that termination occurs when either the maximum number of iterations is reached or $\max_j |A_{lj}^{(k)}| = \max_i |A_{il}^{(k)}| = 1$. For problem 2 (`thread`), termination happened after 4 iterations and did not reduce the number of delayed pivots or entries in the factor. In Table 5.6, we report detailed results for problems 4 and 9 (0 iterations corresponds to no scaling). On the first iteration, η (given by (3.2)) is approximately equal to $(nz(A) * \|A\|_\infty)/n$; on the second iteration it is close to 1, and then decreases steadily, with an asymptotic linear rate of 1/2 (see [20]).

Table 5.5: Comparison between rook and partial threshold pivoting with equilibration and $u = 0.01$. For the first four pairs of results, one is in bold if it is significantly better than the other.

Problem	Time		flops$*10^9$		$nz(L)*10^6$		delay$*10^3$		Searched$*10^3$	
									Rows/Cols	Cols
	rook	partial	rook	partial	rook	partial	rook	partial	rook	partial
1. ship_001	16.7	16.3	22	22	15.6	15.6	0	0	35/35	35
2. thread	**55.0**	64.6	**93**	154	**27.4**	32.9	**4**	8	33/46	60
3. x104	24.9	**23.0**	29	29	27.2	27.2	1	1	111/114	111
4. m_t1	**45.0**	63.0	**60**	104	**37.6**	46.7	**24**	45	109/182	273
5. shipsec8	51.6	**45.3**	74	74	36.3	36.3	0	0	117/117	115
6. shipsec1	49.3	**44.4**	68	68	38.7	38.7	0	0	144/144	141
7. shipsec5	78.0	**69.8**	114	114	54.2	54.2	0	0	183/183	180
8. ship_003	98.4	**89.1**	156	156	60.3	60.3	0	0	123/123	122
9. raju_001	344	**255**	635	**567**	163	**149**	81	**54**	208/420	297

For problem mt_1, it is beneficial to use 2 iterations; with 3 or 4 iterations, the factors are sparser but the increased cost of the scaling leads to an increase in the total time. For problem raju_001, the number of iterations has little effect on the performance of HSL_MA78 but each extra iteration adds a significant cost.

Table 5.6: The effect of the number of iterations used by the equilibration algorithm on the performance of HSL_MA78 for problems m_t1 and raju_001 with partial pivoting and $u = 0.01$. η is defined in (3.2).

Problem	Number Iterations	η	Time		flops$*10^9$	$nz(L)*10^6$	delay	Searched$*10^3$
			Scaling	Total				Cols
m_t1	0	–	0.00	94.9	158	56.2	67467	358
	1	$2.32*10^{11}$	5.73	63.0	104	46.7	45127	273
	2	0.99	10.1	56.8	86	43.3	37384	227
	3	0.92	13.7	57.8	81	42.3	35124	218
	4	0.63	17.4	60.8	78	41.6	34279	212
	5	0.40	20.8	65.8	82	42.3	34945	220
	10	0.14	37.8	80.8	78	41.7	33748	212
raju_001	0	–	0.00	227	579	147.3	78669	345
	1	$1.22*10^7$	30.8	255	568	149.5	53235	297
	2	1.00	59.1	284	564	147.0	34279	250
	3	0.98	81.6	302	564	147.0	34945	247
	4	0.85	99.5	320	564	147.0	30073	248
	5	0.61	122	342	564	147.0	30341	247
	10	0.03	219	439	564	147.0	30206	247

5.3 Static pivoting results

Finally, we present results for static pivoting. We consider only those problems for which our earlier experiments reported a significant number of delayed pivots (if there are no delayed pivots, static pivoting will not have an effect). We start the factorization with the default threshold $u = 0.01$ and set the control parameter static $= 10^{-12}$ (see Section 4.2.4). With this choice of static, in our tests it was not necessary to replace any small pivots. In Table 5.7, we report the elapsed times, flop count, the value of the resulting threshold (usmall), and the residuals before and after one step of iterative refinement. The number of entries in the factor is equal to the predicted number of entries (see Table 5.1). Comparing the results with those in Table 5.2, we see that static pivoting can lead to significant savings and, for our test problems, a

single step of iterative refinement is generally able to recover accuracy.

Table 5.7: The elapsed times, flops, the value of the threshold used (`usmall`), and the residuals before and after iterative refinement with static pivoting ($u = 0.01$).

	Problem	Time	flops$*10^9$	`usmall`	Residual	
					Before	After
No Scaling	3. `x104`	18.4	28	$3.84 * 10^{-5}$	$2.2 * 10^{-13}$	$5.2 * 10^{-17}$
	4. `m_t1`	25.3	47	$5.46 * 10^{-4}$	$4.6 * 10^{-15}$	$3.1 * 10^{-16}$
	5. `shipsec8`	39.1	74	$1.95 * 10^{-4}$	$7.0 * 10^{-15}$	$1.2 * 10^{-16}$
	6. `shipsec1`	35.5	68	$2.17 * 10^{-4}$	$1.8 * 10^{-14}$	$9.1 * 10^{-17}$
	7. `shipsec5`	60.1	114	$1.30 * 10^{-4}$	$8.4 * 10^{-14}$	$1.6 * 10^{-16}$
	8. `ship_003`	73.4	156	$8.23 * 10^{-5}$	$2.8 * 10^{-14}$	$9.4 * 10^{-17}$
	9. `raju_001`	211	542	$7.54 * 10^{-5}$	$2.9 * 10^{-15}$	$2.8 * 10^{-16}$
With scaling	2. `thread`	36.7	72	$1.94 * 10^{-3}$	$4.1 * 10^{-16}$	$8.0 * 10^{-17}$
	3. `x104`	22.6	28	$8.73 * 10^{-4}$	$6.2 * 10^{-16}$	$8.0 * 10^{-17}$
	4. `m_t1`	30.0	47	$9.66 * 10^{-4}$	$2.3 * 10^{-15}$	$2.4 * 10^{-16}$
	9. `raju_001`	241	542	$9.93 * 10^{-5}$	$2.6 * 10^{-15}$	$3.3 * 10^{-17}$

We also performed experiments with `static` $= 10^{-8}$. In this case, for problems `x104` and `raju_001` with scaling, a few pivots (7 and 13, respectively) were replaced by `static`. This resulted in a loss of accuracy. For `x104`, performing further iterations of iterative refinement improved the accuracy of the solution but for `raju`, the computed solution had infinity norm $2.5 * 10^4$ (recall the exact solution is the vector of ones), and iterative refinement was unable to improve this. This illustrates the importance of using static pivoting with care.

6 Code availability

`HSL_MA78` is available as part of the 2007 release of the mathematical software library HSL. All use of HSL requires a licence. Individual HSL packages (together with their dependencies and accompanying documentation) are available without charge to individual academic users for their personal (non-commercial) research and for teaching; licences for other uses involve a fee. Details of the packages and how to obtain a licence plus conditions of use are available at `www.cse.clrc.ac.uk/nag/hsl/`.

7 Acknowledgements

I am very grateful to my colleague John Reid, who collaborated with me on the development of `HSL_MA78` and commented on a draft of this paper. I would also like to thank Michael Saunders of Stanford University, both for interesting discussions on scaling and rook pivoting at the Householder Symposium XVII and for his careful reading of this paper.

References

[1] M. Arioli, I. S. Duff, S. Gratton, and S. Pralet. A note on GMRES preconditioned by a perturbed LDLT decomposition with static pivoting. *SIAM J. Scientific Computing*, 29(5):2024–2044, 2007.

[2] X.-W. Chang. Some features of Gaussian elimination with rook pivoting. *BIT*, 42:66–83, 2002.

[3] I.S. Duff. MA57– a new code for the solution of sparse symmetric definite and indefinite systems. *ACM Transactions on Mathematical Software*, 30:118–144, 2004.

[4] I.S. Duff and S. Pralet. Towards a stable static pivoting strategy for the sequential and parallel solution of sparse symmetric indefinite systems. Technical Report RAL-TR-2005-07, Rutherford Appleton Laboratory, 2005.

[5] 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.

[6] L.V. Foster. The growth factor and efficiency of Gaussian elimination with rook pivoting. *Journal of Computational and Applied Mathematics*, 86:177–194, 1997.

[7] P.E. Gill, W. Murray, and M.A. Saunders. SNOPT: An SQP algorithm for large-scale constrained optimization. *SIAM Review*, 47:99–131, 2005.

[8] N.J. Higham and D.J. Higham. Large growth factors in Gaussian elimination with pivoting. *SIAM J. Matrix Analysis and Applications*, 10:155–164, 1989.

[9] HSL. A collection of Fortran codes for large-scale scientific computation, 2007. See http://www.cse.scitech.ac.uk/nag/hsl/.

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

[11] X.S. Li and J.W. Demmel. Making sparse Gaussian elimination scalable by static pivoting. In *Proceedings of Supercomputing*, Orlando, Florida, 1998.

[12] L. Neal and G. Poole. A geometric analysis of Gaussian elimination, II. *Linear Algebra and Applications*, 173:239–264, 1992.

[13] M.J. O'Sullivan and M.A. Saunders. Sparse rank-revealing LU factorization (via threshold complete pivoting and threshold rook pivoting), 2002. Presented at Householder Symposium XV on Numerical Linear Algebra, Peebles, Scotland; available from http://www.stanford.edu/group/SOL/talks.html.

[14] G. Poole and L. Neal. The rook's pivoting strategy. *J. Comp. Appl. Math.*, 123:353–369, 2000.

[15] J.K. Reid and J.A. Scott. HSL_OF01, a virtual memory system in Fortran. Technical Report RAL-TR-2006-026, Rutherford Appleton Laboratory, 2006. Revised Nov. 2007. To appear in *ACM Transactions on Mathematical Software*.

[16] J.K. Reid and J.A. Scott. An out-of-core sparse Cholesky solver. Technical Report RAL-TR-2006-013, Rutherford Appleton Laboratory, 2006. Revised Nov. 2007. Submitted to *ACM Transactions on Mathematical Software*.

[17] J.K. Reid and J.A. Scott. An efficient out-of-core multifrontal solver for large-scale unsymmetric element problems. Technical Report RAL-TR-2007-014, Rutherford Appleton Laboratory, 2007. To appear in *Inter. Journal on Numerical Methods in Engineering*.

[18] E. Rothberg and R. Schreiber. Efficient methods for out-of-core sparse Cholesky factorization. *SIAM J. Scientific Computing*, 21:129–144, 1999.

[19] V. Rotkin and S. Toledo. The design and implementation of a new out-of-core sparse Cholesky factorization method. *ACM Transactions on Mathematical Software*, 30(1):19–46, 2004.

[20] D. Ruiz. A scaling algorithm to equilibrate both rows and columns norms in matrices. Technical Report RAL-TR-2001-034, Rutherford Appleton Laboratory, 2001.

[21] R.D. Skeel. Scaling for numerical stability in Gaussian elimination. *J. ACM*, 26:494–526, 1979.

[22] J.H. Wilkinson. Error analysis of direct methods for matrix inversion. *J. Soc. Indust. Appl. Math*, 10:162–195, 1962.