Preprint

RAL-P-2013-004

# HSL_MI28: an efficient and robust limited-memory incomplete Cholesky factorization code

J Scott, M Tuma

April 2013

Submitted for publication in ACM Transactions on Mathematical Software

RAL Library
STFC Rutherford Appleton Laboratory
R61
Harwell Oxford
Didcot
OX11 0QX

Tel: +44(0)1235 445384
Fax: +44(0)1235 446403
email: libraryral@stfc.ac.uk

Science and Technology Facilities Council preprints are available online at: http://epubs.stfc.ac.uk

**ISSN 1361- 4762**

Neither the Council nor the Laboratory accept any responsibility for loss or damage arising from the use of information contained in any of their reports or in any communication about their tests or investigations.

# HSL_MI28: an efficient and robust limited-memory incomplete Cholesky factorization code

Jennifer Scott[1] and Miroslav Tůma[2]

## ABSTRACT

This paper focuses on the design and development of a new robust and efficient general-purpose incomplete Cholesky factorization package HSL_MI28, which is available within the HSL mathematical software library. It implements a limited memory approach that exploits ideas from the positive semidefinite Tismenetsky-Kaporin modification scheme and, through the incorporation of intermediate memory, is a generalisation of the widely-used ICFS algorithm of Lin and Moré. Both the sparsity density of the incomplete factor and the amount of memory used in its computation are under the user's control. The performance of HSL_MI28 is demonstrated using extensive numerical experiments involving a large set of test problems arising from a wide range of real-world applications. The numerical experiments are used to isolate the effects of the semidefinite modifications and of scaling, ordering and dropping strategies so as to assess their usefulness in the development of robust algebraic incomplete factorization preconditioners and to select default settings for HSL_MI28. They also illustrate the significant advantage of employing a modest amount of intermediate memory. Furthermore, the results demonstrate that, with limited memory, high quality yet sparse general-purpose preconditioners are obtained. Comparisons are made with ICFS, with a level-based incomplete factorization code and, finally, with a state-of-the-art direct solver.

**Keywords:** sparse matrices, sparse linear systems, positive-definite symmetric systems, iterative solvers, preconditioning, incomplete Cholesky factorization.

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

---

[1] Computational Science and Engineering Department, Rutherford Appleton Laboratory, Harwell Oxford, Oxfordshire, OX11 0QX, UK.
Correspondence to: jennifer.scott@stfc.ac.uk
Supported by EPSRC grant EP/I013067/1.

[2] Institute of Computer Science, Academy of Sciences of the Czech Republic.
Partially supported by the Grant Agency of the Czech Republic Project No. P201/13-06684 S. Travel support from the Academy of Sciences of the Czech Republic is also acknowledged.

April 18, 2013

# 1 Introduction

Incomplete Cholesky ($IC$) factorizations have long been an important tool in the armoury of methods for the numerical solution of large sparse symmetric linear systems $Ax = b$. Many papers, books and reports on the development and performance of incomplete factorizations as preconditioners for iterative methods for solving problems from a wide range of practical applications have been published over the last 50 years (for an introduction or overview see, for instance, [5, 36, 38] and the long lists of references therein). A number of software packages have also been developed and made available; the recent paper by George, Gupta and Sarin [14] provides an insightful empirical analysis of some widely-used iterative solvers that include an $IC$ factorization preconditioner option.

An incomplete Cholesky factorization takes the form $LL^T$ in which some of the fill entries (entries that were zero in $A$) that would occur in a complete factorization are ignored. There are many different types of incomplete factorization. An important class are level-based $IC(\ell)$ methods in which the location of permissible fill entries using only the sparsity pattern of $A$ is prescribed in advance. The aim is to partially mimic the way in which the pattern of $A$ is developed during the complete factorization. A symbolic factorization phase is used to assign each potential fill entry a level and an entry is only permitted in the factor if its level is at most $\ell$. Unfortunately, while entries of the error matrix $E = A - LL^T$ are zero inside the prescribed sparsity pattern, outside they can be very large, and the pattern of $IC(\ell)$ (even for large $\ell$) may not guarantee that $L$ is a useful preconditioner (particularly if the matrix entries do not decay significantly with distance from the diagonal). Furthermore, although $IC(1)$ can be a significant improvement over $IC(0)$, the fill-in resulting from increasing $\ell$ can be prohibitive in terms of both storage requirements and the time to compute and then apply the preconditioner.

Another widely-used class of $IC$ factorizations are threshold-based $IC(\tau)$ methods in which the locations of permissible fill entries are determined in conjunction with the numerical factorization of $A$; entries of the computed factors or intermediate quantities that exceed a prescribed drop tolerance $\tau$ are discarded. Success depends on being able to choose a suitable $\tau$ but this can be highly problem dependent. Intuitively, using a small $\tau$ is more likely to produce a high quality preconditioner (measured in terms of the iteration count of a preconditioned Krylov subspace method) than using a larger $\tau$. But as the fill-in increases as $\tau$ is reduced, there is generally a trade-off between sparsity and quality. This approach may work when dropping is determined by the sparsity pattern. Gustafson [16] used the terminology that a preconditioner is more accurate if its structure is a superset of another preconditioner sparsity structure. But this dropping inclusion property, although often valid when solving simple problems, can be very far from reality for tougher problems. Also, improvements in robustness through extending the preconditioner structure by adding simple patterns or by reducing $\tau$ have their limitations and, importantly, for algebraic preconditioning, do not address the problem of memory consumption.

A straightforward approach to overcome the memory problem is to simply prescribe the maximum number of entries allowed in each column of the incomplete factor $L$ and retain only the largest entries. In this case, the dropping inclusion property is often satisfied, that is, by increasing the maximum number of entries, a higher quality preconditioner is obtained. This strategy appears to have been proposed first by Axelsson and Munksgaard [3]. It enabled them to significantly simplify their right-looking implementation as it allowed simple bounds on the amount of memory required. Dropping based on a prescribed upper bound for the largest number of entries in a column of $L$ combined with an efficient strategy to keep track of left-looking updates was implemented in a successful and influential $IC$ code by Jones and Plassman [24, 25]. They retained the $n_j$ largest entries in the strictly lower triangular part of the $j-$th column of $L$, where $n_j$ is the number of entries in the $j-$th column of the strictly lower triangular part of $A$. The code has predictable memory demands and, in the event of factorization breakdown (that is, a zero or negative pivot is encountered), it uses the strategy proposed by Manteuffel [31] of applying a global diagonal shift (so that $A + \alpha I$ is factorized for some positive $\alpha$).

The combination of dropping by magnitude with bounding the number of entries in a column first appeared in [13] but a very popular concept that has predictable memory requirements is the dual threshold

$ILUT(p,\tau)$ factorization of Saad [35]. This is designed for non symmetric problems and combines the use of a drop tolerance $\tau$ with the strategy of prescribed maximum column and row counts. All entries in the computed factors that are smaller in magnitude than $\tau_l$ are discarded, where $\tau_l$ is the product of $\tau$ times the $l_2$-norm of the $l-$th row of $A$. Additionally, only the $p$ largest entries in each column of $L$ and row of $U$ are retained. This approach ignores symmetry in $A$ and, if $A$ is symmetric, the sparsity patterns of $L$ and $U^T$ are normally different.

A widely-used limited-memory $IC$ factorization implementation is provided by the ICFS code of Lin and Moré [28]. It aims to exploit the best features of both the Jones and Plassmann and the Saad factorizations, incorporating $l_2$-norm based scaling and adding a loop for efficiently changing the Manteuffel diagonal shift to prevent breakdown. Given a user-controlled parameter $p$, ICFS retains the $n_j + p$ largest entries in the lower triangular part of the $j-$th column of $L$ and uses only memory as the criterion for dropping entries (thus having both the advantage and disadvantage of not requiring a drop tolerance). Reported results for large-scale trust region subproblems indicate that using additional memory by selecting $p > 0$ can substantially improve performance on difficult problems. However, following [37], let us define the *efficiency* of an $IC$ preconditioner $LL^T$ to be

$$iter \times nz(L), \tag{1.1}$$

where *iter* is the iteration count (number of iterations required by the Krylov subspace method using the preconditioner to achieve the requested accuracy) and $nz(L)$ is the number of entries in $L$. Numerical experimentation reported in [38] has shown that, for a general set of problems arising from a range of applications (see Section 4.1 for details), increasing $p$ improves reliability but the efficiency is not very sensitive to the choice of $p$ (although, of course, the time to compute the factorization and the storage for $L$ increase with $p$).

An alternative approach to incomplete factorizations focuses on avoiding breakdown by incorporating positive semidefinite modifications. The scheme introduced by Jennings and Malik [22, 23] in the late 1970s modifies the corresponding diagonal entries every time an off-diagonal entry is discarded. It can be shown that the sequence of these modifications leads to a breakdown-free factorization for which the error matrix $E$ is a sum of positive semidefinite matrices with non-positive off-diagonal entries and is thus itself positive semidefinite. This approach has been adopted in some engineering applications (and can give good results on moderately ill-conditioned problems) but in the experiments reported on in [38] for a more general test set, it was observed that, in general, the quality of the computed preconditioner was less than was obtained using a global diagonal shift.

A more sophisticated modification scheme is one due to Tismenetsky [42] and which was later significantly enhanced by Kaporin [26]. This approach, which we discuss further in Section 2, introduces the use of intermediate memory that is employed during the construction of $L$ but is then discarded. It has been shown to be very robust but, as Benzi [5] remarks, it "has unfortunately attracted surprisingly little attention". This may be because it suffers from a serious drawback: its memory requirements can be prohibitively high (in some cases, it has been reported that it uses more than 70 per cent of the storage required for a complete Cholesky factorization, see [7]). The dropping strategy of Kaporin may help to alleviate this but a breakdown factorization is no longer guaranteed.

Thi leads us to the two main motivations that lie behind our work. The first is that we would like to develop a generalisation of the successful ICFS algorithm that is such that the efficiency of the $IC$ preconditioner improves with the prescribed memory. Secondly, we would like a memory-efficient variant of the Tismenetsky-Kaporin approach, without diagonal compensation but using a global shift to avoid breakdown. We want to combine both these aims within the development a single "black-box" $IC$ factorization code that is demonstratively robust and efficient on a wide range of problems, while also being flexible in allowing experienced users to tune the parameters to potentially enhance performance for their particular problems of interest. The new package is called `HSL_MI28` and is available within the HSL mathematical software library [20].

The remainder of this paper is organised as follows. In Section 2, we present a brief overview of the

Tismenetsky-Kaporin scheme. We also explain how we can limit the memory requirements and, in so doing, we obtain a generalisation of the ICFS algorithm. An outline of the algorithm implemented by HSL_MI28 is given in Section 3; implementation details and user parameters are discussed. In Section 4, numerical results for HSL_MI28 are presented. In particular, we report on the effects of using intermediate memory and of scaling, ordering and dropping, and provide comparisons with a level-based preconditioner and a modern sparse direct solver. Concluding remarks are made in Section 5.

# 2 A limited memory Tismenetsky-Kaporin modification scheme

## 2.1 The approach of Tismenetsky

The Tismenetsky scheme [42] is based on a matrix decomposition of the form

$$A = LL^T + LR^T + RL^T + E, \tag{2.1}$$

where $L$ is a lower triangular matrix with positive diagonal entries that is used for preconditioning, $R$ is a strictly lower triangular matrix with small entries that is used to stabilise the factorization process, and $E$ has the structure

$$E = RR^T. \tag{2.2}$$

Although HSL_MI28 uses a left-looking implementation, for simplicity, consider for a moment an equivalent right-looking approach: at each step, the next column of $L$ is computed and then the remaining Schur complement is modified. On the $j-$th step, the first column of the Schur complement can be decomposed into a sum of two vectors

$$l_j + r_j,$$

such that $|l_j|^T|r_j| = 0$ (with the first entry in $l_j$ nonzero), where $l_j$ (respectively, $r_j$) contains the entries that are retained in (respectively, discarded from) the incomplete factorization. On the next step of a complete decomposition, the Schur complement of order $n-j$ is updated by subtracting the outer product of the pivot row and column. That is, by subtracting

$$(l_j + r_j)(l_j + r_j)^T.$$

The Tismenetsky incomplete factorization does not compute the complete update as it does not subtract

$$E_j = r_j r_j^T. \tag{2.3}$$

Thus, the positive semidefinite modification $E_j$ is implicitly added to $A$.

As proposed by Tismenetsky, the obvious choice for $r_j$ is the smallest off-diagonal entries in the column (those that are smaller in magnitude than a chosen tolerance). Then at each step of the right-looking formulation, implicitly adding $E_j$ is combined with the standard steps of the Cholesky factorization, with entries dropped from $L$ *after* the updates have been applied to the Schur complement. The approach is naturally breakdown-free because the only modification of the Schur complement that is used in the later steps of the factorization is the addition of the positive semidefinite matrices $E_j$.

## 2.2 Kaporin's second order incomplete Cholesky factorization scheme

While the fill in $L$ can be controlled through the use of a drop tolerance, this does not limit the memory required to compute $L$. A right-looking implementation of a sparse factorization is generally very demanding from the point of view of memory as it is necessary to store all the fill-in for column $j$ until the modification is applied in the step $j$, as follows from (2.3). Hence, a left-looking implementation (or, as in [26], an upward-looking implementation) might be thought preferable. But to compute column $l_j$ and $r_j$ in a left-looking implementation and to apply the modification (2.3) correctly, all the vectors $l_k$ and

$r_k$ for $k = 1, \ldots, j-1$ have to be available. Therefore, the dropped entries have to be stored throughout the left-looking factorization and the $r_k$ cannot be discarded until the factorization is finished (and similarly for an upward-looking implementation). These $r_k$ vectors thus represent intermediate memory. Note the need for intermediate memory is caused not just by the fill in the factorization: it is required because of the structure of the positive semidefinite modification that forces the use of the $r_k$. Sparsity allows some of the $r_k$ to be discarded before the factorization is complete but essentially the total memory is as for a complete factorization, without the other tools that direct methods offer. This memory problem was discussed by Kaporin [26], who proposed using two drop tolerances $\tau_1$, $\tau_2$ with $\tau_1 > \tau_2$. Only entries of magnitude at least $\tau_1$ are kept in $L$ and entries smaller than $\tau_2$ are dropped from $R$. In this case, the error matrix $E$ has the structure

$$E = RR^T + F + F^T,$$

where $F$ is a strictly lower triangular matrix that is not computed while $R$ is used in the computation of $L$ but is then discarded.

When non-zero drop tolerances are introduced, the factorization is no longer guaranteed to be breakdown-free. To avoid breakdown, diagonal compensation (as in the Jennings-Malik scheme discussed in Section 1) for the entries that are dropped from $R$ may be used. Kaporin coined the term *second order incomplete Cholesky factorization* to denote this combined strategy.

In recent years, the Tismenetsky-Kaporin approach has been used to provide a robust preconditioner for some practical applications, see, for example, [2, 4, 26, 29, 30]. We note, however, that there are no reported comparisons with other approaches that take into account not only iteration counts but also the size of the preconditioner; providing some comparisons is one of our aims (see Sections 4.6 and 4.7).

### 2.3 A limited memory Tismenetsky-Kaporin approach

The use of drop tolerances $\tau_1$ and $\tau_2$ can help to reduce the amount of intermediate memory needed. However, picking suitable tolerances can be highly problem dependent: too small and the dropping has little effect, too large and the resulting preconditioner is ineffective. Applying an appropriate scaling can help significantly. In [43], Yamazaki et al use the Tismenetsky-Kaporin approach without diagonal compensation; in the event of breakdown, they employ a global diagonal shift. By restricting their attention to a specific class of problems, they are able to determine an interval of useful drop tolerances that limit the size of the computed factor and to obtain good preconditioners for their examples.

In a recent study [38], we found that, in terms of efficiency (recall (1.1)), using unlimited intermediate memory leads to a good preconditioner. However, efficiency as a measure for assessing how well an algorithm works does not take into account the time required to compute the incomplete factorization and this can be prohibitive (or at least uncompetitive) for the Tismenetsky-Kaporin approach, unless the factorization time can be amortised over several problems. Furthermore, for a number of the largest test problems used in [38], the incomplete factorization failed because of insufficient memory, that is, a memory allocation error was returned. Thus, the Tismenetsky-Kaporin approach can only be considered robust if the problems to be solved are not too large for the available memory. These key issues highlight that the Tismenetsky-Kaporin approach can be impractical for the very problems that we want to solve using an iterative method and motivates us impose on it the use of limited memory.

The memory predictability of our approach depends on specifying a parameter $lsize$ that limits the maximum number of nonzero off-diagonal fill entries in a column of $L$. In practice, we use a slightly more general concept since we may extend $lsize$ for some columns if the maximum allowed space has not been used in a previous step of the factorization. Thus we retain at most $nz(A) + (n-1) * lsize$ off-diagonal entries in the incomplete factorization (where $nz(A)$ is the number of entries in the lower triangular part of $A$ and $n$ is the order of $A$). We employ a second parameter $rsize \geq 0$ that controls the amount of intermediate memory that is used for $R$; it is limited to at most $(n-1) * rsize$ entries. At step $j$, the candidate entries for inclusion in the $j-$th column of $L$ are held in a temporary array. The entries of this array are sorted and (at most) the largest $n_j + lsize$ (plus the diagonal) are retained in $L$; the next $rsize$

largest entries form the $j-$th column of $R$ and all other entries are discarded. Following Kaporin, our implementation offers the option of employing two drop tolerances; entries in $L$ are retained only if they are at least $\tau_1$ in magnitude while those in $R$ must be at least $\tau_2$. Numerical results (see Section 4.4) will illustrate the potential benefits of employing dropping according to the magnitude of the entries.

# 3 An enhanced limited-memory $IC$ factorization

Based on the above discussion, we now summarise our limited-memory $IC$ factorization algorithm and briefly describe its user interface and implementation; further details of how to use the code are supplied in the user documentation that is distributed with the software and is also available at `http://www.hsl.rl.ac.uk/catalogue/`. Given a symmetric sparse matrix $A$, `HSL_MI28` computes an $IC$ factorization $(QL)(QL)^T$, where $Q$ is a permutation matrix. The matrix $A$ is optionally scaled and, if necessary, shifted to avoid breakdown of the factorization, so that the incomplete factorization of $\overline{A} = SAS + \alpha I$ is computed, where $S = \{s_i\}$ is a diagonal scaling matrix and $\alpha$ is a positive shift. The user supplies the lower triangular part of $A$ in compressed sparse column format and the computed $L$ is returned to the user in the same format; a separate entry performs the preconditioning operation $y = Pz$, where $P = (\overline{L}\overline{L}^T)^{-1}$, $\overline{L} = S^{-1}QL$, is the incomplete factorization preconditioner.

## 3.1 Algorithm outline

In Figure 3.1, we present a summary outline of our left-looking $IC$ algorithm. Here $A = \{a_{ij}\}$ and $a_j$, $l_j$ and $r_j$ denote the $j-$th columns of the lower trinagular parts of $A$, $L$ and $R$, respectively; $w$ is a workarray of length $n$. For simplicity of explanation, we assume that each $l_j$ can have at most $lsize$ fill entries (that is, we ignore the use of any spare space that has been passed from a previous column). The scalar $small$ is used to determine whether a diagonal entry is sufficiently large; if at any stage a diagonal entry is less than $small$, the factorization is considered to have broken down and in this case, the shift $\alpha$ is increased and the factorization restarted. Details of the shift strategy are given in Section 3.2.5 (note that in Figure 3.1 we omit the possibility of decreasing the chosen $\alpha$ but this is discussed in Section 3.2.5). $\tau_1 > \tau_2 \geq 0$ are chosen drop tolerances.

The ICFS algorithm of Lin and Moré [28] is a special case in which ordering is not incorporated, $\tau_1 = \tau_2 = 0$ and $rsize = 0$ (so that there is no dropping of entries by size, $R = 0$ and only $LL^T$ updates are applied).

## 3.2 User interface and implementation details

### 3.2.1 Setting the memory limits

The user must supply the following parameters that determine the amount of memory and work involved in computing the incomplete factorization:

`lsize` is the maximum number off fill entries within each column of the incomplete factor $L$. The number of entries in the computed factor is at most $nz(A) + \texttt{lsize} * (n-1)$.

`rsize` is the maximum number of entries within each column of the strictly lower triangular matrix $R$. A rank-1 integer array and a rank-1 real array of size $\texttt{rsize} * (n-1)$ are allocated internally to hold $R$ thus the amount of intermediate memory used, as well as the amount of work involved in computing the preconditioner, depends on `rsize`. If `rsize` = 0, $R$ is not used (and the code is then our implementation of the ICFS algorithm [28]).

### 3.2.2 Control parameters

A number of control parameters provide the experienced user with the means of experimenting with different settings and to tune the code for a particular application. These parameters have default values

Figure 3.1: Outline of the `HSL_MI28` algorithm

*! Reorder (see Section 3.2.4)*
Compute an ordering $Q$ for $A$
Permute the matrix: $A \leftarrow Q^T AQ$

*! Scale (see Section 3.2.3)*
Compute a diagonal scaling $S$
Scale the matrix: $A \leftarrow SAS$

*! Diagonal shift (see Section 3.2.5)*
Choose $\alpha$ such that $\min(a_{ii}) + \alpha > small$
Initialise breakdown = false and $\alpha_0 = 0$

*! Loop over shifts*
Set $w = 0$
**do**
    Set $A \leftarrow A + (\alpha - \alpha_0)I$ and $d(1 : n) = (a_{11}, a_{22}, ..., a_{nn})$

    *! Factorization : loop over columns*
    **for** $j = 1 : n$ **do**
        Copy $a_j$ into $w$
        Apply $LL^T + RL^T + LR^T$ updates from columns $1 : j - 1$ to $w$
        Apply $LL^T + RL^T + LR^T$ updates from columns $1 : j - 1$ to $d(j + 1 : n)$
        Optionally apply $RR^T$ updates from columns $1 : j - 1$ that cause
            no additional fill-in in $w$    *! User-controlled option (see Section 3.2.6)*
        **if** $(\min(d(j + 1 : n)) < small)$ **then**
            Set breakdown = true, $\alpha_0 = \alpha$ and increase $\alpha$    *! (see Section 3.2.5)*
            **exit**
        **end if**
        Sort entries in $w$ by magnitude
        Keep at most $n_j + lsize$ entries of largest magnitude in $l_j$ such that
            they are all at least $\tau_1$
        Keep at most $rsize$ additional entries that are next largest in magnitude
            in $r_j$ such that they are all at least $\tau_2$
        Reset entries of $w$ to zero
    **end do**
    **if** breakdown = false **exit**
**end do**

that we have chosen on the basis of numerical experimentation (see Section 4 for results that support our choices); for some problems, selecting different values can be beneficial but the default settings have been found to be generally robust. In particular, we have two parameters, `tau1` ($= \tau_1$) and `tau2` ($= \tau_2$), that control dropping of small entries from $L$ and $R$, respectively. Note that, by allowing `lsize` and `rsize` to be sufficiently large, the user can force the code to simulate the dropping strategy of Suarjana and Law [41] and Kaporin [26]. But, as we will see from our experimental results, for our test examples this is unnecessary since limiting memory appears to afford a more practical approach. The dropping parameters have default values `0.001` and `0.0001`, respectively; dropping based on the size of entries is disabled by setting both to zero. Observe that these values, that were chosen on the basis of our general set of scaled matrices, are significantly smaller that those that are typically recommended (see, for example, [5, 26]).

#### 3.2.3 Scaling options

For scaling and ordering $A$ before the factorization begins, `HSL_MI28` is able to offer a range of options by taking advantage of existing HSL routines. The default scaling is $l_2$ scaling, in which the entries in column $j$ of $A$ are normalised by the 2-norm of column $j$; this is used in ICFS [28]. We also offer diagonal scaling, scaling based on maximum matching using a symmetrized version of the package `MC64` [11, 12], and equilibration scaling using `MC77` [33, 34]; in addition, there is a facility for the user to supply a scaling.

#### 3.2.4 Ordering options

By default, a profile reduction ordering using a variant of the Sloan algorithm [32, 39, 40] is employed (this variant is implemented by the HSL package `MC61`). Other orderings that are currently offered are reverse Cuthill McKee (RCM) [8] (again, implemented within `MC61`), approximate minimum degree [1] (`HSL_MC68`), nested dissection (currently, provided by routine `METIS_NodeND` from the METIS package [27] but it is anticipated that this may be replaced by an HSL package in the future), and ordering of the rows by ascending degree. In addition, an option is available for the user to supply an ordering; this is convenient if a series of problems having the same sparsity pattern is to be solved.

#### 3.2.5 Coping with breakdown

In the event of breakdown within the factorization, `HSL_MI28` employs a global diagonal shift. It is important to try and use as small a shift as possible but also to limit the number of breakdowns. An option exists for the user to supply an initial (positive) shift $\alpha_0$. Otherwise, $\alpha_0$ is computed as in [28] so that, if $\beta = \min(s_i^2 a_{ii}) > 0$, then $\alpha_0 = 0.0$; otherwise, $\alpha_0 = -\beta + \texttt{lowalpha}$, where $\texttt{lowalpha} > 0$ may be chosen by the user. The incomplete factorization algorithm is applied to $\overline{A}_0 = SAS + \alpha_0 I$. If breakdown occurs, a larger shift

$$\alpha_1 = \max(\texttt{lowalpha},\ \ \alpha_0 \times \texttt{shift_factor}), \tag{3.1}$$

with $\texttt{shift_factor} > 1$, is tried. The process continues until an incomplete factorization of $\overline{A}_k = SAS + \alpha_k I$ is successful. If breakdown occurs at the same (or nearly the same) stage of the factorization for two successive shifts, we try to limit the number of restarts by more rapidly increasing the shift. In this case, we increase $\alpha$ by a factor of $2 \times \texttt{shift_factor}$. Conversely, if $\alpha_k = \texttt{lowalpha}$, to prevent an unnecessarily large shift from being used, we try decreasing $\alpha$. We first take a copy of the successful factorization (if there is insufficient memory to do this, we accept the successful factorization), then set

$$\alpha_{k+1} = \alpha_k / \texttt{shift_factor2}, \tag{3.2}$$

with $\texttt{shift_factor2} > 1$, and apply the incomplete factorization algorithm to $\overline{A}_{k+1} = SAS + \alpha_{k+1} I$. If this factorization is also breakdown free, we repeat (up to `maxshift` times); otherwise we use the stored factorization. In all cases, the value of the final shift is returned to the user, along with the number of shifts tried and the number of restarts.

In summary, the parameters within `HSL_MI28` that determine the initial and subsequent choice of the shift $\alpha$ and their default settings are as follows:

`alpha` holds the initial shift $\alpha$. It has default value zero.

`lowalpha` controls the choice of the first non-zero shift. The default value is 0.001.

`maxshift` determines the maximum number of times the shift can be decreased after a successful factorization with a positive shift. Limiting `maxshift` may reduce the factorization time but may result in a poorer quality preconditioner. It has default value `3`.

`shift_factor` controls how rapidly the shift is increased after a breakdown. Increasing `shift_factor` rapidly may reduce the factorization time but may result in a poorer quality preconditioner. It has default value `2`.

`shift_factor2` controls how rapidly the shift is decreased after a successful factorization with $\alpha =$ `lowalpha`. It has default value `4`.

In all the experiments reported on in Section 4, we use the default settings for these parameters (which were selected on the basis of experimentation).

#### 3.2.6 Other control parameters

As well as the parameters already described and parameters that are used to control diagnostic printing, `HSL_MI28` uses the following control parameters:

`rrt` is used to control whether entries of $RR^T$ that cause no additional fill-in are included (`rsize` $>$ `0` only). Experiments in [38] found that allowing such entries can improve the quality of the preconditioner but this is not guaranteed. Such entries are allowed if `rrt` = `.true.`. The default is `rrt` = `.false.`

`small` is used to decide when factorization breakdown has occurred. Any pivot whose modulus is less than `small` is treated as zero and, if such a pivot is encountered, the factorization breaks down, the shift is increased and the factorization restarted (see Figure 3.1). The default value in the double precision version of the package is $10^{-20}$ and in the single version it is $10^{-12}$.

# 4 Numerical experiments

## 4.1 Test environment

All the numerical results reported on in this paper are performed (in serial) on our test machine that has two Intel Xeon E5620 processors with 24 GB of memory. The ifort Fortran compiler (version 12.0.0) with option -O3 is used. The implementation of the conjugate gradient (CG) algorithm offered by the HSL routine `MI22` is employed, with starting vector $x_0 = 0$, the right-hand side vector $b$ computed so that the exact solution is $x = 1$, and stopping criteria

$$\|A\hat{x} - b\|_2 \leq 10^{-10}\|b\|_2, \tag{4.1}$$

where $\hat{x}$ is the computed solution. In addition, for each test we impose a limit of 2000 CG iterations.

In assessing the effectiveness of a preconditioner we use the definition of efficiency given by (1.1). A weakness of this measure is that it does not taken into account the number of entries in $R$. If `lsize` is fixed, increasing `rsize` will generally lead to a more efficient preconditioner. However, this will be at the cost of additional work in the incomplete factorization. Thus we record the time to compute the preconditioner together with the time for convergence of the iterative method: the sum of these is referred to as the *total time* and is also be used to assess the quality of the preconditioner. Note that in our tests, a simple matrix-vector product routine is used with the lower triangular part of $A$ held in compressed sparse column format: we have not attempted to perform either the matrix-vector products or the application of the preconditioner in parallel and all times are serial times.

Our test problems are real positive-definite matrices of order at least 1000 taken from the University of Florida Sparse Matrix Collection [9]. Many papers on preconditioning techniques and iterative solvers select a small set of test problems that are somehow felt to be representative of the applications of interest. However, as in [38], our interest is more general and we want to test our new software on as wide a range of problems as we can. Thus we took all such problems and then removed any that were diagonal matrices and, where there was more than one problem with the same sparsity pattern, we chose only one representative problem. This resulted in a test set of 153 problems of order up to 1.5 million. Following initial experiments, 8 problems were removed from this set as we were unable to achieve convergence to the required accuracy within our limit of 2000 iterations without allowing a large amount of fill. To assess performance on our test set and compare different settings and options, we use performance profiles [10].

## 4.2 Results for `lsize` fixed and `rsize` varying

As reported in [38], increasing `lsize` with `rsize` = `0` does little to improve the preconditioner quality (in terms of efficiency), although it can improve reliability. We thus start by examining the effects of introducing intermediate memory. We set the drop tolerances to zero, employ $l_2$ scaling, no reordering, and fix `lsize` = `5`. We run with no intermediate memory, `rsize` = `2`, `5`, and `10`, and with unlimited intermediate memory (all entries in $R$ are retained, which we denote by `rsize` = `-1`). We remark that the latter is not an option offered by `HSL_MI28` but corresponds to the original Tismenetsky approach. Figure 4.1 presents the efficiency performance profile (on the right) and total time performance profile (on the left). Since `lsize` is the same for all runs, the fill in $L$ is essentially the same in each case and thus comparing the efficiency here is equivalent to comparing the iteration counts. Note that the asymptotes of the performance profile provide a statistic on reliability (that is, the proportion of problems that are successfully solved) and as the curve for `rsize` = `-1` lies below the others on the right-hand axis of the profiles in Figure 4.1, this indicates poorer reliability when using unlimited memory. This poor reliability is because, as anticipated, in this case there was insufficient memory to factorize a number of our large test problems. Moreover, in terms of time as well as memory, unlimited intermediate memory is the most

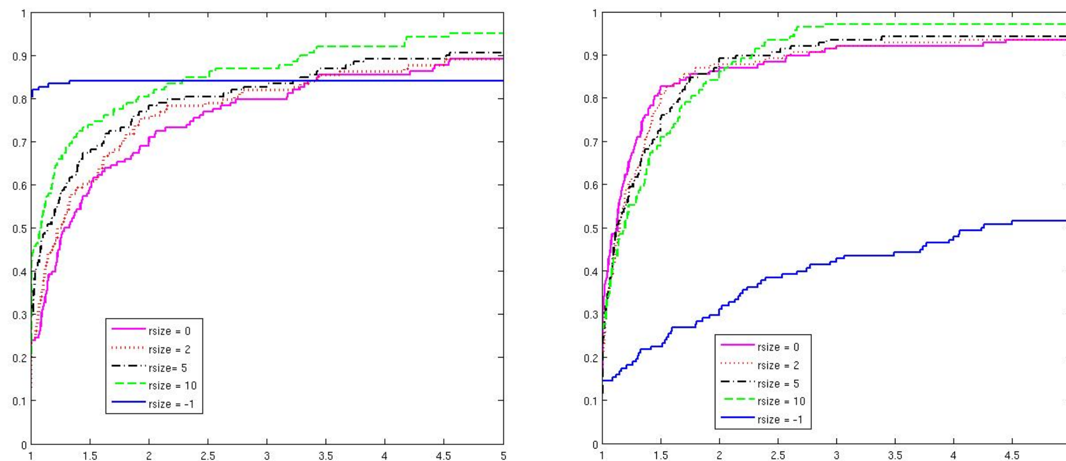


Figure 4.1: Efficiency (left) and total time (right) performance profiles for `rsize` varying.

expensive option. With the introduction of a limit on memory, we see that, as `rsize` is increased from `0` to `10`, the efficiency and robustness of the preconditioner steadily increases, without significantly increasing the total time. Since a larger value of `rsize` reduces the number of iterations required, if more than one problem is to be solved with the same preconditioner, it may be worthwhile to increase `rsize` in this case

(but the precise choice of `rsize` is not important).

### 4.3 Effect of scaling

The importance of scaling in the solution of linear systems is well known. In Figure 4.2, we illustrate how scaling effects the performance of `HSL_MI28`. The code is run with `lsize` = `rsize` = `5`, drop tolerances `tau1` = `0.001`, `tau2` = `0`, and no ordering. We see that, considering the test set as a whole (a set that contains some problems that are initially well-scaled and others that are initially poorly scaled), scaling can significantly improve performance. Interestingly, each of the tested scalings gives a preconditioner of similar quality. If no scaling is used, the number of restarts following factorization breakdown can be large, the final shift can be large and the computed $IC$ preconditioner of poor quality. We remark that the study by Hogg and Scott [17, 19] into the effects of scalings on the performance of direct solvers for the solution of sparse symmetric indefinite systems found `MC64` to be expensive but it produced the highest quality scalings, particularly for some "tough" problems (problems that a direct solver can struggle to solve efficiently as well as accurately). `MC64` has also been used to advantage in the non symmetric case (for example, Benzi, Haws and Tuma [6] report on the beneficial effects of scalings to place large entries on the diagonal when computing incomplete factorization preconditioners for use with Krylov subspace methods). However, since it is expensive, does not significantly reduce the number of factorization breakdowns compared with the other scalings, and does not lead to higher quality $IC$ preconditioners for our test problems, it is not the default scaling within `HSL_MI28`; the default is the much cheaper and simpler $l_2$ scaling.

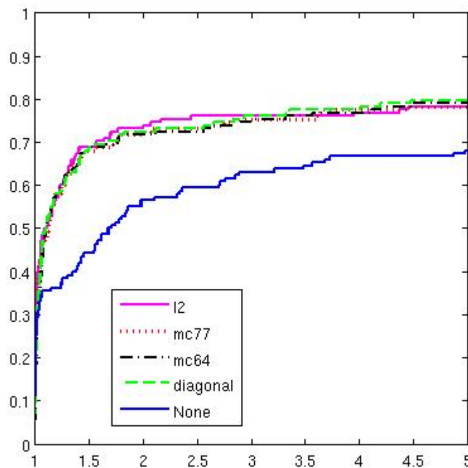


Figure 4.2: Efficiency performance profile for different scalings.

### 4.4 Effect of dropping

Figure 4.3 presents an efficiency performance profile (on the left) and an iteration performance profile (on the right) for the drop tolerance `tau1` in the range `0` (no dropping) and `0.0005`. Here we use `lsize`=`rsize`=`5`, $l_2$ scaling, no reordering and `tau2` = `0` (so that small entries are dropped from $L$ but dropping is not used on $R$). In terms of efficiency, it is clearly advantageous to use a small drop tolerance greater than `0` but, if `tau1` is increased too far, reliability of the computed preconditioner is reduced. Looking at the iteration performance profile, we see that, as expected, dropping entries reduces the quality of the preconditioner but for small `tau1` the number of extra iterations is generally modest. We have selected `tau1` = `0.001` as the default setting in `HSL_MI28`. Provided the problem has been scaled, we

found that also using a non-zero value for tau2 has a relatively small but beneficial effect on the efficiency. We have chosen the default value to be tau2 = 0.0001.

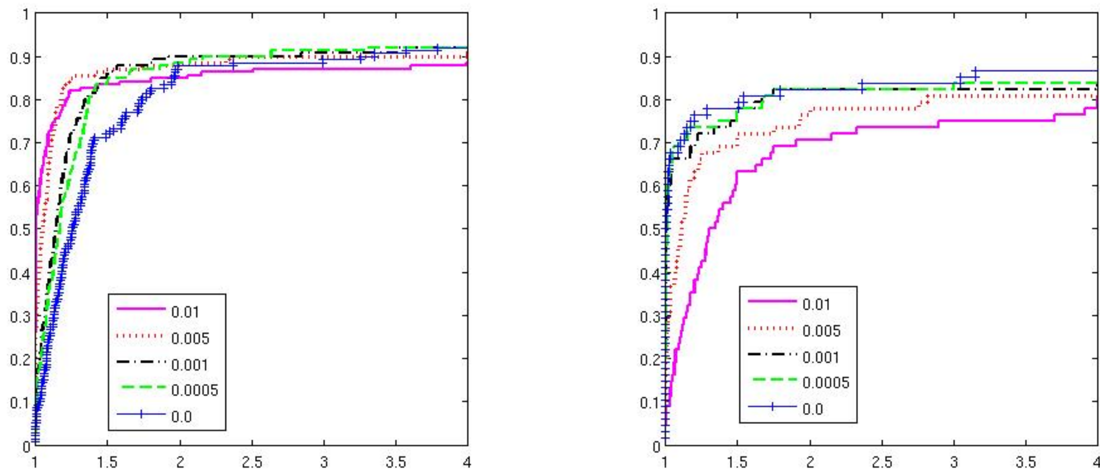


Figure 4.3: Efficiency (left) and iteration count (right) performance profiles for tau1 varying.

## 4.5 Effect of ordering

The effects of sparse matrix orderings on the convergence of preconditioned Krylov subspace methods have been widely reported on in the literature (the survey by Benzi [5], for example, contains a large number of references and a brief summary discussion). In Figure 4.4, we compare the performance of the ordering algorithms offered by HSL_MI28 (with the same settings as above and $l_2$ scaling). Our results agree with previously reported findings; in particular, minimum degree and nested dissection perform less well than the natural order and this, in turn, is outperformed by RCM and the Sloan algorithms. It is interesting to observe how well the Sloan algorithm does: it is significantly better (in terms of efficiency and reliability) than the more widely-used RCM ordering and, as its use adds very little to the time to compute the factorization, we have made it the default ordering within HSL_MI28. Closer examination of the results shows that the Sloan ordering gives the best results across the range of problems tested (in particular, it gives the same kind of overall performance improvement for the largest problems as for the smallest).

## 4.6 Comparison with a level-based approach

In an earlier paper, Scott and Tůma [37] considered level-based preconditioning and presented an improved strategy that considers the individual entries of the system matrix and restricts small entries to contributing to fewer levels of fill than the largest entries. Their numerical results showed that preassigning levels of fill can be beneficial. In Figure 4.5, we present an efficiency performance profile (on the left) and an iteration performance profile (on the right) that compare the performance of the level-based preconditioner $IC(\ell)$ with $\ell = 3$ with that of HSL_MI28. The settings for computing $IC(3)$ were those found by Scott and Tůma to give the best performance. For $IC(3)$ and HSL_MI28 we use $l_2$ scaling and no reordering. We see that the HSL_MI28 preconditioner is more efficient than $IC(3)$. However, while the $IC(3)$ incomplete factor is much denser than that produced by HSL_MI28, it requires fewer iterations. In terms of reliability, $IC(3)$ failed to give convergence for 19 problems, while with lsize = rsize = 5, the HSL_MI28 preconditioner failed on 8 problems and with lsize = 20 and 40, all the problems were solved. These findings suggest that the use of intermediate memory can have a positive effect compared with using the concept of levels.

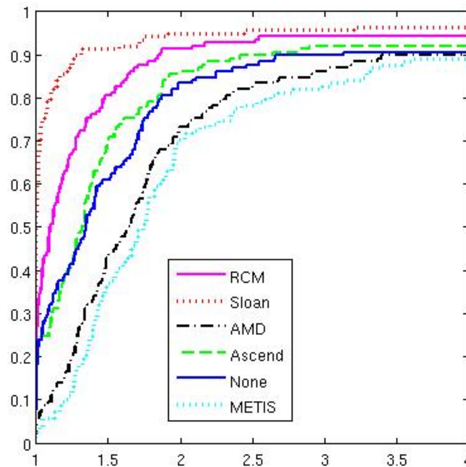


Figure 4.4: Efficiency performance profile for different orderings. RCM is the reverse Cuthill-McKee bandwidth minimisation algorithm, Sloan is a profile reduction algorithm, AMD is an approximate minimum degree ordering, Ascend is ordering by ascending degree, METIS is a nested dissection ordering, and None indicates the supplied (natural) ordering.

However, as level-based factors are denser and are more structured and they offer the possibility of being computed in parallel [21], they remain a useful approach to incomplete factorization.

### 4.7 Comparisons with a direct solver

It is of interest to examine how well the preconditioned conjugate method performs in comparison with a modern direct solver. In Figure 4.6, we present total time performance profiles that compare `HSL_MI28` with the ICFS code of Lin and Moré [28] and the modern direct solver `HSL_MA97` [18] (Version 2.1.0). Although `HSL_MA97` is a parallel code, here we run it in serial (since our current incomplete factorization code is a serial code); all parameter settings within `HSL_MA97` are the default settings and we use the Intel MKL BLAS (10.3.0). ICFS is run with $p = 5$ (a maximum of 5 fill entries in each column of $L$) and for `HSL_MI28` we use `lsize=rsize=5` plus default settings for the control parameters. The left-hand profile includes all problems in our test set while that on the right is restricted to the set of 43 problems for which the total ICFS time is at least 1 second. On small problems, ICFS performs well; it is a much simpler code than the others and includes no overhead for ordering or for the other options that are available within `HSL_MI28`. For larger problems, we see the benefits of incorporating reordering and using `rsize` $>$ `0`, with `HSL_MI28` performing significantly better than ICFS. For these problems, the direct solver is clearly the best approach; it employs a block algorithm and is able to make extensive use of Level 3 BLAS to enhance performance. However, memory limitations mean `HSL_MA97` was unable to solve two of the largest problems that were successfully solved by `HSL_MI28` and ICFS. Thus the results demonstrate the potential of `HSL_MI28` to be used to efficiently solve problems that are too large to be tackled by a direct solver but also suggests a need to incorporate techniques used by the direct solver into the incomplete factorization (see, for example, [15]).

## 5 Concluding remarks

In this paper, we have discussed the design and development of a new software package `HSL_MI28` for incomplete Cholesky factorizations. The algorithm implemented within `HSL_MI28` is based on the robust

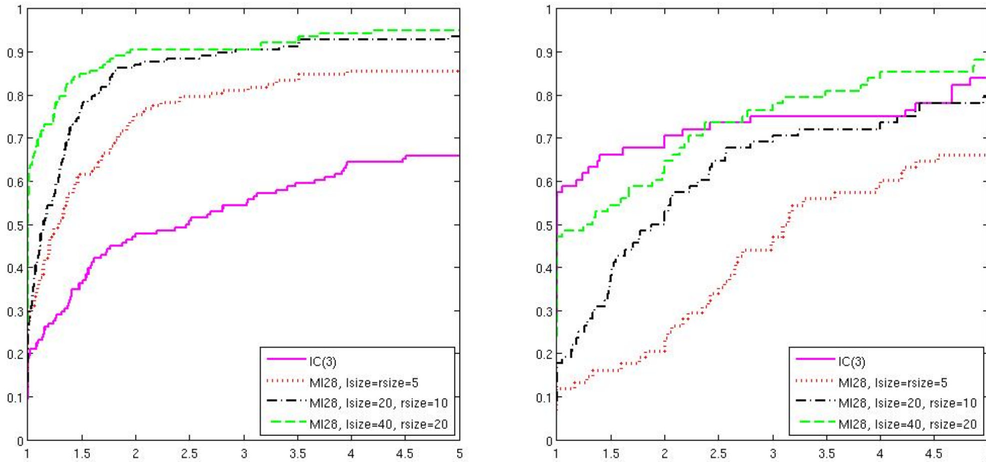


Figure 4.5: Efficiency and iteration performance profiles for level-based and memory-based methods.

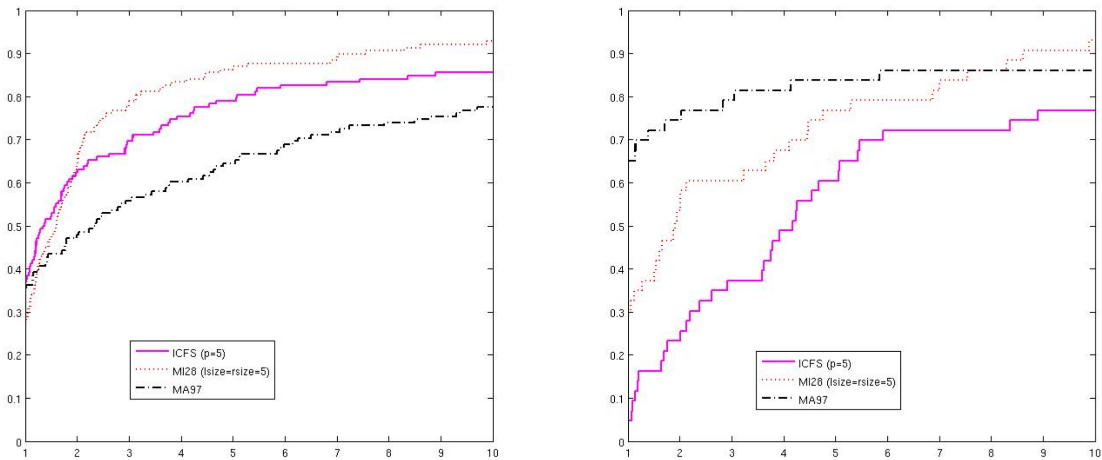


Figure 4.6: Total time performance profile for direct and iterative methods (all problems on left, large problems only on right).

Tismenetsky-Kaporin scheme, which we have made into a practical approach through the introduction of limited memory for both the incomplete factor $L$ and the lower triangular matrix $R$ that is used to stabilise the factorization but is subsequently discarded. Through the use of $R$ and the option to drop entries by size, HSL_MI28 generalises the ICFS code. Because our previous work [38] found that, in many instances, the use of a diagonal compensation scheme to prevent factorization breakdown can lead to a poor quality preconditioner, HSL_MI28 follows ICFS in using a global diagonal shift.

HSL_MI28 can be used as a black box package. However, through the inclusion of control parameters that have default settings but that can be reset by the user, the design is such that the experienced user can tune the code to optimise its performance on his or her problems of interest.

We have presented the results of extensive numerical experiments. In our opinion, using a large test set drawn from a range of practical applications allows us to make general and authoritative conclusions. The experiments emphasise the concept of the efficiency of a preconditioner (defined by (1.1)) as a measure that can be employed to help capture preconditioner usefulness, although we recognise that it can also be necessary to consider other statistics (such as the time and the number of iterations). The experiments have been used to isolate the effects of ordering, scaling and dropping. An interesting observation is that the ordering that gives the best results overall is the Sloan profile reduction algorithm, rather than the more widely-used RCM ordering.

Our results suggest that Tismenetsky-type updates are better able to stabilise the decomposition than the concept of levels (even in the enhanced form introduced recently in [37]). Thus we conclude that memory-limited decompositions are not only possible but are also highly effective and a potentially useful addition to the armoury of tools for solving sparse linear systems.

HSL_MI28 is available as part of the 2013 release of the HSL mathematical software library. All use of HSL requires a licence; licences are available to academics without charge for individual research and teaching purposes. Details of how to obtain a licence and the code are available at http://www.hsl.rl.ac.uk or by email to hsl@stfc.ac.uk.

# References


[1] P. R. Amestoy, T. A. Davis, and I. S. Duff. Algorithm 837: AMD, an approximate minimum degree ordering algorithm. *ACM Transactions on Mathematical Software*, pages 381–388, 2004.

[2] O. Axelsson, I. Kaporin, I. Konshin, A. Kucherov, M. Neytcheva, B. Polman, and A. Yeremin. Comparison of algebraic solution methods on a set of benchmark problems in linear elasticity. Final Report of the STW project NNS.4683, Department of Mathematics, University of Nijmegen, 2000.

[3] O. Axelsson and N. Munksgaard. Analysis of incomplete factorizations with fixed storage allocation. In *Preconditioning Methods: Analysis and Applications*, volume 1 of *Topics in Computational Mathematics*, pages 219–241. Gordon & Breach, New York, 1983.

[4] L. Beirão da Veiga, V. Gyrya, K. Lipnikov, and G. Manzini. Mimetic finite difference method for the Stokes problem on polygonal meshes. *J. of Computational Physics*, 228(19):7215–7232, 2009.

[5] M. Benzi. Preconditioning techniques for large linear systems: a survey. *J. of Computational Physics*, 182(2):418–477, 2002.

[6] M. Benzi, J. C. Haws, and M. Tůma. Preconditioning highly indefinite and nonsymmetric matrices. *SIAM J. on Scientific Computing*, 22(4):1333–1353, 2000.

[7] M. Benzi and M. Tůma. A robust incomplete factorization preconditioner for positive definite matrices. *Numerical Linear Algebra with Applications*, 10(5-6):385–400, 2003.

[8] E. H. Cuthill and J. McKee. Reducing the bandwidth of sparse symmetric matrices. In *Proc. $24^{th}$ National Conference of the ACM*, pages 157–172. ACM Press, 1969.

[9] T. A. Davis and Y. Hu. The University of Florida sparse matrix collection. *ACM Transactions on Mathematical Software*, 38(1), 2011.

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

[11] I. S. Duff and J. Koster. On algorithms for permuting large entries to the diagonal of a sparse matrix. *SIAM J. on Matrix Analysis and Applications*, 22:973–996, 2001.

[12] I. S. Duff and S. Pralet. Strategies for scaling and pivoting for sparse symmetric indefinite problems. *SIAM J. on Matrix Analysis and Applications*, 27:313–340, 2005.

[13] R. W. Freund and N. M. Nachtigal. An implementation of the look-ahead Lanczos algorithm for non-Hermitian matrices, part II. Technical Report TR 90-46, RIACS, NASA Ames Research Center, 1990.

[14] T. George, A. Gupta, and V. Sarin. An empirical analysis of the performance of preconditioners for SPD systems. *ACM Transactions on Mathematical Software*, 38(4):24:1–24:30, August 2012.

[15] A. Gupta and T. George. Adaptive techniques for improving the performance of incomplete factorization preconditioing. *SIAM J. on Scientific Computing*, 1:84–110, 2010.

[16] I. Gustafsson. Modified incomplete Cholesky (MIC) methods. In *Preconditioning methods: analysis and applications*, volume 1 of *Topics in Computational Mathematics*, pages 265–293. Gordon & Breach, New York, 1983.

[17] J. D. Hogg and J. A. Scott. The effects of scalings on the performance of a sparse symmetric indefinite solver. Technical Report RAL-TR-2008-007, 2008.

[18] J. D. Hogg and J. A. Scott. HSL_MA97: a bit-compatible multifrontal code for sparse symmetric systems. Technical Report RAL-TR-2011-024, 2011.

[19] J. D. Hogg and J. A. Scott. A study of pivoting strategies for tough sparse indefinite systems. Technical Report RAL-TR-2012-009, 2012.

[20] HSL. A collection of Fortran codes for large-scale scientific computation, 2013. `http://www.hsl.rl.ac.uk`.

[21] D. Hysom and A. Pothen. A scalable parallel algorithm for incomplete factor preconditioning. *SIAM J. on Scientific Computing*, 22:2194–2215, 2001.

[22] A. Jennings and G. M. Malik. Partial elimination. *J. of the Institute of Mathematics and its Applications*, 20(3):307–316, 1977.

[23] A. Jennings and G. M. Malik. The solution of sparse linear equations by the conjugate gradient method. *International J. of Numerical Methods in Engineering*, 12(1):141–158, 1978.

[24] M. T. Jones and P. E. Plassmann. Algorithm 740: Fortran subroutines to compute improved incomplete Cholesky factorizations. *ACM Transactions on Mathematical Software*, 21(1):18–19, 1995.

[25] M. T. Jones and P. E. Plassmann. An improved incomplete Cholesky factorization. *ACM Transactions on Mathematical Software*, 21(1):5–17, 1995.

[26] I. E. Kaporin. High quality preconditioning of a general symmetric positive definite matrix based on its $U^TU + U^TR + R^TU$ decomposition. *Numerical Linear Algebra with Applications*, 5:483–509, 1998.

[27] G. Karypis and V. Kumar. METIS: A software package for partitioning unstructured graphs, partitioning meshes, and computing fill-reducing orderings of sparse matrices (version 3.0). Technical report, University of Minnesota, Department of Computer Science and Army HPC Research Center, October 1997.

[28] C.-J. Lin and J. J. Moré. Incomplete Cholesky factorizations with limited memory. *SIAM J. on Scientific Computing*, 21(1):24–45, 1999.

[29] K. Lipnikov, M. Shashkov, D. Svyatskiy, and Yu. Vassilevski. Monotone finite volume schemes for diffusion equations on unstructured triangular and shape-regular polygonal meshes. *J. of Computational Physics*, 227(1):492–512, 2007.

[30] K. Lipnikov, D. Svyatskiy, and Y. Vassilevski. Interpolation-free monotone finite volume method for diffusion equations on polygonal meshes. *J. of Computational Physics*, 228(3):703–716, 2009.

[31] T. A. Manteuffel. An incomplete factorization technique for positive definite linear systems. *Mathematics of Computation*, 34:473–497, 1980.

[32] J. K. Reid and J. A. Scott. Ordering symmetric sparse matrices for small profile and wavefront. *International J. of Numerical Methods in Engineering*, 45:1737–1755, 1999.

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

[34] D. Ruiz and B. Uçar. A symmetry preserving algorithm for matrix scaling. Technical Report INRIA RR-7552, INRIA, Grenoble, France, 2011.

[35] Y. Saad. ILUT: a dual threshold incomplete *LU* factorization. *Numerical Linear Algebra with Applications*, 1(4):387–402, 1994.

[36] Y. Saad. *Iterative Methods for Sparse Linear Systems.* PWS Publishing Co., Boston, 1996.

[37] J. A. Scott and M. Tůma. The importance of structure in incomplete factorization preconditioners. *BIT Numerical Mathematics*, 51:385–404, 2011.

[38] J. A. Scott and M. Tůma. On positive semidefinite modification schemes for incomplete Cholesky factorization. Technical Report RAL-P-2013-005, 2013.

[39] S. W. Sloan. An algorithm for profile and wavefront reduction of sparse matrices. *International J. of Numerical Methods in Engineering*, 23:239–251, 1986.

[40] S. W. Sloan. A Fortran program for profile and wavefront reduction. *International J. of Numerical Methods in Engineering*, 28:2651–2679, 1989.

[41] M. Suarjana and K. H. Law. A robust incomplete factorization based on value and space constraints. *International J. of Numerical Methods in Engineering*, 38:1703–1719, 1995.

[42] M. Tismenetsky. A new preconditioning technique for solving large sparse linear systems. *Linear Algebra and its Applications*, 154–156:331–353, 1991.

[43] I. Yamazaki, Z. Bai, W. Chen, and R. Scalettar. A high-quality preconditioning technique for multi-length-scale symmetric positive definite linear systems. *Numerical Mathematics: Theory, Methods and Applications*, 2(4):469–484, 2009.