BIT

The importance of structure in incomplete factorization preconditioners

Jennifer Scott · Miroslav Tůma

Abstract In this paper, we consider level-based preconditioning, which is one of the basic approaches to incomplete factorization preconditioning of iterative methods. It is well-known that while structure-based preconditioners can be very useful, excessive memory demands can limit their usefulness. Here we present 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. Using symmetric positive-definite problems arising from a wide range of practical applications, we show that the use of variable levels of fill can yield incomplete Cholesky factorization preconditioners that are more efficient than those resulting from the standard level-based approach. The concept of level-based preconditioning, which is based on the structural properties of the system matrix, is then transferred to the numerical incomplete decomposition. In particular, the structure of the incomplete factorization determined in the symbolic factorization phase is explicitly used in the numerical factorization phase. Further numerical results demonstrate that our level-based approach can lead to much sparser but efficient incomplete factorization preconditioners.

J. Scott (✉)
Computational Science and Engineering Department, Atlas Centre, Rutherford Appleton Laboratory, Oxfordshire OX11 0QX, UK
e-mail: jennifer.scott@stfc.ac.uk

M. Tůma
Institute of Computer Science, Academy of Sciences of the Czech Republic, Pod Vodárenskou věží 2, 18207 Praha 8, Libeň, Czech Republic
e-mail: tuma@cs.cas.cz

Mathematics Subject Classification (2000) 65F08 · 65F50

1 Introduction

Incomplete Cholesky factorizations are an important tool in the solution of large sparse symmetric linear systems of equations $Ax = b$. Preconditioners based on an incomplete factorization of A (that is, a factorization in which some of the fill entries and possibly some of the entries of A are ignored) fall into three main classes:

(i) 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 that exceed a prescribed threshold τ are dropped. Success of this approach depends on being able to choose a suitable τ and this is highly problem dependent.

(ii) Memory-based $IC(m)$ methods in which the amount of memory available for the incomplete factorization is prescribed and only the largest entries in each column are retained.

(iii) Structure-based $IC(\ell)$ methods in which an initial symbolic factorization phase determines the location of permissible fill entries using only the sparsity pattern of A. Each potential fill entry is assigned a level and an entry is only permitted in the factor if its level is at most ℓ. This kind of fixed fill strategy allows the memory requirements to be determined before the second phase that performs an incomplete numerical factorization.

Many refinements, variants and hybrids of the above approaches have been proposed and used to solve problems from a wide range of application areas. In Sect. 2, we provide a brief historical overview and highlight some of the important developments in the field over the past 50 years. We are interested in structure-based incomplete factorization preconditioners that have both predictable memory requirements and depend on the entries of A. We propose a general class of methods based on computing an incomplete LDL^T factorization $IC(\ell, \tau, m)$, where the factor L is unit lower triangular and D is diagonal, $\ell \geq 0$ is the target number of levels of fill, τ is a drop tolerance and m controls the maximum number of entries allowed in the factor. In Sect. 3, we introduce a modification to the standard level-based approach. Rather than allowing all the non-zero entries of A to contribute to ℓ levels of fill, we restrict small entries to contributing to fewer levels and allow the largest entries to contribute to more than ℓ levels. We explain how this variable level approach can be implemented using a minor change to an existing algorithm for computing a symbolic incomplete factorization. Then, in Sect. 4, we consider transferring the structure of the symbolic incomplete factorization to the numerical factorization, allowing extra entries outside the symbolic pattern if sufficient memory is available and the entries are large enough. Numerical results that illustrate the effectiveness of our proposed level-based strategy for practical applications are presented in Sect. 5 and, finally, some concluding remarks are made in Sect. 6.

2 Background

Sparsity structure was the main ingredient of the first algebraic preconditioners that were developed in the late 1950s. At that time, the sparsity structure essentially expressed the stencils for discretized partial differential equations on structured grids. In particular, the EBM-2 method of Buleev [12] interpolated values of the function at a grid point using a combination of the function values at neighbouring grid points. The solution process was accelerated by additional parametrization derived from smoothness assumptions. The resulting system expressed what we call now the preconditioned system of equations, in which the preconditioner was directly combined with the system matrix. The method was generalized to stencils for three dimensional problems in [13]. An independent derivation and its interpretation as an incomplete factorization (that is, a factorization in which some of the fill entries are ignored) for a matrix from a simple 5-point stencil was given by Varga [42] (see also [3, 34]). Note that [42] is also well-known for introducing the concept of regular splittings. Further early developments included additional corrections that led to heavily parametrized procedures and included more complicated stencils. Later, modifications of the interpolation that implicitly expressed incomplete decompositions were proposed to change in individual steps of the iterative procedure [41]. An overview of the early procedures and the motivations behind them may be found in [26, 27].

Further developments for incomplete factorizations included their classification by the order (first or second order) of the polynomial defining the interpolation on the grid points, extensions to larger stencils, and the development of early matrix formulations and existence criteria for breakdown-free factorizations. The key relation that has been gradually better understood is that between stencil multiplications, local interpolation and extrapolation on a grid, and the combinatorial elimination process (that is, the elimination process based on a graph structure).

The real breakthrough in the practical use of preconditioning using an incomplete factorization came with two important papers. Firstly, Meijerink and van der Vorst [31] recognised the importance of preconditioning for the conjugate gradient method. This paper also implied an understanding of the crucial role of the separate computation of the incomplete factorization as well as recognizing the possibility of prescribing the sparsity structure of the preconditioner by allowing additional diagonals. Discussing the sparsity structure in the form of diagonals was very natural since simple matrix stencils typically restrict nonzeros to a few diagonals [21, 32]. The other key paper that helped to popularize incomplete factorizations was that of Kershaw [29]. Kershaw introduced the idea of locally replacing pivots by a small positive number to prevent breakdown of the factorization, and this led the way to incomplete factorizations in which dropping is based solely on the size of the computed entries [1, 2] (see also the detailed experimental results in [35]).

The hierarchy of sparsity structures that can be prescribed for incomplete factorizations of general matrices was introduced by Watts in 1981 [43]. Since that time, the notation $IC(\ell)$ for an incomplete Cholesky factorization (or, for general systems, $ILU(\ell)$) based on the concept of levels of fill that we discuss in Sect. 3, has become commonplace. It was soon realised that although $IC(1)$ can be a significant improvement over $IC(0)$ (that is, an appropriate iterative method preconditioned using $IC(1)$

generally requires fewer iterations to achieve the requested accuracy than $IC(0)$), the fill-in resulting from increasing ℓ can be prohibitive in terms of both storage requirements and time to compute and then apply the preconditioner (see, for example, [17]). Moreover, the amount of fill-in is difficult to predict. It is easy to explain this increase in density with ℓ since, while entries of the error matrix $A - LL^T$ (where L is the exact Cholesky factor) are zero inside the prescribed sparsity pattern, outside they can be very large, and the pattern of $IC(\ell)$ (even for large ℓ) may not adequately represent the pattern of L. The error can be particularly large for matrices in which the entries do not decay significantly with distance from the diagonal. D'Azevedo, Forsyth, Tang [15] started to solve the problem by combining the approach by levels with dropping by values. Other early global tools to correct dropping by values were introduced by Munksgaard [33], who tried to get the fill-in curve close to that of the exact decomposition by dynamic changes in the drop tolerance.

Around the same time, an important strategy based on combining dropping entries by value with keeping a prescribed number of the largest entries was proposed [20]. A columnwise algorithm based on a similar concept was presented by Jones and Plassmann [28]. They retain the n_l largest entries in the strictly lower triangular part of the l-th column of L, where n_l is the number of entries in the l-th column of the strictly lower triangular part of A. Another approach that has predictable storage requirements and depends on the matrix entries is the dual threshold $ILUT(p, \tau)$ factorization of Saad [39]. A drop tolerance τ is used to drop all entries in the computed factors that are smaller than τ_l, where τ_l is the product of τ 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. For general unsymmetric matrices, $ILUT(p, \tau)$ has proved very popular but note that it ignores symmetry in A and, if A is symmetric, the sparsity patterns of L and U^T will normally be different.

The algorithm of Lin and Moré [30] for symmetric matrices aims to exploit the best features of the Jones and Plassmann factorization and the $ILUT(p, \tau)$ factorization of Saad. This approach retains the $n_l + p$ largest entries in the lower triangular part of the l-th column of L (p is a chosen memory parameter) and uses only memory as the criterion for dropping entries (thus having the advantage of not requiring a drop tolerance). The reported results of Lin and Moré for large-scale trust region subproblems indicate that allowing additional memory can substantially improve performance on difficult problems.

Recently, a new strategy was developed by Bollhöfer and Saad [7–9]. Here the dropping is relative to the estimated norms of the rows and columns of the factors of the inverse matrix. They have shown both theoretically (by perturbation arguments) and experimentally that preconditioners based on this strategy are very reliable. Extended dropping of this kind that mutually balances direct and inverse factors has been introduced in the last few years by Bru, Marín, Mas, and Tůma [10]; see also their comparison of recent incomplete factorization schemes [11].

These later approaches do not take into account the structure of the levels. One reason for this is that, as already observed, the structure may fill in quickly as ℓ increases and, importantly, until relatively recently it was not clear how this structure could be computed efficiently, especially for larger ℓ. A significant advancement came with the work of Hysom and Pothen [25] (see also [24]). They describe the relationship

between level-based factorizations and lengths of fill paths and propose a fast method of efficiently computing the sparsity pattern of $IC(\ell)$ (and $ILU(\ell)$) factorizations, opening the way to the further development of structure-based preconditioners.

Among recent results, the usefulness of level-based preconditioners in parallel computing environments was emphasised in [24]. Their efficiency in the context of a Newton-Krylov method was shown in [6, 36]. Efficiency of block level-based preconditioners is illustrated in [22].

The main goal of this paper is to show that sparsity structure plays an important role in incomplete factorization preconditioners. While the progress that has been achieved in the field of incomplete factorization preconditioners is substantial, we strongly believe that constructing such preconditioners by considering only the size of the entries, possibly complemented by limits on the overall memory or on the number of additional entries, has important limitations. We are persuaded that to increase robustness we need to use other available tools. In particular, we need to exploit the sparsity structure of the factors. As mentioned above, the work of Hysom and Pothen offers relatively cheap tools for computing level-based factorizations. These are sufficiently general to allow changes to the general strategy of the level-based approach. We propose one possible generalization. The structure of levels that we obtain represents a symbolic incomplete factorization.

Furthermore, we believe that it can be necessary to combine the decomposition by levels with a dropping strategy based on the magnitudes of entries. Our approach starts with the level-based structure obtained by the symbolic incomplete factorization. We then use two additional parameters: a memory multiplier m and a drop tolerance τ. The memory multiplier determines the maximum memory allowed for the preconditioner in terms of the incomplete factor size computed by the symbolic factorization. Any additional memory is predistributed to the individual columns of the final factor. The drop tolerance is then used to decide whether an entry should be dropped or kept in the factor. The implementation keeps track separately of the entries inside the structure returned by the symbolic factorization and those outside it. Entries that are removed either from the symbolic structure or from the additional space available if $m > 1$ provides further space for the incomplete factor. The details are explained in Sect. 4. By using a combination of these approaches, our aim is to obtain an incomplete factorization that retains some of the global characteristics of the full factorization. and provides a good preconditioner.

3 Variable levels of fill in an $IC(\ell)$ preconditioner

In this section, we briefly recall the concept of levels of fill in an incomplete matrix factorization and summarise the approach of Hysom and Pothen [25] for efficiently performing a symbolic $IC(\ell)$ factorization. We then propose a simple generalization that encourages the dropping of small entries from the incomplete factorization by preassigning small entries in A an initial level greater than 0 and we explain how our modification can be incorporated into the symbolic factorization. We use the notation $L = \{l_{ij}\}$ to denote the complete factor of A and $\hat{L} = \{\hat{l}_{ij}\}$ to denote an incomplete factor.

3.1 The incomplete fill path theorem and symbolic $IC(\ell)$ factorization

It is convenient to use some basic concepts and notation from graph theory. The pattern of a sparse symmetric matrix $A = \{a_{ij}\}$ of order n can be represented by an undirected graph $\mathscr{G} = (V, E)$ with vertices $V = \{1, \ldots, n\}$ and edges E. An edge $\{i, j\}$ is present in E if and only if $a_{ij} \neq 0$ and $i \neq j$. Vertices i and j in V are *neighbours* (or are *adjacent* to each other) if edge $\{i, j\} \in E$. The *adjacency set* for i is the set of its neighbours, that is,

$$adj(i) = \{j \mid j \leftrightarrow i, \ i, j \in V\},$$

where we use the notation $i \leftrightarrow j$ to denote that i and j are neighbours. A *path* of length k in $\mathscr{G}$ is an ordered set of distinct vertices $(v_1, v_2, \ldots, v_k, v_{k+1})$, with $v_i \leftrightarrow v_{i+1}$ $(1 \leq i \leq k)$. A path in $\mathscr{G}$ connecting vertices i and j is a *fill path* if the index of each of the intermediate vertices is less than $\min(i, j)$.

An important result that characterizes the fill in the complete factor of A is the *fill path* theorem of Rose, Tarjan and Lueker [37, 38]. This states that l_{ij} is non zero if and only if there is a fill path connecting i and j in $\mathscr{G}$.

Two rules appear in the literature for assigning levels to fill entries, referred to as the *sum* rule [15] and the *max* rule [21]. Following the work of Hysom and Pothen [25], we use the more common sum rule, which states that entries of the factor that correspond to nonzero entries of A are assigned the level 0 while each potential fill entry is assigned a level

$$level(i, j) = \min_{1 \leq l \leq \min\{i, j\}} \{level(i, l) + level(l, j) + 1\}.$$

That is, a level is assigned that is one more than the sum of the levels of the two causative entries. A fill entry is permitted in the incomplete factor provided $level(i, j) \leq \ell$.

The *incomplete fill path* theorem of Hysom and Pothen [25] states that, if the sum rule is used, $level(i, j) = \ell$ if and only if there exists a shortest fill path of length $\ell + 1$ joining i and j in $\mathscr{G}$. Hysom and Pothen use this result to develop the scheme outlined in Algorithm 1 for computing the sparsity pattern of a single column of the incomplete factor $\hat{L}$. The procedure uses a breadth first search that finds a shortest path between vertex k and vertices reachable from k via a traversal of at most $\ell + 1$ edges. A key feature is that the structure of each column of $\hat{L}$ can be computed independently (and hence in parallel). Note that since the number of entries in each column of $\hat{L}$ is not known initially, Algorithm 1 may first be used with line 15 omitted and then repeated after allocating the adjacency set $adj'(k)$ for column k of $\hat{L}$ to have size nz_k.

3.2 Preassigning levels: Strategy I

It is convenient to define $ilev(i, j)$ to be the number of levels of fill to which each nonzero entry a_{ij} of A may contribute. In a standard $IC(\ell)$ algorithm, $ilev(i, j)$ is set to ℓ for each nonzero a_{ij}. To try and ensure that small entries contribute to fewer

---

Algorithm 1 Symbolic $IC(\ell)$ factorization: computes the sparsity pattern of column k of $\hat{L}$. The row indices of the entries in column k are returned in $adj'(k)$ and nz_k is the number of such entries. *length* is an array of size n.

---

1   **Input:** $\mathscr{G}$, ℓ and k.
2   **Initialise:** initialise the queue to hold only k;
3   flag k as visited;
4   set $length(k) = 0$ and $nz_k = 0$.
5   **do**
6   **if** (the queue is empty) **exit**
7   take i from the queue
8   **forall** (unvisited $j \in adj(i)$)
9   flag j as visited
10   **if** ($j < k$ **and** $length(i) < \ell$) **then**
11   add j to the queue
12   set $length(j) = length(i) + 1$
13   **else if** ($j > i$) **then**
14          $nz_k = nz_k + 1$
15   add j to $adj'(k)$
16   **end if**
17   **end forall**
18   **end do**

---

levels of fill in the incomplete factorization than larger entries, the approach we propose preassigns $ilev(i, j)$ for each entry of A individually to have an integer value that depends on $|a_{ij}|$.

We begin by computing the absolute values of the smallest and largest nonzero entries of A, which we denote by *msmall* and *mbig*, respectively. We then take the logarithm of each nonzero $|a_{ij}|$ and distribute these between the $mgrp = [\log(mbig) - \log(msmall)] + 1$ groups that uniformly span the set of logarithm matrix values $\{\log|a_{ij}|\}$. In practice, we have observed that a number of the groups can be empty so that the entries of A are distributed between $ngrp \leq mgrp$ non-empty groups, which we refer to as *slots*. We index the slots as 1 to *ngrp*, with the entries of smallest absolute value in slot 1 and those of largest absolute value in the slot with index *ngrp*. How the initial levels are preassigned then depends on whether $\ell < ngrp$ or $\ell \geq ngrp$.

When $\ell < ngrp$ we uniformly decrease the number of slots to ℓ and set

$$ilev(i, j) = \begin{cases} [k_{ij}/q] & \text{if } \mod(k_{ij}, q) = 0, \\ \min(l, [k_{ij}/q] + 1) & \text{otherwise,} \end{cases} \quad (3.1)$$

where $q = [ngrp/\ell]$ and k_{ij} ($1 \leq k_{ij} \leq ngrp$) is the index of the slot $\log|a_{ij}|$ belongs to. Thus the smallest entries may contribute to a single level of fill and the largest to ℓ levels. For $\ell \geq ngrp$, we set

$$ilev(i, j) = \ell - (ngrp - k_{ij}), \quad (3.2)$$

with k_{ij} is as before. In this case, the largest entries again contribute to ℓ levels of fill while smaller entries contribute to fewer levels.

Since we want to ensure very small entries of A do not contribute to fill entries in the sparsity pattern of $\hat{L}$, for all entries that are smaller in absolute value than the square root of machine precision multiplied by the entry of largest absolute value belonging to the slot with index 1, we set $ilev(i, j) = -(n + 1)$. This has the effect of removing these small entries from A during the symbolic factorization.

We will refer to the strategy we have described for preassigning the levels as **Strategy I**. Having preassigned the levels, we can compute the sparsity pattern of each column of $\hat{L}$ using a simple modification to Algorithm 1. In addition to inputting $ilev(i, j)$ for each nonzero a_{ij} of A, the only modifications we need to make are replace line 8 by the line

8_{new} **forall** (unvisited $j \in adj(i)$ with $ilev(i, j) \neq -(n + 1)$)

and to replace line 10 by the line

10_{new} **if** ($j < k$ **and** $length(i) < ilev(i, j)$) **then**

Line 8_{new} ensures very small entries that have been preassigned a level of $-(n + 1)$ are skipped over while line 10_{new} results in entries with $ilev < \ell$ potentially contributing to fewer levels of fill than they would in the original Hysom and Pothen algorithm. We will refer to this variant of Algorithm 1 using either Strategy I or Strategy II (see below) as the *modified HP algorithm*.

3.3 Strategy II

Numerical results for Strategy I show that preassigning levels so that small entries contribute to fewer than ℓ levels of fill can be advantageous (see Sect. 5). However, the gains are often small. To try and improve the effectiveness of the preconditioner further, we have experimented with allowing the largest entries to contribute to more than ℓ levels of fill. Recall that we distributed the set of logarithm matrix values $\{\log |a_{ij}|\}$ between $mgrp = [\log(mbig) - \log(msmall)] + 1 \geq ngrp$ groups. Let m_{ij} be the group that $\log |a_{ij}|$ belongs to. If $m_{ij} \geq ngrp$, we set

$$ilev(i, j) = \min(m_{ij}, \nu * \ell) \tag{3.3}$$

for some $\nu > 1$. Thus, the largest entries may contribute up to a maximum of $\nu * \ell$ levels of fill and rather than being the maximum number of levels of fill allowed, ℓ becomes the *target* number of levels of fill, with small entries restricted to contributing to fewer than ℓ levels of fill while the largest entries may contribute to more levels. We will refer to this approach as **Strategy II**. Note that m_{ij} plays a similar role to k_{ij} in Strategy I, but the two indices are generally different since they correspond to the distribution of logarithm matrix values into different numbers of groups.

4 The $IC(\ell, \tau, m)$ preconditioner

For general matrices that are not diagonally dominant, the size of an entry of L is not necessarily related to its level of fill. We therefore want a strategy that offers greater flexibility during the numerical factorization. Our basic approach will be to allow

entries outside the pattern predicted by the symbolic factorization to be included provided there is sufficient space available in the preconditioner and, optionally, all entries must be greater in absolute value than a chosen tolerance τ. We will also drop computed entries within the predicted pattern if they are too small.

The (modified) HP algorithm is first used to compute the number nzl of entries in the sparsity pattern of the $IC(\ell)$ incomplete factor $\hat{L}$. Based on the storage available for the preconditioner P, a memory multiplier m is then chosen. If $m > 0$, the number of entries in P will be at most $m * nzl$; choosing $m \leq 0$ indicates there is no restriction on the number of entries in P, which will be controlled only by the drop tolerance τ. In the following subsections, we consider the possible choices for m, with and without a drop tolerance.

4.1 Special case: $m = 1$, $\tau = 0.0$

In the special case in which no entries are dropped because of their size ($\tau = 0$) and the number of entries in P is equal to nzl, the sparsity pattern of P is determined using the (modified) HP algorithm, the entries of the original matrix A are copied into the data structure for P and then a right-looking algorithm is used to compute the entries of P. The resulting preconditioner is a classical $IC(\ell)$ preconditioner if all entries of A are allowed to contribute to ℓ levels of fill.

4.2 $m \geq 1$

Choosing $m > 1$ (or $m = 1$ with $\tau > 0$) allows entries outside the sparsity structure of $\hat{L}$ to be retained. We begin by allocating arrays for the values and row indices of the entries of P to be of size $[m * nzl]$ and define $eroom = [(m - 1) * nzl]$ to be the extra space that is not required by $\hat{L}$. The sparsity pattern of $\hat{L}$ is determined using the (modified) HP algorithm and P is initially given this sparsity pattern. The entries of the original matrix A are copied into the data structure for P, leaving $eroom$ locations free at the start of the arrays. If nz_k is the number of entries in column k of $\hat{L}$, the space provisionally assigned to column k of P is $sp_k = nz_k + [eroom/n]$ (that is, the spare locations are shared equally between the columns).

The incomplete factorization is computed one column at a time using a left-looking algorithm. The entries within each column are always sorted by increasing row index. This enables the strategy proposed in the Yale sparse package [18, 19] to be followed. This keeps track of the columns that are required to update the current column using a simple linked list, which is updated after each major step of the left-looking algorithm. As each column is computed, it is moved forward so that its first entry occupies the first available location in the arrays holding P. Any entries that are smaller in absolute value than τ are dropped as they are computed and not included in P. Additional entries outside the sparsity pattern of $\hat{L}$ that was computed by the symbolic factorization are permitted provided there is sufficient room to accommodate them and they are greater than τ. If there is insufficient space to include all such additional entries, they are sorted and the largest are included in P. Conversely, if the number of accepted entries for column k is less than sp_k, the spare space is added to the space sp_{k+1} available for the next column. Note that if $\tau = 0$, memory is the only criteria for dropping fill entries from P.

4.3 $0 < m < 1$

If $0 < m < 1$, the number of entries in each column k of P must, in general, be less than the corresponding number nz_k in the $IC(\ell)$ incomplete factor $\hat{L}$, and we therefore need to decide how much space to initially assign to each column of P. We perform a complete symbolic Cholesky factorization $A = LL^T$ and compute the number of entries in each column of L. We then share out the $[m * nzl]$ entries allowed for P so that the distribution for the individual columns is approximately proportional to the column counts for L. We denote by nzp_k the number of entries provisionally assigned to column k.

The incomplete factorization again proceeds column by column, using a left-looking algorithm. The computation of column k starts by computing the sparsity pattern of column k of $\hat{L}$ using the (modified) HP algorithm. A temporary array of size nz_k is allocated, initialised to zero and the entries of column k of A then copied into it. If nz_k is greater than the space sp_k available for column k, the entries in the temporary array are sorted and only the sp_k entries of largest absolute value are kept.

Candidate entries with absolute value less than the drop tolerance τ are not included in P. If $\tau > 0$, this may mean that, when column k is processed, the final number of entries that are retained is less than the space available for that column. In this case, the spare space s_k is passed to the next column so that the space for column $k + 1$ becomes $sp_{k+1} = nzp_{k+1} + s_k$.

4.4 $m < 0$, $\tau \neq 0$

We use $m < 0$ to indicate that there are no memory restrictions on the size of P and entries are only dropped because of their size relative to τ. In this case, we perform an incomplete factorization without distinguishing between entries inside the pattern predicted by the symbolic $IC(\ell)$ factorization and those outside it. The storage requirements are not predictable. We initially allocate arrays for the values and row indices of the entries of P to be of size $\max(2, |m|) * nzl$. If these arrays are subsequently found to be too small, we reallocate them with larger size (saving the already computed columns using temporary arrays) and then continue the incomplete factorization. Reallocation can be needed more than once and failure only occurs if we do not have sufficient memory available to successfully allocate larger arrays. The final incomplete factorization depends only on τ (and not on ℓ or m); we denote this by $IC(\tau)$.

4.5 Dropping strategies

The dropping strategy we use is absolute dropping so that a potential entry of P is dropped if its absolute value is less than the chosen tolerance τ. An alternative approach is relative dropping (see, for example, [40]). In this case, an entry is dropped whenever its absolute value is less than τ multiplied by some quantity that expresses the average size of the computed entries. An appropriate choice for this might be a norm of the computed column. Our preference is to use absolute dropping in incomplete factorizations and this is used in the numerical experiments reported on in

Sect. 5. Both absolute and relative dropping have potential advantages and disadvantages. A drawback of relative dropping is that it can hide significant growth in entries of the incomplete factor. This growth, which may result in a very unstable preconditioner, can then be detected only numerically. However, for absolute dropping the growth can be detected by monitoring the size of fill-in. We believe that this may be more useful for future adaptive strategies. Another reason for offering absolute dropping is that some problems can involve large and small entries that are coupled by subtle properties of the physical model. This may happen, for example, when solving shell problems from structural engineering (see, for example, [5]).

5 Numerical experiments

The numerical results reported in this section were performed on a single processor of a 2-way quadcore Harpertown machine. All the software was written in Fortran; the g95 compiler with option -O was used. The implementation of the conjugate gradient algorithm offered by the HSL [23] routine `MI22` was employed, with starting vector $x_0 = 0$, the right-hand side vector b computed so that the exact solution was $x = 1$, and stopping criteria

$$\|A\hat{x} - b\|_2 \leq 10^{-6}\|b\|_2$$

where $\hat{x}$ is the computed solution. In addition, for each test we imposed a limit of 800 iterations.

We define the *iteration count* for preconditioner P for a given problem to be the number of iterations required by the iterative method using the preconditioner P to achieve the requested accuracy and we define the *preconditioner size* to be the number of entries $nz(P)$ in the lower triangular part of P.

While we are well aware that the number of entries in the preconditioner may increase but its effectiveness decrease, in many practical situations, the mutual relation between the iteration count and preconditioner size provides an important insight into the usefulness of an incomplete factorization preconditioner if we assume that the following two important conditions are fulfilled:

1. the preconditioner is sufficiently *robust* with respect to changes to the parameters of the decomposition, such as with respect to the drop tolerance or number of levels,
2. the time required to compute the preconditioner grows *slowly* with the problem dimension n.

We define the *efficiency* of P to be

$$iter \times nz(P),$$

where *iter* is the iteration count for P. Assuming the preconditioners P_k ($q = 1, \ldots, r$) each satisfy the above conditions, we say that, for solving a given problem, P_i is the most *efficient* of the r preconditioners if

$$iter_i \times nz(P_i) \leq \min_{q \neq i}(iter_q \times nz(P_q)).$$

We use this measure of efficiency in our numerical experiments.

Unless stated otherwise, all our test problems are real positive-definite matrices of order at least 1000 taken from the University of Florida Sparse Matrix Collection [14]. 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 147 problems of order up to 1.5 million. In the tables of results, n denotes the order of A; $nz(\hat{L})$ is the number of entries in the lower triangular part of $\hat{L}$ (measured in thousands); *iter* and *effic* are the iteration count and efficiency, respectively.

5.1 The effects of preassigning levels

In our first experiment, we look at the effects of preassigning levels of fill. Since we want to isolate these effects from those caused by allowing additional memory and/or using a drop tolerance during the numerical factorization, we restrict our attention to the case $m = 1$, $\tau = 0.0$ (see Sect. 4.1). To illustrate the potential benefits of preassigning levels, we start by presenting results for problem `carsten3`, which arises from a finite-difference discretization of a Kohn–Sham equation of physical chemistry in two dimensions (see [4]). The matrix dimension is 250500 and it has 750998 nonzeros. In Fig. 1, the number of iterations needed for CG to achieve the requested accuracy as a function of the number of entries in the incomplete factor $IC(\ell, 0, 1)$ is presented for $\ell = 1, \ldots, 15$, both with preassigning levels (using Strategy II with $\nu = 2$) and without preassigning levels (that is, standard constant levels). As ℓ increases, the number of nonzeros increases and the number of iterations decreases. We see that, for this example, the efficiency is consistently improved by preassigning the levels.

To assess the effect of preassigning the levels on a large set of problems, it is convenient to use performance profiles [16]. A performance profile measures performance of two or more preconditioners on a set $\mathscr{T}$ of problems. Let $e_{k,P}$ be the efficiency of using preconditioner P to solve problem k and define the efficiency performance ratio to be $ratio_{k,P} = e_{k,P} / \min\{e_{k,P_i} : \text{for all } P_i\}$. If the number of problems in $\mathscr{T}$ is N, the efficiency performance profile

$$\rho_P(\alpha) = (1/N)|\{k \in \mathscr{T} : ratio_{k,P} \leq \alpha\}|$$

is the probability that an efficiency performance ratio $ratio_{k,P}$ is within a factor α of the best possible ratio. For instance, $\rho_P(1)$ gives the fraction of the test problems for which P is the most efficient preconditioner and $\rho_P(2)$ gives how often P can get results with an efficiency that is within twice that of the best preconditioner. The closer ρ_P is to 1, the greater the probability that preconditioner P can solve all problems from $\mathscr{T}$.

For a fixed value of ℓ, for each problem we computed the $IC(\ell, 0, 1)$ preconditioner with and without preassigning levels. Any problem for which the resulting preconditioned CG method failed to converge with Strategy I and with Strategy II and without preassigning levels was removed from the test set $\mathscr{T}(\ell)$. Since the costs associated with computing and applying as well as storing an $IC(\ell, 0, 1)$ preconditioner increase with ℓ, we are normally interested in small values of ℓ. Here we consider $\ell = 3$ and use $\nu = 2$ for Strategy II. The set $\mathscr{T}(3)$ comprises 120 problems.
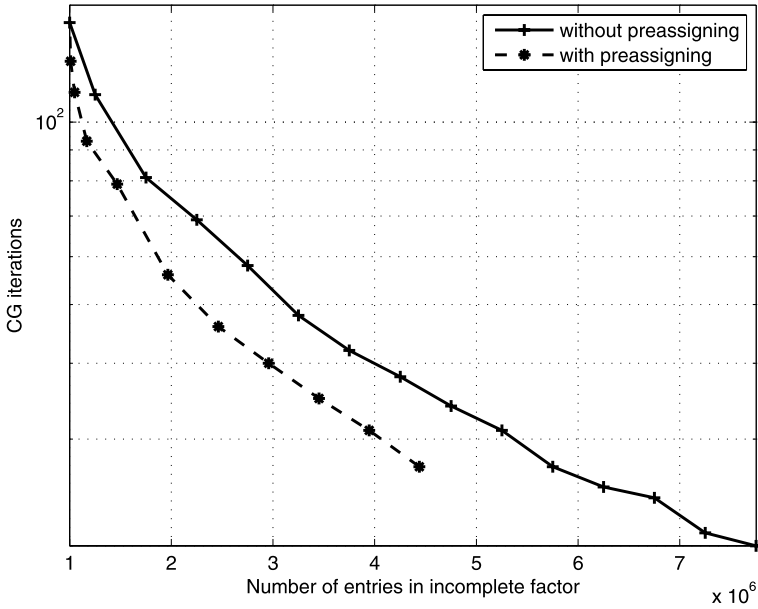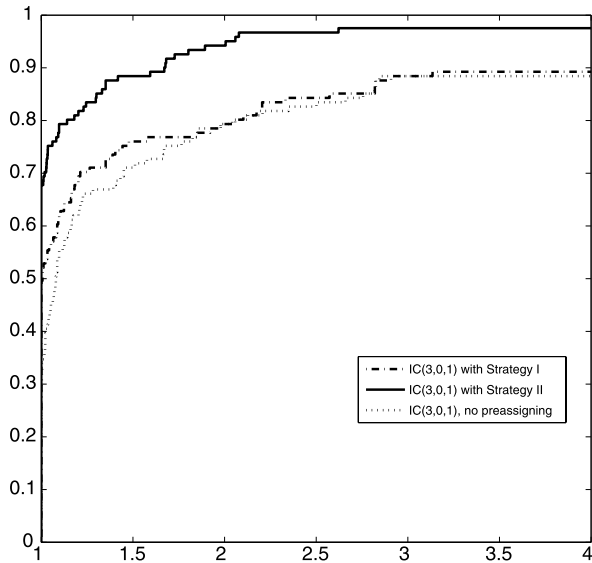
Fig. 1 The effect of allowing variable levels for problem `carsten` comparing preassigning (using Strategy II) with standard level-based preconditioning (no preassigning). Results are given for $\ell = 1, \ldots, 15$

Fig. 2 Efficiency performance profile for $IC(3, 0, 1)$

The efficiency performance profile for $IC(3, 0, 1)$ is given in Fig. 2. It is clear that overall there is an advantage in preassigning levels. The improvement is often modest, particularly for Strategy I and, in some instances, it is better not to preassign levels. Unfortunately, we are currently unable to predict when this is the case.

Fig. 3 Efficiency performance profile comparing $IC(3, 0, 1)$ with Strategy II with $IC(6, 0, 1)$ without preassigning levels
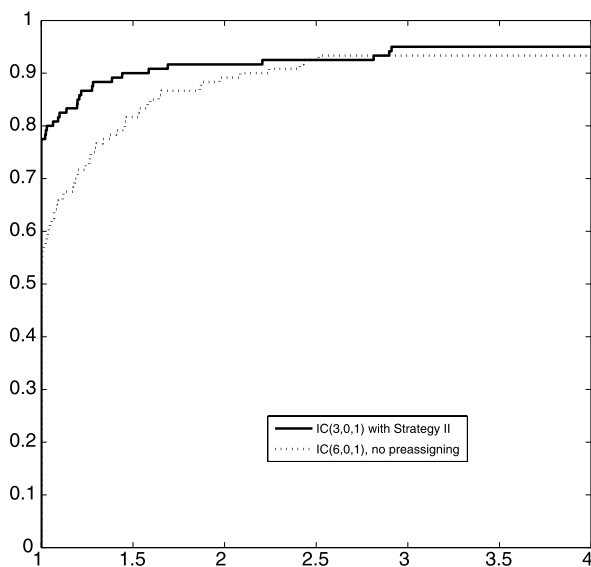

Table 1 Results for $IC(3, 0, 1)$ for a subset of our test set. n denotes the order of A and $nz(A)$ is the number of entries in the lower triangular part of A. $nz(A)$ and $nz(\hat{L})$ are in thousands

Problem	n	$nz(A)$	No preassigning		Strategy I		Strategy II	
			$nz(\hat{L})$	iter	$nz(\hat{L})$	iter	$nz(\hat{L})$	iter
Boeing/msc01440	1440	23	76	12	76	12	86	5
Nasa/nasa2910	2910	88	236	18	235	17	352	5
Boeing/ct20stif	52329	1326	6704	73	6608	73	16591	23
Wissgott/parabolic_fem	525825	2099	5926	299	5175	289	7968	244
DNVS/ship_003	121728	1949	17146	113	15625	110	37529	57

Looking in more detail at the results for Strategy I, we find that in many examples the number of iterations is the same as for not preassigning levels: the improvement in efficiency comes from having fewer entries in $\hat{L}$. Thus, as was our intention, Strategy I improves the sparsity of $\hat{L}$ without reducing its quality as a preconditioner. The achieved reduction in $\hat{L}$ will be particularly beneficial if the preconditioner is used to solve more than one system. On the other hand, Strategy II can produce denser preconditioners that require fewer iterations. Some examples that illustrate this are given in Table 1. With $\ell = 3$ and $\nu = 2$, Strategy II allows the largest entries to contribute up to 6 levels of fill. In Fig. 3, we compare this with $IC(6, 0, 1)$ without preassigning levels. We again see that preassigning levels is advantageous. $IC(6, 0, 1)$ produces preconditioners with more entries than Strategy II with $\ell = 3$ and, in some cases, $\hat{L}$ can be significantly denser (and more expensive to compute and to apply). Based on our findings, in the rest of the paper, we preassign levels.
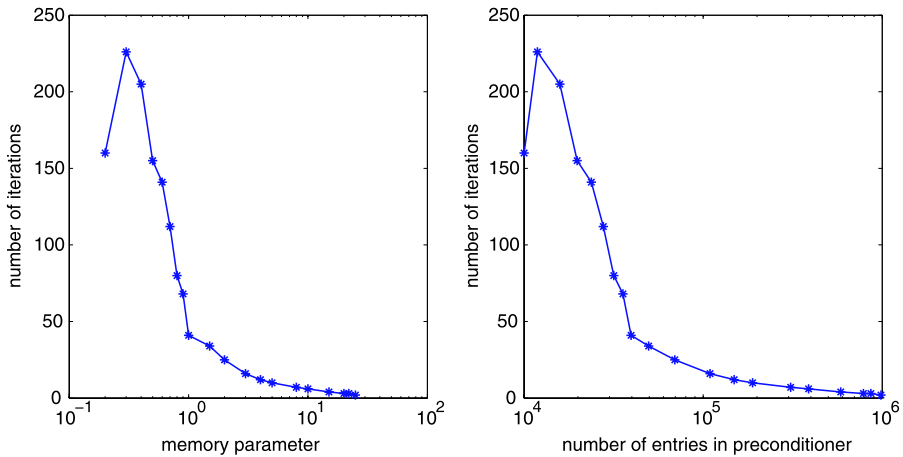
Fig. 4 The iteration count for $IC(1, 0, m)$ for the simple Laplace equation as m increases (*left-hand plot*) or $nz(\hat{L})$ increases (*right-hand plot*)

5.2 Memory control

In this section, we illustrate the importance and usefulness of the memory control parameter m. We first consider a simple five-point discretization of the 2D Laplace equation on a unit square with homogeneous Dirichlet boundary conditions using a 100×100 grid. For m ranging from 0.2 to 25, Fig. 4 shows the dependence of the number of iterations required for the convergence of $IC(1, 0, m)$ on m and on $nz(\hat{L})$. We see that, as m increases, so too does $nz(\hat{L})$ while the number of iterations steadily decreases. Note in particular that our strategy for $m < 1$ yields IC-like preconditioners that have fewer entries than the initial level-based structure but that nevertheless yield convergence. However, in practice, extreme values of m (either very small or very large values) are unlikely be useful. Small m may require prohibitively many iterations while large m may be infeasible from the memory point of view.

In Table 2, we present results for $IC(\ell, 0, m)$ for a tougher problem HB/bcsstk17 from our test set. With $\ell = 1$ we found it was necessary to set m to be greater than 2.7 to achieve convergence while $m = 5$ gave a complete factorization. Thus results are given for m in the range 2.7 to 5 and, for comparison, we ran $\ell = 0, 1, 2$. We see that, for fixed ℓ, as m increases so too does $nz(\hat{L})$ and, in general, the number of iterations decreases. Note that if ℓ is increased to 3, for all the values of m in the given range $nz(\hat{L}) = 1596 * 10^3$ and a single iteration is required.

5.3 A comparison with $IC(\tau)$ and symmetric $ILUT(p, \tau)$

Tables 3 and 4 show results of experiments for problem TKK/tube1, a symmetric positive-definite matrix that comes from solving thin shell problems in three-dimensional structural analysis. The matrix has dimension 21498 and 459277 nonzeros. The first of these tables presents results for $IC(\ell, \tau, 1)$ with ℓ ranging from 4 to 15 and different drop tolerances τ (smaller values of ℓ did not give convergence). We

Table 2 Results for $IC(\ell, 0, m)$ for $\ell = 0, 1, 2$ and a range of values of m applied to problem HB/bcsstk17. $nz(\hat{L})$ is in thousands. † indicates convergence not achieved

m	$\ell = 0$		$\ell = 1$		$\ell = 2$	
	$nz(\hat{L})$	iter	$nz(\hat{L})$	iter	$nz(\hat{L})$	iter
2.7	592	†	945	†	1271	11
2.75	603	†	966	23	1292	10
2.78	603	†	977	18	1303	10
2.8	614	†	988	33	1314	10
2.85	625	†	999	†	1336	10
2.9	636	†	1021	17	1358	9
3	657	†	1054	17	1412	8
3.1	679	†	1086	17	1455	8
3.15	691	†	1108	16	1477	7
3.2	701	†	1119	70	1499	7
3.25	712	†	1141	14	1520	6
3.3	723	†	1163	15	1553	4
3.5	767	†	1228	12	1596	1
4	876	†	1402	9	1596	1
4.3	942	71	1510	6	1596	1
4.5	985	26	1576	3	1596	1
5	1094	16	1597	1	1596	1

Table 3 Results for $IC(\ell, \tau, 1)$ for a range of values of ℓ and small drop tolerances τ applied to problem TKK/tube1. $nz(\hat{L})$ is in thousands. † indicates convergence not achieved

ℓ	$\tau = 0.0$		$\tau = 10^{-10}$		$\tau = 10^{-8}$		$\tau = 10^{-7}$	
	$nz(\hat{L})$	iter	$nz(\hat{L})$	iter	$nz(\hat{L})$	iter	$nz(\hat{L})$	iter
4	1654	520	1653	501	1639	506	1608	499
5	2188	283	2186	278	2158	313	2105	287
6	2863	223	2857	224	2800	197	2711	197
7	3705	159	3691	156	3584	158	3431	159
8	4662	†	4630	†	4458	744	4222	†
9	5628	†	5574	†	5322	†	4999	†
10	7383	230	7271	231	6835	204	6346	239
11	7624	325	7480	252	7030	261	6519	236
12	10532	158	10221	154	9344	123	8527	159
13	10588	135	10270	155	9386	139	8563	123
14	10612	135	10293	151	9405	139	8580	165
15	13667	83	13018	80	11619	61	10404	59

see that for fixed ℓ, using a small non-zero drop tolerance can reduce the number of entries in $\hat{L}$ without adversely effecting its performance. The second table presents results for $IC(\tau)$ (see Sect. 4.4). For this problem, to get convergence we found that the drop tolerance needed to be approximately $5 * 10^{-5}$ or less; by only varying τ,

Table 4 Results for $IC(\tau)$ for a range of values of the drop tolerance τ applied to problem TKK/tube1. $nz(\hat{L})$ is in thousands. † indicates convergence not achieved

τ	$nz(\hat{L})$	iter	τ	$nz(\hat{L})$	iter
100	88	†	1e-2	14262	†
60	168	†	1e-3	16140	†
55	281	†	1e-4	9001	†
50	1458	†	5e-5	9649	471
45	2077	†	2e-5	9611	87
40	2253	†	1e-5	10050	18
10	4624	†	5e-6	10741	6
1	7151	†	1e-6	12451	2
1e-1	11565	†	0	21803	1

it was not possible to achieve convergence with $nz(\hat{L})$ less than $9.6 * 10^6$. This contrasts with the level-based preconditioner results in Table 3, where convergence was achieved with significantly sparser $\hat{L}$. A partial explanation is based on the fact that the finite-element discretization of thin shell problems couples unknown displacements and bending moments that strongly differ in magnitudes and this fact makes the uni-parametric preconditioner $IC(\tau)$ difficult to tune. Note that for $IC(\tau)$ we tried values of the drop tolerance τ greater than one. Such values are meaningful if there is growth in the entries during the computation. In fact, checking sensitivity of the drop tolerance against the number of dropped entries which reflects the growth factor can be a tool for adaptive dropping schemes.

Comparing the results for $IC(\ell, \tau, 1)$ with those for $IC(\tau)$ demonstrates that incorporating a level-based strategy can lead to very different results. Furthermore, finding a suitable τ is both highly problem dependent and strategy dependent ($IC(\ell, \tau, 1)$ and $IC(\tau)$ used different τ), which means that it is hard to perform a unified comparison on the kind of large test set that was reported on using performance profiles in Sect. 5.1. To illustrate further the difficulties of choosing appropriate drop tolerances, we compare our level-based strategy $IC(\ell, \tau, 1)$ with the symmetric dual parameter $ILUT(p, \tau)$ preconditioner [39] for two of our test problems. Figure 5 presents results for the thin shell problem Cylshell/s1rmt3m1 ($n = 5489$, $nz = 112505$). The figure shows the dependence of the number of iterations required for the convergence of $IC(\ell, 0.5, 1)$ with $\ell = 1, 2, \ldots, 15$ and for the symmetric $ILUT(p, \tau)$ preconditioner with parameters varying from $(78, 9)$ to $(120, 0)$. Note that it was very difficult to find parameters for which $ILUT$ converged and these were found by repeated experimentation. Similarly, Fig. 6 shows results for the structural mechanics problem Nasa/nasasrb problem ($n = 54870$, $nz = 1366097$). The behaviour of $IC(\ell, 100, 1)$ is shown for ℓ in the range 5 and 15 (for smaller ℓ convergence was not achieved) and $ILUT(p, \tau)$ with p, τ varying from $(258, 119)$ to $(1000, 0)$. Again, finding parameters that gave convergence of $ILUT(p, \tau)$ was difficult. The superior performance of the level-based approach is clear in both examples. This confirms our belief that starting the construction of an efficient preconditioner by improving a relatively stable level-based strategy may be a reasonable strategy.

Fig. 5 Comparison of
$ILUT(p, \tau)$ preconditioning
with variable parameters and
$IC(\ell, 0.5, 1)$ for problem
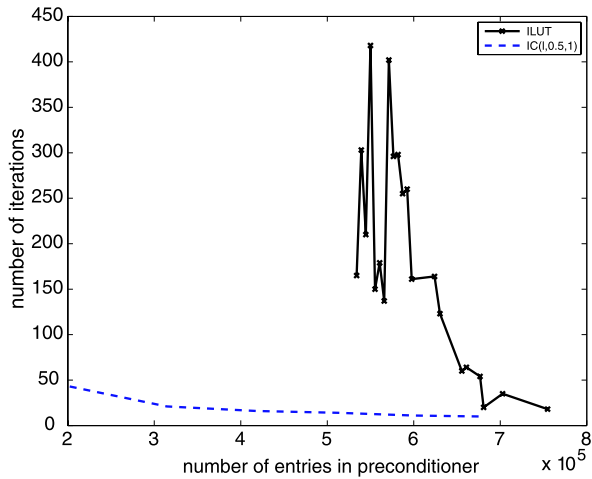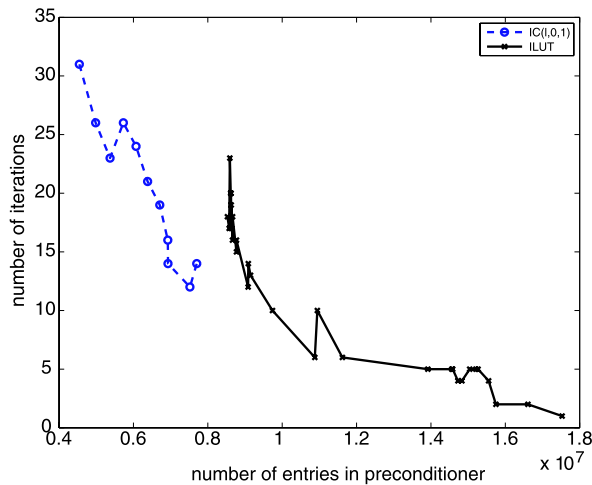`Cylshell/s1rmt3m1`

Fig. 6 Comparison of
$ILUT(p, \tau)$ preconditioning
with variable parameters and
$IC(\ell, 0, 1)$ for problem
`Nasa/nasasrb`

6 Concluding remarks

In this paper, we have presented a new strategy for computing an incomplete
Cholesky factorization preconditioner that is derived from the level-based approach.
In particular, we have proposed new strategies for setting the levels and then ex-
ploited the sparsity structure computed during the symbolic factorization throughout
the numerical factorization. The numerical experiments confirmed that the proposed
approach is viable and can be regarded as one step in improving basic incomplete fac-
torization preconditioning strategies. It is generally assumed that a universal incom-
plete factorization preconditioning strategy for all types of problems is not possible,
but there still seems to be scope for improving the computational paradigms that we
have. Note that the scheme is easily extended to the nonsymmetric case, that is, for
ILU preconditioners. Symbolic procedures by Hysom and Pothen were, in fact, pro-

posed for the general nonsymmetric case. Nevertheless, it is well known that the resulting preconditioned iterative methods can behave much more erratically. Because of this, our next target will be first to embed our ideas into a more comprehensive scheme that will exploit blocks, pivoting, efficient nonsymmetric reorderings, possibly also a multilevel framework. We believe that these enhancements are necessary for getting more consistent improvements and comparisons also in the nonsymmetric case.

References

1. Ajiz, M.A., Jennings, A.: A robust incomplete Choleski-conjugate gradient algorithm. Int. J. Numer. Methods Engrg. **20**(5), 949–966 (1984)
2. Axelsson, O., Munksgaard, N.: Analysis of incomplete factorizations with fixed storage allocation. In: Preconditioning Methods: Analysis and Applications. Topics in Comput. Math., vol. 1, pp. 219–241. Gordon & Breach, New York (1983)
3. Baker, G. Jr., Oliphant, T.: An implicit, numerical method for solving the two-dimensional heat equation. Quart. Appl. Math. **17**, 361–373 (1960)
4. Benzi, M., Haws, J.C., Tůma, M.: Preconditioning highly indefinite and nonsymmetric matrices. SIAM J. Sci. Comput. **22**(4), 1333–1353 (2000)
5. Benzi, M., Kouhia, R., Tůma, M.: Stabilized and block approximate inverse preconditioners for problems in solid and structural mechanics. Comput. Methods Appl. Mech. Engrg. **190**(49–50), 6533–6554 (2001)
6. Blanco, M., Zingg, D.: A Newton-Krylov algorithm with a loosely coupled turbulence model for aerodynamic flows. AIAA J. **45**, 980–987 (2007)
7. Bollhöfer, M.: A robust ILU with pivoting based on monitoring the growth of the inverse factors. Linear Algebra Appl. **338**, 201–218 (2001)
8. Bollhöfer, M.: A robust and efficient *ILU* that incorporates the growth of the inverse triangular factors. SIAM J. Sci. Comput. **25**(1), 86–103 (2003)
9. Bollhöfer, M., Saad, Y.: On the relations between ILUs and factored approximate inverses. SIAM J. Matrix Anal. Appl. **24**(1), 219–237 (2002)
10. Bru, R., Marín, J., Mas, J., Tůma, M.: Balanced incomplete factorization. SIAM J. Sci. Comput. **30**(5), 2302–2318 (2008)
11. Bru, R., Marín, J., Mas, J., Tůma, M.: Improved balanced incomplete factorization. SIAM J. Matrix Anal. Appl. **31**(5), 2431–2452 (2010)
12. Buleev, N.I.: A numerical method for solving two-dimensional diffusion equations. Atomnaja Energija **6**, 338–340 (1959)
13. Buleev, N.I.: A numerical method for solving two-dimensional and three-dimensional diffusion equations. Mat. Sb. **51**, 227–238 (1960)
14. Davis, T.A.: The university of Florida sparse matrix collection. Technical Report, University of Florida (2007). http://www.cise.ufl.edu/~davis/techreports/matrices.pdf
15. D'Azevedo, E., Forsyth, P., Tang, W.: Drop tolerance preconditioning for incompressible viscous flow. Int. J. Comput. Math. **44**, 301–312 (1992)
16. Dolan, E., Moré, J.: Benchmarking optimization software with performance profiles. Math. Program. **91**(2), 201–213 (2002)
17. Duff, I.S., Meurant, G.A.: The effect of ordering on preconditioned conjugate gradients. BIT **29**, 635–657 (1989)
18. Eisenstat, S.C., Gursky, M.C., Schultz, M.H., Sherman, A.H.: The Yale sparse matrix package (YSMP)—II: The non-symmetric codes. Tech. Rep. No. 114, Department of Computer Science, Yale University (1977)
19. Eisenstat, S.C., Gursky, M.C., Schultz, M.H., Sherman, A.H.: Yale Sparse Matrix Package (YSMP)—I: The symmetric codes. Int. J. Numer. Methods Engrg. **18**, 1145–1151 (1982)
20. Freund, R.W., Nachtigal, N.M.: 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)
21. Gustafsson, I.: A class of first order factorization methods. BIT **18**(2), 142–156 (1978)

22. Hénon, P., Ramet, P., Roman, J.: On finding approximate supernodes for an efficient block-ILU(k) factorization. Parallel Comput. **34**(6–8), 345–362 (2008)
23. HSL: A collection of Fortran codes for large-scale scientific computation (2007). See http://www.cse.scitech.ac.uk/nag/hsl/
24. Hysom, D., Pothen, A.: A scalable parallel algorithm for incomplete factor preconditioning. SIAM J. Sci. Comput. **22**, 2194–2215 (2001)
25. Hysom, D., Pothen, A.: Level-based incomplete LU factorization: Graph model and algorithms. Technical Report UCRL-JC-150789, Lawrence Livermore National Labs (November 2002)
26. Il'in, Y.M.: Difference Methods for Solving Elliptic Equations (in Russian). Novosibirskij Gosudarstvennyj Universitet, Novosibirsk (1970)
27. Il'in, Y.M.: Iterative Incomplete Factorization Methods. World Scientific, Singapore (1992)
28. Jones, M.T., Plassmann, P.E.: An improved incomplete Cholesky factorization. ACM Trans. Math. Softw. **21**(1), 5–17 (1995)
29. Kershaw, D.S.: The incomplete Cholesky-conjugate gradient method for the iterative solution of systems of linear equations. J. Comput. Phys. **26**, 43–65 (1978)
30. Lin, C.J., Moré, J.J.: Incomplete Cholesky factorizations with limited memory. SIAM J. Sci. Comput. **21**(1), 24–45 (1999)
31. Meijerink, J.A., van der Vorst, H.A.: An iterative solution method for linear systems of which the coefficient matrix is a symmetric M-matrix. Math. Comp. **31**, 148–162 (1977)
32. Meijerink, J.A., van der Vorst, H.A.: Guidelines for the usage of incomplete decompositions in solving sets of linear equations as they occur in practical problems. J. Comput. Phys. **44**(1), 134–155 (1981)
33. Munksgaard, N.: Solving sparse symmetric sets of linear equations by preconditioned conjugate gradients. ACM Trans. Math. Softw. **6**(2), 206–219 (1980)
34. Oliphant, T.A.: An extrapolation process for solving linear systems. Quart. Appl. Math. **20**, 257–265 (1962)
35. Østerby, O., Zlatev, Z.: Direct Methods for Sparse Matrices. Lecture Notes in Computer Science, vol. 157. Springer, Berlin (1983)
36. Pueyo, A., Zingg, D.: Efficient Newton-Krylov solver for aerodynamic computations. AIAA J. **36**, 1991–1997 (1998)
37. Rose, D.J., Tarjan, R.E.: Algorithm aspects of vertex elimination on directed graphs. SIAM J. Appl. Math. **34**(1), 176–197 (1978)
38. Rose, D.J., Tarjan, R.E., Lueker, G.S.: Algorithm aspects of vertex elimination on graphs. SIAM J. Comput. **5**, 266–283 (1976)
39. Saad, Y.: ILUT: a dual threshold incomplete *LU* factorization. Numer. Linear Algebra Appl. **1**(4), 387–402 (1994)
40. Saad, Y.: Iterative Methods for Sparse Linear Systems. PWS Publishing, Boston (1996)
41. Sabinin, V.I.: An algorithm of the incomplete factorization method. Chisl. Metody Mekh. Sploshn. Sredy **16**(2), 103–117 (1985)
42. Varga, R.S.: Factorizations and normalized iterative methods. In: Boundary Problems in Differential Equations, pp. 121–142. University of Wisconsin Press, Madison (1960)
43. Watts, J.W. III: A conjugate gradient truncated direct method for the iterative solution of the reservoir simulation pressure equation. Soc. Pet. Eng. J. **21**, 345–353 (1981)