ITERATIVE ILU PRECONDITIONERS FOR LINEAR SYSTEMS AND EIGENPROBLEMS

DANIELE BOFFI *, ZHONGJIE LU*, AND LUCA F. PAVARINO *

Abstract. Iterative ILU factorizations are constructed, analyzed and applied as preconditioners to solve both linear systems and eigenproblems. The computational kernels of these novel Iterative ILU factorizations are sparse matrix-matrix multiplications, which are easy and efficient to implement on both serial and parallel computer architectures and can take full advantage of existing matrix-matrix multiplication codes. We also introduce high-level and threshold algorithms in order to enhance the accuracy of the proposed Iterative ILU factorizations. The results of several numerical experiments illustrate the efficiency of the proposed preconditioners to solve both linear systems and eigenvalue problems.

Key words: Iterative ILU factorizations, matrix-matrix multiplication, fill-in, eigenvalue problems, parallel preconditioners

1. Introduction. The LU factorization is an efficient direct method to solve linear systems. For sparse problems, the computed L and U factors might suffer from fill-in and lose sparsity compared with the original matrix. For large-scale problems, this fact generates factors that are both time consuming to compute and memory consuming to store. In this case, iterative methods become attractive alternatives, see e.g. [5, 18, 31], since they usually require less memory and less float computations. However, iterative methods might suffer from lack of robustness and slow convergence, and preconditioning techniques become necessary in order to make iterative methods faster and more reliable. While many preconditioners are designed for special problems, incomplete LU (ILU) preconditioners are relatively general, since they are based on the LU factorization.

There are a large number of references for the analysis and applications of ILU preconditioners in the literature of mathematics, physics and computer science, see e.g. [6, 23, 34], and ILU preconditioners are implemented in many numerical libraries [4, 25]. The convergence speed of most iterative methods depends to the condition number of the matrices involved. In [7, 29, 34], it has been proven that incomplete factorization preconditioner can reduce the condition number of finite difference discretizations from $O(n^2)$ to $O(n)$ for two-dimensional elliptic problems. There are many modifications of standard ILU based on various techniques. For example, high-level ILU(p) [31] enhances the accuracy of the factorization by allowing more fill-ins, modified ILU (MILU) [19, 26]. compensates diagonal entries to make the row or column sums of the multiplication LU equal to the ones of the original matrix. Threshold ILU (ILUT(p, τ)) [30] introduces a tolerance and a number of fill-ins into the dropping rules, which provide more options to balance the accuracy and storage of the factors. There are also many references about the stability and implementation of ILUs, see e.g. [3, 15, 32].

Gaussian elimination, the core technique of ILU factorization, is a highly sequential procedure, which is an obstacle to ILUs' parallelization. Consequently, many other techniques have been introduced in order to improve the parallelism of LU factorizations. Graph coloring techniques and domain decomposition techniques have

*Dipartimento di Matematica, Università di Pavia, via Ferrata 5, 27100 Pavia, Italy. Email: daniele.boffi@unipv.it, zhongjie.lu@unipv.it, luca.pavarino@unipv.it. This work was partially supported by the Italian Ministry of Education, University and Research (MIUR) through the "Dipartimenti di Eccellenza Program 2018-22 - Dept. of Mathematics, University of Pavia", and by the Istituto Nazionale di Alta Matematica (INdAM - GNCS), Italy.

been used to partition a matrix into several submatrices, where factorizations can then be executed on each submatrix, see e.g. [20, 21, 28]. In [12], a factorization is designed based on residual correction, which can be implemented in parallel. Recently, a type of fine-grained parallel incomplete factorization based on fixed-point iterations, has been proposed in [2, 13] and it has been successfully implemented in share-memory environments.

In this paper, we construct and analyze an iterative ILU factorization of a given matrix. The main procedure of new algorithms is based on sparse matrix-matrix multiplications. This matrix operation is nowaday contained in many computing libraries and implemented in efficient codes, which can be much faster than computing the entries one by one according to the definition. If we take full advantage of these techniques and libraries, the new algorithms are easy to implement on both share-memory and distributed-memory systems without the need to enter into the complex details of optimized serial and parallel implementations, in particular interprocessor communications. We also present two modifications of the new Iterative ILU algorithms based on high-level and threshold variants aimed at improving the efficiency.

Throughout this paper, we will not study the parallelism of the new algorithms explicitly, since the the proposed preconditioners and iterative solvers require only matrix-matrix and matrix-vector multiplications, which have intrinsically fine-grained parallelism and have been implemented efficiently on different computing architectures.

The paper is organized as follows. In Section 2, we derive the conventional LU factorization in a matrix form and recall its fill-in property. We then propose several new Iterative ILU preconditioners and prove some convergence results in Section 3. In Section 4, we discuss some issues involved in the new algorithms, such as matrix-matrix multiplication and solving triangular systems. In Section 5 and 6, we apply the new preconditioner to solving linear systems and eigenvalue problems, respectively, showing the results of several numerical experiments. Some conclusions and limitations are reported in Section 7.

2. Iterative LU factorization. Let A be a square matrix that can be factorized as a multiplication of a lower and a upper triangular matrices, i.e.

$$A = LU, \tag{2.1}$$

where all the diagonal entries of L are 1. We write the triangular matrices L and U in the following forms:

$$L = L_0 + I \qquad \text{and} \qquad U = U_0 + D,$$

where I is the identity matrix and D is the diagonal matrix of U. We rewrite (2.1) as

$$A = (L_0 + I)(D + U_0) = L_0 D + D + U_0 + L_0 U_0$$

or

$$L_0 D + D + U_0 = A - L_0 U_0. \tag{2.2}$$

Based on the equation (2.2), we obtain an iterative form for LU factorization:

$$L_0^k D^k + D^k + U_0^k = A - L_0^{k-1} U_0^{k-1}, \tag{2.3}$$

where $L_0^k D^k$, D^k and U_0^k are strictly lower triangular, diagonal and strictly upper triangular matrix of the matrix $A - L_0^{k-1} U_0^{k-1}$, respectively, and L_0^k is easy to obtain from $L_0^k D^k$. In this paper, we assume that all the diagonal entries of D^k $(k \geq 1)$ are not zero.

For a square matrix B, we use Matlab-like notations $diag(diag(B))$, $tril(B, -1)$ and $triu(B, 1)$ to denote the diagonal, strictly lower triangular and strictly upper triangular part of B, respectively. This iterative form (2.3) can be implemented as Algorithm 1.

Algorithm 1 Iterative LU factorization

1: Set the initial data $L_0 = D = U_0 = 0$.
2: **for** $k = 1, 2, \cdots, p$ **do**
3: $\quad B = A - L_0 U_0$
4: $\quad D = diag(diag(B))$
5: $\quad U_0 = triu(B, 1)$
6: $\quad L_0 = tril(B, -1) D^{-1}$
7: **end for**
8: $L = L_0 + I$ and $U = U_0 + D$.

The convergence properties of this algorithm are analyzed in the following theorem. Since it is a special case of Theorem 3.3, we will discuss it after proving Theorem 3.3 in Section 3.

THEOREM 2.1. *The factors L, U in Algorithm 1 converge to the exact factorization of A within at most n steps.*

2.1. Fill-in of the factors. Let S_A be the set of index pairs (i, j), $i, j \leq n$ associated to the sparsity pattern of the matrix A. For large sparse matrices, we only allocate memory for the entries in a sparsity pattern. We defined the sparsity pattern of the sum and product of two matrices L and U:

$$S_{L+U} = \{(i,j) \mid (i,j) \in S_L \text{ or } (i,j) \in S_U\} = S_L \cup S_A, \tag{2.4}$$

$$S_{LU} = \{(i,j) \mid \exists\, k, \;\; s.t.\; (i,k) \in S_L \text{ and } (k,j) \in S_U\}\,. \tag{2.5}$$

The number of nonzeros of the factors increases with the increasing iteration number k, as it involves the matrix-matrix multiplication $L_0 U_0$ in step 3 of in Algorithm 1. We call the new nonzeros in the factors compared with the original matrix A *fill-ins*. To study the fill-in property of the factors, we introduce the concepts of *fill-path* and *level-of-fill* in sparse direct solution methods [31]. If a matrix is regarded as a oriented graph, a *fill-path* of two vertexes i and j is a path between them, such that all the vertexes in this path, except the end points i and j, are numbered less than i and j. If the length of the shortest path between i and j is $p+1$, the *level-of-fill* value of the position (i, j) is p. If the level-of-fills of all the positions in a sparsity pattern of a matrix are less than or equal to p, we call this pattern *level-p* pattern of this matrix. Then the level-of-fills of the nonzero positions of a matrix are 0, and the sparsity pattern consisting of all the nonzero positions is a level-0 pattern of this matrix. Also, there is the following result about the level-1 pattern.

LEMMA 2.2. *[2, 31] If the sparsity patterns of L and U are the same as the lower and upper part of A, respectively, then the sparsity pattern of the product LU is the level-1 pattern of A.*

THEOREM 2.3. *For $k \geq 2$, the sparsity pattern of $L_0^k + D^k + U_0^k$ is the level-1 pattern of $L_0^{k-1} + D^{k-1} + U_0^{k-1}$.*

Proof. In the k^{th} $(k \geq 1)$ iteration, by (2.4) and (2.5) $B^k = A - L_0^{k-1}U_0^{k-1}$ (step 3) implies

$$S_{B^k} = S_A \cup S_{L_0^{k-1}U_0^{k-1}}, \tag{2.6}$$

and $L_0^k D^k + D^k + U_0^k = B^k$ (step 4, 6 and 5) implies

$$S_{B^k} = S_{L_0^k} \cup S_{D^k} \cup S_{U_0^k} \tag{2.7}$$

Also, by (2.4), (2.5) and (2.7), we have an expansion

$$\begin{aligned} S_{L^kU^k} &= S_{(L_0^k+I)(D^k+U_0^k)} \\ &= S_{L_0^k D} \cup S_{D^k} \cup S_{U_0^k} \cup S_{L_0^k U_0^k} \\ &= S_{L_0^k} \cup S_{D^k} \cup S_{U_0^k} \cup S_{L_0^k U_0^k} \\ &= S_{B^k} \cup S_{L_0^k U_0^k}. \end{aligned} \tag{2.8}$$

By Lemma 2.2, the conclusion of this theorem is equivalent to

$$S_{B^k} = S_{L^{k-1}U^{k-1}}.$$

In the following, we prove this by induction.

For $k = 1$, we have $B = A$ as the initial data. Then by (2.8), we have

$$S_{L^1U^1} = S_B \cup S_{L_0^1U_0^1} \equiv S_A \cup S_{L_0^1U_0^1} \tag{2.9}$$

For $k = 2$, from step 3 and (2.9), we have

$$S_B = S_A \cup S_{L_0^1U_0^1} = S_{L^1U^1}.$$

For $k \geq 3$, we assume by induction that

$$S_{B^k} = S_{L^{k-1}U^{k-1}}.$$

By (2.7) and (2.8), we expand its both sides

$$\begin{aligned} S_{L_0^k} \cup S_{D^k} \cup S_{U_0^k} = S_{B^k} &= S_{L^{k-1}U^{k-1}} \\ &= S_{L_0^{k-1}} \cup S_{D^{k-1}} \cup S_{L_0^{k-1}} \cup S_{L_0^{k-1}U_0^{k-1}}. \end{aligned}$$

which implies

$$S_{L_0^k} \supset S_{L_0^{k-1}} \quad \text{and} \quad S_{U_0^k} \supset S_{U_0^{k-1}}.$$

and consequently

$$S_{L_0^k U_0^k} \supset S_{L_0^{k-1}U_0^{k-1}}. \tag{2.10}$$

Then for the next iteration $k+1$, we have

$$\begin{aligned} S_{B^{k+1}} &= S_A \cup S_{L_0^k U_0^k} \\ &= S_A \cup S_{L_0^{k-1}U_0^{k-1}} \cup S_{L_0^k U_0^k} && \text{(using (2.10))} \\ &= S_{B^k} \cup S_{L_0^k U_0^k} && \text{(using (2.6))} \\ &= S_{L^kU^k}, && \text{(using (2.8))} \end{aligned}$$

which proves the claim for the index $k+1$. □

REMARK 2.1. *In Algorithm 1, the sparsity pattern of the factorizations* (L^1, U^1) *and* (L^2, U^2) *are the level-*0 *and level-*1 *patterns of the original matrix. However, it is a progressive relation in Theorem 2.3, which is different from the conventional concept of level-*$(k-1)$ *pattern, when* $k \geq 3$*. Usually, in* k^{th} $(k \geq 3)$ *iteration, the pattern* (L^k, U^k) *is larger than the level-*$(k-1)$ *pattern.*

3. Iterative incomplete LU. The number of fill-ins in the factors of Algorithm 1 increases rapidly when the number of iterations increases, because of the matrix-matrix multiplication L_0U_0 in step 3. In order to control the storage of the factors, we drop some entries in each iteration according to some rules and stop the iteration with an appropriate number, which is the reason that we call it the incomplete LU (ILU) factorization. We summarize this idea in Algorithm 2. In the following of this section, we introduce two dropping strategies, based on pattern and threshold, respectively.

Algorithm 2 Iterative ILU factorization

1: Set the initial data $L_0 = D = U_0 = 0$.
2: **for** $k = 1, 2, \cdots, p$ **do**
3: $B = L_0U_0$
4: $B = A - B$
5: $D = diag(diag(B))$
6: $U_0 = triu(B, 1)$
7: $L_0 = tril(B, -1)D^{-1}$
8: execute dropping strategies on L_0 and U_0.
9: **end for**
10: $L = L_0 + I$ and $U = U_0 + D$.

3.1. Pattern-based. In Algorithm 3, we set a sparsity pattern $\boldsymbol{S}$ in advance, and drop then entries of L_0 and U_0 out of $\boldsymbol{S}$ in each iteration. If $\boldsymbol{S}$ contains the sparsity pattern of A, after the first iteration, $L_0 = A_{L0}A_D^{-1}$ and $U_0 = A_{U0}$, where A_{L0}, A_D and A_{U0} are the strictly lower triangular, diagonal and strictly upper triangular parts of the original matrix A, respectively. Then, the corresponding ILU preconditioner $LU = (A_{L0} + A_D)A_D^{-1}(A_D + A_{U0})$ is the SSOR(1) preconditioner [31], which can be regarded as the de facto initial data of this algorithm. This is a reasonable initial guess, as the SSOR(1) preconditioner already works well for some problems.

Algorithm 3 Iterative ILU factorization

1: Set the initial data $L_0 = D = U_0 = 0$ and the sparsity pattern $\boldsymbol{S}$.
2: **for** $k = 1, 2, \cdots, p$ **do**
3: $B = L_0U_0$
4: $B = A - B$
5: $D = diag(diag(B))$
6: $U_0 = triu(B, 1)$
7: $L_0 = tril(B, -1)D^{-1}$
8: drop the entries of L_0 and U_0 out of the pattern $\boldsymbol{S}$.
9: **end for**
10: $L = L_0 + I$ and $U = U_0 + D$.

In the following, we prove the convergence of Algorithm 3. We use Figure 3.1 to illustrate the procedure of the proof.

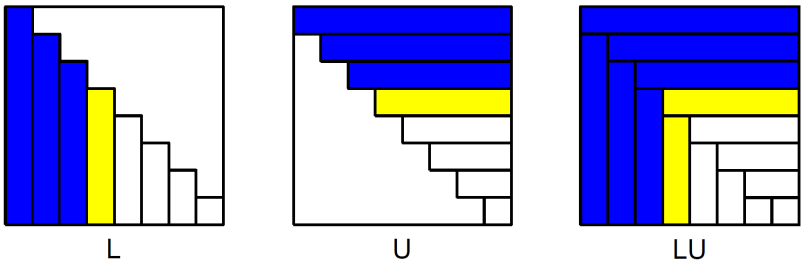

Fig. 3.1: The entries in the blue part of each matrix are exact values. In the next iteration, these exact entries will not change (Lemma 3.1), and the yellow rows and columns will become exact (Lemma 3.2).

LEMMA 3.1. *In Algorithm 3, if the product LU is equal to A on the first k rows and the first k columns of the sparsity pattern $\boldsymbol{S}$, then the first k columns of L and first k rows of U do not change in the next iteration.*

Proof. As the dropping rule in step 8, we only consider the entries in the pattern $\boldsymbol{S}$ during the proof. Let l_{ij} and u_{ij} denote the entries of the lower and upper triangular factors L and U, respectively. As in the settings of Section 2, all the diagonal entries $l_{ii} = 1$, and the diagonal entries of D and U are the same. Let l'_{ij} and u'_{ij} be the updated values of l_{ij} and u_{ij} in the next iteration, respectively. From the assumption, we have (see the blue part in the third figure in Figure 3.1)

$$a_{ij} = \sum_{m=1}^{\min(i,j)} l_{im}u_{mj} \quad \text{for } i \le k \text{ or } j \le k, \quad (i,j) \in \boldsymbol{S},$$

and divide the sum into two parts:

$$\begin{aligned} a_{ij} &= \sum_{m=1}^{i-1} l_{im}u_{mj} + u_{ij} \quad \text{for } i \le k, i \le j, \quad (i,j) \in \boldsymbol{S}, \\ a_{ij} &= \sum_{m=1}^{j-1} l_{im}u_{mj} + l_{ij}u_{jj} \quad \text{for } j \le k, i > j, \quad (i,j) \in \boldsymbol{S}, \end{aligned} \tag{3.1}$$

(here we use the setting $l_{ii} = 1$).

In the next iteration,

$$b_{ij} = a_{ij} - \sum_{m=1}^{\min(i-1,j-1)} l_{im}u_{mj}. \quad \text{(step 3 and 4)} \tag{3.2}$$

The new upper factor becomes

$$u'_{ij} = b_{ij} = a_{ij} - \sum_{m=1}^{i-1} l_{im}u_{mj} \quad \text{for } i \le k, i \le j, \quad \text{(step 6)}$$

and by the first equation of (3.1), we have

$$u'_{ij} = u_{ij}, \quad \text{for} \quad i \le k. \tag{3.3}$$

The new lower factor becomes

$$l'_{ij}u'_{jj} = b_{ij} = a_{ij} - \sum_{m=1}^{j-1} l_{im}u_{mj} \quad \text{for } j \le k, i > j, \quad \text{(step 5 and 7)}$$

and by the second equation of (3.1), we have $l'_{ij}u'_{jj} = l_{ij}u_{jj}$. From (3.3), we obtain $l'_{ij} = l_{ij}$ for $j \le k$. □

LEMMA 3.2. *In Algorithm 3, if the product LU is equal to A on the first k ($k \le n-1$) rows and the first k columns of the sparsity pattern $\boldsymbol{S}$, then in the next iteration, LU is equal to A on the first $k+1$ rows and the first $k+1$ columns of the sparsity pattern $\boldsymbol{S}$.*

Proof. By Lemma 3.1, we only need to investigate the $(k+1)^{th}$ row and $(k+1)^{th}$ column of the new factors.

From (3.2), for $j = k+1, \cdots, n$ and $(k+1, j) \in \boldsymbol{S}$,

$$u'_{k+1,j} = b_{k+1,j} = a_{k+1,j} - \sum_{m=1}^{k} l_{k+1,m}u_{m,j}.$$

We then have

$$a_{k+1,j} = \sum_{m=1}^{k} l'_{k+1,m}u'_{m,j} + l'_{k+1,k+1}u'_{k+1,j} = \sum_{m=1}^{n} l'_{k+1,m}u'_{m,j}.$$

Here we use the setting $l'_{k+1,k+1} = 1$ and the result of Lemma 3.1, i.e. $l_{k+1,m} = l'_{k+1,m}$ and $u_{m,j} = u'_{m,j}$ because of $m \le k$. Then the $(k+1)^{th}$ row of U, i.e. $u_{k+1,j}$ for $j = k+1, \cdots, n$, become exact values (see the yellow part of the second panel in Figure 3.1).

Similarly, for $i = k+2, \cdots, n$ and $(i, k+1) \in \boldsymbol{S}$, we have

$$l'_{i,k+1}u'_{k+1,k+1} = b_{i,k+1} = a_{i,k+1} - \sum_{m=1}^{k} l_{i,m}u_{m,k+1},$$

$$a_{i,k+1} = \sum_{m=1}^{k} l'_{i,m}u'_{m,k+1} + l'_{i,k+1}u'_{k+1,k+1} = \sum_{m=1}^{n} l'_{i,m}u'_{m,k+1}.$$

Then the $(k+1)^{th}$ column of L, i.e. $l_{i,k+1}$ for $i = k+2, \cdots, n$, become exact values (see the yellow part of the first panel in Figure 3.1). □

THEOREM 3.3. *The factors in Algorithm 3 converge to the standard incomplete factors on the sparsity pattern $\boldsymbol{S}$ within n iterations.*

Proof. After the iteration $k = 1$, the entries in the first column of L^1 and the first row of U^1 are, respectively,

$$l^1_{i1} = \begin{cases} a_{i1}/a_{11} & (i,1) \in \boldsymbol{S} \\ 0 & (i,1) \notin \boldsymbol{S} \end{cases} \quad \text{and} \quad u^1_{1j} = \begin{cases} a_{1j} & (1,j) \in \boldsymbol{S} \\ 0 & (1,j) \notin \boldsymbol{S}. \end{cases} \tag{3.4}$$

Then it can be verified that the multiplication L^1U^1 is equal to A on the first row and the first column of the sparsity pattern $\boldsymbol{S}$.

By Lemma 3.1 and 3.2, we know that one more iteration makes one more row and column exact until we obtain the exact ILU factorization on $\boldsymbol{S}$. □

Theorem 2.1 is a special case of Theorem 3.3, when $\boldsymbol{S}$ contains all the positions of the matrix A.

The level-0 sparsity pattern is the most commonly used in ILU factorizations. To obtain more accurate factors, we can enlarge the sparsity pattern to allow more fill-ins. If we iterate Algorithm 3 several times without dropping any entries, the resulting factors are of a larger sparsity pattern by Theorem 2.3 and Remark 2.1. Then the factors and their sparsity pattern can be taken as the initial data and a fixed pattern, respectively, when using Algorithm 3. We summarize this strategy in Algorithm 4, and call it IterILU(p, m). Here p indicates the size of the pattern and m is the iteration number to enhance the accuracy of the factors on this pattern. We obverse that Algorithms 1 and 3 can be regarded as special cases of Algorithm 4:

- $p = 1$ and $m = 0$, SSOR(1) preconditioner;
- $p = 1$ and $m \geq 1$, Algorithm 3 with m iterations on level-0 sparsity pattern;
- $p \geq 2$ and $m = 0$, Algorithm 1 with p iterations;
- $p \geq 2$ and $m \geq 1$, Algorithm 1 with p iterations, then on the sparsity pattern of the results, execute Algorithm 3 with m enhancement iterations.

Usually, the storage increases rapidly with increasing p. When setting p, the memory in the computing resource and the time for solving the corresponding triangular systems should be taken into consideration. We write a short code for this algorithm in Matlab, pasted in Appendix A.

Algorithm 4 Iterative ILU factorization IterILU(p, m)

Set the initial data $L_0 = D = U_0 = 0$.
for $k = 1, 2, \cdots, p$ **do**
  $B = A - L_0U_0$
  $D = diag(diag(B))$
  $U_0 = triu(B, 1)$
  $L_0 = tril(B, -1)D^{-1}$
end for
get the sparsity pattern $\boldsymbol{S}$ of B
for $k = 1, 2, \cdots, m$ **do**
  $B = L_0U_0$
  $B = A - B$
  $D = diag(diag(B))$
  $U_0 = triu(B, 1)$
  $L_0 = tril(B, -1)D^{-1}$
  drop the entries of L_0 and U_0 out of the pattern $\boldsymbol{S}$.
end for
$L = L_0 + I$ and $U = U_0 + D$.

3.2. Threshold-based. We propose another dropping strategy for L_0 and U_0, which is based on only the magnitudes of the entries in each row or column:

- Find the largest absolute value Γ in each row of L, delete the entries whose absolute values are smaller than $\tau\Gamma$ in the corresponding row of L_0.

- Find the largest absolute value Γ in each column of U, delete the entries whose absolute values are smaller than $\tau\Gamma$ in the corresponding column of U_0.

Here, the threshold τ is uniformly set for L_0 and U_0. We call this algorithm IterILUT(τ, p) in Algorithm 5, where p is the iteration number.

Algorithm 5 Iterative threshold ILU factorization IterILUT(τ, p)

1: Set the initial data $L_0 = D = U_0 = 0$.
2: **for** for $k = 1, 2, \cdots, p$ **do**
3: $B = L_0 U_0$
4: $B = A - B$
5: $D = diag(diag(B))$
6: $U_0 = triu(B, 1)$
7: $L_0 = tril(B, -1)D^{-1}$
8: execute dropping strategies on L_0 and U_0 according to the threshold τ.
9: **end for**
10: $L = L_0 + I$ and $U = U_0 + D$.

The application range of the threshold-based factorization should be smaller than the pattern-based ones. In some matrices, the magnitudes of their entries may cluster in some small regions, not relatively uniform distribute on the number axis. For example, the matrices generated by FDM or FEM on uniform meshes contain many entries with the same values. If the threshold τ is too large, an important part of the matrix may be dropped, which heavily damages the accuracy of the factors. If τ is too small, there may be no effect in reducing the storage. This strategy should be applied to those matrices whose entries distribute smoothly. And the threshold also should be chosen deliberately according to the desired accuracy and storage of the factors. Of course, there should be hybrid dropping strategies based on positions, numbers and magnitudes of entries.

4. Some other issues involved in the algorithms. In this section, we discuss some details in the implementation of the new algorithms.

4.1. Matrix-matrix multiplication. The main computational task of the new algorithms is the sparse matrix-matrix multiplication which is a fundamental operation in the field of high-performance computing. The existing algorithms and codes for this operation have been studied by many experts in various aspects for many years [11, 17, 27]. Many theories and techniques have been introduced for the optimization of its algorithms, especially for sparse cases. Most computing libraries contain efficient codes for this operation on both sequential and parallel computer architectures.

A risk in performing this operation is that the product may contain too many fill-ins, even though the two matrices are very sparse. In some extreme cases, it can even result in a full matrix, for example, the first column of L_0 and the first row of U_0 are nonzero. The reordering techniques can reduce the number of fill-ins for some cases[22], and also possibly improve the accuracy of the factors. Another method is that we only compute the entries contained in the given pattern during the matrix-matrix multiplication, which we call **the Incomplete Matrix-Matrix Multiplication** (IMMM). Algorithm 6 implements this idea, which is mathematically equivalent to, but theoretically cheaper than Algorithm 3. However, to the

Algorithm 6 Iterative ILU factorization

Set the initial data $L_0 = D = U_0 = 0$.
for $k = 1, 2, \cdots, p$ **do**
  $B = L_0 U_0$ (IMMM on $\boldsymbol{S}$)
  $B = A - B$ (on $\boldsymbol{S}$)
  $D = diag(diag(B))$
  $U_0 = triu(B, 1)$
  $L_0 = tril(B, -1)D^{-1}$
end for
$L = L_0 + I$ and $U = U_0 + D$.

authors' knowledge, there is no such operation in the existing libraries. It is easy to implement in parallel according to the definition on shared-memory platforms, while on distributed-memory platforms, the users have to deal with the communications between different processors. If the memory and runtime can afford these fill-ins, Algorithm 3 and 4 are recommended.

4.2. Triangular system solvers. After obtaining the factors of an incomplete factorization, the preconditioning procedure requires solving two triangular linear systems $L(Uy) = x$. The forward or backward substitution is a common method, which can be parallelized by level scheduling [1]. Another method is to use Jacobi iterations [2]. In the iteration schemes:

$$\begin{aligned} z^k &= x - L_0 z^{k-1}, \\ y^k &= D^{-1} z - D^{-1} U_0 y^{k-1}, \end{aligned} \tag{4.1}$$

the iteration matrices $-D^{-1}U_0$ and $-L_0$ are strictly upper and lower triangular matrix. We summarize this method in Algorithm 7. The spectral radii of the two matrices are both 0, which means fast convergence of the schemes. By using this method, the main task of the preconditioned iterative solver remains only matrix-vector products, which benefits from the parallel implementation. Moreover, the preconditioned solution is only a quite rough approximation for the exact solution, so there is no need to iterate the schemes (4.1) until reaching a very small error. In general, a few iteration steps for each scheme are accurate enough for preconditioning purposes.

Algorithm 7 Jacobi method for the systems $L(Uy) = x$ with p iterations.

Set the initial data $z^0 = 0$ and $y^0 = 0$.
for for $k = 1, 2, \cdots, q$ **do**
  $z^k = x - L_0 z^{k-1}$
end for
for for $k = 1, 2, \cdots, q$ **do**
  $y^k = D^{-1} z^p - D^{-1} U_0 y^{k-1}$
end for
output $y = y^p$

5. Numerical experiments I: linear systems. In this section, we apply the new preconditioners to solving linear systems with the Preconditioned Conjugate Gradient method (PCG). The main purpose is to determine the parameters in the new algorithms and to compare the new preconditioners with the standard ILU.

	2D Laplace 100×100		3D Laplace $100 \times 100 \times 100$	
	nnz	ratio	nnz	ratio
$p = 1$	29800	1	3970000	1
$p = 2$	39601	1.33	6910300	1.74
$p = 3$	49303	1.65	12721996	3.20
$p = 4$	68608	2.30	28972351	7.30
$p = 5$	97025	3.26	72694564	18.31
$p = 6$	143276	4.81	201462286	50.75

Table 5.1: Numbers of nonzeros in the lower triangular factor L generated by IterILU(p,m) for increasing p and $m = 0$ for the 2D and 3D finite difference Laplacian.

The code is implemented in Matlab and run on a laptop with an Intel i7-6700HQ CPU and 16 GB RAM.

5.1. IterILU(p,m). In this section, we consider linear system with coefficient matrices originating from finite difference discretizations of the Laplacian with homogeneous Dirichlet boundary condition and we investigate the basic properties and effects of the iterative ILU preconditioners IterILU(p,m) in Algorithm 4. The two- and three-dimensional grids are 102×102 and $102 \times 102 \times 102$, so the corresponding square matrices have 10^4 and 10^6 rows, respectively.

Firstly, we test the fill-in property of the factors generated by IterILU(p,m). Table 5.1 shows the numbers of nonzeros of L for different p, and the ratios between these number and the number of nonzeros of the case $p = 1$. The numbers of nonzeros increase with p and the 3D ratios increase more rapidly than the 2D ratios, as its matrices have more nonzero entries in each row or column. In order to keep the storage required by the factors under control, we restrict our tests to the values $p = 1, 2, 3$.

The next two tests focus instead on the effects of varying the enhancement number m. The test matrix is the 3D Laplacian. In order to show the difference between incomplete factors and the exact one, we introduce a relative error:

$$\text{relative error}(A, LU) = \max_i \left\{ \frac{\sum_j^n |A_{ij} - (LU)_{ij}|}{\sum_j^n |A_{ij}|} \right\}. \tag{5.1}$$

Figure 5.1 (left) shows that the relative errors decrease with increasing m, but reach a plateau after a few enhancement iterations m. We then employ the factors of IterILU(p,k) as preconditioners within the PCG method with right-hand side given by random values in $[-1, 1]$ and 0 initial gauss. The PCG iteration numbers reach a stable value after a few enhancement iterations, shown in Figure 5.1 (right). From the two panels of Fig. 5.1, we conclude that 3 enhancement iterations are enough for both the relative errors and iteration numbers to reach stable values.

In the test above, the triangular systems $L(Uy) = x$ in the preconditioning procedures of PCG are solved using a direct method. In the next numerical experiment, we use instead the Jacobi iterative method (4.1), Algorithm 7 to solve the two triangular systems. Figure 5.2 shows the relation between the PCG and Jacobi iteration counts. From the results, we observe that the PCG iteration counts with IterILU(p,m) preconditioners become stable after 6 iterations for $(p, m) = (1, 3)$, 8 iterations for $(p, m) = (2, 3)$ and 12 iterations for $(p, m) = (3, 3)$. Hence, it is not appropriate to

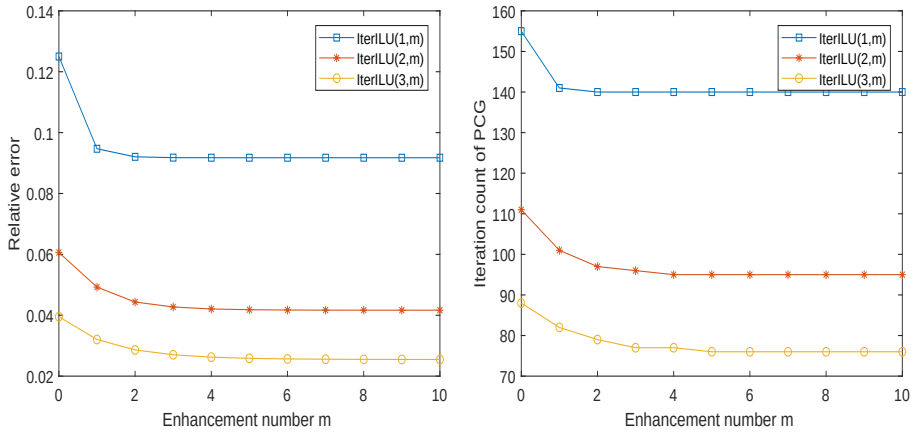

Fig. 5.1: Left: relative errors of IterILU(p,m) with $p = 1, 2, 3$ and different enhancement numbers m. Right: PCG iteration counts with different values of p and m.

set a uniform value for the iteration number q in the Jacobi method. If q is too small, the accuracy of preconditioning may be not enough for some problems, while if it is too large, it is possible to waste computation. This number should be calibrated according to the specific problem considered. In order to eliminate the influence of different settings for this number, we use an exact solver for the triangular systems in the remaining tests.

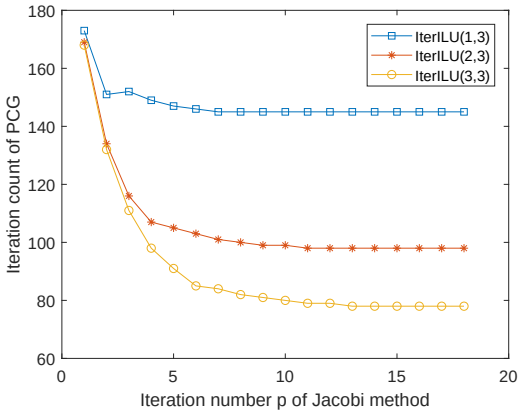

Fig. 5.2: PCG iteration counts using the Jacobi method of Algorithm 7 with different iteration numbers for the preconditioner triangular solves.

We plot the PCG convergence history using IterILU(p,m) with different parameters and compare them with the standard level-0 ILU preconditioner, Figure 5.3. IterILU(1,3) and standard ILU yield the same results, which means they have the similar effects for this problem. The differences are that IterILU(1,3) is easier to generate and has naturally fine-grained parallelism. IterILU(2,3) and IterILU(3,3) yield

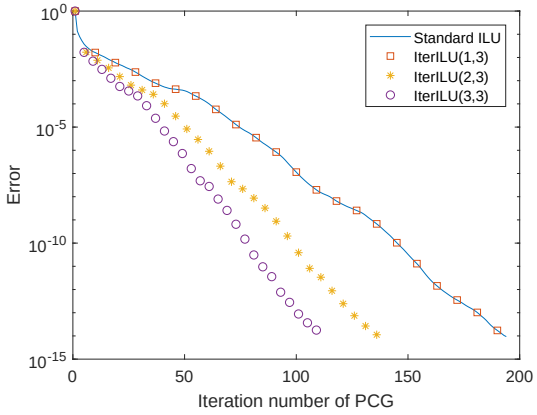

Fig. 5.3: Convergence history for solving the 3D Laplace problem using PCG with different ILU preconditioners.

	row/column	nnz	$p=1$	$p=2$	$p=3$
apache1	80800	542184	1	1.70	3.01
apache2	715176	4817870	1	1.71	3.02
thermal1	82654	574458	1	1.61	2.39
thermal2	1228045	8580313	1	1.62	2.42
parabolic_fem	525825	3674625	1	1.65	2.59
G3_circuit	1585478	7660826	1	1.31	1.69
thermomech_dM	204316	1423116	1	1.68	2.77
thermomech_TC	102158	711558	1	1.68	2.77
ecology2	999999	4995991	1	1.33	1.67
offshore	259789	4242673	1	2.21	6.16
af_shell3	504855	17562051	1	1.42	1.97

Table 5.2: The second and third columns in this table are the rows and numbers of nonzeros of some SPD matrices from the University of Florida sparse matrix collection [14], respectively. The rightmost two columns are the ratios of nonzeros of the IterILU(p,m) lower factor L with different p and the nonzeros with $p=1$.

faster PCG convergence than the level-0 case, at the cost of allowing more entries in their triangiular factors.

Next, we carry out the same study for some symmetric matrices from the University of Florida sparse matrix collection [14]. Table 5.2 lists the dimension, number of nonzero entries and ratios of numbers of nonzeros in the IterILU(p,m) factors generated with different p. Table 5.3 shows the iteration counts of PCG using different preconditioners. For most cases, the IterILU(1,3) preconditioner yields similar counts with the standard level-0 ILU. The iteration counts decrease with increasing p. The IterILU(p,m) preconditioners do not work for the last two test cases.

5.2. IterILUT(τ,k). We test the threshold type factorization IterILUT(τ,p) in Algorithm 5. As discussed in the previous section, the parameters of IterILUT(τ,p)

	no p.c.	level-0 ILU	IterILU(1,3)	IterILU(2,3)	IterILU(3,3)
apache1	3774	364	359	281	249
apache2	5468	874	874	592	473
thermal1	1734	659	659	369	252
thermal2	6727	2598	2599	1443	984
parabolic_fem	3495	1432	1432	853	553
G3_circuit	19975	1152	1178	667	550
thermomech_dM	89	10	10	6	4
thermomech_TC	89	10	10	6	4
ecology2	7154	2125	2127	1338	1085
	no p.c.	level-0 ILU	IterILU(1,10)	IterILU(2,10)	IterILU(3,10)
offshore	-	567	574	-	-
af_shell3	4700	1172	-	-	-

Table 5.3: PCG iteration counts of different ILU preconditioners for linear systems from the University of Florida sparse matrix collection [14].

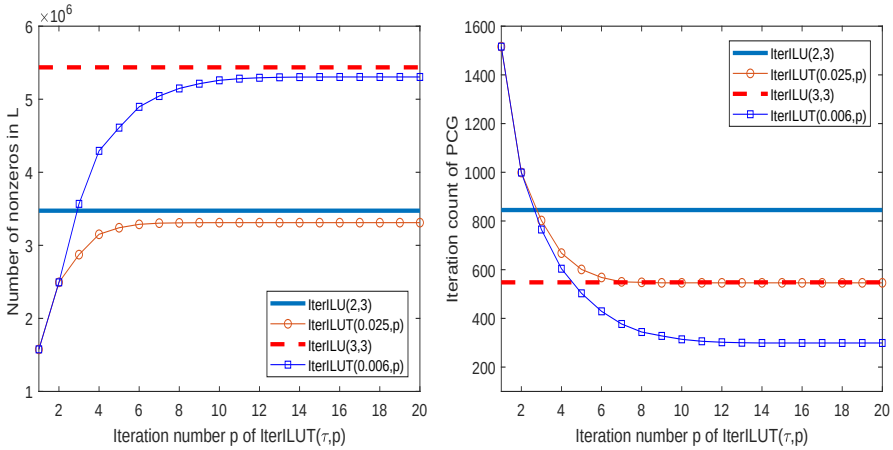

Fig. 5.4: Left: the number of nonzeros in the lower factor L generated by IterILUT(τ,p) with different threshold parameters and iteration numbers. Rigth: PCG iteration numbers using the IterILUT(τ,p) preconditioners with different τ and p.

should be determined carefully. Here, we consider the *parabolic_fem* matrix in [14] and choose $\tau = 0.025$ and $\tau = 0.006$. The reason we choose these two parameters is that the factors generated by them have similar numbers of nonzeros compared with IterILU(2,m) and IterILU(3,m), respectively. Figure 5.4 (left) shows the numbers of nonzeros in the lower factor L of IterILUT(τ,p) as a function of the enhancement number k. While the storage required is similar, we can observe the advantage of the threshold ILU preconditioners. Here, we set the stopping tolerance to 10^{-10}. Figure 5.4 (right) shows the PCG iteration counts as a function of the enhancement number m. Both the numbers of nonzeros in the factors and the PCG iteration counts tend to flatten out after $p = 5$ for $\tau = 0.025$ and $p = 10$ for $\tau = 0.006$. Then we choose

these values of p for each τ and we plot the PCG convergence history with different preconditioners in Figure 5.5. These results indicate that while the number of nonzeros is similar, IterILUT(τ,p) performs better than IterILU(p,m) for this problem.

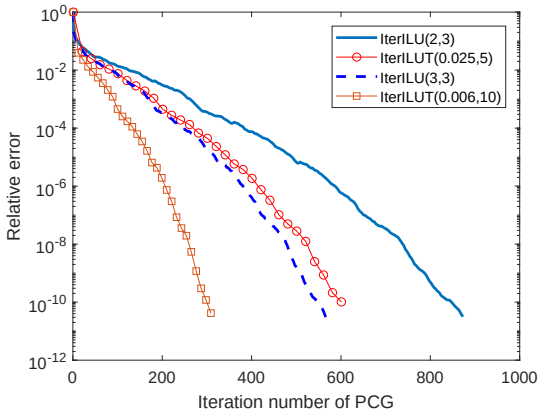

Fig. 5.5: PCG convergence history with IterILUT(τ,p) and IterILU(p,m) preconditioners for the *parabolic_fem* problem.

6. Numerical experiments II: eigenvalue problems. In this section, we apply the IterILU(p,m) preconditioner in Algorithm 4 to solve eigenvalue problems. We use the Locally Optimal Block Preconditioned Conjugate Gradient method (LOBPCG) [24, 16, 33] to compute part of the spectrum. In addition to preconditioners, there are many factors that impact the convergence behaviors of eigensolvers, such as the initial data, stopping criterion, the number of eigenvalue computed simultaneously and random errors. When the eigensolver convergence degenerates or even fails, the eigensolver can be restarted with different initial data and parameters which may lead to better convergence.

We start by computing some of the smallest eigenvalues of the 3D finite difference Laplacian matrix with 10^6 rows considered in the previous section. We compute the four smallest eigenvalues simultaneously, set the stopping criterion to 10^{-12} and use random initial data. Figure 6.1 shows the LOBPCG convergence history with different preconditioners. The results are similar to solving linear systems with PCG. The level-0 iterative preconditioner has almost the same performance of the standard level-0 ILU preconditioner. The convergence rates improve with increasing levels of fill-ins.

6.1. An eigenvalue problem in mixed form. In this subsection, we compute the Laplace eigenvalue problem in mixed form, i.e. we consider a generalized eigenvalue problem generated by mixed finite element discretizations. To this end, we rewrite the Laplace eigenvalue problem

$$\begin{cases} -\Delta u = \lambda u & \text{in } \Omega \\ u = 0 & \text{on } \partial\Omega \end{cases}$$

as a first-order system:

$$\begin{cases} \sigma - \operatorname{grad} u = 0 & \text{in } \Omega \\ \operatorname{div}\sigma = -\lambda u & \text{in } \Omega \\ u = 0 & \text{on } \partial\Omega. \end{cases}$$

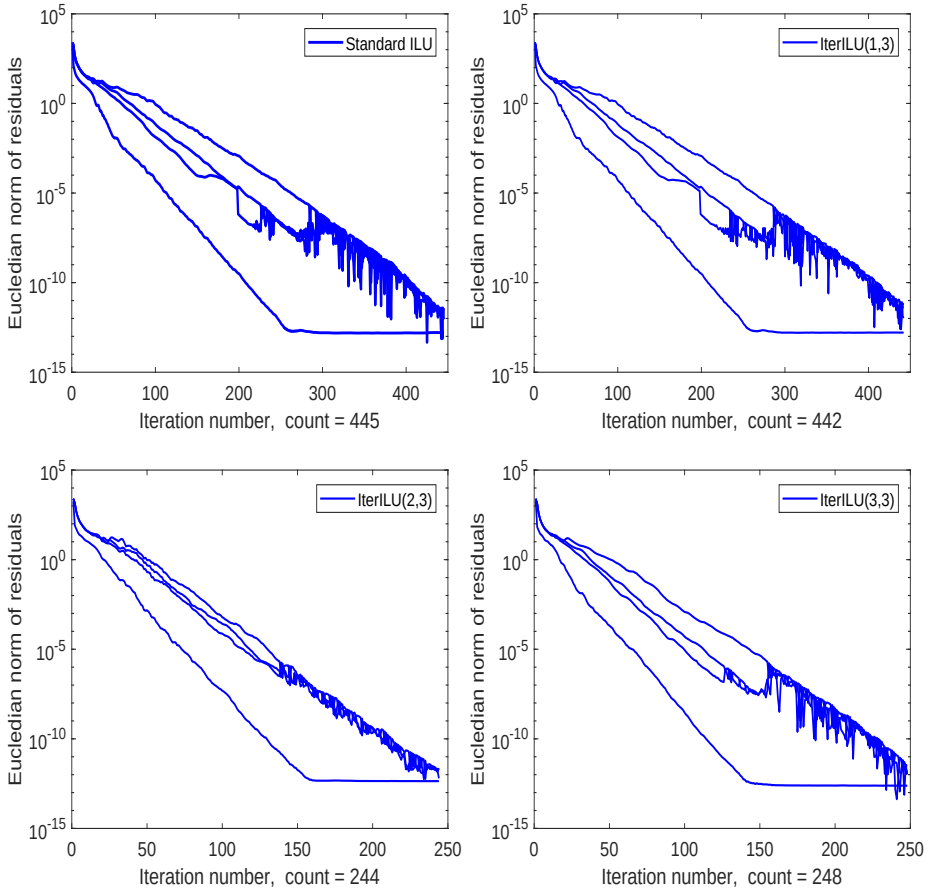

Fig. 6.1: LOBPCG convergence history in solving the Laplace eigenvalue problem with 10^6 rows using different ILU preconditioners.

We then consider the two-dimensional case and approximate this differential system with Raviat-Thomas elements; see [8, 9, 10] and the references therein for more details. The resulting algebraic system can be written in block matrix form as

$$\begin{pmatrix} A & B^T \\ B & 0 \end{pmatrix} \begin{pmatrix} \sigma \\ u \end{pmatrix} = \mu \begin{pmatrix} 0 & 0 \\ 0 & M \end{pmatrix} \begin{pmatrix} \sigma \\ u \end{pmatrix}, \qquad \lambda = -\mu, \tag{6.1}$$

where A and M are Hermitian matrices.

The standard LOBPCG method finds the eigenvalues at the two ends of the spectrum of Hermitian systems by using a Rayleigh-Ritz procedure. However, the spectrum of (6.1) distributes as follows

$$\mu_M \leq \cdots \leq \mu_2 \leq \mu_1 (\leq 0) \leq \infty_1 = \infty_2 = \cdots = \infty_N.$$

Here, the infinite eigenvalues are generated by the eigenpairs $(\sigma, u)^T$ with $\sigma \neq 0$ and $u = 0$. The desired eigenvalues are the ones with smallest magnitude values,

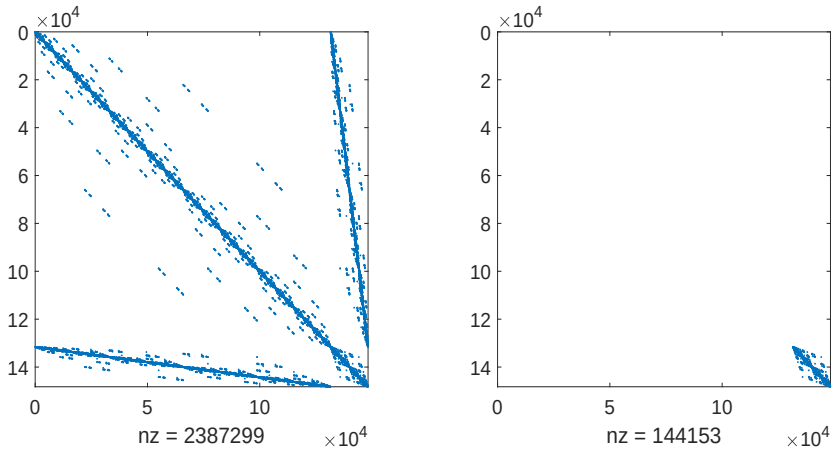

Fig. 6.2: The sparsity patterns of the matrices generated by mixed FEM.

which are interior eigenvalues. In order to approximate these aigenvalues, we replace the Rayleigh-Ritz procedure by a Harmonic Rayleigh-Ritz procedure in LOBPCG. Another problem is that the bottom left block in the left-hand side is zero, while the new algorithm generating our preconditioners requires that the diagonal entries are all nonzeros. We then use a spectral transform method to add a mass matrix at both sides of (6.1), shown in Figure 6.2, and obtain an equivalent generalized eigenvalue problem:

$$\begin{pmatrix} A & B^T \\ B & M \end{pmatrix} \begin{pmatrix} \sigma \\ u \end{pmatrix} = \mu \begin{pmatrix} 0 & 0 \\ 0 & M \end{pmatrix} \begin{pmatrix} \sigma \\ u \end{pmatrix}, \qquad \lambda = -\mu + 1. \tag{6.2}$$

Then we can generate preconditioners for the the left-hand side matrix by using the IterILU(p,m) preconditioner in Algorithm 4.

In this test example, the block matrices are generated by the first order RT elements on a square domain $[0, \pi]^2$ with a mesh 128×128, and have 148225 rows. We set 10^{-12} as the stopping criterion tolerance and compute the six smallest magnitude eigenvalues. The results reported in Figure 6.3 indicate that the IterILU(1,3) preconditioner has a similar performance as the standard ILU preconditioner. Again, the convergence rates improve with increasing levels of fill-ins.

7. Conclusions and limitations. In this paper, we derived an iterative matrix form of the conventional LU factorization and constructed several iterative ILU preconditioners based on this iterative procedure. The new preconditioners have been successfully applied to solving both linear systems and eigenvalue problems. As the main task of the algorithms is a matrix-matrix multiplication, the proposed preconditioners can benefit from fine-grained parallelism. Leveraging the basic matrix operations in existing numerical libraries, the algorithm can be implemented efficiently on both serial and parallel computing platforms.

Limitations. The proposed algorithms, especially with higher levels of fill-in, may become unstable and fail for some matrices, as shown in Table 5.3. As the algorithms involve divisions in each iteration, we choose the matrices without zero diagonal entries

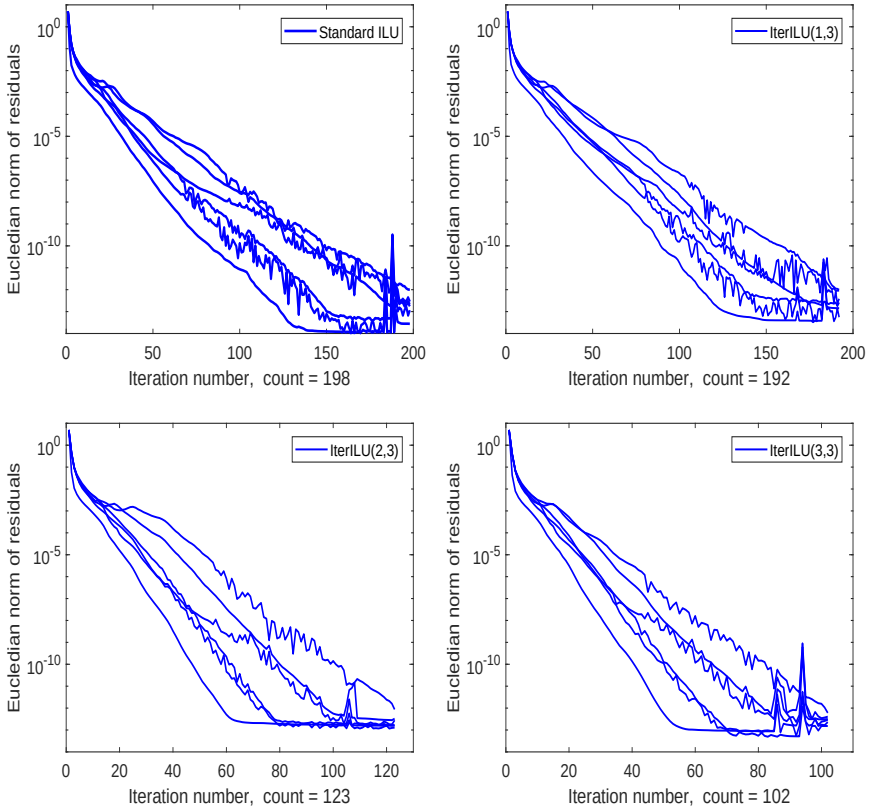

Fig. 6.3: The convergence history in solving mixed eigenvalue problem with different ILU preconditioners.

and assume that there is no breakdown during the computation. Future work is needed in order to deal with these singular cases and to enhance the robustness of the iterative factorizations. We only present two dropping strategies based on positions and magnitudes, respectively, but it would be interesting to explore hybrid strategies combining positions, magnitudes and numbers of nonzeros together.

Appendix A. IterILU code in Matlab.

```
function [L,U] = IterILU(A,p,m)

n = size(A,1);
L = sparse(n); U = L;
D = sparse(n,1);

for k = 1 : p
    B = L * U;
    B = A - B;
    D = diag(B); U = triu(B,1); L = tril(B,-1);
```

```
    L = L * diag(D.^(-1));
end

SP = (B~=0); % obtain the sparsity pattern SP.

for k = 1 : m
    B = L * U;
    B = A - B;
    % delete the entries out of the sparsity pattern SP.
    B = B.* SP;
    D = diag(B); U = triu(B,1); L = tril(B,-1);
    L = L * diag(D.^(-1));
end
L = L + speye(n);
U = U + diag(D);

end
```

REFERENCES

[1] Edward Anderson and Youcef Saad. Solving sparse triangular linear systems on parallel computers. *Internat. J. High Speed Comput.*, 1(01):73–95, 1989.

[2] Hartwig Anzt, Edmond Chow, and Jack Dongarra. ParILUT—a new parallel threshold ILU factorization. *SIAM J. Sci. Comput.*, 40(4):C503–C519, 2018.

[3] C. Cleveland Ashcraft and Roger G. Grimes. On vectorizing incomplete factorization and ssor preconditioners. *SIAM J. Sci. Statist. Comput.*, 9(1):122–151, 1988.

[4] Satish Balay, Shrirang Abhyankar, Mark Adams, Jed Brown, Peter Brune, Kris Buschelman, LD Dalcin, Victor Eijkhout, W. Gropp, Dinesh Kaushik, et al. Petsc users manual revision 3.8. Technical report, Argonne National Lab.(ANL), Argonne, IL (United States), 2017.

[5] Richard Barrett, Michael W Berry, Tony F Chan, James Demmel, June Donato, Jack Dongarra, Victor Eijkhout, Roldan Pozo, Charles Romine, and Henk Van der Vorst. *Templates for the solution of linear systems: building blocks for iterative methods*, volume 43. SIAM, 1994.

[6] Michele Benzi. Preconditioning techniques for large linear systems: a survey. *J. Comput. Phys.*, 182(2):418–477, 2002.

[7] Marshall Bern, John R. Gilbert, Bruce Hendrickson, Nhat Nguyen, and Sivan Toledo. Support-graph preconditioners. *SIAM J. Matrix Anal. Appl.*, 27(4):930–951, 2006.

[8] Daniele Boffi. Finite element approximation of eigenvalue problems. *Acta Numer.*, 19:1–120, 2010.

[9] Daniele Boffi, Franco Brezzi, and Michel Fortin. *Mixed finite element methods and applications*, volume 44. Springer, 2013.

[10] Daniele Boffi, Dietmar Gallistl, Francesca Gardini, and Lucia Gastaldi. Optimal convergence of adaptive fem for eigenvalue clusters in mixed form. *Math. Comp.*, 86(307):2213–2237, 2017.

[11] Aydin Buluç and John R. Gilbert. Parallel sparse matrix-matrix multiplication and indexing: Implementation and experiments. *SIAM J. Sci. Comput.*, 34(4):C170–C191, 2012.

[12] Caterina Calgaro, Jean-Paul Chehab, and Yousef Saad. Incremental incomplete LU factorizations with applications. *Numer. Linear Algebra Appl.*, 17(5):811–837, 2010.

[13] Edmond Chow and Aftab Patel. Fine-grained parallel incomplete LU factorization. *SIAM J. Sci. Comput.*, 37(2):C169–C193, 2015.

[14] Timothy A. Davis and Yifan Hu. The University of Florida sparse matrix collection. *ACM T. Math. Software (TOMS)*, 38(1):1, 2011.

[15] Howard C Elman. A stability analysis of incomplete LU factorizations. *Math. Comp.*, pages 191–217, 1986.

[16] Luca Ghezzi, Luca F. Pavarino, and Elena Zampieri. Overlapping schwarz preconditioned eigensolvers for spectral element discretizations. *Appl. Math. Comput.*, 218(15):7700–7710, 2012.

[17] John R. Gilbert, Cleve Moler, and Robert Schreiber. Sparse matrices in matlab: Design and implementation. *SIAM J. Matrix Anal. Appl.*, 13(1):333–356, 1992.
[18] Anne Greenbaum. *Iterative methods for solving linear systems*, volume 17. SIAM, 1997.
[19] Ivar Gustafsson. A class of first order factorization methods. *BIT*, 18(2):142–156, 1978.
[20] Pascal Hénon and Yousef Saad. A parallel multistage ILU factorization based on a hierarchical graph decomposition. *SIAM J. Sci. Comput.*, 28(6):2266–2293, 2006.
[21] David Hysom and Alex Pothen. A scalable parallel algorithm for incomplete factor preconditioning. *SIAM J. Sci. Comput.*, 22(6):2194–2215, 2001.
[22] David Hysom and Alex Pothen. Level-based incomplete LU factorization: Graph model and algorithms. *Preprint UCRL-JC-150789, US Department of Energy, Nov*, 2002.
[23] David S Kershaw. The incomplete Cholesky conjugate gradient method for the iterative solution of systems of linear equations. *J. Comput. Phys.*, 26(1):43–65, 1978.
[24] Andrew V. Knyazev. Toward the optimal preconditioned eigensolver: Locally optimal block preconditioned conjugate gradient method. *SIAM J. Sci. Comput.*, 23(2):517–541, 2001.
[25] Xiaoye S. Li. An overview of superlu: Algorithms, implementation, and user interface. *ACM T. Math. Software (TOMS)*, 31(3):302–325, 2005.
[26] Scott MacLachlan, Daniel Osei-Kuffuor, and Yousef Saad. Modification and compensation strategies for threshold-based incomplete factorizations. *SIAM J. Sci. Comput.*, 34(1):A48–A75, 2012.
[27] Michael McCourt, Barry Smith, and Hong Zhang. Sparse matrix-matrix products executed through coloring. *SIAM J. Matrix Anal. Appl.*, 36(1):90–109, 2015.
[28] Mardochée Magolu monga Made and Henk A. van der Vorst. A generalized domain decomposition paradigm for parallel incomplete LU factorization preconditionings. *Future Gener. Comp. Sy.*, 17(8):925–932, 2001.
[29] Yvan Notay. Conditioning analysis of modified block incomplete factorizations. *Linear Algebra Appl.*, 154:711–722, 1991.
[30] Yousef Saad. ILUT: A dual threshold incomplete LU factorization. *Numer. Linear Algebra Appl.*, 1(4):387–402, 1994.
[31] Yousef Saad. *Iterative methods for sparse linear systems*, volume 82. SIAM, 2003.
[32] Jennifer Scott and Miroslav Tuma. Improving the stability and robustness of incomplete symmetric indefinite factorization preconditioners. *Numer. Linear Algebra Appl.*, 24(5):e2099, 2017.
[33] Eugene Vecharynski and Andrew V. Knyazev. Preconditioned locally harmonic residual method for computing interior eigenpairs of certain classes of hermitian matrices. *SIAM J. Sci. Comput.*, 37(5):S3–S29, 2015.
[34] Andrew J. Wathen. Preconditioning. *Acta Numer.*, 24:329–376, 2015.