CROUT VERSIONS OF THE ILU FACTORIZATION WITH PIVOTING FOR SPARSE SYMMETRIC MATRICES [*]

NA LI [†] AND YOUSEF SAAD[†]

Abstract. The Crout variant of ILU preconditioner (ILUC) developed recently has been shown to have a number of advantages over ILUT, the conventional row-based ILU preconditioner [14]. This paper explores pivoting strategies for sparse symmetric matrices to improve the robustness of ILUC. This paper shows how to integrate two symmetry-preserving pivoting strategies, the diagonal pivoting and the Bunch-Kaufman pivoting, into ILUC without significantly overheads. The performances of the pivoting methods are compared with ILUC and ILUTP ([17]) on a set of problems, including a few arising from saddle-point (KKT) problems.

Key words. Incomplete LU factorization, ILU, ILUC, Sparse Gaussian Elimination, Crout factorization, Preconditioning, Diagonal pivoting, Bunch-Kaufman pivoting, ILU with threshold, Iterative methods, Sparse symmetric matrices.

AMS subject classifications. 65F10, 65N06.

1. Introduction.

Incomplete LU (ILU) factorization based preconditioners combined with Krylov subspace projection processes are often considered as the best "general purpose" iterative solvers available. Such ILU factorizations normally perform a Gaussian elimination algorithm while dropping some fill-ins. However, some variants of the Gaussian elimination algorithm may be more suitable than others for solving a given problem. For example, when dealing with sparse matrices, the traditional KIJ version [10, p 99] is impractical since all remaining rows are modified at each step k. In this case, another variant of Gaussian elimination namely the IKJ version, which implements a delayed-update version for the matrix elements, is preferred. Though the IKJ version of Gaussian elimination is widely used for implementing ILU factorizations (see, e.g. ILUT in SPARSKIT [17]), it has an inherent disadvantage: its requirement to access to the entries of a row of a matrix in a topological order during the factorization. Due to the fill-ins introduced during the factorization of a sparse matrix, a search is needed at each step in order to locate a pivot with the smallest index [18]. These searches often result in high computational costs, especially when the number of nonzero entries in A and/or the factors is large. A strategy to reduce the cost of these searches is to construct a binary tree for the current row and utilize binary searches [19]. Recently, a more efficient Crout version of ILU (termed ILUC) based on the Crout version of Gaussian elimination has been developed [14].

In the process of the Crout version of Gaussian elimination, the k-th column of L ($L_{k:n,k}$) and the k-th row of U ($U_{k,k:n}$) are calculated at the k-th step. Unlike the IKJ version, all elements that will be used to update $L_{k:n,k}$ and $U_{k,k:n}$ at the k-th step in the Crout version have already been calculated, i.e., the fill-ins will not interfere with the row updates at the k-th step. Therefore, searching for the next pivot in the Crout version is avoided. In ILUC, a bi-index data structure has been developed to address two implementation difficulties in sparse matrix operations (see Section 3 for details), following earlier work by Eisenstat et al. [9] in the context of the Yale Sparse Matrix Package (YSMP), and Jones and Plassmann [13]. Other than efficiency, another advantage of ILUC is that it also enables some more rigorous dropping strategies (e.g. [3, 2]), hence, improved robustness. However, there are still many situations where sparse linear systems are difficult to solve by iterative methods with the ILUC preconditioning, especially when the coefficient matrices are very ill-conditioned and/or highly indefinite. In such situations, pivoting techniques can be used to further improve robustness. Nevertheless, a pivoting method that is both efficient and effective must be carefully designed to fit within the data structure used by the incomplete factorization algorithm, which may not be a trivial task.

In this paper, symmetry-preserving pivoting strategies that are suitable for the data structure used in ILUC are explored and implemented for symmetric matrices. We begin our discussion with a review of related pivoting techniques in Section 2. We then discuss in detail ILUC with pivoting methods in Section 3. Finally, we compare the performances of the pivoting methods with ILUC and ILUTP ([17]) on some general symmetric matrices in Section 4 and some KKT matrices in Section 5.

[*]This work was supported in part by the Army Research Office under grant DA/DAAD19-00-1-0485, in part by the National Science Foundation under grant ACI-0305120, and by the Minnesota Supercomputer Institute.

[†]Department of Computer Science and Engineering, University of Minnesota, 200 Union Street S.E., Minneapolis, MN 55455. email: {nli,saad}@cs.umn.edu

2. Related Pivoting Techniques.

For many linear systems that arise from real applications, the ILU factorization may encounter either a zero or a small pivot. In case a zero pivot occurs, one strategy to avoid a break-down of the factorization process is to replace the zero pivot with a very small element, which leads to the second case as well. However, this remedy works only in very limited cases. Indeed, small pivots can cause an exponential growth of the entries in the factors and eventually, the ILU process will break down due to an overflow or underflow condition. Even when this does not happen, the factors L and U produced by the process are generally of poor quality and will not lead to convergence in the iteration phase.

A common solution to ensure a moderate growth in the factors is to use pivoting techniques. However, pivoting techniques are often in conflict with the underlying data structures of the factorizations. For example, for a symmetric matrix, a column partial pivoting method such as the ILUTP algorithm of SPARSKIT [17], will destroy symmetry. Pivoting methods that preserve symmetry are desirable. The elimination in ILUC is symmetric, i.e., at step k, the kth row of U and the kth column of L are computed. Moreover, the bi-index data structure used to implement ILUC is also symmetric. Thus, the symmetric elimination process and the symmetric data structure are ideal to incorporate symmetry-preserving pivoting methods into ILUC.

In this paper, we explore symmetry-preserving pivoting methods that can be integrated into the existing ILUC process without significant overheads. Our goal is to improve the robustness of ILUC for symmetric systems while preserving symmetry. For this reason, we use a revised version of ILUC (termed ILDUC) to compute the factorization $A = LDU$ instead of $A = LU$. In ILDUC, L and U^T are unit lower triangular matrices, and D is diagonal.

One simple pivoting method that preserves symmetry is to select the largest diagonal entry as the next pivot and interchange the corresponding row and column simultaneously. This method, referred to as 1×1 diagonal pivoting in the rest of the paper, works well for many symmetric matrices according to our tests. However, it fails for a matrix as simple as

$$A = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}.$$

In 1971, Bunch and Parlett proposed a pivoting method based on Kahan's generalized pivot to include 2×2 principal submatrices [4]. They also proved that the bound of this method, in terms of the growth of elements in the reduced matrices, is almost as good as that of the Gaussian elimination with complete pivoting. However, this method requires searching the entire reduced matrix at each step during the factorization, which makes it impractical for large sparse matrices. In 1977, Bunch and Kaufman proposed a partial pivoting method, now known as the Bunch-Kaufman pivoting method, where a 1×1 or 2×2 pivot can be determined by searching at most two columns of the reduced matrix at each step [6]. In 1998, Ashcraft et al. proposed two alternatives of the Bunch-Kaufman algorithm, providing better accuracy by bounding the triangular factors [1]. In 2000, Higham proved the stability of the Bunch-Kaufman pivoting method [12]. Because of the efficiency and effectiveness of the Bunch-Kaufman pivoting method, it has been used in LAPACK and LINPACK to solve dense symmetric indefinite linear systems.

In this paper, we show that the Bunch-Kaufman pivoting method can be integrated with ILDUC to solve *sparse* symmetric indefinite linear systems. For the sake of completeness, we provide some details of the Bunch-Kaufman pivoting algorithm. Let $A^{(k)}$ denote the matrix at step k. To decide the next pivot,

1. Determine
$$\lambda = \max_{k+1 \le i \le n} |A^{(k)}_{ik}|,$$
which is the largest off-diagonal element in absolute value in the k-th column. If $\lambda = 0$, then go to step $k+1$. Otherwise let r $(r > k)$ be the smallest integer such that $|A^{(k)}_{rk}| = \lambda$.
2. If $|A^{(k)}_{kk}| \ge \alpha\lambda$ where $\alpha = (1+\sqrt{17})/8$, then use $A^{(k)}_{kk}$ as a 1×1 pivot. Otherwise,
3. Determine
$$\sigma = \max_{\substack{k \le m \le n \\ m \ne r}} |A^{(k)}_{m,r}|.$$
4. If $\alpha\lambda^2 \le |A^{(k)}_{kk}|\sigma$, then use $A^{(k)}_{kk}$ as a 1×1 pivot.

5. Else if $|A^{(k)}_{rr}| \geq \alpha\sigma$, then interchange the k-th and the r-th rows and columns of $A^{(k)}$, use $A^{(k)}_{rr}$ as a 1×1 pivot.
6. Else interchange the $(k+1)$-th and the r-th rows and columns of $A^{(k)}$ so that $|A^{(k)}_{k+1,k}| = |A^{(k)}_{k,k+1}| = \lambda$, use $\begin{pmatrix} A^{(k)}_{kk} & \lambda \\ \lambda & A^{(k)}_{k+1,k+1} \end{pmatrix}$ as a 2×2 pivot.

In the next section, we present the methods that integrate the 1×1 diagonal pivoting and the Bunch-Kaufman pivoting algorithms into the existing ILDUC process without significant overhead.

3. ILDUC with Pivoting for Sparse Symmetric Matrices.

Algorithm 3.1 shows the ILDUC process to calculate the factorization $A = LDU$, modified from the ILUC algorithm proposed in [14].

ALGORITHM 3.1. *ILDUC - Crout version of Incomplete LDU factorization*

```
1.  $d_k = a_{kk},\ k = 1:n$
2.  For $k = 1:n$ Do :
3.      Initialize row $z$: $z_{1:k} = 0, \quad z_{k+1:n} = a_{k,k+1:n}$
4.      For $\{i \mid 1 \leq i \leq k-1$ and $l_{ki} \neq 0\}$ Do :
5.          $z_{k+1:n} = z_{k+1:n} - l_{ki} * d_i * u_{i,k+1:n}$
6.      EndDo
7.      Initialize column $w$: $w_{1:k} = 0, \quad w_{k+1:n} = a_{k+1:n,k}$
8.      For $\{i \mid 1 \leq i \leq k-1$ and $u_{ik} \neq 0\}$ Do :
9.          $w_{k+1:n} = w_{k+1:n} - u_{ik} * d_i * l_{k+1:n,i}$
10.     EndDo
11.     Apply a dropping rule to row $z$
12.     Apply a dropping rule to column $w$
13.     $u_{k,:} = z/d_k, \quad u_{kk} = 1$
14.     $l_{:,k} = w/d_k, \quad l_{kk} = 1$
15.     For $\{i \mid k+1 \leq i \leq n$ , $l_{ik} \neq 0$ and $u_{ki} \neq 0\}$ Do:
16.         $d_i = d_i - l_{ik} * d_k * u_{ki}$
17.     EndDo
18. EndDo
```

One property of the Crout version of LU is that only previously calculated elements are used to update the k-th column of L and the k-th row of U at step k. For sparse matrices, this means new fill-ins will not interfere with the updates of the k-th column of L and the k-th row of U. Therefore, the updates to the k-th row of U (resp., the k-th column of L) can be made in any order, i.e., the variable i can be chosen in any order in Line 4 (resp., in Line 8). This avoids the expensive searches in the standard ILUT. However, as pointed out in [14], two implementation difficulties regarding sparse matrix operations have to be addressed. The first difficulty lies in Lines 5 and 9 in Algorithm 3.1. At step k, only the section $(k+1:n)$ of the i-th row of U is needed to calculate the k-th row of U. Similarly, only the section $(k+1:n)$ of the i-th column of L is needed to calculate the k-th column of L. Accessing entire rows of U or columns of L and then extracting the desired part is an expensive option. The second difficulty lies in lines 4 and 8 in Algorithm 3.1. L is stored column-wise, but the nonzeros in the k-th row of L must be accessed. Similarly, U is stored row-wise, but the nonzeros in the k-th column of U must be accessed.

A carefully designed bi-index data structure was used in [14] to address these difficulties, inspired from early work by Eisenstat et al. [9] and Jones and Plassmann [13]. The bi-index data structure consists of four arrays of size n: $Lfirst$, $Llist$, $Ufirst$, and $Ulist$. At step k, $Lfirst(i)$, $1 \leq i \leq k-1$, points to the first entry with the row index greater than k in the i-th column of L. In this way, the section $L_{k+1:n,i}$ can be efficiently accessed, which addresses the first difficulty. It is worth pointing out that the elements in a column of L needs to be sorted by their indices to achieve this. However, in contrast with the searches needed in the standard ILUT, this sorting is much more efficient for two reasons. First, the sort is performed only after a large number of elements in a column are dropped, unlike in ILUT, where searching is performed on all of the elements. Second, a fast sorting algorithm such as Quick Sort can be easily applied, unlike in ILUT where some overhead in performing the search (e.g., when building the binary search trees). To address the second difficulty, the array $Llist$ is used to maintain linked lists of elements in the i-th row of L, where $i \geq k$. The linked lists are updated in such a way that the linked list of the elements in the k-th row of L

is guaranteed to be ready at step k. This linked list will be used to update the k-th row of U. $Ufirst$ and $Ulist$ are formed in a similarly way.

In the following, we discuss how we integrate pivoting methods to the bi-index data structure used in ILDUC for sparse symmetric matrices. One critical issue is that the new methods with pivoting should have a similar cost and complexity as that of ILDUC.

3.1. The 1×1 Diagonal Pivoting. For the 1×1 diagonal pivoting, we need to locate the largest diagonal element in absolute value at each step. For sparse matrices, a straightforward linear search would lead to a computational cost of $O(n^2)$, which is not acceptable especially when n is large. Alternatively, we build a binary heap for the diagonal elements in order to locate the largest diagonal entry efficiently. The binary heap is formed and maintained so that each node is greater than or equal to its children in absolute value. Therefore, the root node is always the largest diagonal entry in absolute value. Algorithm 3.2 summarizes ILDUC with 1×1 diagonal pivoting (termed ILDUC-DP). Note that for a symmetric matrix A the factorization is $PAP^T \approx LDL^T$, so only L needs to be calculated in the algorithm.

Algorithm 3.2. *ILDUC-DP - Incomplete LDL^T with 1×1 diagonal pivoting*

```
1.  $d_k = a_{kk},\ k = 1:n$
2.  Initialize a binary heap for $d_k,\ k = 1, \cdots, n$
3.  For $k = 1:n$ Do :
4.      Locate the largest diagonal $d_r$ in absolute value from the root node of the heap
5.      Interchange column $r$ and column $k$ if $r \neq k$
6.      Remove the root node from the heap and reorder the heap
7.      Initialize column $w$: $w_{1:k} = 0, \quad w_{k+1:n} = a_{k+1:n,k}$
8.      For $\{i \mid 1 \le i \le k-1$ and $l_{ki} \neq 0\}$ Do :
9.          $w_{k+1:n} = w_{k+1:n} - l_{ki} * d_i * l_{k+1:n,i}$
10.     EndDo
11.     Apply a dropping rule to column $w$
12.     $l_{:,k} = w/d_k, \quad l_{kk} = 1$
13.     For $\{i \mid k+1 \le i \le n\ ,\ l_{ik} \neq 0\}$ Do:
14.         $d_i = d_i - l_{ik} * d_k * l_{ik}$
15.         Reorder the heap for $d_i$
16.     EndDo
17. EndDo
```

In the above algorithm, the cost of initializing the binary heap is $O(n)$ (a bottom-up approach). The total cost of removing the root node and reordering the heap is $O(n \log n)$. The total cost of maintaining the heap is at most $O(mn \log n)$ (where $m << n$ is the dropping parameter defining the maximum number of fill-ins allowed in each column) since whenever a diagonal element is modified (line 15) the heap has to be reordered. Therefore, the total cost is bounded by $O(mn \log n)$.

3.2. The Bunch-Kaufman Diagonal Pivoting. In the Bunch-Kaufman pivoting method, to determine the next pivot at each step only requires searching for at most two columns in the reduced matrix. This is feasible for a sparse symmetric matrix as the number of nonzero entries in each column is very small. Algorithm 3.3 describes ILDUC with the Bunch-Kaufman pivoting (termed ILDUC-BKP). Since ILDUC uses a delayed-update strategy, notice that the two columns must be updated before proceeding with the search in the algorithm.

Algorithm 3.3. *ILDUC-BKP - Incomplete LDL^T with the Bunch-Kaufman pivoting method*

```
1.  For $k = 1:n$ Do :
2.      Load and update $a_{k+1:n,k}$. Let $\lambda = ||a_{k+1:n,k}||_\infty$ and $|a_{rk}| = \lambda$.
3.      If $|a_{kk}| \ge \alpha\lambda$ Then
4.          Let $s = 1$.
5.      Else
6.          Load and update $a_{k+1:n,r}$. Let $\sigma = \max_{m \neq r} |a_{mr}|$.
7.          If $|a_{kk}|\sigma \ge \alpha\lambda^2$ Then
8.              $s = 1$
9.          Else If $a_{rr} \ge \alpha\sigma$ Then
10.             $s = 1$; interchange the $k$-th and the $r$-th rows and columns.
```

```
11.             Else
12.                 $s = 2$; interchange the $(k+1)$-th and the $r$-th rows and columns.
13.             End If
14.         End If
15.         Perform ILDUC elimination process using the $s \times s$ pivot
16.     EndDo
```

3.3. Implementation. Similarly to ILUTP in SPARSKIT [17], ILDUC-DP and ILDUC-BKP use a permutation array along with a reverse permutation array to hold the new ordering of the variables. This strategy ensures that during the elimination the matrix elements are kept in their original labeling.

For ILDUC with pivoting, new strategies are needed to address the two implementation difficulties mentioned earlier in this section due to pivoting. First, we need to efficiently access $L_{k+1:n,i}$ for any $i < k$ to calculate the k-th column of L. In the implementation of ILUC, recall that an n-size array $Lfirst$ is maintained so that $Lfirst(i)$ always points to the first element with a row index greater than k in column i. This also requires that the elements in a column of L be stored in the order of increasing row indices. However, with pivoting enabled, at the time when the i-th column is calculated, the order of the elements is unknown as they may be repositioned later due to pivoting. Thus, this method will not work with pivoting. Our new strategy to handle this issue is as follows. Observe that the positions of elements in section $L_{i:k-1,i}$, $i < k$, will not be changed after step $k-1$. We use $Lfirst(i)$ to record the number of nonzero elements in section $L_{i:k-1,i}$. Since the nonzero elements in $L_{i:n,i}$ are stored compactly in a linear buffer, we always store the nonzero elements in section $L_{i:k-1,i}$ in the first $Lfirst(i)$ positions in the buffer. In this way, the nonzero elements in $L_{k:i,i}$ are continuously stored (although they may be in any order) starting from position $Lfirst(i)+1$ in the linear buffer, which can be efficiently accessed. After $L_{k:i,i}$ is used to calculate the k-th column of L, we need to ensure the above property for the next step. Specifically, we need to scan the linear buffer starting from position $Lfirst(i)+1$ to access the nonzero elements in $L_{k:i,i}$. If $L_{k,i}$ is zero, we do nothing. Otherwise, it can be located during the scan. We then swap $L_{k,i}$ with the elements stored at position $Lfirst(i)$ and let $Lfirst(i) := Lfirst(i)+1$. Thus, at step $k+1$, the nonzero elements in section $L_{k+1:i,i}$ are guaranteed to be stored continuously starting at position $Lfirst(i)$ as well.

Second, we need to access some rows of L but it is stored column-wise. In ILUC, recall that a n-size array $Llist$ is carefully designed and maintained such that all elements in the k-th row of L are guaranteed to form a linked list when needed at the k-th step. This linked list is embedded in $Llist$. However, when pivoting is allowed, the availability of only the k-th row of L is not enough. For any $j > k$, the j-th row may be interchanged with the k-th row at step k due to pivoting. Therefore, we need to maintain a linked list for each row of L with a row index greater than or equal to k.

We can also use strategies such as preordering and equilibration to further improve the stability of our methods. We apply the Reverse Cuthill-McKee (RCM) algorithm to preorder the matrices [7] in our implementation. A matrix is equilibrated if all its rows and columns have the same length in some norm. We use the method proposed by Bunch in [5] to equilibrate a symmetric matrix so that its rows/columns are normalized under the max-norm. It is worth pointing out that for sparse symmetric matrices the two difficulties in ILDUC exist in the equilibration method as well. We address them by using the same bi-index data structure in our implementation.

4. Experiments.

In this section, we compare the performances of ILDUC, ILDUC with 1×1 pivoting (ILDUC-DP), and ILDUC with the Bunch-Kaufman pivoting (ILDUC-BKP). The computational codes were written in C, and the experiments were conducted on a 1.7GHz Pentium 4 PC with 1GB of main memory. All codes were compiled with the -O3 optimization option.

We tested ILDUC, ILDUC-DP, and ILDUC-BKP on 11 symmetric indefinite matrices selected from the Davis collection [8]. Some generic information on these matrices is given in Table 4.1, where n is the matrix size and nnz is the total number of nonzeros in a full matrix. In the tests, artificial right-hand sides were generated, and GMRES(60) was used to solve the systems using a zero initial guess. The iterations were stopped when the residual norm was reduced by 8 orders of magnitude or when the maximum iteration count of 300 was reached. The dual criterion dropping strategy was used for all preconditioners, i.e., any element of column k whose magnitude was less than a tolerance $\tau * ||L_{k+1:n,k}||_1$ was dropped; and only "Lfil" largest elements were kept. The parameter "Lfil" was selected as a multiple of the ratio $\frac{nnz}{n}$, the average number of nonzero elements per column in the original matrix. We used a parameter τ to determine the dropping tolerance and a parameter γ to set the value of "Lfil": $Lfil = \gamma * \frac{nnz}{n}$. Nevertheless, even under the same control parameters, the number of fill-ins may be very different from method to method. To better compare the preconditioners, we used an indicator called a fill-factor, i.e., the value of $nnz(L+D+L^T)/nnz(A)$, to represent the sparsity of the ILU factors. A good method should yield convergence for a small fill-factor.

Matrix	n	nnz	Source
aug3dcqp	35543	128115	3D PDE
bloweya	30004	150009	Cahn-Hilliard problem
bratu3d	27792	173796	3D Bratu problem
dixmaanl	60000	299998	Dixon-Maany optimization example
mario001	38434	206156	Discretization
mario002	389874	2101242	Discretization
olesnik0	88263	744216	Straz pod Ralskem mine model
sit100	10262	61046	Straz pod Ralskem mine model
stokes64	12546	140034	Stokes equation
tuma1	22967	87760	Mine model
tuma2	12992	49365	Mine model

TABLE 4.1
Symmetric Indefinite Matrices from the Davis Collection

Table 4.2 shows the results of the three preconditioners on the 11 matrices, where the RCM ordering and equilibration were applied. To ensure the preconditioners were comparable, we fixed the dropping tolerance to $\tau = 0.001$ and artificially selected a fill-in parameter γ for each preconditioner such that the resulting fill-factors were similar for each matrix. For references, we also tested the linear systems with ILUTP under similar fill-in parameters. Since ILUTP does not take advantage of symmetry and the ILUTP code we used was written in FORTRAN, we only compared the convergence and ignore the execution time for ILUTP. In the table, the values in the "Fill-in" field are the fill-factors. The symbol "-" in the "Fill-in" field indicates that the preconditioner failed due to a zero pivot encountered during ILU. "ITS" denotes the number of iterations for GMRES(60) to convergence. A symbol "-" in the "ITS" field indicates that convergence was not obtained in 300 iterations. "Tm.(s)" denotes the total time in seconds used for each method (Preconditioning time +GMRES(60) time).

Matrix	ILDUC			ILDUC-DP			ILDUC-BKP			ILUTP	
Name	Fill-in	ITS	Tm.(s)	Fill-in	ITS	Tm.(s)	Fill-in	ITS	Tm.(s)	Fill-in	ITS
aug3dcqp	1.508	57	2.10	1.479	51	3.41	1.504	52	3.10	2.225	-
bloweya	-	-	-	1.156	20	0.95	1.001	4	0.15	1.134	7
bratu3d	-	-	-	1.553	152	8.57	1.571	75	4.10	1.623	41
dixmaanl	1.445	20	1.02	1.589	-	-	1.434	7	0.47	1.789	-
mario001	-	-	-	2.001	82	6.27	2.020	60	5.64	2.169	-
mario002	-	-	-	2.006	266	208.78	2.025	205	157.84	2.182	-
olesnik0	-	-	-	2.020	156	30.65	2.016	165	31.83	2.068	-
sit100	-	-	-	1.471	71	0.72	1.415	70	0.71	1.465	-
stokes64	2.080	144	1.52	2.078	124	2.22	2.079	143	3.12	2.174	-
tuma1	-	-	-	1.792	230	9.98	1.817	71	2.68	1.989	-
tuma2	-	-	-	1.453	-	-	1.424	254	3.49	1.443	-

TABLE 4.2
Performances on Symmetric Indefinite Matrices: $\tau = 0.001$

From the table, it is clear that ILUTP was not robust and ILDUC failed in most cases due to zero pivots encountered. ILDUC with the Bunch-Kaufman is better than ILDUC with diagonal pivoting in general. Next we offer a few additional comments on these performances.

1. *aug3dcqp, stokes64*: For these two matrices, with a similar amount of fill-ins allowed (e.g. 1.508nnz, 1.479nnz, and 1.504nnz for *aug3dcqp* respectively), all three ILDUC-based methods converged. The number of iterations required for convergence was similar for all methods (ILDUC-DP required slightly fewer iterations to converge for matrix *stokes64*). As expected, the cost of ILDUC with pivoting is of the same order as that of ILDUC. Even with larger fill-in factors, ILUTP did not converge.
2. *bloweya, bratu3d*: For these two matrices, ILDUC-DP, ILDUC-BKP and ILUTP converged, but ILDUC failed due to zero pivots encountered. Nevertheless, ILDUC-BKP and ILUTP had better performances than ILDUC-DP.
3. *mario001, mario002, tuma1*: For these matrices, ILDUC-DP and ILDUC-BKP both converged, but ILDUC and ILUTP failed. ILDUC-BKP had better performances than ILDUC-DP. With a similar or even less amount of fill-ins, ILDUC-BKP required much fewer iterations to converge. For example, for matrix *tuma1*, ILDUC-DP required 230 iterations to converge with a fill-factor of 1.7924, while ILDUC-BKP only required 71 iterations with a fill-factor of 1.817. Table 4.3 compares ILDUC-DP and ILDUC-BKP with various fill-in factors on matrix *tuma1*. From the table, we see that ILDUC-BKP is more efficient than ILDUC-DP. Figure 4.1 (a) and (b) compare the preconditioning cost and the total cost of the two methods respectively.

Preconditioner			GMRES(60)		Total
Method	Fill-in	Time(s)	ITS	Time(s)	Time(s)
ILDUC-DP	1.792	0.27	230	10.18	10.45
ILDUC-BKP	1.817	0.14	71	2.84	2.98
ILDUC-DP	2.101	0.31	118	5.37	5.68
ILDUC-BKP	2.110	0.16	56	2.38	2.54
ILDUC-DP	2.390	0.36	69	2.98	3.34
ILDUC-BKP	2.400	0.20	41	1.44	1.64
ILDUC-DP	2.670	0.40	52	2.26	2.66
ILDUC-BKP	2.676	0.23	36	1.19	1.42
ILDUC-DP	2.955	0.45	45	1.80	2.25
ILDUC-BKP	2.938	0.26	31	0.97	1.23

Table 4.3
Matrix tuma1. $\tau = 0.001$.

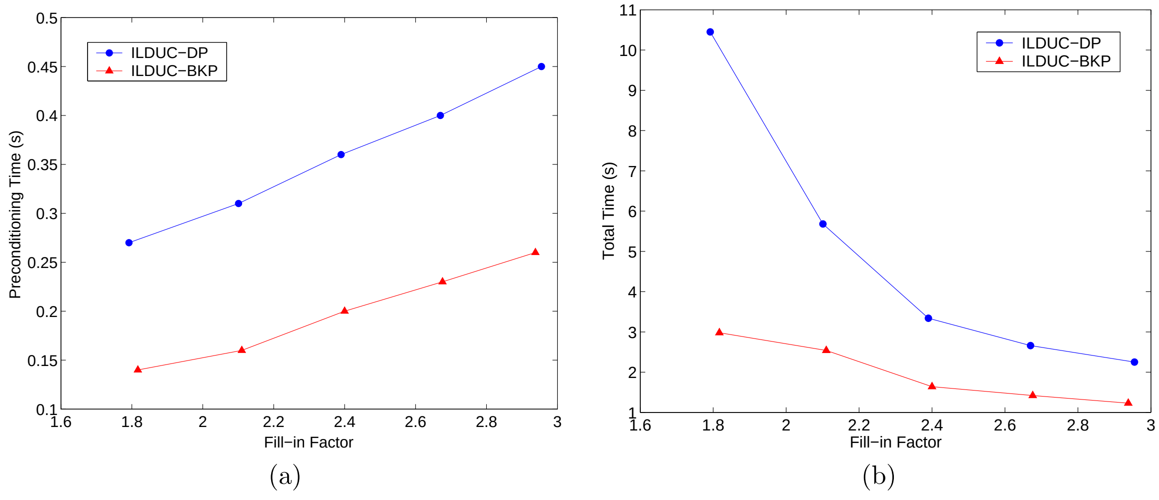

Fig. 4.1. *Comparison on the preconditioning time and the total time for matrix* tuma1

4. *olesnik0, sit100*: ILDUC and ILUTP failed on these two matrices, but ILDUC-DP and ILDUC-BKP converged and had similar performances.

5. *dixmaanl*: ILDUC-DP and ILUTP failed on this matrix. ILDUC-BKP had a slightly better performance than ILDUC.
6. *tuma2*: ILDUC-BKP was the only preconditioning method that converged for matrix *tuma2*.

5. Experiments with KKT Matrices. In this section, we test the preconditioners on 14 KKT matrices (in reference to the Karush-Kuhn-Tucker first-order necessary optimality conditions for the solution of general nonlinear programming problems [11]) as shown in Table 5.1. KKT matrices have the form:

$$A = \begin{bmatrix} H & B^{\mathrm{T}} \\ B & 0 \end{bmatrix}$$

and are often generated from equality and inequality constrained nonlinear programming, sparse optimal control, and mixed finite-element discretizations of partial differential equations [11]. The 14 KKT matrices were provided by Little [15] and Haws [11]. Matrices P2088 and P14480 were permuted from their original form [16] in order to obtain a more narrow bandwidth for the H part. The experiments were conducted on a 866MHz Pentium III PC with 1GB of main memory.

Matrix	n	nnz	Source
P2088	2088	15480	Magnetostatic problem (2D coarse discretization)
P14880	14880	113880	Magnetostatic problem (2D fine discretization)
CHOI	9225	168094	Particles in fluid simulator (five descending particles)
CHOI-L	22128	417156	Particles in fluid simulator (five descending particles)
LCAV-S1	14531	169972	Full Navier-Stokes equations in an L shaped cavity
OPT	9800	72660	Optimization problem
STIFF4	8496	41318	Stiffness problem
STIFF5	33410	177256	Stiffness problem
MASS04	8496	56818	Mass problem
MASS05	33410	241012	Mass problem
MASS06	33794	257220	Mass problem
TRAJ27	17148	235141	Sparse optimal control problem
TRAJ33	20006	496945	Sparse optimal control problem
LNTS09	17990	95295	Sparse optimal control problem

TABLE 5.1
KKT Matrices

Table 5.2 summaries the results of ILDUC, ILDUC-DP, ILDUC-BKP, and ILUTP on these matrices. Equilibration was applied. From the table, we make the following observations. First, ILDUC-BKP has the best overall performance on these matrices. ILUTP solved 6 problems, ILDUC solved 7 problems, ILDUC-DP solved 8 problems, and ILDUC-BKP solved 12 problems. Second, as expected, the cost of ILDUC-BKP is of the same order as that of ILDUC. This is evidenced in matrices CHOI-L, LCAV-S1, STIFF5, MASS04, MASS05 and MASS06, where the number of iterations for the two methods were identical or very close. Third, ILDUC-DP is the best method for matrices P2088 and P14880. Especially for P14880, ILDUC-DP is the only preconditioner that converged. However, for matrices such as CHOI-L, LCAV-S1, STIFF4 and MASS05, ILDUC-DP did not converge or required significantly more iterations to converge. Finally, although ILUTP had the worst performance on these matrices and it did not take advantage of symmetry, it was the only method that converged for matrix TRAJ33.

6. Conclusion. We have explored two symmetry-preserving pivoting methods and shown how to integrate them into a Crout version of the ILU factorization. As expected, this implementation results in better quality symmetric incomplete factorization for symmetric matrices. The overhead associated with this implementation is not significant. In addition, the pivoting methods have been demonstrated to fit the underlying data structure used in ILUC. Our experiments show that the Bunch-Kaufman pivoting method can be efficiently and effectively integrated with a sparse symmetric iterative solver.

7. Acknowledgements. We would like to thank Leigh Little, John Haws, Masha Sosonkina and Ilaria Perugia for providing many of the test matrices used in the numerical tests Section 4 and 5.

Matrix	ILDUC			ILDUC-DP			ILDUC-BKP			ILUTP	
Name	Fill-in	ITS	Tm.(s)	Fill-in	ITS	Tm.(s)	Fill-in	ITS	Tm.(s)	Fill-in	ITS
P2088	2.656	-	-	2.634	33	0.17	2.655	201	1.06	2.663	-
P14880	2.627	-	-	2.618	97	4.38	2.627	-	-	2.638	-
CHOI	-	-	-	2.511	41	1.84	2.402	20	1.12	2.334	20
CHOI-L	2.280	20	2.16	2.504	40	5.12	2.280	20	2.88	1.756	20
LCAV-S1	1.777	23	0.95	2.269	41	2.24	1.777	23	1.18	2.085	-
OPT	2.685	-	-	2.690	-	-	2.685	-	-	2.892	-
STIFF4	2.289	41	0.67	2.277	79	1.32	2.285	58	1.14	2.322	68
STIFF5	2.224	75	7.60	2.161	-	-	2.221	76	8.10	2.224	-
MASS04	2.196	5	0.12	2.188	13	0.26	2.193	5	0.15	2.181	10
MASS05	2.328	7	0.74	2.266	180	20.79	2.328	7	0.91	2.406	-
MASS06	2.292	11	1.11	2.254	-	-	2.289	11	1.31	2.409	-
TRAJ27	0.894	-	-	1.292	-	-	0.897	120	7.10	0.897	-
TRAJ33	0.885	-	-	1.075	-	-	0.891	-	-	0.841	96
LNTS09	-	-	-	-	-	-	0.849	15	0.48	0.967	55

TABLE 5.2
Performances on KKT Matrices: $\gamma = 2.5$, $\tau = 0.01$

REFERENCES

[1] C. Ahscraft, R. G. Grimes, and J. G. Lewis. Accurate Symmetric Indefinite Linear Equation Solvers. *SIAM Journal on Matrix Analysis and Applications*, 20(2):513–561, 1998.

[2] M. Bollhöfer and Y. Saad. A factored approximate inverse preconditioner with pivoting. *SIAM Journal on Matrix Analysis and Applications*, 23(3):692–705, 2001.

[3] M. Bollhöfer. A robust ILU with pivoting based on monitoring the growth of the inverse factors. *Linear Algebra and its Applications*, 338(1–3):201–218, 2001.

[4] J. R. Bunch and B. N. Parlett. Direct Methods for Solving Symmetric Indefinite Systems of Linear Equations. *SIAM Journal on Matrix Analysis and Applications*, 8:639–655, 1971.

[5] J. R. Bunch. Equilibration of Symmetric Matrices in the Max-Norm. *Journal of the Association for Computing Machinery*, 18(4): 566–572, 1971.

[6] J. R. Bunch and L. Kaufman. Some Stable Methods for Calculating Inertia and Solving Symmetric Linear Systems. *Mathematics of Computation*, 31(137):163–179, 1977.

[7] E. Cuthill and J. McKee. Reducing the bandwidth of sparse symmetric matrices. *In Proceedings of the 24th National Conference of the ACM*, ACM Publication P-69, Association for Computing Machinery, NY, 1969.

[8] T. Davis. University of Florida Sparse Matrix Collection, http://www.cise.ufl.edu/research/sparse/matrices. *NA Digest*, 92(42), October 16, 1994, *NA Digest*, 96(28), July 23, 1996, and *NA Digest*, 97(23), June 7, 1997.

[9] S. C. Eisenstat, M. H. Schultz, and A. H. Sherman. Algorithms and data structures for sparse symmetric Gaussian elimination. *SIAM Journal on Scientific Computing*, 2:225–237, 1981.

[10] G. H. Golub and C. F. Van Loan. *Matrix Computations.* The Johns Hopkins University Press, Baltimore, Maryland, third edition, 1996.

[11] J. C. Haws and C. D. Meyer. Preconditioning KKT systems. *Numerical Linear Algebra with Applications*, 00:1–6, 2001.

[12] N. J. Higham. Stability of the diagonal pivoting method with partial pivoting. *SIAM Journal on Matrix Analysis and Applications*, 18(1):52-65, 2000.

[13] M. Jones and P. Plassmann. An improved incomplete Cholesky factorization. *ACM Transactions on Mathematical Software*, 21:5–17, 1995.

[14] N. Li, Y. Saad, and E. Chow. Crout versions of ILU for general sparse matrices. *SIAM Journal on Scientific Computing*, 25(2):716–728, 2003.

[15] L. Little. KKT matrices for testing. Personal communication, 2003.

[16] I. Perugia, V. Simoncini, and M. Arioli Linear Algebra Methods in a Mixed Approximation of Magnetostatic Problems *SIAM Journal on Scientific Computing*, 21(3):1085–1101, 1999.

[17] Y. Saad. SPARSKIT: A basic tool kit for sparse matrix computations. Technical Report RIACS-90-20, Research Institute for Advanced Computer Science, NASA Ames Research Center, Moffet Field, CA, 1990.

[18] Y. Saad. ILUT: a dual threshold incomplete ILU factorization. *Numerical Linear Algebra with Applications*, 1:387–402, 1994.

[19] Y. Saad. *Iterative Methods for Sparse Linear Systems.* SIAM, Philadelphia, second edition, 2003.