SYM-ILDL: Incomplete LDL^T Factorization of Symmetric Indefinite and Skew-Symmetric Matrices

CHEN GREIF, SHIWEN HE, and PAUL LIU, University of British Columbia

SYM-ILDL is a numerical software package that computes incomplete LDL^T (ILDL) factorizations of symmetric indefinite and real skew-symmetric matrices. The core of the algorithm is a Crout variant of incomplete LU (ILU), originally introduced and implemented for symmetric matrices by Li and Saad [2005]. Our code is economical in terms of storage, and it deals with real skew-symmetric matrices as well as symmetric ones. The package is written in C++ and is templated, is open source, and includes a MATLAB^{TM} interface. The code includes built-in RCM and AMD reordering, two equilibration strategies, threshold Bunch-Kaufman pivoting, and rook pivoting, as well as a wrapper to MC64, a popular matching-based equilibration and reordering algorithm. We also include two built-in iterative solvers: SQMR, preconditioned with ILDL, and MINRES, preconditioned with a symmetric positive definite preconditioner based on the ILDL factorization.

CCS Concepts: • **Mathematics of computing → Solvers**; **Mathematical software performance**; **Computations on matrices**

Additional Key Words and Phrases: Symmetric indefinite, Skew-symmetric matrices

ACM Reference Format:
Chen Greif, Shiwen He, and Paul Liu. 2017. SYM-ILDL: Incomplete LDL^T factorization of symmetric indefinite and skew-symmetric matrices. ACM Trans. Math. Softw. 44, 1, Article 1 (April 2017), 21 pages.
DOI: http://dx.doi.org/10.1145/3054948

1. INTRODUCTION

For the numerical solution of symmetric and real skew-symmetric linear systems of the form

$$Ax = b,$$

stable (skew-)symmetry-preserving decompositions of A often have the form

$$PAP^T = LDL^T,$$

where L is a (possibly dense) lower triangular matrix and D is a block-diagonal matrix with 1-by-1 and 2-by-2 blocks [Bunch 1982; Bunch and Kaufman 1977]. The matrix P is a permutation matrix, satisfying $PP^T = I$, and the right-hand side vector b is permuted accordingly: in practice, we solve $(PAP^T)(Px) = Pb$.

In the context of incomplete LDL^T (ILDL) decompositions of sparse and large matrices for preconditioned iterative solvers, various element-dropping strategies are commonly used to impose sparsity of the factor, L. Fill-reducing reordering strategies are also used to encourage the sparsity of L, and various scaling methods are applied to improve conditioning. For a symmetric linear system, several methods have

Authors' addresses: C. Greif, S. He, and P. Liu, Department of Computer Science, The University of British Columbia, 2366 Main Mall, Vancouver BC V6T 1Z4, Canada; emails: greif@cs.ubc.ca, hswlmx@hotmail.com, paul.liu.ubc@gmail.com.

DOI: http://dx.doi.org/10.1145/3054948

been developed. Approaches have been proposed that perturb or partition A so that incomplete Cholesky may be used [Lin and Moré 1999; Orban 2014; Scott and Tuma 2014a]. Whereas Lin and Moré [1999] is designed for positive definite matrices, the works of Orban [2014] and Scott and Tuma [2014a] are applicable to a large set of 2×2 block structured indefinite systems.

We present SYM-ILDL—a software package based on a left-looking Crout version of LU, which is stabilized by pivoting strategies such as Bunch-Kaufman and rook pivoting. The algorithmic principles underlying our software are based on (and extend) an ILDL factorization approach proposed by Li and Saad [2005], which itself extends work by Li et al. [2003] and Jones and Plassmann [1995]. We offer the following new contributions:

—A Crout-based ILDL factorization for real skew-symmetric matrices is introduced in this article for the first time. It features a similar mechanism to the one for symmetric indefinite matrices, but there are notable differences. Most importantly, for real skew-symmetric matrices, the diagonal elements of D are zero and the pivots are always 2×2 blocks.
—We offer two integrated preconditioned solvers. The first solver is a preconditioned MINRES solver, specialized to our ILDL code. The main challenge here is to design a positive definite preconditioner, even though ILDL produces an indefinite (or skew-symmetric) factorization. To that end, for the symmetric case, we implement the technique presented in Gill et al. [1992]. For the skew-symmetric case, we introduce a positive definite preconditioner based on exploiting the simple 2×2 structure of the pivots. The second solver is a preconditioned SQMR solver based on the work of Freund and Nachtigal [1994]. For SQMR, we use ILDL to precondition it directly.
—The code is written in C++, and it is templated and easily extensible. As such, it can be easily modified to work in other fields of numbers, such as $\mathbb{C}$. SYM-ILDL is self-contained and includes implementations of reordering methods (AMD and RCM), equilibration methods (in the max-norm, 1-norm, and 2-norm), and pivoting methods (Bunch-Kaufman and rook pivoting). Additionally, we provide a wrapper that allows the user to use the popular `HSL_MC64` library to reorder and equilibrate the matrix. To facilitate ease of use, a MATLAB™ MEX file is provided that offers the same performance as the C++ version. The MEX file simply bundles the C++ library with the MATLAB™ interface (i.e., the main computations are still done in C++).

Incomplete factorizations of symmetric indefinite matrices have received much attention recently, and a few numerical packages have been developed in the past few years. Scott and Tuma [2014a] have developed a numerical software package based on signed incomplete Cholesky factorization preconditioners due to Lin and Moré [1999]. For saddle-point systems, Scott and Tuma [2014a] have extended their limited-memory incomplete Cholesky algorithm [Scott and Tuma 2014b] to a signed incomplete Cholesky factorization. Their approach builds on the ideas of Tismenetsky [1991] and Kaporin [1998]. In the case of breakdown (a zero pivot), a global shift is applied (see also Lin and Moré [1999]).

In Section 6.4 of their work, Scott and Tuma [2014a] have made comparisons with our code, finding that in general, the two codes are comparable in performance for several of the test problems; however, for some of the problems, each code outperforms the other. But the package they compared was an earlier release of SYM-ILDL. Given the numerous improvements made on the code since then, we repeat their comprehensive comparisons and show that SYM-ILDL now performs better than the preconditioner of Scott and Tuma [2014a].

Orban [2014] has developed `LLDL`, a generalization of the limited-memory Cholesky factorization of Lin and Moré [1999] to the symmetric indefinite case with special

interest in symmetric quasidefinite matrices. The code generates a factorization of the form LDL^T with D diagonal. We are currently engaged in a comparison of our code to LLDL.

The remainder of this article is structured as follows. In Section 2, we outline a Crout-based factorization for symmetric and skew-symmetric matrices, symmetry-preserving pivoting strategies, equilibration approaches, and reordering strategies. In Section 3, we discuss how to modify the output of SYM-ILDL to produce a positive definite preconditioner for MINRES. In Section 4, we discuss the implementation of SYM-ILDL and how the pivoting strategies of Section 2 may be efficiently implemented within SYM-ILDL's data structures. Finally, we compare SYM-ILDL to other software packages and show the performance of SYM-ILDL on some general (skew-)symmetric matrices and some saddle-point matrices in Section 5. In Section 6, we provide information on how to contribute to SYM-ILDL, and in the Appendix we provide full information on the numerical experiments.

2. LDL AND ILDL FACTORIZATIONS

SYM-ILDL uses a Crout variant of LU factorization. To maintain stability, SYM-ILDL allows the user to choose one of two symmetry-preserving pivoting strategies: Bunch-Kaufman partial pivoting [Bunch and Kaufman 1977] (Bunch in the skew-symmetric case [Bunch 1982]) and rook pivoting. The details of the factorization and pivoting procedures, as well as simplifications for the skew-symmetric case, are provided in the following sections. See also [Duff 2009] for more details on the use of direct solvers for solving skew-symmetric matrices.

2.1. Crout-Based Factorizations

The Crout order is an attractive way for computing an ILDL factorization of symmetric or skew-symmetric matrices, because it naturally preserves structural symmetry, especially when dropping rules for the incomplete factorization are applied. As opposed to the IKJ-based approach [Li and Saad 2005], Crout relies on computing and applying dropping rules to a column of L and a row of U simultaneously. The Crout procedure for a symmetric matrix is outlined later in Algorithm 2, using a delayed update procedure for the factors that is laid out in Algorithm 1. (As shown in Algorithm 2, the procedure in Algorithm 1 may be called multiple times when various pivoting procedures are employed.)

ALGORITHM 1: k-th Column Update Procedure

Input: A symmetric matrix A, partial factors L and D, matrix size n, current column index k
Output: Updated factors L and D

1 $L_{k:n,k} \leftarrow A_{k:n,k}$
2 $i \leftarrow 1$
3 **while** $i < k$ **do**
4 $\quad s_i \leftarrow$ size of the diagonal block with $D_{i,i}$ as its top left corner
5 $\quad L_{k:n,k} \leftarrow L_{k:n,k} - L_{k:n,i:i+s_i-1} D^{-1}_{i:i+s_i-1,i:i+s_i-1} L^T_{k,i:i+s_i-1}$
6 $\quad i \leftarrow i + s_i$
7 **end**

For computing the ILDL factorization, we apply dropping rules (see line 10 of Algorithm 2). These are the standard rules: we drop all entries below a prespecified tolerance (referred to as drop_tol throughout the article), multiplied by the norm of a column of L, keeping up to a prespecified maximum number of the largest nonzero entries in every column. Here we use the term fill_factor to signify the maximum

ALGORITHM 2: Crout Factorization, LDL$^{\mathrm{T}}$C

Input: A symmetric matrix A
Output: Matrices P, L, and D such that $PAP \approx LDL^T$

$k \leftarrow 1$
$L \leftarrow \mathbf{0}$
$D \leftarrow \mathbf{0}$
while $k < n$ **do**
  Call Algorithm 1 to update L
  Find a pivoting matrix in $A_{k:n,k:n}$ and permute A and L accordingly
  $s \leftarrow$ size of the pivoting matrix
  $D_{k:k+s-1,k:k+s-1} \leftarrow L_{k:k+s-1,k:k+s-1}$
  $L_{k:n,k:k+s-1} \leftarrow L_{k:n,k:k+s-1} D^{-1}_{k:k+s-1,k:k+s-1}$
  Apply dropping rules to $L_{k+s:n,k:k+s-1}$
  $k \leftarrow k + s$
end

allowed ratio between the number of nonzeros in any column of L and the average number of nonzeros per column of A.

In Algorithm 2, the $s \times s$ pivot is typically 1×1 or 2×2, as per the strategy devised by Bunch and Kaufman [1977], which we briefly describe next.

2.2. Symmetric Partial Pivoting

Pivoting in the symmetric or skew-symmetric setting is challenging, as we seek to preserve the (skew-)symmetry and it is not sufficient to use 1×1 pivots to maintain stability. Much work has been done in this front (e.g., see Duff et al. [1989, 1991] and Hogg and Scott [2014] and the references therein).

Bunch and Kaufman [1977] proposed a partial pivoting strategy for symmetric matrices, which relies on finding 1×1 and 2×2 pivots. The cost of finding a pivot is $\mathcal{O}(n)$, as it only involves searching up to two columns. We provide this procedure in Algorithm 3. For all pivoting algorithms that follow, we assume that we are pivoting on the Schur complement (i.e., column 1 is the k-th column if we are on the k-th step of Algorithm 2).

ALGORITHM 3: Bunch-Kaufman LDL$^{\mathrm{T}}$ Using the Partial Pivoting Strategy

$\alpha \leftarrow (1 + \sqrt{17})/8 \ (\approx 0.64)$
$\omega_1 \leftarrow$ maximum magnitude of any subdiagonal entry in column 1
if $|a_{11}| \geq \alpha \omega_1$ **then**
  Use a_{11} as a 1×1 pivot ($s = 1$)
else
  $r \leftarrow$ row index of first (subdiagonal) entry of maximum magnitude in column 1
  $\omega_r \leftarrow$ maximum magnitude of any off-diagonal entry in column r
  if $|a_{11}|\omega_r \geq \alpha \omega_1^2$ **then**
    Use a_{11} as a 1×1 pivot ($s = 1$)
  else if $|a_{rr}| \geq \alpha \omega_r$ **then**
    Use a_{rr} as a 1×1 pivot ($s = 1$, swap rows and columns 1, r)
  else
    Use $\begin{pmatrix} a_{11} & a_{r1} \\ a_{r1} & a_{rr} \end{pmatrix}$ as a 2×2 pivot ($s = 2$, swap rows and columns 2, r)
  end
end

The constant $\alpha = (1+\sqrt{17})/8$ in line 1 of the algorithm controls the growth factor, and a_{ij} is the ij-th entry of the matrix A after computing all delayed updates in Algorithm 1 on column i. Although the partial pivoting strategy is backward stable [Higham 2002], the possibly large elements in the unit lower triangular matrix L may cause numerical difficulty. Rook pivoting provides an alternative that in practice proves to be more stable, at a modest additional cost. This procedure is presented in Algorithm 4. The algorithm searches the pivots of the matrix in spiral order until it finds an element that is largest in absolute value in both its row and its column, or terminates if it finds a relatively large diagonal element. Although theoretically rook pivoting could traverse many columns, we found that it is as fast as Bunch-Kaufman in practice, and we use it as the default pivoting scheme of SYM-ILDL.

ALGORITHM 4: LDL^T Using the Rook Pivoting Strategy

1 $\alpha \leftarrow (1+\sqrt{17})/8\ (\approx 0.64)$
2 $\omega_1 \leftarrow$ maximum magnitude of any subdiagonal entry in column 1
3 **if** $|a_{11}| \geq \alpha\omega_1$ **then**
4 Use a_{11} as a 1×1 pivot ($s = 1$)
5 **else**
6 $i \leftarrow 1$
7 **while** *a pivot is not yet chosen* **do**
8 $r \leftarrow$ row index of first (subdiagonal) entry of maximum magnitude in column i
9 $\omega_r \leftarrow$ maximum magnitude of any off-diagonal entry in column r
10 **if** $|a_{rr}| \geq \alpha\omega_r$ **then**
11 Use a_{rr} as a 1×1 pivot ($s = 1$, swap rows and columns 1 and r)
12 **else if** $\omega_i = \omega_r$ **then**
13 Use $\begin{pmatrix} a_{ii} & a_{ri} \\ a_{ri} & a_{rr} \end{pmatrix}$ as a 2×2 pivot ($s = 2$, swap rows and columns 1 and i, and 2 and r)
14 **else**
15 $i \leftarrow r$
16 $\omega_i \leftarrow \omega_r$
17 **end**
18 **end**
19 **end**

2.3. Equilibration and Reordering Strategies

In many cases of practical interest, the input matrix is ill conditioned. For these cases, equilibration schemes have been shown to be effective in lowering the condition number of the matrix. Symmetric equilibration schemes rescale entries of the matrix by computing a diagonal matrix D such that DAD has equal row norms and column norms.

SYM-ILDL offers three equilibration schemes. Two of the equilibration schemes are built in: Bunch's equilibration in the max-norm [Bunch 1971] and Ruiz's iterative equilibration in any L_p-norm [Ruiz 2001]. Additionally, a wrapper is provided so that one can use MC64, a matching-based reordering and equilibration algorithm.

2.3.1. Bunch's Equilibration. Bunch's equilibration allows the user to scale the max-norm of every row and column to 1 before factorization. Let T be the lower triangular part of A in absolute value (diagonal included)—that is, $T_{ij} = |A_{ij}|,\ 1 \leq j \leq i \leq n$. Then Bunch's algorithm runs in $\mathcal{O}(\text{nnz}(A))$ time and is based on the following greedy procedure:

For $1 \leq i \leq n$, set

$$D_{ii} := \left(\max \left\{ \sqrt{T_{ii}}, \max_{1 \leq j \leq i-1} D_{jj} T_{ij} \right\} \right)^{-1}.$$

2.3.2. Ruiz's Equilibration. Ruiz's equilibration allows the user to scale every row and column of the matrix to 1 in any L_p norm, provided that $p \geq 1$ and the matrix has support [Ruiz 2001]. For the max-norm, Ruiz's algorithm scales each column's norm to within ε of 1 in $\mathcal{O}(\text{nnz}(A) \log \frac{1}{\varepsilon})$ time for any given tolerance ε. We use a variant of Ruiz's algorithm that is similar in spirit but produces different scaling factors.

Let $r(A, i)$ and $c(A, i)$ denote the i-th row and column of A, respectively, and let $D(i, \alpha)$ to be the diagonal matrix with $D_{jj} = 1$ for all $j \neq i$ and $D_{ii} = \alpha$. Using this notation, our variant of Ruiz's algorithm is shown in Algorithm 5.

ALGORITHM 5: Equilibrating General Matrices in the Max-Norm

Input: A general matrix A
Output: Diagonal matrices R and C such that RAC has max-norm 1 in every row and column

$R \leftarrow \mathbf{I}$
$C \leftarrow \mathbf{I}$
$\tilde{A} \leftarrow A$
while R **and** C *have not yet converged* **do**
  for $i := 1$ *to* n **do**
    $\alpha_r \leftarrow \frac{1}{\sqrt{||r(\tilde{A}, i)||_\infty}}$
    $\alpha_c \leftarrow \frac{1}{\sqrt{||c(\tilde{A}, i)||_\infty}}$
    $R \leftarrow R \cdot D(i, \alpha_r)$
    $C \leftarrow C \cdot D(i, \alpha_c)$
    $\tilde{A} \leftarrow D(i, \alpha_r) \tilde{A} D(i, \alpha_c)$
  end
end

Our presentation differs from Ruiz's original algorithm in that it operates on one row and column at a time as opposed to operating on the entire matrix in each iteration. We implemented the algorithm this way because it naturally adapts to our storage structures; our code is more easily amenable to single-column operations rather than matrix-vector products. Hence, Ruiz's original implementation and ours produce quite different scaling matrices. However, a proof of correctness similar to that of of Ruiz's algorithm applies, with the same guarantee for the running time.

2.3.3. Matching-Based Equilibration. The use of weighted matchings provides an effective technique to improve the stability of computing factorizations. In many cases, reorderings based on weighted matchings provided an effective form of static pivoting for tough indefinite symmetric problems [Hagemann and Schenk 2006]. Our code provides a wrapper to the well-known `HSL_MC64` software package, which implements a matching-based equilibration algorithm. When MC64 is installed, our code will use a symmetrized variant of it to generate a scaling matrix that scales the max-norm of every row and column to 1. More details regarding the functionality of MC64 can be found in Duff and Koster [2001] and Duff and Pralet [2005], as well as in the manual of the HSL Mathematical Software Library.

2.3.4. Comparison of Equilibration Strategies. Ruiz's strategy seems to perform well in terms of preserving diagonal dominance when no reordering strategy is used. In fact, we observed that for certain skew-symmetric systems, Ruiz's equilibration leads to convergence of the iterative solver, whereas Bunch's approach does not. However, Bunch's equilibration strategy is faster, being a one-pass procedure. When MC64 is available, its speed and scaling are comparable with Bunch's algorithm. However, there are some matrices in our test suite for which MC64 provides a suboptimal equilibration. In our experiments, we use Bunch's algorithm as the default.

2.3.5. Fill-Reducing Reorderings. After equilibration, we carry out a reordering strategy. The user is given the option of choosing from approximate minimum degree (AMD) [Amestoy et al. 1996], reverse Cuthill-McKee (RCM) [George and Liu 1981], and MC64. AMD and RCM are built into SYM-ILDL, but MC64 requires the installation of an external library. Whereas RCM and AMD are meant to reduce fill, MC64 computes a symmetric reordering of the matrix so that larger elements are placed near the diagonal. This has the effect of improving stability during the factorization but may increase fill. Although there are matrices in our tests for which MC64 reordering is effective, we found reducing fill to be more important, as our pivoting procedures deal with stability issues already. For the purpose of improving diagonal dominance while reducing fill, a common strategy is to combine MC64 with a fill-reducing reordering such as AMD or METIS [Schenk and Gärtner 2006; Hagemann and Schenk 2006]. We use the procedure described in Hagemann and Schenk [2006], which first preprocesses the matrix with MC64 and then compresses 2×2 pivots identified during the matching. The rows/columns corresponding to the pivots have their zero patterns merged and are replaced by a single row/column. Then AMD is run on the condensed matrix, after which the pivots are expanded back into two rows and columns. This procedure is implemented in `HSL_MC80`. Although MC80 is not built into our package, the results obtained by MC80 are comparable to those obtained by AMD and MC64. These results can be found in the appendix. For reducing fill, we found both MC80 and AMD to be effective for our test cases. As MC80 requires an external library, AMD is set as the default in the code.

2.4. LDL and ILDL Factorizations for Skew-Symmetric Matrices

The real skew-symmetric case is different from the symmetric indefinite case in the sense that here we must always use 2×2 pivots because diagonal elements of real skew-symmetric matrices are zero. This simplifies both the Bunch-Kaufman and Rook pivoting procedures: we have only one case in both scenarios. Algorithm 6 illustrates the simplification for rook pivoting (the simplification for Bunch is similar). Furthermore, as opposed to a typical 2×2 symmetric matrix, which is defined by three parameters, the analogous real skew-symmetric matrix is defined by one parameter only. As a result, at the k-th step, the computation of the multiplier and the subsequent update of pair of columns associated with the pivoting operation can be expressed as follows:

$$\begin{aligned} A_{k+2:n,k:k+1} A_{k:k+1,k:k+1}^{-1} &= A_{k+2:n,k:k+1} \begin{pmatrix} 0 & -a_{k+1,k} \\ a_{k+1,k} & 0 \end{pmatrix}^{-1} \\ &= \frac{1}{a_{k+1,k}} A_{k+2:n,k:k+1} \begin{pmatrix} 0 & 1 \\ -1 & 0 \end{pmatrix}, \end{aligned}$$

which can be trivially computed by swapping columns k and $k+1$ and scaling.

The ILDL factorization for skew-symmetric matrices can thus be carried out similarly to the manner in which it is developed for symmetric indefinite matrices, but the eventual algorithm gives rise to the simplifications described previously. Skew-symmetric

ALGORITHM 6: LDL$^{\mathrm{T}}$ Using the Rook Pivoting Strategy for Skew-Symmetric Matrices

$\omega_1 \leftarrow$ maximum magnitude of any subdiagonal entry in column 1
$i \leftarrow 1$
while *a pivot is not yet chosen* **do**
  $r \leftarrow$ row index of first (subdiagonal) entry of maximum magnitude in column i
  $\omega_r \leftarrow$ maximum magnitude of any off-diagonal entry in column r
  if $\omega_i = \omega_r$ **then**
    Use $\begin{pmatrix} 0 & -a_{ri} \\ a_{ri} & 0 \end{pmatrix}$ as a 2×2 pivot (swap rows and columns 1 and i, and 2 and r)
  else
    $i \leftarrow r$
    $\omega_i \leftarrow \omega_r$
  end
end

matrices are often ill conditioned, and we experimentally found that computing a numerical solution effectively for those systems is challenging. More details are provided in Section 5.

3. ITERATIVE SOLVERS FOR SYM-ILDL

In SYM-ILDL, we implement two preconditioned iterative solvers: SQMR and MINRES. For SQMR, we can use the ILDL factorization as a preconditioner directly. For MINRES, we require the preconditioner to be positive definite and modify our LDL$^{\mathrm{T}}$ as described in Section 3.

In our experiments, SQMR usually took fewer iterations to converge, with the same arithmetic cost per iteration. However, we found MINRES to be useful and more stable in difficult problems where SQMR stagnates (see Section 5). In these problems, MINRES returns a solution vector with a much smaller residual than SQMR in fewer iterations.

3.1. A Specialized Preconditioner for MINRES

In the following, we describe techniques for generating MINRES preconditioned iterations using positive definite versions of the incomplete factorization. For the symmetric indefinite case, we apply the method presented in Gill et al. [1992]. Given $M = LDL^T$, let us focus our attention on the various options for the blocks of D. Our ultimate goal is to modify D and L such that D is diagonal with only 1 or -1 as its diagonal entries. If a block of the matrix D from the original LDL$^{\mathrm{T}}$ factorization was 2×2, then the corresponding modified (diagonal) block would become

$$\begin{pmatrix} \pm 1 & 0 \\ 0 & \mp 1 \end{pmatrix}.$$

For a diagonal entry of D that appears as a 1×1 block, say, $d_{i,i}$, we rescale the i-th row of L: $L(i,:) \rightarrow L(i,:)\sqrt{|d_{i,i}|}$. We can then set the new value of $d_{i,i}$ as $\mathrm{sgn}(d_{i,i})$. In practice, there is no need to perform a multiplication of a row of L by $\sqrt{|d_{i,i}|}$; instead, this scalar is stored separately, and its multiplicative effect is computed as an $\mathcal{O}(1)$ operation for every matrix vector product.

Now consider a 2×2 block of D, say D_j. For this case, we compute the eigendecomposition

$$D_j = Q_j \Lambda_j Q_j^T,$$

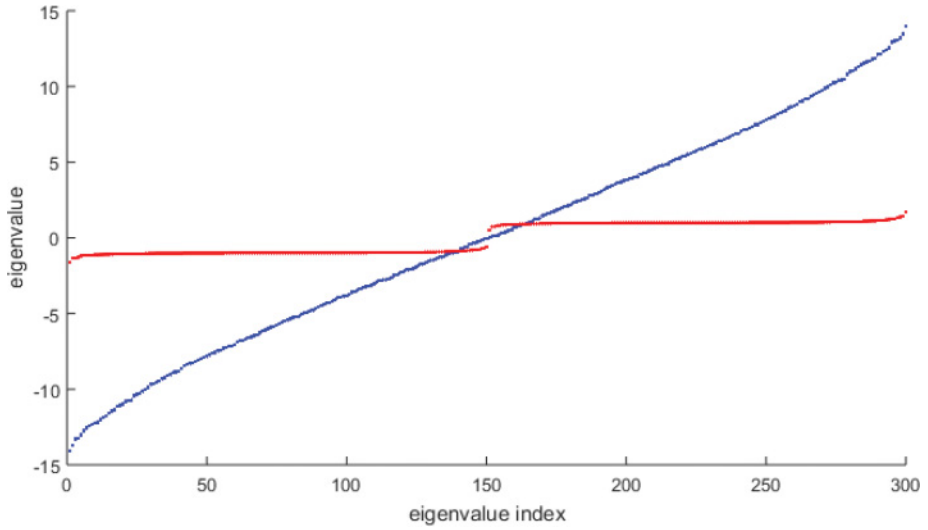

Fig. 1. Eigenvalues of a preconditioned symmetric random 300 × 300 matrix (in red). Note the clustering at 1 and –1. Entries in the matrix are drawn from the standard normal distribution. The unpreconditioned eigenvalues are in blue.

and similarly to the case of a 1×1 block, we implicitly rescale two rows of L by $Q_j\sqrt{|\Lambda_j|}$. This means that L is no longer triangular; it is in fact lower Hessenberg, as some values above the main diagonal may become nonzero. But the solve is just as straightforward, as the decomposition is explicitly given.

Since L was originally a unit lower triangular matrix that was scaled by positive scalars, LL^T is symmetric positive definite and we use it as a preconditioner for MINRES. Note that if we were to compute the full LDL^T decomposition and scale L as described earlier, then MINERS would converge within two iterations (in the absence of round-off errors), thanks to the two eigenvalues of D, namely 1 and -1.

In the skew-symmetric case, we may use a specialized version of MINRES [Greif and Varah 2009]. We only have 2×2 blocks, and for those we know that

$$\begin{pmatrix} 0 & a_{j,j} \\ -a_{j,j} & 0 \end{pmatrix} = \begin{pmatrix} \sqrt{|a_{j,j}|} & 0 \\ 0 & \sqrt{|a_{j,j}|} \end{pmatrix} \begin{pmatrix} 0 & \pm 1 \\ \mp 1 & 0 \end{pmatrix} \begin{pmatrix} \sqrt{|a_{j,j}|} & 0 \\ 0 & \sqrt{|a_{j,j}|} \end{pmatrix}.$$

Therefore, we do not need an eigendecomposition (as in the symmetric case), and instead we just scale the two affected rows of L by $\sqrt{|a_{j,j}|}I_2$.

Figure 1 shows the clustering effect that the proposed preconditioning approach has. We generate a random real symmetric 300×300 matrix A (with entries drawn from the standard normal distribution) and compute the eigenvalues of $(LDL^T)^{-1}A$, where L and D are the matrices generated in the preconditioning procedure described previously. Our fill factor is 2.0, and the drop tolerance was 10^{-4}. We note that the eigenvalue distribution in the figure is typical for other cases tested.

4. IMPLEMENTATION

4.1. Matrix Storage in SYM-ILDL

Since we are dealing with symmetric or skew-symmetric matrices, one of our goals is to avoid duplicating data. At the same time, it is necessary for SYM-ILDL to have both fast column access and fast row access. In terms of storage, we deal with these requirements by generating a format similar to standard compressed sparse column

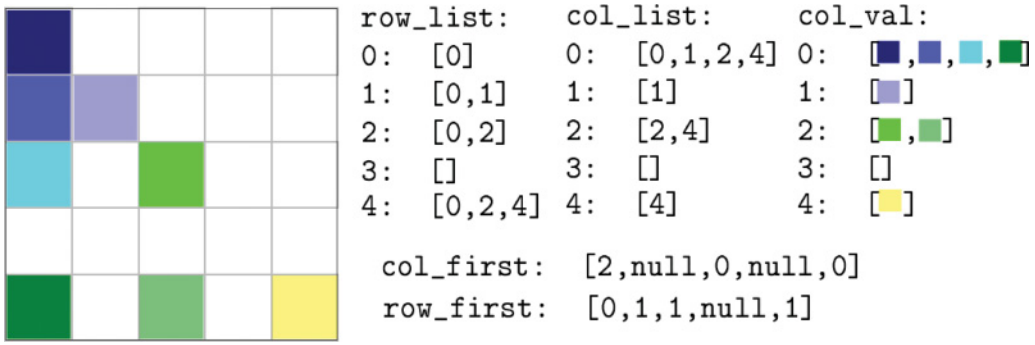

Fig. 2. Graphical representation of the data structures of SYM-ILDL. `col_first` and `row_first` are shown during the third iteration of the factorization. Hence, `col_first` holds the values of indices in `col_list` for the first element under or on the third row of the matrix. Similarly, `row_first` holds the values of indices in `row_list` for the last element not exceeding the third column of the matrix.

form, along with compressed sparse row form without the nonzero floating point matrix values. Matrices are stored in a *list-of-arrays* format. Each column is represented internally as two dynamically sized arrays, storing both its nonzero values `col_val` and row indices (`col_list`). These arrays facilitate fast random accesses and removals from the middle of the array (by simply swapping the value to be deleted to the end of the array and decrementing its size by 1). Meanwhile, another array holds pairs of pointers to the two column arrays of each column. One advantage of this format is that swapping columns and deallocating their memory is much easier, as we only need to operate on the array holding column pointers. Additionally, a row-major data structure (`row_list`) is used to maintain fast access across the nonzeros of each row (Figure 2). This is obtained by representing each row internally as a single array, storing the column indices of each row in an array (the nonzero values are already stored in the column-major representation).

Our format is an improvement over storing the full matrix in standard CSC, as used in Li and Saad [2005]. Assuming that the row and column indices are stored in 32-bit integers and the nonzero values are stored in 64-bit doubles, this gives us an overall 33% saving in storage if we were to store the factorization in-place. This is an easy modification of Algorithm 3. In the default implementation, we find it more useful to store an equilibrated and permuted copy of the original matrix so that we may use it for MINRES after the preconditioner is computed. An in-place version that returns only the preconditioner is included as part of our package.

4.2. Data Structures for Matrix Access

In ILUC [Li and Saad 2005], a bi-index data structure was developed to address two implementation difficulties in sparse matrix operations, following earlier work by Eisenstat et al. [1981] in the context of the Yale Sparse Matrix Package (YSMP) and Jones and Plassmann [1995]. Our implementation uses a similar bi-index data structure, which we briefly describe in the following.

Internally, the column and row indices in the matrix are stored in partial order, with one array per column and row. On the k-th iteration, elements are partially sorted so that all row indices less than k are stored in a contiguous segment per column, and all row indices greater or equal to k are stored in another contiguous segment. Within each segment, the elements are unsorted. This avoids the cost of sorting whenever we need to pivot. Since elements are partially sorted, accessing specific elements of the matrix is difficult and requires a slow linear search. Luckily, because Algorithm 3

Table I. Variable Names with Data Structure Types

Variable Name	Data Structure Type	Purpose
col_first	n length array	Speeds up access to $L_{k+1:n,i}$ (i.e., row_list)
row_first	n length array	Speeds up access to $A_{i,1:k}$ (i.e., col_list)
row_list	n dynamic arrays (row-major)	Stores indices of A across the rows
col_list	n dynamic arrays (col-major)	Stores indices of A across the columns
col_val	n dynamic arrays (col-major)	Stores nonzero coefficients of A

accesses elements in a predictable fashion, we can speed up access to subcolumns required during the factorization to $\mathcal{O}(1)$ amortized time. The strategy we use to speed up matrix access is similar to that of Jones and Plassmann [1995]. To ensure fast access to the submatrix $L_{k+1:n,1:k}$ and the row $L_{k,:}$ during factorization, we use one additional length n array: col_first. On the k-th iteration, the i-th element of col_first holds an offset that stores the dividing index between indices less than k and greater or equal to k. In effect, col_first gives us fast access to the start of the submatrix $L_{k+1:n,i}$ in col_list and speeds up Algorithm 1, allowing us access to the submatrix in $\mathcal{O}(1)$ time. To get fast access to the list of columns that contribute to the update of the $(k+1)$-st column, we use the row structure row_list discussed in Section 4.1. To speed up access to row_list, we maintain a row_first array that is implemented similarly to col_first. Overall, this reduces the access time of the submatrix $L_{k+1:n,1:k}$ and row L_k down to a cost proportional to the number of nonzeros in these submatrices.

Before the first iteration, col_first(i) is initialized to an array of all zeros. To ensure that col_first(i) stores the correct offset to the start of the subcolumn $L_{k+1:n,i}$ on step k, we increment the offset for col_first(i) (if needed) at the end of processing the k-th column. Since the column indices in col_list are unsorted, this step requires a linear search to find the smallest element in col_list. Once this element is found, we swap it to the correct spot so that the column indices for $L_{k+1:n,i}$ are in a contiguous segment of memory. We found it helpful to speed up the linear search by ensuring that the indices of A are sorted before beginning the factorization. This has the effect that A remains roughly sorted when there are few pivot steps.

Similarly, we will also need to access the subrows $A_{k,1:k}$ and $A_{r,1:k}$ during the pivoting stage (lines 11 through 15 in Algorithms 3 and 4). This is sped up by an analogous row_first(i) structure that stores offsets to the end of the subrow $A_{i,1:k}$ ($A_{i,1:k}$ is the memory region that encompasses everything from the start of memory for that row to row_first(i)). At the end of step k, we also increment the offsets for row_first if needed.

A summary of data structures can be found in Table I.

5. NUMERICAL EXPERIMENTS

All of our experiments were run single threaded on a machine with a 2.1GHz Intel Xeon CPU and 128GB of RAM. In our experiments, we follow the conventions of Li and Saad [2005] and Li et al. [2003], and we define the *fill* of a factorization as $\text{nnz}(L + D + L^T)/\text{nnz}(A)$. Recall that fill_factor is the maximum allowed ratio between the number of nonzeros in any column of L and the average number of nonzeros per column of A. Therefore, the fill of our preconditioner is bounded by approximately $2 \cdot$ fill_factor; the factor of 2 arises from the symmetry.

5.1. Results for Symmetric Matrices

5.1.1. Tests on General Symmetric Indefinite Matrices. For testing our code, we use the University of Florida (UF) collection [Davis and Hu 2011] as well as our own matrices. The UF collection provides a variety of symmetric matrices, which we specify in Tables II

Table II. Factorization Timings and SQMR Iterations for Test Matrices

Matrix	n	nnz(A)	Fill	Time (s)	Type	Iterations
aug3dcqp	35,543	128,115	1.9	0.05+0.15	ILDL(B+AMD)	24
			7.3	2.66+0.20	ILUTP(B+AMD)	6
bloweya	30,004	150,009	1.0	0.07+0.02	ILDL(MC64+MC64R)	3
			3.2	7.86+0.10	ILUTP(B+MC64R)	3
bratu3d	27,792	173,796	3.8	0.25+0.11	ILDL(B+MC64R)	18
			8.1	8.50+0.54	ILUTP(B+MC64R)	11
tuma1	22,967	87,760	3.0	0.05+0.13	ILDL(MC64+MC64R)	35
			7.8	2.68+0.58	ILUTP(B+AMD)	14
tuma2	12,992	49,365	3.0	0.03+0.09	ILDL(MC64+MC64R)	28
			6.9	0.72+0.23	ILUTP(B+AMD)	13
boyd1	93,279	1,211,231	1.0	0.10+0.50	ILDL(B+AMD)	3
			0.8	0.26+0.86	ILUTP(B+MC64R)	10
brainpc2	27,607	179,395	1.0	0.31+0.10	ILDL(MC64+MC64R)	31
			0.6	0.54+38.7	ILUTP(B+AMD)	NC
mario001	38,434	204,912	3.7	0.13+0.56	ILDL(B+MC64R)	52
			8.0	2.47+0.54	ILUTP(B+AMD)	8
qpband	20,000	45,000	1.1	0.008+0.004	ILDL(B+AMD)	1
			1.1	0.008+0.021	ILUTP(B+AMD)	1
nlpkkt80	1,062,400	28,192,672	8.0	153+53	ILDL(B+MC64R)	34
			4.1	6,803+2,502	ILUTP(B+AMD)	NC
nlpkkt120	3,542,400	95,117,792	8.0	525+334	ILDL(B+MC64R)	58
			—	—	ILUTP	—

The experiments were run with `fill_factor` $= 2.0$ for the smaller matrices and `fill_factor` $= 4.0$ for matrices larger than 1 million in dimension. The tolerance was `drop_tol` $= 10^{-4}$, and we used rook pivoting to maintain stability. The iteration was terminated when the norm of the relative residual went below 10^{-6}. The time is reported as $x + y$, where x is the preconditioning time and y is the iterative solver time. Times labeled with a dash (—) took more than 10 hours to run and were terminated before completion. Iteration counts labeled with NC indicate that the problem did not converge within 1,000 iterations.

Table III. Comparison of MATLAB's ILUTP and SYM-ILDL for Helmholtz Matrices

Matrix	n	nnz(A)	ILU Fill	ILU GEMRES Iterations	ILDL Fill	ILDL GMRES Iterations
$\alpha = 0.3/h^2$, Extra memory for ILUTP						
helmholtz80	6,400	31,680	7.7	11	7.6	8
helmholtz120	14,400	71,520	10.6	13	10.3	8
helmholtz160	25,600	127,360	12.3	17	12.3	8
helmholtz200	40,000	199,200	14.1	24	14.0	11
$\alpha = 0.7/h^2$, Extra memory for ILUTP						
helmholtz80	6,400	31,680	7.4	5	7.5	8
helmholtz120	14,400	71,520	13.4	10	14.0	18
helmholtz160	25,600	127,360	16.4	15	16.7	43
helmholtz200	40,000	199,200	19.5	20	20.8	86
$\alpha = 0.7/h^2$, Equal memory for ILDL and ILUTP						
helmholtz80	6,400	31,680	7.4	5	11.0	6
helmholtz120	14,400	71,520	13.4	10	18.6	6
helmholtz160	25,600	127,360	16.4	15	22.8	8
helmholtz200	40,000	199,200	19.5	20	33.0	11

The parameter α in Equation (1) is indicated above. GMRES was terminated when the relative residual decreased below 10^{-6}.

and IV. We used some of the same matrices used in the work of Li and Saad [2005], Li et al. [2003], and Scott and Tuma [2014a].

In Table II, we show the results of experiments with a set of matrices from Davis and Hu [2011] and comparisons with MATLAB's ILUTP. The matrix dimensions go up

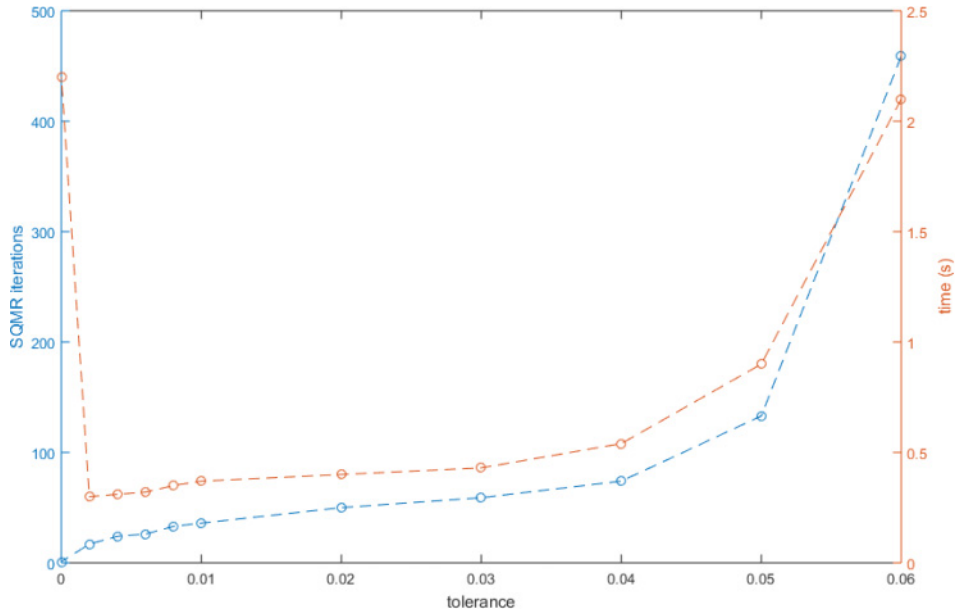

Fig. 3. Number of SQMR iterations and total precondition+solve time as a function of tolerance. Tests were performed on the `tuma1` matrix with Bunch equilibration, AMD reordering, and `fill_factor` set to ∞ (to measure only the tolerance dropping rule).

to approximately 4 million, with the number of nonzeros going up to approximately 100 million. We show timings for constructing the ILDL factorization and an iterative solution, applying preconditioned SQMR for SYM-ILDL and GMRES(20) for ILUTP with drop tolerance 10^{-3} for a maximum of 1,000 iterations. We apply either Bunch's equilibration or MC64 scaling and either AMD or MC64 reordering (MC64R) before generating the ILDL factorization. Preconditioned SQMR is run with SYM-ILDL for a maximum of 1,000 iterations. We show the best results in Table II out of the four possible reordering and equilibration combinations for both ILUTP and ILDL. We also tested the ILDL preconditioner with `HSL_MC80` reordering and equilibration and found it to be comparable with the best of the four preceding combinations. The full test data for all four combinations and tests with MC80 can be found later in Tables VI and VII in the appendix. For the incomplete factorization, we apply rook pivoting. We observe that ILDL achieves similar iteration counts with a far sparser preconditioner. Furthermore, even for cases where ILDL was beaten on iteration count, we see that the denser factor of ILUTP causes the overall solve time to be much slower than ILDL. When ILDL and ILUTP have similar fill, ILDL converges in fewer iterations.

In Figure 3, we examine the sensitivity of ILDL to input tolerance. We plot the number of iterations and the timings for a changing value of the tolerance. We observe the expected behavior. As the tolerance decreases, there is a trade-off between preconditioning time and iteration count. Thus, the total computational time is high at both extremes. That said, there is a large range of values of tolerance for which both time and iterations are modest. Altogether, ILDL works well for all test cases with fairly generic parameters.

5.1.2. Further Comparisons with MATLAB*'s ILUTP.* To show the memory efficiency of our code, we consider matrices associated with the discrete Helmholtz equation,

$$-\Delta u - \alpha u = f, \tag{1}$$

Table IV. GMRES Comparisons Between SYM-ILDL and HSL_MI30

Matrix			MI30			SYM-ILDL		
Name	n	nnz(A)	$\text{Fill}_{\text{MI30}}$	Iterations	Time (s)	$\text{Fill}_{\text{SYM-ILDL}}$	Iterations	Time (s)
c-55	32,780	403,450	3.45	49	1.25+0.94	2.95	15	0.23+0.15
c-59	41,282	480,536	3.62	70	1.59+1.84	2.99	15	0.36+0.20
c-63	44,234	434,704	4.10	51	1.53+1.23	2.92	15	0.29+0.21
c-68	64,810	565,996	4.12	37	1.87+1.12	2.31	9	0.31+0.17
c-69	67,458	623,914	4.33	43	4.07+1.47	2.65	9	0.35+0.18
c-70	68,924	658,986	4.26	38	3.77+1.30	2.67	11	0.40+0.24
c-71	76,638	859,520	3.58	61	3.93+2.71	3.00	12	0.74+0.32
c-72	84,064	707,546	4.18	54	3.05+2.40	2.69	9	0.40+0.31
c-big	345,241	2,340,859	4.82	67	23.4+25.3	2.54	8	1.20+0.93

For each test case, we report the time it takes to compute the preconditioner, as well as the GMRES time and the number of GMRES iterations. The time is reported as $x + y$, where x is the preconditioning time and y is the GMRES time. GMRES was terminated when the relative residual decreased below 10^{-6}.

subject to Dirichlet boundary conditions on the unit square, discretized using a uniform mesh of size h. Here we choose a moderate value of α so that a symmetric indefinite matrix is generated. The choice of α may have a significant impact on the conditioning of the matrix. In particular, if α is an eigenvalue, then the shifted matrix is singular. In SYM-ILDL, a singular matrix will trigger static pivoting and may add a significant computational overhead. In our numerical experiments, we stayed away from choices of α such that the shifted matrix is singular. Although we could have used the same matrices as in Table II, additional tests using Helmholtz matrices provide a greater degree of insight since we know its spectra and can easily control the dimension and number of nonzeros.

In Table III, we present results for the Helmholtz model problem. We compare SYM-ILDL to MATLAB's ILUTP. For ILUTP, we used a drop tolerance of 10^{-3} in all test cases. For ILDL, `fill_factor` was set to ∞ (since ILUTP does not limit its intermediate memory by a fill factor), and the `drop_tol` parameter was then chosen to get roughly the same fill as that of ILUTP. In the context of the ILUTP preconditioner, the *fill* is defined as $\text{nnz}(L + U)/\text{nnz}(A)$.

For both ILDL and ILUTP, GMRES(100) was used as the iterative solver and the input matrix was scaled with Bunch equilibration and reordered with AMD. During the computation of the preconditioner, the in-place version of ILDL uses only about 2/3 of the memory used by ILUTP. During the GMRES solve, the ILDL preconditioner only uses about 1/2 of the memory used by ILUTP. We note that ILDL could also be used with SQMR, which has a much smaller memory footprint than GMRES.

We observe that the performance of ILDL on the Helmholtz model problem is dependent on the value of α chosen, but that if ILDL is given the same memory resources as ILUTP, ILDL outperforms ILUTP. The memory usage of ILUTP and ILDL are measured through the MATLAB profiler. For $\alpha = 0.3/h^2$, the ILDL approach leads to lower iteration counts even when approximately 1/2 of the memory is allocated (i.e., when the same fill is allowed), whereas for $\alpha = 0.7/h^2$, ILUTP outperforms ILDL when the fill is roughly the same. If we allow ILDL to have memory usage as large as ILUTP (i.e., up to roughly 3/2 the fill), we see that ILDL clearly has lower iteration counts for GMRES.

5.1.3. Comparisons with `HSL_MI30`. In Table IV, we compare our code to that of Scott and Tuma [2014a], implemented in the package `HSL_MI30`. This comparison was already done in Scott and Tuma [2014a] with an older version of SYM-ILDL. However, with recent improvements, we see that SYM-ILDL generally takes two to six times fewer iterations than `HSL_MI30`. The matrices with which we compare are a subset of the

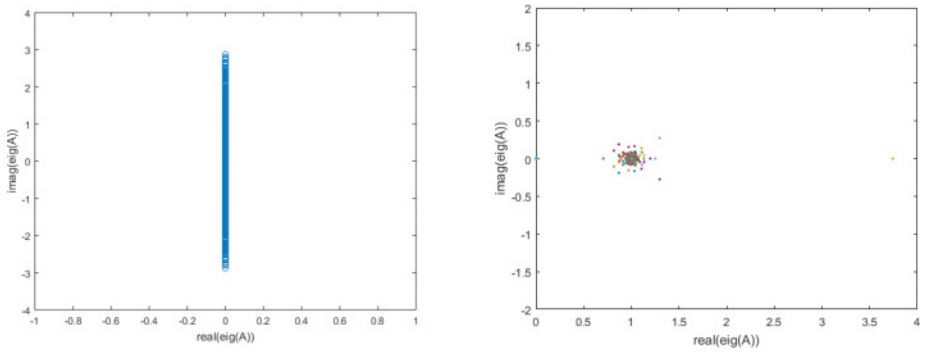

Fig. 4. Eigenvalues of an unpreconditioned (left) and preconditioned (right) skew-symmetric 1,000 × 1,000 matrix A arising from a convection-diffusion model problem.

Table V. Comparison of MATLAB's ILUTP and SYM-ILDL for a Skew-Symmetric Matrix Arising from a Model Convection-Diffusion Equation

n	nnz(A)	Method	Drop Tolerance	Fill	GMRES(20)	Time (s)
$20^3 = 8{,}000$	45,600	ILDL-rook	$4 \cdot 10^{-4}$	7.008	6	0.130+0.041
		ILDL-partial	$5 \cdot 10^{-4}$	6.861	6	0.138+0.041
		ILUTP	10^{-3}	7.758	8	0.406+0.038
$30^3 = 27{,}000$	156,600	ILDL-rook	$2 \cdot 10^{-4}$	10.973	8	0.936+0.246
		ILDL-partial	$3 \cdot 10^{-4}$	11.235	10	1.162+0.331
		ILUTP	10^{-3}	11.758	13	4.475+0.307
$40^3 = 64{,}000$	374,400	ILDL-rook	$9 \cdot 10^{-5}$	15.205	9	3.820+0.855
		ILDL-partial	$3 \cdot 10^{-4}$	15.686	18	4.971+1.582
		ILUTP	10^{-3}	15.654	19	26.63+1.40
$50^3 = 125{,}000$	735,000	ILDL-rook	$2 \cdot 10^{-5}$	21.560	6	15.39+1.76
		ILDL-partial	$2 \cdot 10^{-4}$	22.028	62	21.11+17.95
		ILUTP	10^{-3}	22.691	58	151.14+11.60
$60^3 = 216{,}000$	1,274,400	ILDL-rook	$2 \cdot 10^{-5}$	22.595	9	34.82+4.02
		ILDL-partial	$4 \cdot 10^{-4}$	22.899	NC	36.17+NC
		ILUTP	10^{-3}	23.483	NC	356.60+NC
$70^3 = 343{,}000$	2,028,600	ILDL-rook	$5 \cdot 10^{-6}$	32.963	5	106.81+3.25
		ILDL-partial	$4 \cdot 10^{-4}$	36.959	NC	156.52+NC
		ILUTP	10^{-3}	33.861	NC	876.33+NC

The parameters used were $\beta = 20,\ \gamma = 2,\ \delta = 1$. The MATLAB ILUTP used a drop tolerance of 0.001. NC stands for no convergence.

matrices used in the original comparison. In particular, these matrices were ones for which SYM-ILDL performed the most poorly in the original comparison. The matrices were obtained from the UF matrix collection [Davis and Hu 2011].

The parameters used here are almost the same as in the original comparison. For `HSL_MI30`, we used the built-in MATLAB interface, set `lsize` and `rsize` to 30, $\alpha(1:2)$ both to 0.1, the drop tolerances τ_1 and τ_2 to 10^{-3} and 10^{-4}, and used the built-in `MC77` equilibration (which performed the best out of all possible equilibration options, including MC64). We also tried all possible reordering options for `HSL_MI30` and found that the natural ordering performed the best. For SYM-ILDL, we used `fill_factor` of 12.0, `drop_tol` of 0.003, as in the original comparison. The only difference between the original comparison and this one is that rook pivoting is used for stability and MC80 is used for equilibration and reordering. We also performed additional tests using MC64 for equilibration and AMD for reordering and found a comparable number of iterations with higher fill. All tests can be found in Table VIII of the appendix.

We note that although we set `fill_factor` to be 12.0 in all comparisons with `HSL_MI30`, SYM-ILDL can have similar performance with a much smaller `fill_factor`.

5.2. Results for Skew-Symmetric Matrices

We test with a skew-symmetrized version of a model convection-diffusion equation, which is a discrete version of

$$-\Delta u + (\sigma, \tau, \mu)\nabla u = f, \tag{2}$$

with Dirichlet boundary conditions on the unit square, discretized using a uniform mesh of size h. We define the mesh Péclet numbers

$$\beta = \sigma h/2, \quad \gamma = \tau h/2, \quad \delta = \mu h/2.$$

We use the skew-symmetric part of this matrix (i.e., given A, form $\frac{A-A^T}{2}$) for our skew-symmetric experiments.

In our tests, we found that equilibration has not been particularly effective. We speculate that this might have to do with a property related to block diagonal dominance that these matrices have for certain values of the convective coefficients. Specifically, the norm of the tridiagonal part of the matrix is significantly larger than the norm of the remaining part. Equilibration tends to adversely affect this property by scaling down entries near the diagonal; as a result, the performance of an iterative solver often degrades. We thus do not apply equilibration in our skew-symmetric solver.

In Table V, we manipulate the drop tolerance for ILDL to obtain a fill nearly equal to that of ILUTP. For the latter, we fix the drop tolerance at 0.001. This is done for the purpose of comparing the performance of the iterative solvers when the memory requirements of ILUTP and ILDL are similar. Prior to preconditioning, we apply AMD as a fill-reducing reordering. We apply preconditioned GMRES(100) to solve the linear system until either a residual of 10^{-6} is reached or until 1,000 iterations are used. If the linear system fails to converge after 1,000 iterations, we mark it as NC. We see that the iteration counts are significantly better for ILDL, especially when rook pivoting is used. Note that our ILDL still consumes only about 2/3 of the memory of ILUTP due to the fact that the floating point entries of only half of the matrix are stored.

In Figure 4, we show the (complex) eigenvalues of the preconditioned matrix $(LDL^T)^{-1}A$, where A is the skew-symmetric part of 2 with convective coefficients $(\beta, \gamma, \delta) = (0.4, 0.5, 0.6)$, and LDL^T is the preconditioner generated by running SYM-ILDL with a drop tolerance of 10^{-3} and a fill-in parameter of 20.

For the purpose of comparison, we also show the unpreconditioned eigenvalues. As seen in the figure, most of the preconditioned eigenvalues are very strongly clustered around 1, which indicates that a preconditioned iterative solver is expected to rapidly converge. The unpreconditioned eigenvalues are pure imaginary and follow the formula

$$2\,(\beta \cos(j\pi h) + \gamma \cos(k\pi h) + \delta \cos(\ell\pi h))\,\imath,$$

where $1 \le j, k, \ell \le 1/h$.

6. OBTAINING AND CONTRIBUTING TO SYM-ILDL

SYM-ILDL is open source, and documentation can be found at `http://www.cs.ubc.ca/~greif/code/sym-ildl.html`. We essentially allow free use of our software with no restrictions. To this end, SYM-ILDL uses the MIT Software License.

We welcome any contributions to our code. Details on the contribution process can be found using the preceding URL. Certainly, more code optimization is possible, such as parallelization; such tasks remain as items for future work.

APPENDIX

The following table uses HSL_MC80 on matrices from Tables I and III in this article. For MC80, we chose AMD as the fill-reducing reordering after the matching stage. Only matrices from these two tables were used, as all other matrices in our tests were well scaled and block-diagonally dominant to begin with (e.g., the Helmholtz problem of Table II).

Table VI. Factorization Timings and Iterative Solver Iterations for Test Matrices

Matrix	n	nnz(A)	Fill	Time (s)	Type	Iterations
			1.9	0.051+0.148	ILDL(B+AMD)	24
			3.3	0.063+0.442	ILDL(MC64+MC64R)	55
aug3dcqp	35, 543	128, 115	2.1	0.068+0.261	ILDL(MC64+AMD)	33
			3.2	0.063+0.223	ILDL(B+MC64R)	33
			7.3	2.655+0.198	ILUTP(B+AMD)	6
			21.2	11.674+0.890	ILUTP(MC64+MC64R)	14
			36.0	11.513+0.397	ILUTP(MC64+AMD)	6
			7.4	1.753+0.215	ILUTP(B+MC64R)	7
			0.9	0.030+0.081	ILDL(B+AMD)	18
			1.0	0.071+0.014	ILDL(MC64+MC64R)	3
bloweya	30, 004	150, 009	1.0	0.023+0.019	ILDL(MC64+AMD)	5
			0.9	0.152+0.126	ILDL(B+MC64R)	18
			2.8	38.817+0.101	ILUTP(B+AMD)	4
			6.1	2.726+0.109	ILUTP(MC64+MC64R)	4
			2.9	39.537+0.104	ILUTP(MC64+AMD)	4
			3.2	7.858+0.100	ILUTP(B+MC64R)	3
			3.8	0.358+0.155	ILDL(B+AMD)	23
			3.6	0.155+0.124	ILDL(MC64+MC64R)	24
bratu3d	27, 792	173, 796	3.6	0.231+0.272	ILDL(MC64+AMD)	36
			3.8	0.245+0.105	ILDL(B+MC64R)	18
			8.6	22.237+0.214	ILUTP(B+AMD)	9
			10.3	13.114+0.962	ILUTP(MC64+MC64R)	18
			9.8	32.717+0.500	ILUTP(MC64+AMD)	10
			8.1	8.480+0.540	ILUTP(B+MC64R)	11
			2.9	0.044+0.201	ILDL(B+AMD)	50
			3.0	0.051+0.132	ILDL(MC64+MC64R)	35
tuma1	22, 967	87, 760	3.0	0.077+0.299	ILDL(MC64+AMD)	54
			3.0	0.046+0.220	ILDL(B+MC64R)	59
			7.8	2.686+0.582	ILUTP(B+AMD)	14
			40.0	19.476+0.495	ILUTP(MC64+MC64R)	8
			20.7	7.268+0.242	ILUTP(MC64+AMD)	6
			17.7	7.750+51.991	ILUTP(B+MC64R)	NC
			2.8	0.023+0.084	ILDL(B+AMD)	41
			3.0	0.029+0.087	ILDL(MC64+MC64R)	28
tuma2	12, 992	49, 365	3.0	0.045+0.104	ILDL(MC64+AMD)	34
			3.0	0.041+0.218	ILDL(B+MC64R)	55
			6.9	0.720+0.226	ILUTP(B+AMD)	13
			33.8	4.140+0.192	ILUTP(MC64+MC64R)	7
			19.0	1.936+0.106	ILUTP(MC64+AMD)	5
			15.5	2.082+12.341	ILUTP(B+MC64R)	697

(Continued)

Table VI. (Continued)

Matrix	n	nnz(A)	Fill	Time (s)	Type	Iterations
			1.0	0.155+0.077	ILDL(B+AMD)	3
			0.6	0.102+0.505	ILDL(MC64+MC64R)	42
boyd1	93, 279	1, 211, 231	1.0	0.123+0.088	ILDL(MC64+AMD)	6
			0.6	0.144+0.437	ILDL(B+MC64R)	36
			0.8	0.219+1.021	ILUTP(B+AMD)	10
			0.8	0.257+0.875	ILUTP(MC64+MC64R)	12
			0.8	0.233+1.656	ILUTP(MC64+AMD)	14
			0.8	0.188+0.481	ILUTP(B+MC64R)	10
			1.0	0.878+0.094	ILDL(B+AMD)	31
			1.8	0.315+0.100	ILDL(MC64+MC64R)	31
brainpc2	27, 607	179, 395	1.5	1.661+0.085	ILDL(MC64+AMD)	28
			1.8	0.481+0.983	ILDL(B+MC64R)	214
			0.6	0.541+38.711	ILUTP(B+AMD)	NC
			961.5	373.210+1210.140	ILUTP(MC64+MC64R)	NC
			88.7	15.434+180.070	ILUTP(MC64+AMD)	NC
			0.6	0.925+38.263	ILUTP(B+MC64R)	NC
			3.7	0.163+0.541	ILDL(B+AMD)	54
			3.6	0.234+0.629	ILDL(MC64+MC64R)	55
mario001	38, 434	204, 912	3.6	0.213+0.603	ILDL(MC64+AMD)	54
			3.7	0.129+0.557	ILDL(B+MC64R)	52
			8.0	2.474+0.542	ILUTP(B+AMD)	8
			9.3	26.39+0.612	ILUTP(MC64+MC64R)	8
			9.0	2.552+0.555	ILUTP(MC64R+AMD)	8
			8.6	21.73+0.325	ILUTP(B+MC64)	9
			1.1	0.008+0.004	ILDL(B+AMD)	1
			1.1	0.007+0.004	ILDL(MC64+MC64R)	1
qpband	20, 000	45, 000	1.8	0.014+0.004	ILDL(MC64+AMD)	1
			1.1	0.016+0.004	ILDL(B+MC64R)	1
			1.1	0.008+0.026	ILUTP(B+AMD)	1
			1.1	0.008+0.021	ILUTP(MC64+MC64R)	1
			1.2	0.011+0.028	ILUTP(MC64+AMD)	1
			1.1	0.010+0.013	ILUTP(B+MC64R)	1
			9.5	113+1308	ILDL(B+AMD)	998*
			14.5	176+1580	ILDL(MC64+MC64R)	854*
nlpkkt80	1, 062, 400	28, 192, 672	12.3	153+53	ILDL(MC64+AMD)	34
			10.6	121+NC	ILDL(B+MC64R)	NC
			4.1	6,803+2,502	ILUTP(B+AMD)	NC
			—	—	ILUTP(MC64+MC64R)	—
			—	—	ILUTP(MC64+AMD)	—
			—	—	ILUTP(B+MC64)	—
			9.8	401+NC	ILDL(B+AMD)	NC
			14.5	533+NC	ILDL(MC64+MC64R)	NC
nlpkkt120	3, 542, 400	95, 117, 792	12.4	525+334	ILDL(MC64+AMD)	58
			10.9	460+NC	ILDL(B+MC64R)	NC
			—	—	ILUTP(B+AMD)	—
			—	—	ILUTP(MC64+MC64R)	—
			—	—	ILUTP(MC64+AMD)	—
			—	—	ILUTP(B+MC64)	—

The experiments were run with `fill_factor` = 2.0 for the smaller matrices and `fill_factor` = 4.0 for matrices larger than 1 million in dimension. The tolerance was `drop_tol` = 10^{-4}, and we used rook pivoting to maintain stability. The iteration was terminated when the norm of the relative residual went below 10^{-6}. For iteration counts labeled with an asterisk (*), MINRES was used (as SQMR failed to converge). Iteration counts labeled with NC indicate nonconvergence for both MINRES and SQMR. Times labeled with a dash (—) took more than 10 hours to run and were terminated before completion.

Table VII. Results with HSL_MC80 for Matrices in Tables I and III

Matrix	n	nnz(A)	Fill	Time (s)	Iterations
aug3dcqp	35,543	128,115	2.0	0.051+0.188	28
bloweya	30,004	150,009	0.9	0.036+0.023	4
bratu3d	27,792	173,796	2.9	0.118+0.106	26
tuma1	22,967	87,760	3.0	0.063+0.227	44
tuma2	12,992	49,365	2.9	0.033+0.094	35
boyd1	93,279	1,211,231	1.0	0.120+0.062	4
brainpc2	27,607	179,395	1.5	0.086+0.119	26
mario001	38,434	204,912	3.6	0.110+0.501	59
qpband	20,000	45,000	1.1	0.015+0.004	1
nlpkkt80	1,062,400	28,192,672	7.1	133+86	49
nlpkkt120	3,542,400	95,117,792	—	—	—
c-55	32,780	403,450	2.95	0.28+0.15	15
c-59	41,282	480,536	2.99	0.36+0.20	15
c-63	44,234	434,704	2.92	0.29+0.21	15
c-68	64,810	565,996	2.31	0.31+0.17	9
c-69	67,458	623,914	2.65	0.35+0.18	9
c-70	68,924	658,986	2.67	0.40+0.24	11
c-71	76,638	859,520	3.00	0.74+0.32	12
c-72	84,064	707,546	2.69	0.40+0.21	9
c-big	345,241	2,340,859	2.54	1.2+0.93	8

Matrices in the first section (delimited by horizontal lines) were run with `fill_factor` = 2.0 and `drop_tol` = 10^{-4}. Matrices in the second section were run with `fill_factor` = 4.0 and `drop_tol` = 10^{-4}. Matrices in the third section were run with `fill_factor` = 12.0 and `drop_tol` = 0.003. Rook pivoting was used to maintain stability. The iterative solver used for the first two sections was SQMR, and GMRES was used for the third section. These settings maintain consistency with Tables I and III. The iteration was terminated when the norm of the relative residual went below 10^{-6}. Iteration counts labeled with NC indicate nonconvergence.

Table VIII. GMRES Comparisons Between SYM-ILDL and AMD with MC64 Equilibration

Matrix			MI30			SYM-ILDL		
Name	n	nnz(A)	$\text{Fill}_{\text{MI30}}$	Iterations	Time (s)	$\text{Fill}_{\text{SYM-ILDL}}$	Iterations	Time (s)
c-55	32,780	403,450	3.45	49	1.25+0.94	3.85	12	0.49+0.15
c-59	41,282	480,536	3.62	70	1.59+1.84	3.70	13	0.59+0.27
c-63	44,234	434,704	4.10	51	1.53+1.23	4.12	13	0.48+0.25
c-68	64,810	565,996	4.12	37	1.87+1.12	4.00	9	0.69+0.26
c-69	67,458	623,914	4.33	43	4.07+1.47	3.93	11	0.64+0.34
c-70	68,924	658,986	4.26	38	3.77+1.30	3.46	13	0.58+0.42
c-71	76,638	859,520	3.58	61	3.93+2.71	4.09	10	1.13+0.40
c-72	84,064	707,546	4.18	54	3.05+2.40	5.33	14	1.15+0.59
c-big	345,241	2,340,859	4.82	67	23.4+25.3	2.92	11	1.89+1.62

For each test case, we report the time it takes to compute the preconditioner, as well as the GMRES time and the number of GMRES iterations. The time is reported as $x + y$, where x is the preconditioning time and y is the GMRES time. GMRES was terminated when the relative residual decreased below 10^{-6}.

ACKNOWLEDGMENTS

We would like to thank Dominique Orban and Jennifer Scott for their careful reading of an earlier version of this article and Yousef Saad for providing the code from Li et al. [2003]. We would also like to thank the referees for their helpful and constructive comments.

REFERENCES

Patrick R. Amestoy, Timothy A. Davis, and Iain S. Duff. 1996. An approximate minimum degree ordering algorithm. *SIAM Journal on Matrix Analysis and Applications* 17, 4, 886–905. DOI:http://dx.doi.org/10.1137/S0895479894278952

James R. Bunch. 1971. Equilibration of symmetric matrices in the max-norm. *Journal of the ACM* 18, 4, 566–572. DOI:http://dx.doi.org/10.1145/321662.321670

James R. Bunch. 1982. A note on the stable decomposition of skew-symmetric matrices. *Mathematics of Computation* 38, 158, 475–479.

James R. Bunch and Linda Kaufman. 1977. Some stable methods for calculating inertia and solving symmetric linear systems. *Mathematics of Computation* 31, 137, 163–179.

Timothy A. Davis and Yifan Hu. 2011. The University of Florida sparse matrix collection. *ACM Transactions on Mathematical Software* 38, 1, Article No. 1. DOI:http://dx.doi.org/10.1145/2049662.2049663

I. S. Duff. 2009. The design and use of a sparse direct solver for skew symmetric matrices. *Journal of Computational and Applied Mathematics* 226, 1, 50–54. DOI:http://dx.doi.org/10.1016/j.cam.2008.05.016

I. S. Duff, A. M. Erisman, and J. K. Reid. 1989. *Direct Methods for Sparse Matrices* (2nd ed.). Clarendon Press, New York, NY.

I. S. Duff, N. I. M. Gould, J. K. Reid, J. A. Scott, and K. Turner. 1991. The factorization of sparse symmetric indefinite matrices. *IMA Journal of Numerical Analysis* 11, 2, 181–204. DOI:http://dx.doi.org/10.1093/imanum/11.2.181

I. S. Duff and J. Koster. 2001. On algorithms for permuting large entries to the diagonal of a sparse matrix. *SIAM Journal on Matrix Analysis and Applications* 22, 4, 973–996. DOI:http://dx.doi.org/10.1137/S0895479899358443

I. S. Duff and S. Pralet. 2005. Strategies for scaling and pivoting for sparse symmetric indefinite problems. *SIAM Journal on Matrix Analysis and Applications* 27, 2, 313–340. DOI:http://dx.doi.org/10.1137/04061043X

Stanley C. Eisenstat, Martin H. Schultz, and Andrew H. Sherman. 1981. Algorithms and data structures for sparse symmetric Gaussian elimination. *SIAM Journal on Scientific and Statistical Computing* 2, 2, 225–237. DOI:http://dx.doi.org/10.1137/0902019

R. W. Freund and N. M. Nachtigal. 1994. A new Krylov-subspace method for symmetric indefinite linear systems. In *Proceedings of the 14th IMACS World Congress on Computational and Applied Mathematics*.

Alan George and Joseph W. H. Liu. 1981. *Computer Solution of Large Sparse Positive Definite Systems*. Prentice Hall Series in Computational Mathematics. Prentice Hall, Englewood Cliffs, NJ.

Philip E. Gill, Walter Murray, Dulce B. Ponceleón, and Michael A. Saunders. 1992. Preconditioners for indefinite systems arising in optimization. *SIAM Journal on Matrix Analysis and Applications* 13, 1, 292–311. DOI:http://dx.doi.org/10.1137/0613022

Chen Greif and James M. Varah. 2009. Iterative solution of skew-symmetric linear systems. *SIAM Journal on Matrix Analysis and Applications* 31, 2, 584–601.

Michael Hagemann and Olaf Schenk. 2006. Weighted matchings for preconditioning symmetric indefinite linear systems. *SIAM Journal on Scientific Computing* 28, 2, 403–420. DOI:http://dx.doi.org/10.1137/040615614

Nicholas J. Higham. 2002. *Accuracy and Stability of Numerical Algorithms* (2nd ed.). Society for Industrial and Applied Mathematics, Philadelphia, PA.

J. D. Hogg and J. A. Scott. 2014. Compressed threshold pivoting for sparse symmetric indefinite systems. *SIAM Journal on Matrix Analysis and Applications* 35, 2, 783–817. DOI:http://dx.doi.org/10.1137/130920629

Mark T. Jones and Paul E. Plassmann. 1995. An improved incomplete Cholesky factorization. *ACM Transactions on Mathematical Software* 21, 1, 5–17. DOI:http://dx.doi.org/10.1145/200979.200981

Igor E. Kaporin. 1998. High quality preconditioning of a general symmetric positive definite matrix based on its $U^{\mathsf{T}}U + U^{\mathsf{T}}R + R^{\mathsf{T}}U$-decomposition. *Numerical Linear Algebra with Applications* 5, 6, 483–509. DOI:http://dx.doi.org/10.1002/(SICI)1099-1506(199811/12)5:6<483::AID-NLA156>3 .3.CO;2-Z

Na Li and Yousef Saad. 2005. Crout versions of ILU factorization with pivoting for sparse symmetric matrices. *Electronic Transactions on Numerical Analysis* 20, 75–85.

Na Li, Yousef Saad, and Edmond Chow. 2003. Crout versions of ILU for general sparse matrices. *SIAM Journal on Scientific Computing* 25, 2, 716–728. DOI:http://dx.doi.org/10.1137/S1064827502405094

Chih-Jen Lin and Jorge J. Moré. 1999. Incomplete Cholesky factorizations with limited memory. *SIAM Journal on Scientific Computing* 21, 1, 24–45. DOI:http://dx.doi.org/10.1137/S1064827597327334

Dominique Orban. 2014. Limited-memory $\mathrm{LDL}^{\mathrm{T}}$ factorization of symmetric quasi-definite matrices with application to constrained optimization. *Numerical Algorithms* 70, 1, 9–41. DOI:http://dx.doi.org/10.1007/s11075-014-9933-x

Daniel Ruiz. 2001. *A Scaling Algorithm to Equilibrate Both Rows and Columns Norms in Matrices*. Technical Report RAL-TR-2001-034. ENSEEIHT.

Olaf Schenk and Klaus Gärtner. 2006. On fast factorization pivoting methods for sparse symmetric indefinite systems. *Electronic Transactions on Numerical Analysis* 23, 158–179.

J. A. Scott and M. Tuma. 2014a. On signed incomplete Cholesky factorization preconditioners for saddle-point systems. *SIAM Journal on Scientific Computing* 36, 6, A2984–A3010. DOI:http://dx.doi.org/10.1137/140956671

J. A. Scott and M. Tuma. 2014b. On positive semidefinite modification schemes for incomplete Cholesky factorization. *SIAM Journal on Scientific Computing* 36, 2, A609–A633.

M. Tismenetsky. 1991. A new preconditioning technique for solving large sparse linear systems. *Linear Algebra and Its Applications* 154/156, 331–353. DOI:http://dx.doi.org/10.1016/0024-3795(91)90383-8

Received May 2015; revised November 2016; accepted December 2016