INCREMENTAL INCOMPLETE LU FACTORIZATIONS WITH APPLICATIONS TO TIME-DEPENDENT PDES

C. CALGARO *, J. P. CHEHAB †, AND Y. SAAD ‡

Abstract. This paper addresses the problem of computing preconditioners for solving linear systems of equations with a sequence of slowly varying matrices. This problem arises in many important applications. For example, a common situation in computational fluid dynamics, is when the equations change only slightly, possibly in some parts of the physical domain. In such situations it is wastful to recompute entirely any LU or ILU factorizations computed for the previous coefficient matrix. This papers examines a number of techniques for computing incremental ILU factorizations. For example we consider techniques based on approximate inverses as well as alternating techniques for updating the factors L and U of the factorization.

Key words. Preconditioning, Incomplete LU factorization, incremental LU.

AMS subject classifications.

1. Introduction.

A common problem which arises in many complex applications is to solve a sequence of linear systems of the form:

$$A_k x_k = b_k \tag{1.1}$$

for $k = 1, 2, ...$ In these applications A_k does not generally change too much from one step k to the next, as it is often the result of a continuous process (e.g. A_k can represent the discretization of some problem at time t_k.) We are faced with the problem of solving each consecutive system effectively, by taking advantage of earlier systems if possible. This problem arises for example in computational fluid dynamics, when the equations change only slightly possibly in some parts of the domain. A similar problem was recently considered in [4]. We also mention the somewhat related but different problem of updating the (exact) LU factorization of a given matrix when a very small number of its entries change. This problem has received much attention in the power systems literature, see, e.g., [9].

Let us examine a few possible approaches which come to mind for dealing with this situation.

Use of direct solvers. When direct solvers are used, recomputing the LU factorization is most expensive, and the only potential approach to exploiting previous information is to employ a previous LU factorization as a preconditioner for solving the next system. This appears to be still wasteful because computing an *exact* LU factorization is expensive and this cost is not likely to be amortized well in the next iterations. In other words, why compute an exact and expensive factorization which may be beneficial only for one system? In fact for 3-D problems an exact factorization may not be an option due to its prohibitive memory cost.

Incomplete LU (ILU). An economical alternative to using direct solvers is to exploit a good incomplete LU factorization at stage $k-1$. Then at stage k our options are (1) either to use the same factorization as at stage $k-1$ as a preconditioner; (2) or to recompute in incomplete LU factorization; or (3) to try to improve the ILU factorization. If the preconditioner is very inexpensive to compute (e.g., ILU(0)), then the second option is appealing. We will later explore the third option as an alternative to the second for more general situations.

APINV. Another approach which may appear appealing is to use approximate inverse techniques. These techniques generated much interest in the early 1990s as alternatives to ILU factorizations mainly because they are easy to parallelize, see for example, [1, 3, 21, 22, 2]. They consist of approximating directly the inverse of A in the form of a sparse (possibly factored) matrix. One such approach used in APINV ([14]) is to approximately minimize the Frobenius norm,

$$\|I - AM\|_F \tag{1.2}$$

*Université Sciences et Technologies de Lille, Laboratoire Paul Painlevé, UMR 8524, France and INRIA Lille Nord Europe, EPI SIMPAF, France. e-mail: caterina.calgaro@univ-lille1.fr

†Université Jules Verne, LAMFA, UMR 6140, Amiens, France and INRIA Lille Nord Europe, EPI SIMPAF, France. e-mail: jean-paul.chehab@u-picardie.fr

‡University of Minnesota, Computer Science, USA, work supported by DOE under grant DE-FG-08ER25841 and by the Minnesota Supercomputer Institute. e-mail: saad@cs.umn.edu. This work was initiated while the third author was visiting the department of Mathematics of the University of Lille

over sparse matrices M which have a pre-specified pattern. The main advantages of approximate inverse methods are: (1) their high degree of parallelism both during the construction phase and the iteration phase, and (2) their guaranteed stability. Here, the term stability is used in the sense of [16], a notion which which was also studied in [32, 13, 5]. Approximate inverse preconditioners also have several disadvantages and limitations, see [14] for example. Their main limitations are (1) that they require more memory than standard ILU factorizations for producing a satisfactory convergence and (2) it is difficult to predict whether the inverse of A (or of its factors) will be nearly sparse and it is even more difficult to guess their sparsity patterns in advance.

From the angle of updatability considered here, these methods appear as perfect candidates. Indeed, if $A_k M_k \approx I$, then for the next matrix A_{k+1}, we would need to update M_k into M_{k+1} by taking M_k as initial guess. So $M_{k+1} = M_k + \Delta_k$ where Δ_k is some sparse matrix. One drawback of approximate inverse methods is that overall they tend to underperform ILU-type techniques. As a result even though these methods yield preconditioners which are easy to update, the resulting preconditioners are far from optimal.

APINV+ILU. One can also think of adapting approximate inverse methods for the purpose of improving an ILU factorization. If B is a given preconditioner to A, we can seek a matrix M such that AM approximates B instead of the identity matrix. The approximation can be sought in the Frobenius norm sense. For example a matrix M can be computed to (approximately) minimize

$$\|B - AM\|_F.$$

In the above equation M can be obtained by approximately solving the linear systems $Am_j = b_j$, where b_j, the j-th column of B is sparse, in order to find the j-th (sparse) column m_j of M. The preconditioner B is then corrected into MB^{-1}. It is also possible to seek corrections to the factors L and U of an ILU factorization in the same manner and this will discussed later in the paper.

Among these options we will consider techniques for updating the L-U factors of an ILU factorization as well as approximate inverse techniques. Our goal is to show that there are inexpensive alternatives to recomputing a preconditioner and that the tools to be developed can be quite helpful in some specific applications.

2. Background and notation. One of the most common ways to define a preconditioning matrix M is through Incomplete LU factorizations. We will briefly review these factorizations in the next section. Then we will outline the main problem addressed in this paper, namely how to correct an ILU factorization.

2.1. Incomplete LU (ILU) factorizations. An ILU factorization is obtained from an approximate Gaussian elimination procedure. When Gaussian elimination is applied to a sparse matrix A, a large number of nonzero elements may appear in locations originally occupied by zero elements. These fill-ins are often small elements and may be dropped to obtain Incomplete LU factorizations. Thus, ILU is in essence a Gaussian elimination procedure in which fill-ins are dropped.

The simplest of these procedures, called ILU(0), is obtained by performing the standard LU factorization of A and dropping all fill-in elements that are generated during the process. In other words, the L and U factors have the same pattern as the lower and upper triangular parts of A (respectively). More accurate factorizations denoted by ILU(k) and IC (k) have been defined which drop fill-ins according to their 'levels'. Level-1 fill-ins for example are generated from level-zero fill-ins (at most). So, for example, ILU(1) consists of keeping all fill-ins that have level zero or one and dropping any fill-in whose level is higher.

Another class of preconditioners is based on dropping fill-ins according to their numerical values. One of these methods is ILUT (ILU with Threshold). This procedure uses basically a form of Gaussian elimination which generates the rows of L and U one by one. Small values are dropped during the elimination, using a parameter τ. A second parameter, p, is then used to keep the largest p entries in each of the rows of L and U. This procedure is denoted by $ILUT(\tau, p)$ of A. The following is a sketch of the algorithm in 'dense' mode, i.e., without exploiting sparsity. Details on practical implementations can be found in [27, 26].

Algorithm 1 (*ILUT*).

```
1.  Do i=1, n
2.      Do k=1,i-1
3.          A_{i,k} := A_{i,k}/A_{k,k}
4.          If |A_{i,k}| is not too small then
```

5. Do $j = k + 1, n$
6. $A_{i,j} := A_{i,j} - A_{i,k} * A_{k,j}$
7. EndDo
8. EndIf
9. EndDo
10. Drop small elements in row $A_{i,*}$
11. EndDo

ILUT and ILU(k) are fairly robust and cost effective general purpose preconditioners. Other preconditioners using the same principle for dropping [28, 25] can also be effective. One of the most unappealing features of ILU-type preconditioners is that they are not easy to update. If some entries in A change slightly, then the whole LU factorization should, in principle, be recomputed.

2.2. The problem. Suppose we have an approximate factorization which we write in the form

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

where R is the error in the factorization, which corresponds to the matrix of the terms that are dropped during the factorization procedure. Our primary goal is to improve this factorization, i.e., to find sparse matrices X_L, X_U such that $L + X_L$, $U + X_U$ is a better pair of factors than L, U. Consider the new error $A - LU$ for the new factors.

$$A - (L + X_L)(U + X_U) = (A - LU) - X_L U - L X_U - X_L X_U \ . \tag{2.2}$$

Ideally, we would like to make the right-hand side equal to zero. Denoting by R the matrix $A - LU$, this means we would like to solve:

$$X_L U + L X_U + X_L X_U - R = 0 \ , \tag{2.3}$$

for X_L and X_U. The above equation, which can be viewed as a form of Riccati equation, is nonlinear in the pair of unknowns X_L, X_U. In the symmetric case, where LU is replaced by the Cholesky factorization, and where we set $X_U = X_L^T \equiv X$ by symmetry, then (2.3) becomes

$$X L^T + L X^T + X X^T - R = 0 \ , \tag{2.4}$$

the standard Algebraic Riccati Equation.

In our case, we only consider (2.3) which we would like to solve approximately. For this we begin by neglecting the quadratic term $X_L X_U$, leading to:

$$X_L U + L X_U - R = 0 \ , \tag{2.5}$$

Then, several possible approaches are available for solving the above equation a few of which we now outline. First, we can use an approach in the spirit of approximate inverse-type corrections, i.e., we can try to find sparse triangular factors X_L, X_U which approximately minimize the Frobenius norm of the left-hand side of (2.5). A second possibility would be an alternating procedure at the matrix level where we fix U (so we set $X_U = 0$) and solve for X_L, and then fix the resulting L and solve for X_U, and repeat the process. We can also use a similar alternating procedure but at the vector level – where columns of L and rows of U are updated alternatively. This approach is similar to a well-known one used for solving Lyapunov equations. Another possibility is to consider the problem from an optimization viewpoint and exploit a descent method for minimizing the Frobenius norm of the matrix in (2.3). Finally, methods based on differential equations as advocated in [12, 10] can also be exploited. A few of these approaches are discussed in detail in the remainder of this paper.

2.3. Notation. Given an $n \times n$ matrix X we will denote by $X^{▯}$ the vector of dimension n^2 which consists of the n columns of X stacked into an n^2 vector. The reverse operation, converting a vector x of size n^2 into an $n \times n$ array will be denoted by $x^{\square}$. This will be called an *array representation* of x.

We will often need to extract lower and upper triangular parts of matrices. We will denote by $X_{◺}$ and $X_{◹}$ the *strict lower triangular part* and the *upper triangular part* of A respectively. We will denote by $X_{◺\!\!◺}$

the *unit lower triangular matrix* obtained by taking the lower part of X and replacing its diagonal entries by ones. In other words $X_{\backslash\!\backslash} = I + X_{\backslash}$, the identity matrix plus the *strict* lower part of X. Note the following easy to verify properties of these operators

$$(X+Y)_{\backslash} = X_{\backslash} + Y_{\backslash}; \qquad (X+Y)_{\urcorner} = (X+Y)_{\urcorner} \tag{2.6}$$
$$(X_{\backslash\!\backslash} Y_{\backslash})_{\urcorner} = 0; \qquad (X_{\backslash} Y_{\backslash\!\backslash})_{\urcorner} = 0; \tag{2.7}$$
$$(X_{\urcorner} Y_{\urcorner})_{\backslash} = 0 \; . \tag{2.8}$$

The fact that a matrix is upper triangular can be conveniently expressed by the equality $X_{\urcorner} = X$. Similarly, a matrix is strict lower triangular when $X_{\backslash} = X$ and it unit lower triangular when $X_{\backslash\!\backslash} = X$.

We define the inner product on the space of $n \times n$ matrices, viewed as objects in $\mathbb{R}^{n^2}$, as follows:

$$\langle X, Y \rangle = \mathrm{Tr}(Y^T X). \tag{2.9}$$

The Frobenius norm $\|.\|_F$ is the 2-norm associated with this inner product, i.e., $\|X\|_F = [\mathrm{Tr}(X^T X)]^{1/2}$.

3. Techniques based on approximate inverses. Standard approximate inverse techniques attempt to find an approximate inverse to A which is sparse. One such approach used in APINV (see, e.g., [14]) is to approximately minimize the Frobenius norm,

$$\|I - AM\|_F \tag{3.1}$$

over sparse matrices M which have a pre-specified pattern. These techniques attracted a great deal of attention in the 1990s as alternatives to the classical Incomplete LU factorizations. Their main advantages are: (1) their high degree of parallelism both during the construction phase and the iteration phase, and (2) their guaranteed stability. Here, the term stability is used in the sense of [16].

However, approximate inverse preconditioners also have several disadvantages and limitations, which outweigh their advantages for general systems. Their main limitations are that (1) they require more memory than standard ILU factorizations for producing a satisfactory convergence and (2) it is difficult to predict whether the inverse of A (or of its factors) is nearly sparse and it is even more difficult to guess it sparsity pattern.

For the reasons mentioned above, we do not consider approximate inverses as potential techniques here. Nevertheless, we are more interested in a by-product of this general class of methods which has good potential in the current framework and which we refer to as "Sparse Matrix Correction".

3.1. Sparse Matrix Corrections. A Sparse Matrix Correction technique consists of finding a correction to a given matrix by computing sparse approximate solution vectors to systems of the form $Am = f$, where f is a column of a certain sparse matrix F. Sparse approximate inverse methods such as APINV are in this category since M in (3.1) is found by computing approximate solutions of $Am_j = e_j$, for $j = 1, \ldots, n$ to get the columns m_j of M. It is important to note that what makes these methods competitive is the use of 'sparse-sparse' computations, in which the product of a sparse matrix by a sparse vector can be performed very inexpensively.

If B is a given preconditioner to A, we can seek a matrix M such that AM approximates B instead of the identity matrix. The approximation can be sought in the Frobenius norm sense. For example a matrix M can be computed to (approximately) minimize

$$F(M) = \|B - AM\|_F^2 \; . \tag{3.2}$$

Let B be a given preconditioner to A, given for example in the form of an approximate ILU factorization, so B is in the form $B = LU$, which is close to A in the Frobenius norm. We can seek the matrix M such that AM approximates $B = LU$, which is a sparse matrix. Recall that in approximate inverse techniques we set $B = I$. Thus, in this approach we try to (approximately) minimize

$$\| \underbrace{B}_{=LU} - AM \|_F .$$

The preconditioner B is then corrected into $MB^{-1} = MU^{-1}L^{-1}$. It is also possible to precondition from the right. In the above equation M can be obtained by approximately solving the linear systems $Am_j = b_j$, where b_j, the j-th column of B is sparse, in order to find the j-th (sparse) column m_j of M.

An alternative is to seek a global iteration, i.e., an iteration which updates the whole matrix at once as in

$$M_{k+1} = M_k + \alpha_k G_k$$

where G_k is a certain matrix. It is possible to apply a form of the steepest descent algorithm to try to minimize (3.2). For this we need the gradient of $F(M)$. The objective function (3.2) is a quadratic function on the space of $n \times n$ matrices, viewed as objects in $\mathbb{R}^{n^2}$. The natural inner product on the space of matrices, with which the function (3.2) is associated, is the inner product (2.9).

In descent-type algorithms a new iterate M_{new} is defined by taking a step along a selected direction G, i.e.,

$$M_{new} = M + \alpha G$$

in which α is selected to minimize the objective function associated with M_{new}. This is achieved by taking

$$\alpha = \frac{\langle R, AG \rangle}{\langle AG, AG \rangle} = \frac{\mathrm{Tr}(R^T AG)}{\mathrm{Tr}\left((AG)^T AG\right)}. \tag{3.3}$$

Note that the denominator is nothing but $\|AG\|_F^2$.

After a descent step is taken, the resulting matrix M will tend to become denser. In the context of sparse matrix corrections, one can think of applying just one step to improve the current approximation. Otherwise, if several steps are taken, it becomes essential to apply numerical dropping to new M's. However, the descent nature of the step is now lost, i.e., it is no longer guaranteed that $F(M_{new}) \leq F(M)$. An alternative would be to apply numerical dropping to the direction of search G before taking the descent step. However, in this case, the fill-in in M cannot be controlled.

The simplest choice for the descent direction G is to take it to be the residual matrix $R = B - AM$, where M is the new iterate. The corresponding descent algorithm is referred to as the Minimal Residual (MR) algorithm. It is easy to see that R is indeed a descent direction for F. Then we are in effect trying to solve the $n^2 \times n^2$ linear system

$$(I \otimes A)M^{\square} = B^{\square}$$

and for this system written in matrix form $AM = B$, $B - AM$ is simply the residual vector written as a matrix. The global Minimal Residual algorithm, adapted from [14] has the following form:

Algorithm 2 (**Global Minimal Residual descent algorithm**).
1. Select an initial M
2. Until convergence do
3. Compute $G := B - AM$
4. Compute α by (3.3)
5. Compute $M := M + \alpha G$
6. Apply numerical dropping to M
7. End do

Another popular choice is to take G to be the direction of steepest descent, i.e., the direction opposite to the gradient of F. Thinking in terms of n^2 vectors, the gradient of F can be viewed as an n^2 vector c such that

$$F(x + e) = F(x) + (g, e) + O(\|e\|^2)$$

where $(,)$ is the usual Euclidean inner product. If we represent all vectors as 2-dimensional $n \times n$ arrays, then the above relation is equivalent to

$$F(X + E) = F(X) + \langle G, E \rangle + O(\|E\|^2).$$

This allows us to determine the gradient as an operator on arrays, rather than n^2 vectors, as is done in the next proposition.

PROPOSITION 3.1. *The array representation of the gradient of F with respect to M is the matrix*

$$G = -2A^T R$$

in which R is the residual matrix $R = B - AM$.

Proof. This is adapted from [14] where the identity is replaced by the matrix B. A little calculation leads to the relation:

$$F(M+E) - F(M) = -2\langle A^T R, E\rangle + \langle AE, AE\rangle. \tag{3.4}$$

for an arbitrary matrix E. Thus, the differential of F applied to E is the inner product of $-2A^T R$ with E plus a second order term. The gradient is therefore simply $-2A^T R$. ∎

The steepest descent algorithm consists of simply replacing G in line 3 of the MR algorithm described above by $G = A^T R$. It was noted in [14] that this option can lead to a very slow convergence in some cases because it is essentially a steepest descent-type algorithm applied to the normal equations. In either global steepest descent or minimal residual, we need to form and store the G matrix explicitly. The scalars $\|AG\|_F^2$ and $\mathrm{Tr}(G^T AG)$ can be computed from the successive columns of AG, which can be generated, used, and discarded.

Seeking to correct $B = LU$ may appear somewhat unnatural because the matrix LU is usually much denser than either L or U taken separately. This suggests that we should seek instead separate corrections to the factors L and U of the ILU factorization. The same tools as the ones discussed above can be exploited. For example, we can seek upper and lower sparse triangular matrices U_1, L_1 which minimize

$$\|LU - L_1 A U_1\|_F.$$

This leads us naturally to an alterative procedure described next.

3.2. Alternating L-U descent methods. Some of the tools used for extracting approximate inverses will now be exploited to develop algorithms to improve a given ILU factorization. Consider equation (2.3) with the particular choice $X_L = 0$, which corresponds to updating U while L is kept frozen. Then, we would like to find a sparse X_U to minimize,

$$F(X_U) = \|A - L(U + X_U)\|_F^2 = \|R - LX_U\|_F^2 \tag{3.5}$$

where we have set $R \equiv A - LU$, the current factorization error.

Similar to the situation of approximate inverses, the optimum X_U is trivially known, in this case it equals $L^{-1}R$, and this will be exploited in the next section. Here, we seek an approximation only to this exact solution – and we can exploit for this purpose descent-type methods. Recall that in the case of approximate inverses, the goal is to minimize, for example, $\|B - AM\|_F^2$, an expression that is similar to that of (3.5). Note that

$$\|R - LX_U\|_F^2 = \mathrm{Tr}\ ([R - LX_U]^T [R - LX_U]) = \|R\|_F^2 - 2\mathrm{Tr}[R^T LX_U] + \|LX_U\|_F^2\ .$$

Therefore, the gradient of $F(X_U)$ at $X_U = 0$ written in matrix form is simply $G = -2\ L^T R$. What this means is that when X_U is a small multiple of $L^T R$ then $F(X_U)$ will decrease. With this, a steepest descent method can now be devised. The only issue is that $G = L^T R$ is not necessarily upper triangular, so X_U which is a multiple of G in this situation will not be upper triangular as desired. We can define G to be the upper triangular part of $L^T R$, i.e., using the notation defined in Section 2.3,

$$G = [L^T R]_{\nabla}, \tag{3.6}$$

which represents the upper triangular part of $L^T R$ (including the diagonal). In fact we need to sparsify even further this matrix to limit fill-in. An interesting property is that under these changes, the matrix remains a descent direction. Here descent direction is to be interpreted as to mean non-ascent.

Lemma 3.2. *The matrix $G = [L^T R]_{\nabla}$ and any sparsified version $\tilde{G}$ of it is a descent direction for F at $X_U = 0$.*

Proof. Let us denote by X the matrix $L^T R$. Since the gradient is $-2X$, a matrix G is a descent direction if $< -2X, G > \leq 0$, or $< X, G >= tr(G^T X) \geq 0$. Let η_{ij} be the entries of X and consider any diagonal entry of $G^T X$, where G is the upper triangular part of X. We have:

$$[G^T X]_{jj} = \sum_{i=1}^{j} \eta_{ij}^2 \geq 0 \ .$$

Therefore the trace is ≥ 0. If G is further sparsified to $\tilde{G}$, then the above sum is replaced by a sum over the sparsity pattern of the j-th row of G^T, i.e., the sparsified pattern of the j-th column of G, and it remains non-negative. ∎

We can now write an algorithm that is similar to Algorithm 3.1. To determine the best α we must minimize the quadratic form:

$$\|R - \alpha LG\|_F^2 = \text{Tr}\ ([R - \alpha LG]^T [R - \alpha LG]) = \|R\|_F^2 - 2\alpha \text{Tr}[R^T LG] + \alpha^2 \|LG\|_F^2$$

yielding

$$\alpha_U = \frac{\langle R, LG \rangle}{\langle LG, LG \rangle} = \frac{\text{Tr}(R^T LG)}{\text{Tr}\,((LG)^T LG)}. \tag{3.7}$$

Note that the denominator may be computed as $\|LG\|_F^2$.

An alternating algorithm which will update U and L alternatively, can now be derived. Formulas for the L part can be readily obtained by repeating the above argument. Note that the L matrix is unit lower triangular, so we seek a correction X_L that is strict lower triangular. Observe that the objective function (3.5) becomes

$$\|A - (L + X_L)U\|_F^2 = \|R - X_L U\|_F^2$$

and repeating the calculation done above will show that the array representation of the gradient is $-2RU^T$. Therefore,

$$G_L = [RU^T]_{\mathbb{L}}$$

and

$$\alpha_L = \frac{\langle R, G_L U \rangle}{\langle G_L U, G_L U \rangle} = \frac{\text{Tr}(R^T G_L U)}{\text{Tr}\,((G_L U)^T G_L U)}. \tag{3.8}$$

The denominator may be computed as $\|G_L U\|_F^2$.

Algorithm 3 (**Alternating Global Minimal Residual descent algorithm**).

1. Select an initial pair L, U (with $L = L_{\mathbb{L}}$, $U = U_{\nabla}$)
2. Until convergence Do
3. Compute $R := A - LU$
4. Compute $G = [L^T R]_{\nabla}$
5. Apply numerical dropping to G
6. Compute $\alpha = \langle R, C \rangle / \|C\|_F^2$, where $C = LG$.
7. Compute $U := U + \alpha G$
8. Compute $R := A - LU$
9. Compute $G := [RU^T]_{\mathbb{L}}$
10. Apply numerical dropping to G
11. Compute $\alpha = \langle R, C \rangle / \|C\|_F^2$ where $C = GU$.
12. Compute $L := L + \alpha G$
13. EndDo

With the above approach there is a guarantee that the residual norm $\|R_k\|_F$ will not increase at each iteration k. In addition, the method is rather flexible as it is tolerant of dropping as indicated by Lemma 3.2. On the practical side, one possible concern is the cost of this procedure. If we think of performing just one iteration to improve a given preconditioner, the computation will be dominated by the formation of the gradient matrices G in Lines 4 and 9 and the scalars α in lines 6 and 11. These involve the products of two sparse matrices the cost of which depends on the sparsity of these matrices. Note that the matrices C need not be stored. They can be calculated one column at a time, for example. Then the norm of the column and the inner inner product of this column with the corresponding column of R can be evaluated to compute parts of $\langle R, C\rangle$ and $\langle C, C\rangle$ and then the column can be discarded.

4. Alternating Sparse-Sparse iterations. Consider equation (2.3) in the situation where $X_U = 0$. Assuming that U is nonsingular we obtain:

$$X_L U = R \quad \rightarrow \quad X_L = RU^{-1} \tag{4.1}$$

Thus, the correction to L can be obtained by solving a sparse triangular linear system with a sparse right-hand side matrix, i.e., the system $U^T X_L^T = R^T$. However, as was noted before, the updated matrix $L + X_L$ obtained in this way is not necessarily unit lower triangular and it is therefore required to apply the $[\ .\ ]_{\setminus\!\!\!\!\setminus}$ operation for $L + X_L$ or the $[\ .\ ]_{\setminus}$ operation for X_L. This procedure can be repeated by freezing U and updating L and vice-versa alternatingly, leading to an iteration which can be summarized by the following 2 equations:

$$U_{k+1} = U_k + [L_k^{-1}(A - L_k U_k)]_{\nabla} \tag{4.2}$$
$$L_{k+1} = L_k + [(A - L_k U_{k+1})U_{k+1}^{-1}]_{\setminus}. \tag{4.3}$$

A feature of equations (4.2–4.3) is that they simplify. For (4.2) for example, we have

$$U_{k+1} = U_k + [L_k^{-1}(A - L_k U_k)]_{\nabla} = [U_k + L_k^{-1}(A - L_k U_k)]_{\nabla} = [L_k^{-1} A]_{\nabla}$$

A similar expression for L_{k+1} is easily derived and this leads to an iteration of the form:

$$U_{k+1} = [L_k^{-1} A]_{\nabla} \tag{4.4}$$
$$L_{k+1} = [A U_{k+1}^{-1}]_{\setminus\!\!\!\setminus}. \tag{4.5}$$

Note that *the above 2 formulas are not intended for practical use.* In most practical cases, it is better to use (4.2–4.3) because one can sparsify $A - L_k U_k$ (say) to keep only the largest entries in order to make the solve far less inexpensive.

One issue in the above equations is the fact that the inverse of U_{k+1} in (4.3) is not always defined. The algorithm will break down when U_{k+1} is singular. This seldom happens in practice, although the possibility cannot be excluded from theoretical arguments as the following example shows.

Example.

$$A = \begin{pmatrix} 1 & 1 & 1 \\ 1 & 2 & 2 \\ 1 & 2 & 4 \end{pmatrix} \quad L_0 = \begin{pmatrix} 1 & 0 & 0 \\ 2 & 1 & 0 \\ -3 & -1 & 1 \end{pmatrix} \quad \rightarrow \quad L_0^{-1} A = \begin{pmatrix} 1 & 1 & 1 \\ -1 & 0 & 0 \\ 3 & 5 & 7 \end{pmatrix} \quad \rightarrow \quad U_1 = \begin{pmatrix} 1 & 1 & 1 \\ 0 & 0 & 0 \\ 0 & 0 & 7 \end{pmatrix}.$$

This issue is not too different from the one encountered in incomplete LU factorizations with threshold. In fact any form of LU factorization implicitly involves solves with lower and upper triangular matrices and when dropping is applied, these matrices can become singular. However, at the limit U_k will approach the U factor of the LU factorization of A (as will be shown in Proposition 4.4), a nonsingular matrix.

It may be thought that solving a sparse system of the type above could be too expensive. In fact, one must remember that the right-hand side is sparse and there exist effective techniques for this task. The resulting alternating algorithm is given below.

ALGORITHM 4 (**Alternating Lower-Upper Sparse Correction**).

1. Given: A, U_0, L_0 (with $U_0 = [U_0]_{\urcorner}; L_0 = [L_0]_{\llcorner\!\llcorner}$)
2. For $k = 1, \cdots,$ Do:
3. Compute $R_k = A - L_k U_k$
4. Compute $X_U = [L_k^{-1} R_k]_{\urcorner}$
5. $U_{k+1} = U_k + X_U$
6. If $det(U_{k+1}) == 0$ Abort "Singular U reached"
7. Compute $R_{k+1/2} = A - L_k U_{k+1}$
8. Compute $X_L = [R_{k+1/2} U_{k+1}^{-1}]_{\llcorner}$
9. $L_{k+1} = L_k + X_L$
10. EndDo

Expressions (4.4–4.5) can be further simplified by replacing A by one of its triangular parts. To establish this we will use the following simple lemma which will also be helpful in the next section.

LEMMA 4.1. *Let X be any square matrix and let P be a unit lower triangular matrix and Q an upper triangular matrix. Then the following equalities hold:*

$$[PX]_{\urcorner} = [PX_{\urcorner}]_{\urcorner} \quad ; \quad [XQ]_{\llcorner} = [X_{\llcorner} Q]_{\llcorner} \ . \tag{4.6}$$

Proof. The first relation results from the equalities (2.6) and (2.7). Indeed,

$$[PX]_{\urcorner} = [P(X_{\llcorner} + X_{\urcorner})]_{\urcorner} = [PX_{\llcorner}]_{\urcorner} + [PX_{\urcorner})]_{\urcorner} = 0 + [PX_{\urcorner})]_{\urcorner}.$$

The second equality can be shown similarly from (2.6) and (2.8):

$$[XQ]_{\llcorner} = [(X_{\llcorner} + X_{\urcorner})Q]_{\llcorner} = [X_{\llcorner} Q]_{\llcorner} + [X_{\urcorner} Q]_{\llcorner} = [X_{\llcorner} Q]_{\llcorner} + 0.$$

∎

REMARK 4.2. *Others equalities can be deduced in the same manner, for example we have: $[XQ]_{\llcorner} = [X_{\llcorner\!\llcorner} Q]_{\llcorner}$ that can be used below.*

Using the lemma with $P = L_k^{-1}$, $Q = U_{k+1}^{-1}$, and $X = A$, equations (4.4–4.5) can be rewritten as:

$$U_{k+1} = [L_k^{-1} A_{\urcorner}]_{\urcorner} \tag{4.7}$$

$$L_{k+1} = [A_{\llcorner} U_{k+1}^{-1}]_{\llcorner\!\llcorner}. \tag{4.8}$$

It will be seen in Section 4.3 that these sequences can also be obtained by applying a time-integration scheme for solving a coupled system of ordinary differential equations.

4.1. Convergence results in the dense filter case. Filtering consists of keeping only specific wanted entries in a matrix, and dropping the others, i.e., replacing them with zeros. For example, applying an upper triangular filter to a matrix M amounts to only keeping its upper triangular part. An equivalent term often used in the literature is that of a 'mask'. The filter matrix can be dense or sparse. Next, we establish a few properties of the sequences U_k and L_k in the particular case where we use *dense triangular filters*. This only means that the iterates L_k, U_k are evaluated by the expressions (4.4–4.5) exactly (no other dropping is performed). We start with the following lemma.

LEMMA 4.3. *Let two square matrices P and Q be unit lower triangular and upper triangular respectively, with Q nonsingular. Then for any square matrix M we have:*

$$[M]_{\llcorner\!\llcorner} = [[MQ^{-1}]_{\llcorner\!\llcorner} Q]_{\llcorner\!\llcorner} \quad \textit{and} \quad [M]_{\urcorner} = [P[P^{-1}M]_{\urcorner}]_{\urcorner}. \tag{4.9}$$

Proof. The second part of the equation in (4.6) applied to $X = MQ^{-1}$ yields immediately

$$[[MQ^{-1}]_{\llcorner\!\llcorner} Q]_{\llcorner\!\llcorner} = [(MQ^{-1})Q]_{\llcorner\!\llcorner} = [M]_{\llcorner\!\llcorner}.$$

The second relation in (4.9) follows similarly by taking $X = P^{-1}M$ in the first equality of equation (4.6) ∎

We can now prove the following proposition

PROPOSITION 4.4. *Let A be a matrix that admits the LU decomposition $A = LU$, where L is unit lower triangular and U is upper triangular. Then, the sequences U_k and L_k defined by (4.4) and (4.5) starting with an arbitrary (unit lower triangular) L_0, satisfy the following properties*

(i) $[A - L_kU_{k+1}]_{\nabla} = 0, \quad [A - L_{k+1}U_{k+1}]_{\triangle} = 0,\ k \geq 0.$

(ii) If the sequences L_k and U_k converge, then $\lim_{k\to+\infty} L_k = L$ and $\lim_{k\to+\infty} U_k = U$.

(iii) The algorithm converges in one step if $L_0 = L$.

Proof. We first prove assertion (i). From (4.4) it follows that $L_kU_{k+1} = L_k[L_k^{-1}A]_{\nabla}$. Applying an upper triangular filter to both sides, we obtain, thanks to the second part of Equation (4.9)

$$[L_kU_{k+1}]_{\nabla} = [A]_{\nabla}.$$

This proves the first half of (i). To prove the second half first observe that from (4.6) we have $[L_{k+1}U_{k+1}]_{\triangle} = [(L_{k+1})_{\triangle}U_{k+1}]_{\triangle}$. Then, using the relation $L_{k+1} = [AU_{k+1}^{-1}]_{\blacktriangle}$, and invoking Equation (4.9) once more yields:

$$[L_{k+1}U_{k+1}]_{\triangle} = [[AU_{k+1}^{-1}]_{\blacktriangle}U_{k+1}]_{\triangle} = [A]_{\triangle}.$$

To prove assertion (ii), we introduce $\bar{L} = \lim_{k\to+\infty} L_k$ and $\bar{U} = \lim_{k\to+\infty} U_k$. Taking limits in (i) yields the relations

$$[\bar{L}\bar{U}]_{\nabla} = [A]_{\nabla} \quad \text{and} \quad [\bar{L}\bar{U}]_{\triangle} = [A]_{\triangle}\ .$$

Hence $\bar{L}\bar{U} = A$ and the uniqueness of the LU factorization implies that $\bar{L} = L$ and $\bar{U} = U$.
Finally, when we take $L_0 = L$, then $U_1 = [L^{-1}LU]_{\nabla} = [U]_{\nabla} = U$, proving (iii). ∎

In fact, if one of the sequences L_k or U_k converges, then both of them will converge to their limits L and U respectively. This follows from the following relations

$$\begin{aligned} U_{k+1} - U &= [L_k^{-1}A - U]_{\nabla} = [L_k^{-1}A - L^{-1}A]_{\nabla} = [(L_k^{-1} - L^{-1})A]_{\nabla} \\ L_{k+1} - L &= [AU_{k+1}^{-1} - L]_{\triangle} = [AU_{k+1}^{-1} - AU^{-1}]_{\triangle} = [A(U_{k+1}^{-1} - U^{-1})]_{\triangle}. \end{aligned}$$

These two relations show that as long as there is no break-down (i.e., U_{k+1} is nonsingular each time it is computed from (4.7)) the sequences L_k, U_k are well defined, and they converge hand-in hand, in the sense that when L_k converges, then U_k also converges and vice versa.

We can now prove the following convergence result which establishes the surprising fact that under the condition that the filters are triangular and dense, the sequence of L_k's and U_k's terminates in a finite number of steps.

THEOREM 4.5. *Let an initial L_0 be given such that the sequences L_k, U_k are defined for $k = 1, 2, \cdots, n$. Then the sequences U_k and L_k converge to U and L respectively, in no more than n steps.*

Proof. We proceed by induction on the dimension n of the matrices. The result is trivial to establish for the case $n = 1$. Indeed, if $A = [a]$, the trivial LU factorization is $L = [1]$, $U = [a]$ and the sequence L_k, U_k reaches this factorization in one step.

We now assume that the algorithm converges in no more than $n-1$ steps for matrices of dimension $n-1$. We use the notation

$$A = \begin{pmatrix} \tilde{A} & b \\ c & d \end{pmatrix}, \quad L_k = \begin{pmatrix} \tilde{L}_k & 0 \\ \alpha_k & 1 \end{pmatrix}, \quad U_k = \begin{pmatrix} \tilde{U}_k & \gamma_k \\ 0 & \delta_k \end{pmatrix}.$$

We also consider the block LU factorization of A,

$$A = \underbrace{\begin{pmatrix} \tilde{L} & 0 \\ c\,\tilde{U}^{-1} & 1 \end{pmatrix}}_{L} \underbrace{\begin{pmatrix} \tilde{U} & \tilde{L}^{-1}b \\ 0 & \delta \end{pmatrix}}_{U} \quad \text{with} \quad \delta = d - c\tilde{U}^{-1}\tilde{L}^{-1}b. \tag{4.10}$$

The recurrence relations (4.4–4.5) of the algorithm give,

$$U_{k+1} = \begin{pmatrix} \tilde{U}_{k+1} & \gamma_{k+1} \\ 0 & \delta_{k+1} \end{pmatrix} = \begin{pmatrix} [\tilde{L}_k^{-1}\tilde{A}]_{\nabla} & \tilde{L}_k^{-1}b \\ 0 & -\alpha_k\tilde{L}_k^{-1}b + d \end{pmatrix}. \tag{4.11}$$

and

$$L_{k+1} = \begin{pmatrix} \tilde{L}_{k+1} & 0 \\ \alpha_{k+1} & 1 \end{pmatrix} = \begin{pmatrix} [\tilde{A}\tilde{U}_{k+1}^{-1}]_{\setminus\!\setminus} & 0 \\ c\tilde{U}_{k+1}^{-1} & 1 \end{pmatrix}. \tag{4.12}$$

This results in the relations

$$\tilde{U}_{k+1} = [\tilde{L}_k^{-1}\tilde{A}]_{\nabla}, \quad \tilde{L}_{k+1} = [\tilde{A}\tilde{U}_{k+1}^{-1}]_{\setminus\!\setminus}, \tag{4.13}$$

and

$$\alpha_{k+1} = c\tilde{U}_{k+1}^{-1}, \quad \gamma_{k+1} = \tilde{L}_k^{-1}b, \quad \delta_{k+1} = d - \alpha_k\tilde{L}_k^{-1}b. \tag{4.14}$$

Now by assumption, the relations (4.13) show that the sequence of the pairs $\tilde{L}_k, \tilde{U}_k$ converges in no more than $n-1$ steps to the L, U factors of $\tilde{A}$, i.e., $\tilde{L}_{n-1} = \tilde{L}$ and $\tilde{U}_{n-1} = \tilde{U}$. It follows that $\tilde{L}_n = \tilde{L}$, $\tilde{U}_n = \tilde{U}$, $\alpha_n = c\tilde{U}^{-1}$, $\gamma_n = \tilde{L}^{-1}b$, $\delta_n = d - c\tilde{U}^{-1}\tilde{L}^{-1}b$. Taking $k = n-1$ in (4.11) and comparing the result with the U part of the factorization (4.10) we obtain immediately $U_n = U$. Proceeding the same way by taking $k = n-1$ in (4.12) and comparing the result with the L factor in (4.10) shows that $L_n = L$ and completes the proof. ∎

Similar results can be derived when considering the iterative relaxed Cholesky factorization defining the sequence of upper triangular matrices S_k defined recursively as

$$S_{k+1} = S_k + \theta\left([(S_k^T)^{-1}A]_{\nabla} - S_k\right), \ \theta \in]0, 1[.$$

These sequences can be obtained by forward Euler time discretization of a Cholesky flow, see Section 4.3.

4.2. Convergence in the sparse filter case. Consider now the situation where a general sparse filter is applied. This means that a sparsity pattern $\mathcal{F}_k$ is selected at each step k and the matrices L_k, U_k of the factorization are restricted to having the pattern of the lower and upper parts of $\mathcal{F}_k$, respectively, by dropping any entries outside of this pattern. Recall the notation with filters. The filter $\mathcal{F}$ can be considered as an $n \times n$ matrix of zeros and ones. A zero entry corresponds to an unwanted term in the pattern, so a nonzero entry in this location is dropped. We denote by $X \odot \mathcal{F}$ the Hadamard product (i.e., component-wise product), of the matrices X and $\mathcal{F}$. The algorithm with a general constant sparse filter would replace equations (4.4–4.5) by

$$U_{k+1} = [(L_k^{-1}A) \odot \mathcal{F}]_{\nabla} = [(L_k^{-1}A)]_{\nabla} \odot \mathcal{F} \tag{4.15}$$

$$L_{k+1} = [(AU_{k+1}^{-1}) \odot \mathcal{F}]_{\setminus\!\setminus} = [(AU_{k+1}^{-1})]_{\setminus\!\setminus} \odot \mathcal{F} \tag{4.16}$$

Note that the algorithm will break down immediately if the filter has a zero diagonal entry. We will call such filters *singular* and exclude them from consideration. Thus, all filters in the remainder of this paper have all their diagonal entries equal to one.

The most straightforward extension of the convergence result of the previous section would show that the sequences L_k, U_k converge to the sparse LU factors L and U of A respectively. For this to be possible, it is necessary that the patterns of the filter matrices $\mathcal{F}_k$ contain those of the L and U parts respectively at each step. In fact under the assumption that $\mathcal{F}_k$ contains the symbolic pattern of Gaussian elimination, the same result as that of the previous theorem holds. Here, by symbolic pattern, we mean the pattern of Gaussian elimination obtained from the graph, i.e., in the situation where exact cancelations do not take place in Gaussian elimination [15]. The proof of this result is an immediate extension of that of the previous theorem.

COROLLARY 4.6. *Assume that each of the filter matrices $\mathcal{F}_k$, $k \geq 0$ contains the (symbolic) sparsity patterns of L and U respectively and let an initial L_0 be given such that $L_0 \odot \mathcal{F}_0 = L_0$ and the sequences L_k*

and U_k exist for $k = 1, \cdots, n$. Then the sequences U_k and L_k converge to U and L, respectively, in no more than n steps.

Proof. We only give a sketch of the proof which can also be inferred from well-known results on sparse Gaussian elimination. Essentially, under the assumptions, no entries are ever dropped due to the filter, because the patterns are included in those of L and U, the LU factors of A. To see this, let the filter at step $k+1$ be written as

$$\mathcal{F}_{k+1} = \begin{pmatrix} \tilde{\mathcal{F}}_k & f_k \\ g_k & 1 \end{pmatrix}.$$

Then, Equations (4.11) and (4.12) become,

$$U_{k+1} = \begin{pmatrix} [\tilde{L}_k^{-1}\tilde{A}]_{\nabla} \odot \tilde{\mathcal{F}}_k & [\tilde{L}_k^{-1}b] \odot f_k \\ 0 & -\alpha_k \tilde{L}_k^{-1} b + d \end{pmatrix} \quad \text{and} \quad L_{k+1} = \begin{pmatrix} [\tilde{A}\tilde{U}_{k+1}^{-1}]_{\backslash\!\backslash} \odot \tilde{\mathcal{F}}_k & 0 \\ [c\tilde{U}_{k+1}^{-1}] \odot g_k & 1 \end{pmatrix}. \tag{4.17}$$

We can use the same induction argument as in Theorem 4.5. The result is trivially true for $n = 1$. Now assume it is true for $k = n-1$, i.e., $\tilde{L}_k = \tilde{L}$ and $\tilde{U}_k = \tilde{U}$ where the notation of the proof of Theorem 4.5 is adopted. In this situation,

$$\tilde{U}_{k+1} = [\tilde{L}_k^{-1}\tilde{A}]_{\nabla} \odot \tilde{\mathcal{F}}_k = [\tilde{L}^{-1}\tilde{A}]_{\nabla} \odot \tilde{\mathcal{F}}_k = \tilde{U}_{\nabla} \odot \tilde{\mathcal{F}}_k = \tilde{U}.$$

The last equality holds because of the assumption that the filter includes the pattern of U. This shows that

$$\tilde{L}_{k+1} = [\tilde{A}\tilde{U}_{k+1}^{-1}]_{\backslash\!\backslash} \odot \tilde{\mathcal{F}}_k = [\tilde{A}\tilde{U}^{-1}]_{\backslash\!\backslash} \odot \tilde{\mathcal{F}}_k = \tilde{L} \odot \tilde{\mathcal{F}}_k = \tilde{L}.$$

Using a similar argument, we also have $\gamma_{k+1} = [\tilde{L}_k^{-1}b] \odot f_k = [\tilde{L}^{-1}b] \odot f_k = \tilde{L}^{-1}b$ and $\alpha_{k+1} = [c\tilde{U}_{k+1}^{-1}] \odot g_k = c\tilde{U}^{-1} \odot g_k = c\tilde{U}^{-1}$ and finally, $\delta_{k+1} = -\alpha_k \tilde{L}_k^{-1} b + d = d - c\tilde{U}_k^{-1}\tilde{L}_k^{-1}b$ converges to δ. So the factors L_{k+1}, U_{k+1} at step $k+1 = n$ are identical with L and U, the LU factors of A. ∎

It is interesting to consider how the results of Proposition 4.4 change when a general sparse filter is applied. Consider the iteration (4.15), (4.16), in which the filter $\mathcal{F}$ is now constant. We can rewrite the iteration with the help of the upper and lower parts of the filter as follows:

$$U_{k+1} = [(L_k^{-1}A)]_{\nabla} \odot \mathcal{F}_{\nabla} \tag{4.18}$$

$$L_{k+1} = [(AU_{k+1}^{-1})]_{\backslash\!\backslash} \odot \mathcal{F}_{\backslash\!\backslash} \tag{4.19}$$

The filtering operation in (4.18) is equivalent to subtracting an upper triangular sparse matrix S_k from the matrix $(L_k^{-1}A)_{\nabla}$, so it follows that $U_{k+1} = [(L_k^{-1}A)]_{\nabla} - S_k$. Proceeding as in Proposition 4.4 we multiply this relation to the left by L_k, so $L_k U_{k+1} = L_k[L_k^{-1}A]_{\nabla} - L_k S_k$, and then apply the upper triangular filter to obtain, exploiting again Lemma 4.3,

$$[L_k U_{k+1}]_{\nabla} = [L_k(L_k^{-1}A)_{\nabla}]_{\nabla} - [L_k S_k]_{\nabla} = [A]_{\nabla} - [L_k S_k]_{\nabla}. \tag{4.20}$$

Similarly, equation (4.19) yields $L_{k+1} = [(AU_{k+1}^{-1})]_{\backslash\!\backslash} \odot \mathcal{F}_{\backslash\!\backslash} = [(AU_{k+1}^{-1})]_{\backslash\!\backslash} - T_k$, where T_k represents the (sparse) lower triangular matrix of dropped entries. Then, proceeding in the same way as above, we get $L_{k+1}U_{k+1} = [(AU_{k+1}^{-1})]_{\backslash\!\backslash} U_{k+1} - T_k U_{k+1}$ and

$$[L_{k+1}U_{k+1}]_{\backslash} = [[(AU_{k+1}^{-1})]_{\backslash\!\backslash} U_{k+1}]_{\backslash} - [T_k U_{k+1}]_{\backslash} = A_{\backslash} - [T_k U_{k+1}]_{\backslash} \tag{4.21}$$

We have just shown the following two relations

$$[A - L_k U_{k+1}]_{\nabla} = [L_k S_k]_{\nabla} \tag{4.22}$$

$$[A - L_{k+1}U_{k+1}]_{\backslash} = [T_k U_{k+1}]_{\backslash} \tag{4.23}$$

Though these relations do not establish convergence, they tell us something in the case when the process converges.

PROPOSITION 4.7. *Assume that a constant filter $\mathcal{F}$ is used and let S_k and T_k the upper triangular and lower triangular matrices of the dropped entries from U_{k+1} and L_{k+1} due to the filtering operation. Assume that the matrices L_k and U_k converge to $\bar{L}, \bar{U}$, respectively, and that $\bar{U}$ is nonsingular, Then,*

(i) The sequence of matrices S_k converges to $S = [\bar{L}^{-1}A]_{\urcorner} - \bar{U}$.
(ii) The sequence of matrices T_k converges to $T = [A\bar{U}^{-1}]_{\llcorner} - \bar{L}$.
(iii) The sequences of matrices $(A - L_kU_{k+1})_{\llcorner}$ and $(A - L_kU_k)_{\urcorner}$ converge to $[\bar{L}S]_{\urcorner}$ and $[T\bar{U}]_{\llcorner}$, respectively.

Proof. To prove (i) note that under the assumptions, Equation (4.22) implies that $(L_kS_k)_{\urcorner}$ converges to $[A - \bar{L}\bar{U}]_{\urcorner}$. We now invoke Lemma 4.3 again and observe that:

$$[L_k^{-1}(L_kS_k)_{\urcorner}]_{\urcorner} = [S_k]_{\urcorner} = S_k$$

The left hand side, and therefore also S_k, converges to the following limit

$$[\bar{L}^{-1}[A - \bar{L}\bar{U}]_{\urcorner}]_{\urcorner} = [\bar{L}^{-1}[A - \bar{L}\bar{U}]]_{\urcorner} = [\bar{L}^{-1}A]_{\urcorner} - \bar{U}.$$

The proof of (ii) is similar. The proof of (iii) is a direct consequence of relations (4.22–4.23). ∎

The convergence of the matrices of dropped entries may seem counter intuitive. However, one must realize that these matrices represent a sort of cumulative matrix of dropped entries. Our experience with the algorithm shows that it often converges rapidly to a certain limit. With a trivial filter equal to a diagonal matrix, the process generally converges in one or 2 steps. We run a few tests with finite difference discretizations of a Laplacien on an $n \times n \times n$ mesh, by taking the filter to be the original pattern of the matrix. The observation is that the process converges in about n steps. In fact the norm $\|L_k - L_{k+1}\|_F + \|U_k - U_{k+1}\|_F$ drops abruptly at around n steps.

The algorithm ITALU is illustrated in the numerical experiments section within a few practical applications of this technique.

4.3. The ODE viewpoint. It is well known that iterative processes can be viewed as discrete dynamical systems by themselves but also, in certain cases, as originating from a suitable time discretization of a (governing) continuous differential system. In this latter case, the discrete system can "inherit" some properties such as consistency and stability that can be studied in the framework of classical numerical analysis of Ordinary Differential Equations (ODEs). This idea has been proposed to enable the computation of approximate inverses of matrices in [10] or to solve linear systems of equations [12]. More recently, in [11] sparse factorization flows (for LU or Cholesky factorizations) have been introduced. An ILU flow can be defined as follows: let A be a square matrix such that all its minors are nonzero. The time dependent matrix $Q(t)$ defined by

$$\frac{dQ}{dt} = A - Q, \quad t > 0 \tag{4.24}$$

$$Q(0) = Q_0, \tag{4.25}$$

converges to A as t goes to infinity. We can write in particular

$$\begin{cases} \frac{dL}{dt} = \bar{L} - L, \quad t > 0, \\ L_{i,i}(t) = 1, \\ L(0) = L_0, \end{cases} \quad \text{and} \quad \begin{cases} \frac{dU}{dt} = \bar{U} - U, \quad t > 0, \\ U(0) = U_0, \end{cases}$$

with $A = \bar{L}\bar{U}$. Here again, the matrices $L = L(t)$ and $U = U(t)$ converge to $\bar{L}$ and $\bar{U}$ as t goes to infinity, these two limit matrices are asymptotic stable steady states and can then be computed by an explicit time marching scheme, such as forward Euler's scheme for which the stability condition for this particular case is $0 < \Delta t < 2$. Of course neither $\bar{L}$ nor $\bar{U}$ are known *a priori*, and therefore we consider modified systems. We can write

$$\bar{L} - L = (\bar{L}\bar{U} - L\bar{U})\bar{U}^{-1} \approx (A - LU)U^{-1} \text{ and } \bar{U} - U = \bar{L}^{-1}(\bar{L}\bar{U} - \bar{L}U) \approx L^{-1}(A - LU).$$

We now introduce dense upper and lower triangular filters and define the coupled system

$$\begin{cases} \frac{dU}{dt} = [L^{-1}(A - LU)]_{\urcorner}, & t > 0 \\ \frac{dL}{dt} = [(A - LU)U^{-1}]_{\llcorner}, & t > 0 \\ U(0) = U_0; \quad L(0) = L_0 \quad \text{with } (L_0)_{ii} = 1,\ i = 1, \cdots, n. \end{cases} \tag{4.26}$$

Note that Lemma 4.3 insures that the only steady state is $L = \bar{L}$ and $U = \bar{U}$. Of course the filtering can be sparse and the above system can generate a flow of sparse triangular matrices [11]. Indeed, defining U_k and L_k respectively as the approximations of $U(k\Delta t)$ and $L(k\Delta t)$ obtained by the scheme

$$U_{k+1} = U_k + \Delta t[L_k^{-1}(A - L_k U_k)]_{\nabla}, \tag{4.27}$$

$$L_{k+1} = L_k + \Delta t[(A - L_k U_{k+1})U_{k+1}^{-1}]_{\setminus}, \tag{4.28}$$

we recover formally the sequence given by (4.2–4.3) by taking $\Delta t = 1$.

When A is symmetric and positive definite a Cholesky flow can be derived in a similar manner as

$$\begin{cases} \frac{dX}{dt} = [(X^T)^{-1}(A - X^T X)]_{\nabla}, \quad t > 0 \\ X(0) = X_0 \end{cases} \tag{4.29}$$

where X is a upper triangular matrix, [11].

4.4. Practical Implementations. Two key tools must be exploited in any implementation of the methods discussed above. One is concerned with basic sparse-sparse computations, i.e., computations involving products of sparse matrices with sparse vectors or matrices. The other key tool is concerned with effective ways of solving sparse triangular systems with sparse right-hand sides. Basic sparse-sparse computations have been covered in detail in the literature, see e.g., [14]. Techniques for solving sparse triangular systems with sparse right-hand sides are key ingredients of sparse direct solvers and so effective methods for this task have already been exploited. These are based on 'elimination paths' and 'topological sorting', see, for example, [19, 31, 9]. Thus, an LU factorization can be implemented as a sequence of sparse triangular solves where the right-hand sides are sparse. Because the right-hand side is sparse, the solution is also sparse, and it is easy to find a sequence in which the unknowns must be determined by examining a tree structure issued from the sparsity pattern of both the right-hand side and the matrix. Details can be found in, e.g., [19, 31, 15].

5. Applications to linear and stationnary problems. To illustrate the behavior of the incremental ILU, we consider in this section a few problems involving 3D-convection-diffusion like finite difference matrix, as well as standard test matrices from the Harwell-Boeing collection. All experiments were performed in Matlab.

5.1. Iterative ILU as a means to obtain an ILU. We will illustrate a method to compute an ILU factorization by starting from an inaccurate SSOR (or ILU(0)) and then improving the factorization progressively by using one step of the procedure described in Section 4 and labeled ITALU (Iterative Threshold Alternating LU).

In the following experiment we generate a matrix from a convection-diffusion problem on a $15 \times 15 \times 10$ regular grid. This yields a problem of size $n = 2250$. We use finite difference discretization and generate a problem with constant convection terms in each direction. So, in each direction, the discretized operator yields a tridiagonal matrix $tridiag[-1+\alpha, \quad 2, \quad -1-\alpha]$. The whole matrix is obtained by exploiting Kronecker products of the 3 matrices in each direction. The right-hand side of the system to solve is generated artificially as $b = A * ones(n, 1)$ in Matlab notation. In the first test α is taken equal to 0.1 and the matrix was shifted by 0.3 to make it indefinite. This shift introduces a negative eigenvalue and two eigenvalues very close to zero as can be seen from a run with MATLAB's eigs function:

$$\begin{array}{lllllll} \lambda_1 & = & -0.112843039478229, & \lambda_2 & = & -0.000397969025052, \\ \lambda_3 & = & 0.000397969025052, & \lambda_4 & = & 0.113638977528334, \\ \lambda_5 & = & 0.122450476821596 & \cdots & & \end{array}$$

In addition the condition number estimated by Matlab was 1.68e+05.

Figure 5.1 shows the performance of GMRES with 4 different preconditioners. The first preconditioner is SSOR with $\omega = 1$. In fact this is simply the factorization $A \approx L_0 U_0$ with $L_0 = I + A_{\setminus} D_A^{-1}$ and $U_0 = A_{\nabla}$. The second preconditioner takes these two factors and generates an improved factorization $A \approx L_1 U_1$ by performing one iteration of the ITALU algorithm. Dropping is performed in this procedure as follows. First a relative tolerance is set to droptol = 0.2, the effect of which is to ignore entries in any generated column

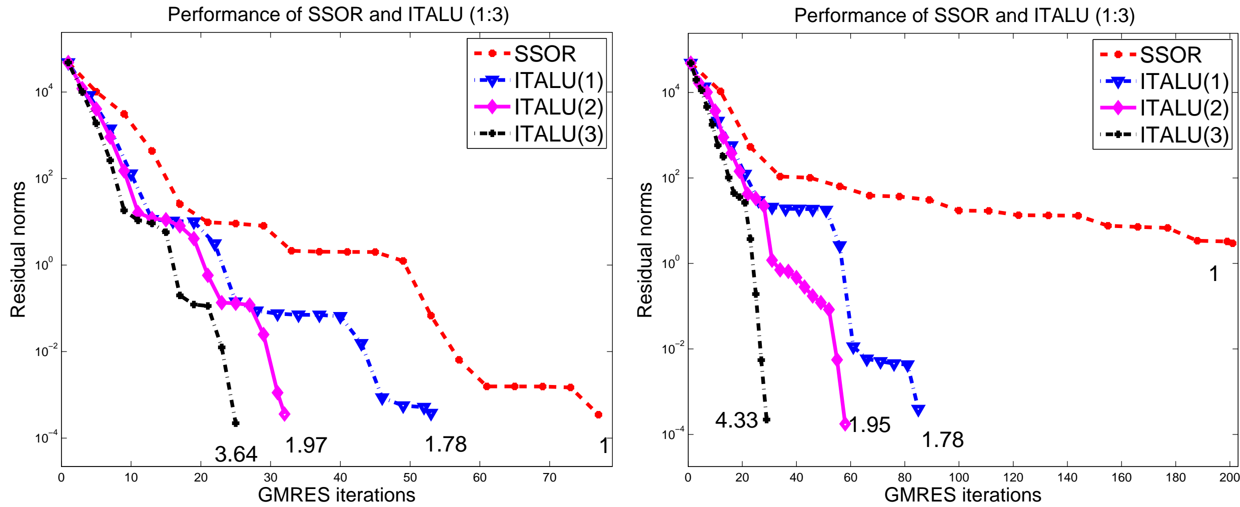

FIG. 5.1. *Performance of GMRES(30) for a convection-diffusion problem with the SSOR and 3 levels of improvements. Left and right plot are for two different problems.*

that is less than droptol times the norm of the original row. In addition, as a precaution against excessive fill-in, only the largest lfil=10 entries are kept (however in this example, this second criterion does not seem to have been triggered). The results, in the form of residual norm versus iteration count, are shown on the left panel of Figure 5.1. In the last run, corresponding to ITALU(3), there was no change from the performance of one preconditioner to the next, so we changed the parameters to show one more run, namely one that corresponds to droptol = 0.1 and lfil = 20 (starting again from the preconditioner obtained in the previous run, i.e., the one labelled ITALU(2) in the figure). The numbers shown at the end of each curve (3.97, 1.97, 1.78, 1) are the "fill-factors" for each preconditioner, i.e., the ratios of the number of nonzero entries in $L+U$ over that of A. These measure the amount of memory required to store the preconditioner. They also give a rough idea of the amount of work required to build the preconditioner.

In a second test, we repeated exactly the same experiment with a problem of the same size but with different coefficients: we took $\alpha = 0.2$ and then shifted the matrix by the diagonal -0.5 I to make the problem more indefinite. In this case SSOR has trouble converging. The results are shown on the right panel of Figure 5.1. The smallest eigenvalues as computed by the function eigs are

$$\begin{array}{llllll} \lambda_1 = & -0.224092132465693 & \lambda_2 = & -.112580095852315 & \lambda_3 = & -0.112580095852314 \\ \lambda_4 = & -0.001068059238936 & \lambda_5 = & .007608905447229 & \lambda_6 = & 0.068505625493865 \\ \lambda_7 = & 0.068505625493866 & \lambda_8 = & .119120942060608 & \ldots & \end{array}$$

In addition, Matlab's condition number estimator yields $condest(A) \approx 4.03E+05$.

Recall that the test corresponding to ITALU(3) used droptol=0.1 and lfil=20 instead of droptol=0.2 and lfil=10. Otherwise the preconditioner resulting from this additional iteration was basically the same as ITALU(2). Note that the fill-ratio is now substantially larger for this last test, but that this results in a much faster convergence.

5.2. Descent-type methods for computing an ILU. The technique illustrated in the preceding section had the disadvantage of requiring solving sparse triangular systems with sparse right-hand side. Effective methods for such tasks are well-known in the context of sparse direct methods. The best known method here is the Gilbert and Peirels (GP) algorithm [19] which utilizes topological sort to perform an effective solve that exploits sparsity in an optimal way.

The Frobenius-based method described in Section 3.2 (labeled ITALUF) is much simpler and it will now be illustrated on the second example seen in the previous section. Recall that in this example, the test matrix is generated from a Convection-Diffusion equation, with the paramter α taken to be 0.2 and the discretized Laplacien is shifted by 0.5 I to make the problem indefinite.

The following test is similar to the one described earlier. First GMRES(30) is attempted with the SSOR preconditioner. This yields the first convergence curve of the left side of Figure 5.2. Then two improved

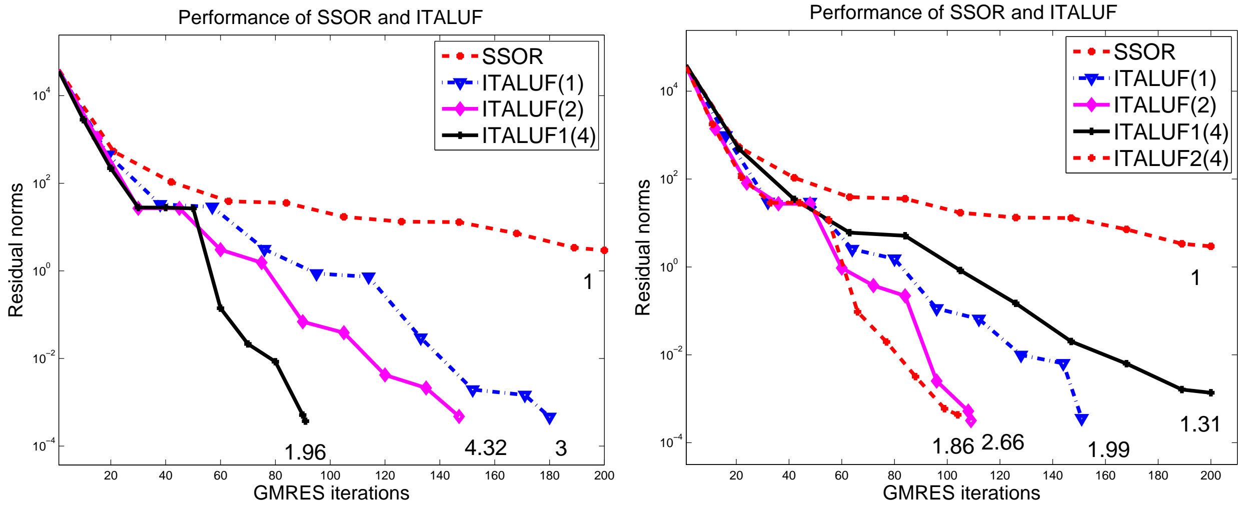

FIG. 5.2. *Performance of GMRES(30) for a convection-diffusion problem with the SSOR and a few levels of improvements using the Frobenius-based approach. The test matrix is the same as the second one of the previous section. The right plot shows tests with different dropping strategies.*

preconditioners are generated by Algorithm 3.2. The same droping parameters are applied in Lines 5 and 10 as in the previous example. We took droptol = 0.2 and lfil=10. As before the number shown at the end of each curve corresponds to the fill-factor for each case. The last curve shows the performance of an algorithm which is a modification of algorithm 3.2 in which the dropping is done differently. In this variation (labeled ITALUF1), the dropping in Lines 5 and 10 is removed. Instead, we drop directly from the factors U and L after each update. From a practical point of view this is more interesting because fill-in is better controled. ¿From a theoretical point of view, we loose the guarantee that the factors generated will have a diminishing residual norm $\|A - LU\|_F$. The last curve in the figure shows this result with the preconditioner obtained from 4 steps of this procedure starting with the SSOR precondition as initial guess. The dropping parameters are the same as above: droptol = 0.2 and lfil = 10.

We repeated the experiment with less restrictive fill-in parameter by changing the value of lfil to 5 instead of 10. This means that fewer entries will be kept each time. The results are shown on the right side of Figure 5.2. The alternative procedure ITALUF1 converges slowly but requires very little fill-in (a fill ratio of 1.31). In fact the last few iterations of the iterative LU procedure to compute the LU factors are not very effective here because we keep too few entries at each iteration. As a result an almost identical convergence curve is obtained if the number of iterations is reduced from 4 to 2. To remedy this, we tested yet another variant which allowed more fill in after each outer iteration. Specifically, we simply increase lfil by 2 after each iterative LU computation. The resulting method is labeled 'ITALUF2' in the plot. As can be seen this variant achieve good convergence relative to the fill-in allowed.

5.3. Improving the factorization within a GMRES run. When solving a linear system we do not know in advance how accurate the preconditioner must be for the iteration to converge fast enough. One can start with an inaccurate factorization which is then improved within the GMRES iteration. For example, as soon as there are signs of stagnation – or slow convergence – one can attempt to improve the current factors by running one step of improvement with the help of Algorithm 4. The following test uses the same example as above with the harder parameters $\alpha = 0.2$ and the shift -0.5. We run GMRES and restart every 30 steps. At the occasion of each restart we update the factorization by improving it with one step of the alternating procedure of algorithm 4. Since the precondioning changes only at restart a standard GMRES is sufficient. However, it is also possible to run a nonstarted GMRES and update the factorization any time the accelerator slows down. The left side of Figure 5.3 show the corresponding results. Two cases are considered with ITALU, one with droptol=0.1 and lfil = 5, and the second with droptol=0.1 and lfil = 10. In each case the initial preconditioner is SSOR. Then the preconditioner is improved internally at each restart of GMRES, by carrying out just one iteration of Algorithm 4. A comparison is also made with a

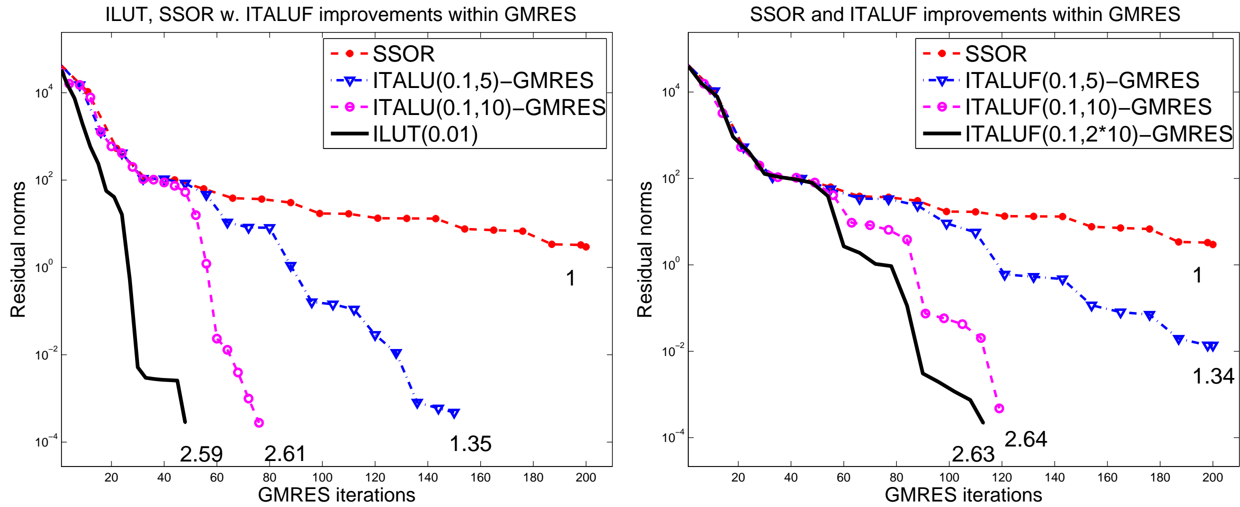

Fig. 5.3. *Performance of GMRES(30) for a convection-diffusion problem with the SSOR and ILUT preconditioners and incremental ILU procedures implemented within GMRES with 2 different parameters. The right side plot uses the Frobenius approach.*

standard ILUT preconditioning with Matlab's ILUT (called `luinc`) using droptol = 0.01 and no pivoting.

The fill-ratios indicated at the tail of each curve are the final fill ratios, i.e., those for the last restart. For tol=0.1, the successive fill ratios were 1.0 (for SSOR), and then they were 1.34, 1.35, 1.35. indicating that little improvement to the preconditioner was actually achieved after the first iteration. For the second run, using lfil=10, they were 1.0 (for SSOR), 2.23 for the first restart and 2.61 for the second (and last) restart. The fill-ratio for Matlab's ILUT is 2.59 as indicated.

We performed a similar experiment with the Frobenius based incremental ILU of Section 3.2 and the results are shown on the right side of Figure 5.3. We show four test cases, in addition to the baseline SSOR preconditioner which is shown for comparison. Here the 'practical' variant discussed in Section 5.2 is used in which the dropping is performed on the L and U factors after the upates instead of on the G matrices. The curves labeled ITALUF(droptol,lfil) are self-explanatory and are similar to the previous test. The last curve ITALUF(0.1,2*10) shows what happens if two iterations are performed at the last restart instead of just one.

5.4. Tests with Harwell-Boeing matrices. We also tested the procedures developed in this paper on well-known test matrices from the Harwell-Boeing collection. Our first test is with the linear system Sherman5 ($n = 3312, nnz = 20793$) which arises from reservoir simulation. We redid the same test as in Section 5.1. SSOR was the base preconditioner. GMRES(30) was tested with SSOR which converged in 55 steps. Then the iterative LU was used to obtain a preconditioner from the L_0, U_0 factors of SSOR using one step of Algorithm 4 with the parameters droptol=0.2, lfil=15. GMRES(30) was tested with this preconditioner and converged in 27 iterations. Then a third preconditioner was obtained with one iteration of Algorithm 4 with the parameters droptol=0.05, lfil=15. GMRES(30) converged in 20 steps. This was repeated one more time with the parameters droptol=0.001, lfil=30. GMRES(30) converged in 15 steps. Finally ILUT from Matlab was tested with a tolerance of 0.0001. Note that larger values yielded poor convergence or non-convergence.

We also tested the Frobenius norm approach for the same problem. While the method worked we noticed that the number of GMRES iterations obtained did not fall as quickly as with the other approaches. We found that this was due to the poor scaling of the matrix. We scaled the rows of the matrix by their 2-norms and then proceeded to scale the columns also by their 2-norms. The right-hand side was scaled accordingly. The results are shown on the right side of Figure 5.4.

6. An application in an evolutive computational fluid dynamics problem. This section discusses applications of the methods proposed in this paper to the numerical simulation of the variable density incompressible Navier-Stokes system given on a domain $\Omega \subset \mathbb{R}^2$. These equations model the motion of two

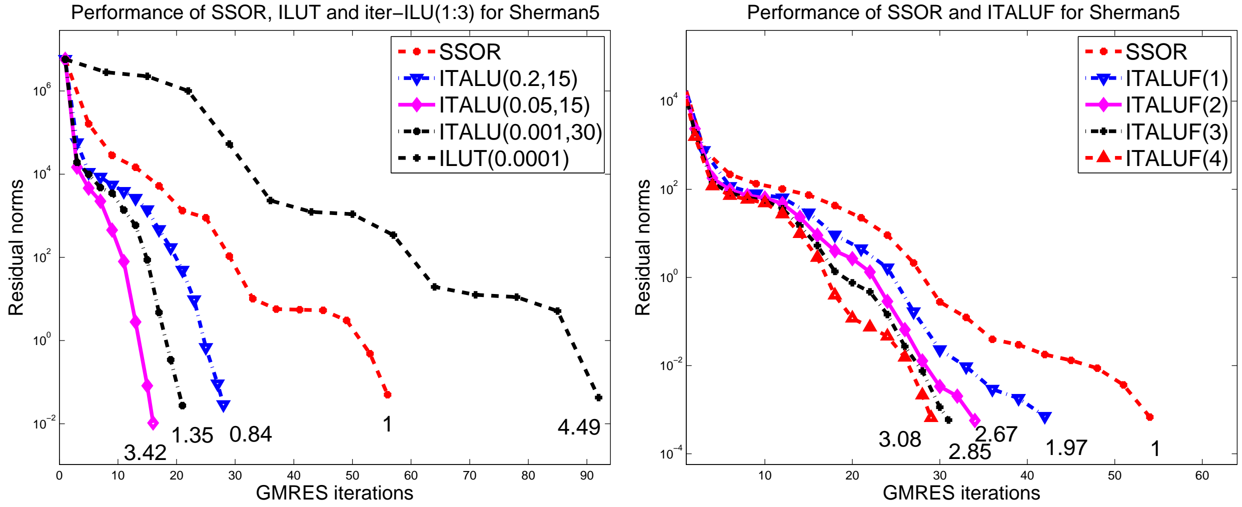

FIG. 5.4. *Performance of GMRES(30) for Sherman-5 problem with the SSOR and ILUT preconditioners the iterative ILU preconditioners. The right side plot uses the Frobenius approach.*

immiscible fluids with varying density, evolving in a domain Ω.

6.1. Governing equations and discrete problems. The variable density incompressible Navier-Stokes system in $\Omega \subset \mathbb{R}^2$ can be written as:

$$\partial_t \rho + \mathrm{div}_{\mathbf{x}}(\rho \mathbf{u}) = 0, \tag{6.1}$$

$$\rho\big(\partial_t \mathbf{u} + (\mathbf{u} \cdot \nabla_{\mathbf{x}})\mathbf{u}\big) + \nabla_{\mathbf{x}} p - \mu \Delta_{\mathbf{x}} \mathbf{u} = \mathbf{f}, \tag{6.2}$$

$$\mathrm{div}_{\mathbf{x}} \mathbf{u} = 0. \tag{6.3}$$

Here $\rho(t, \mathbf{x}) \geq 0$ stands for the density of a viscous fluid whose velocity field is $\mathbf{u}(t, \mathbf{x}) \in \mathbb{R}^2$. The description of the external force is embodied into the right hand side $\mathbf{f}(t, \mathbf{x})$ of (6.2) and $\mu > 0$ is the viscosity of the fluid. The unknowns depend both on time $t \geq 0$ and position $\mathbf{x} \in \Omega \subset \mathbb{R}^2$. The third unknown of the problem is the pressure $p(t, \mathbf{x}) \in \mathbb{R}$; it can be seen as a Lagrange multiplier associated to the incompressibility constraint (6.3). The equations are completed by initial and boundary conditions :

$$\begin{cases} \mathbf{u}_{|\partial\Omega} = \mathbf{g}, & \mathbf{u}_{|t=0} = \mathbf{u}_0, \\ \rho_{|\Gamma_{\mathrm{inc}}} = \rho_{\mathrm{inc}}, & \rho_{|t=0} = \rho_0, \end{cases}$$

where $\mathbf{g}$ and $\rho_{\mathrm{inc}} > 0$ are respectively the velocity and the density prescribed on the boundary, whereas $\mathbf{u}_0$ and ρ_0 are the initial velocity and density. Finally, $\Gamma_{\mathrm{inc}} = \{\mathbf{x} \in \partial\Omega \mid \mathbf{g}(\mathbf{x}) \cdot \mathbf{n}(\mathbf{x}) < 0\}$, where $\mathbf{n}(\mathbf{x})$ denote the outer unit normal vector at $\mathbf{x} \in \partial\Omega$. It is classical that the boundary condition can be transformed into a forcing term so that, without loss of generality, we assume in what follows $\mathbf{g} = 0$.

In [7], different numerical methods have been proposed for the transport equation (6.1) and for evaluating the evolution of the velocity driven by (6.2), (6.3). To be more specific, thanks to a time-splitting, it is possible to solve (6.1) for a given velocity by using a Finite Volume approach (see e.g. [18, 24]) which is efficient when dealing with a pure convection equation and then, compute the divergence free solution of the momentum equation (6.2) by exploiting the advantages of Finite Element (FE) methods (see e.g. [20, 17]). In particular, for the compatibility of the two approaches, the numerical scheme must preserve the divergence free constraint between the two steps of the splitting.

In particular, the domain Ω is approximated by a computational domain Ω_h, discretized by a conforming and isotropic set of triangles $\mathcal{T}_h$, with mesh-size h. The FE spaces $V_h \subset (H_0^1(\Omega_h))^2$ for the velocity $\mathbf{u}_h$ and $Q_h \subset L_0^2(\Omega_h)$ for the pressure p_h must verify the "inf-sup" or LBB condition. In what follows we shall use the $\mathbb{P}_2/\mathbb{P}_1$ elements in order to obtain a "saddle point problem" corresponding to the momentum equation

(6.2–6.3). With this choice, we define

$$V_h = \{\mathbf{v}_h \in \mathcal{C}^0(\bar{\Omega}_h) \mid \mathbf{v}_{h|_K} \in \mathbb{P}_2(K) \;\; \forall K \in \mathcal{T}_h \text{ and } \mathbf{v}_{h|_{\partial\Omega_h}} = 0\},$$
$$\bar{Q}_h = \{q_h \in \mathcal{C}^0(\bar{\Omega}_h) \mid q_{h|_K} \in \mathbb{P}_1(K) \;\; \forall K \in \mathcal{T}_h\},$$
$$Q_h = \{q_h \in \bar{Q}_h \mid \int_{\Omega_h} q_h = 0\}.$$

We also need to define a suitable discrete approximation of the (given) function ρ, compatible with the discretization of the velocity and the pressure. A natural choice is merely $\rho_h \in \bar{Q}_h$.

Let us denote by Δt the time step and $t^n = n\,\Delta t$, $n \geq 0$. We assume that the numerical solution at time t^n, namely (ρ^n, $\mathbf{u}^n$, p^n), is known on the computational domain. The time splitting of the system (6.1–6.2–6.3) is known as the "Strang splitting" [7, 30].

For the time discretization of (6.2), (6.3), we use a classical backward difference method for the time derivative and a semi-implicit linearization scheme to treat the nonlinear convection term in the momentum equation (6.2). This scheme, already used in [23], has a second-order accuracy in time. Specifically, given the approximations $\mathbf{u}^n, \mathbf{u}^{n-1}$ and ρ^* ($\rho^* = \rho^n$ or ρ^{n+1} depending on the time step of the Strang splitting), at time t^{n+1} we compute $(\mathbf{u}^{n+1}, p^{n+1})$ by solving

$$\rho^* \left(\frac{3\mathbf{u}^{n+1} - 4\mathbf{u}^n + \mathbf{u}^{n-1}}{2\Delta t} + (\bar{\mathbf{u}}^{n+1} \cdot \nabla_{\mathbf{x}})\mathbf{u}^{n+1} \right) - \mu \Delta_{\mathbf{x}} \mathbf{u}^{n+1} + \nabla_{\mathbf{x}} p^{n+1} = \mathbf{f}^{n+1}, \tag{6.4}$$

$$\mathrm{div}_{\mathbf{x}} \mathbf{u}^{n+1} = 0. \tag{6.5}$$

Here, $\bar{\mathbf{u}}^{n+1} = 2\mathbf{u}^n - \mathbf{u}^{n-1}$ is the linear second-order extrapolation of the velocity field at the new time t^{n+1}.

The algebraic version of discrete variational form of (6.4), (6.5) may be written in the following saddle point problem:

$$\begin{pmatrix} \mathbf{A}_{n+1} & \mathbf{B}^T \\ \mathbf{B} & \mathbf{0} \end{pmatrix} \begin{pmatrix} U \\ P \end{pmatrix} = \begin{pmatrix} F \\ 0 \end{pmatrix}, \tag{6.6}$$

where $\mathbf{A}_{n+1}$ is a nonsymmetric matrix associated with the convection-diffusion term, $\mathbf{B}$ is the matrix associated with the divergence operator, and the vectors U, P, F correspond, respectively, to the unknowns $\mathbf{u}_h^{n+1}, p_h^{n+1}$ and to the given function $\mathbf{f}_h^{n+1}$ plus other explicit terms depending on the velocity. Here, $\mathbf{A}_{n+1}$ changes from one time step to the next one, due to the evolution of the density and to the linearization of the convective term. More specifically, we have $\mathbf{A}_{n+1} = \frac{3}{2\Delta t}\mathbf{M} + \frac{1}{Re}\mathbf{L} + \mathbf{D}$, where $\mathbf{M}$ is the velocity mass matrix depending on ρ_h^*, $\mathbf{L}$ is the Laplacian matrix for the velocity and the matrix $\mathbf{D}$ corresponds to the convective term and depends on ρ_h^* and $\bar{\mathbf{u}}_h^{n+1}$. Tipically, the matrices $\mathbf{M}$ and $\mathbf{D}$ change at each time step and are not symmetric. Finally, we recall that the Reynolds number Re is proportional to the size of the domain, a reference density and a reference velocity and inversely proportional to the dynamic viscosity μ.

In the case of homogeneous density $\rho(t, \mathbf{x}) = \overline{\rho}$, $\forall t > 0$, the saddle point problem (6.6) has been studied in many papers, see e.g. [8] and references therein. In these references block diagonal or block triangular preconditioning strategies are advocated for the system matrix of (6.6). The eigenvalue analysis in these references shows that the eigenvalues of the preconditioned system are bounded independently of the mesh size h of the underlying grid $\mathcal{T}_h$ and numerical tests show that the convergence rates for GMRES iterations deteriorates roughly like the Reynolds number Re.

Typically, here we are faced with the problem of solving each consecutive system (6.6) by taking advantage of earlier systems. In this section, we study the influence of the alternating correction, Algorithm 4, as preconditioner of the nonsymmetric matrix $\mathbf{A}_{n+1}$, in the case of a block triangular preconditioner applied to the saddle point problem. Block triangular preconditioners are given by

$$\begin{pmatrix} \widehat{\mathbf{A}} & \mathbf{B}^T \\ \mathbf{0} & \widehat{\mathbf{S}} \end{pmatrix} \text{ or } \begin{pmatrix} \widehat{\mathbf{A}} & \mathbf{0} \\ \mathbf{B} & \widehat{\mathbf{S}} \end{pmatrix},$$

where $\widehat{\mathbf{A}}$ is an approximation of the nonsymmetric matrix $\mathbf{A}_{n+1}$ and $\widehat{\mathbf{S}}$ is an approximation of the pressure Schur complement $\mathbf{B}\mathbf{A}_{n+1}^{-1}\mathbf{B}^T$. Since FGMRES, the flexible variant of GMRES, allows variations in the

preconditioner, it is used as the accelerator in our implementation. In FGMRES, preconditioning is applied to the right:

$$\begin{pmatrix} \mathbf{A}_{n+1} & \mathbf{B}^T \\ \mathbf{B} & \mathbf{0} \end{pmatrix} \begin{pmatrix} \widehat{\mathbf{A}} & \mathbf{B}^T \\ \mathbf{0} & \widehat{\mathbf{S}} \end{pmatrix}^{-1} \begin{pmatrix} W_U \\ W_P \end{pmatrix} = \begin{pmatrix} F \\ 0 \end{pmatrix}, \text{ with } \begin{pmatrix} U \\ P \end{pmatrix} = \begin{pmatrix} \widehat{\mathbf{A}} & \mathbf{B}^T \\ \mathbf{0} & \widehat{\mathbf{S}} \end{pmatrix}^{-1} \begin{pmatrix} W_U \\ W_P \end{pmatrix}.$$

We focus our study only on the preconditioner $\widehat{\mathbf{A}}$. For the approximation of the pressure Schur complement $\mathbf{B}\mathbf{A}_{n+1}^{-1}\mathbf{B}^T$, we choose two classical preconditioners : the mass pressure preconditioner [29] $\widehat{\mathbf{S}}^{-1} = \mathbf{M}_p^{-1}$ and the Cahouet-Chabard preconditionner [6] $\widehat{\mathbf{S}}^{-1} = \frac{2\Delta t}{3}\mathbf{M}_p^{-1} + Re\mathbf{L}_p^{-1}$, where $\mathbf{M}_p$ is the mass matrix for the pressure and $\mathbf{L}_p$ is the discretization of the Laplace operator for the pressure. In particular, we will evaluate the abilities of the scheme to recover analytical solution, checking the rates of convergence.

6.2. The ITALU algorithm for saddle point problems. The efficiency of a preconditioner for the saddle point problem (6.6) can be tested numerically by verifying that the convergence rates for FGMRES iterations depend only mildly on the Reynolds number and, most importantly, that the number of iterations does not grow as the mesh size is reduced.

We want to evaluate the ability of the ITALU algorithm to recover analytical solution, comparing the mean number of iterations required by standard ILUT preconditioning (using Matlab `luinc` function with droptol $= 10^{-5}$) with those required by the alternating iterations ITALU, with the same droptol and increasing the lfil and itlu parameters when h decreases. We check for each test the rates of convergence of the velocity and pressure. We consider the example given in [7] by :

$$\begin{cases} \rho_{\text{ex}}(t,x,y) &= 2 + x\cos(\sin t) + y\sin(\sin t), \\ \mathbf{u}_{\text{ex}}(t,x,y) &= \begin{pmatrix} -y\cos t \\ x\cos t \end{pmatrix}, \\ p_{\text{ex}}(t,x,y) &= \sin x \sin y \sin t. \end{cases} \tag{6.7}$$

The fields $\rho_{\text{ex}}(t,x,y)$ and $\mathbf{u}_{\text{ex}}(t,x,y)$ satisfy the mass conservation equation (6.1) identically and $\mathbf{u}_{\text{ex}}(t,x,y)$ is solenoidal. The momentum equation (6.2) is satisfied with the body force defined by

$$\mathbf{f}_{\text{ex}}(t,x,y) = \begin{pmatrix} (y\sin t - x\cos^2 t)\rho_{\text{ex}}(t,x,y) + \cos x \sin y \sin t \\ -(x\sin t + y\cos^2 t)\rho_{\text{ex}}(t,x,y) + \sin x \cos y \sin t \end{pmatrix}. \tag{6.8}$$

The computations are been performed for $0 \le t \le 0.5$ on the square $]-1,1[^2$, using a structured mesh $\mathcal{T}_h$. The convergence results are displayed with respect to h and for different time steps Δt. Table 6.1 shows the mesh size, the corresponding dimensions of vectors U and P and the number of nonzero entries in the matrix $\mathbf{A}_{n+1}$, which is block-diagonal.

TABLE 6.1
Dimensions of the problem with respect to the mesh size.

h	*size*(U)	*size*(P)	*nnz*$(\mathbf{A}_{n+1})$
$h_1 = 0.0312$	2×916	289	$2 \times 10\ 289$
$h_2 = 0.0156$	2×3969	1089	$2 \times 44\ 081$
$h_3 = 0.0078$	2×16129	4225	$2 \times 182\ 321$

For the operation with the matrix $\widehat{\mathbf{S}}^{-1}$, we always use the L and U factors obtained from `luinc` the Matlab version of ILUT, with the droptol set to `droptol=` 10^{-3} and no pivoting. When the mass pressure preconditioner is used, the results are equivalent if we compute $\widehat{\mathbf{S}}^{-1}$ with the iterative ITALU with the same droptol, lfil=10 and itlu=1. This choice is not optimal in the case of the Cahouet-Chabard preconditionner [6].

The results in Figure 6.1 correspond to the mass pressure preconditioner $\widehat{\mathbf{S}}^{-1} = \mathbf{M}_p^{-1}$ for the saddle point problem, when $Re = 1$ and $h = h_1, h_2, h_3$. In order to compare the results of a standard approach and the new procedure presented in section 4, algorithms are used for operations with the matrix $\widehat{\mathbf{A}}$: the

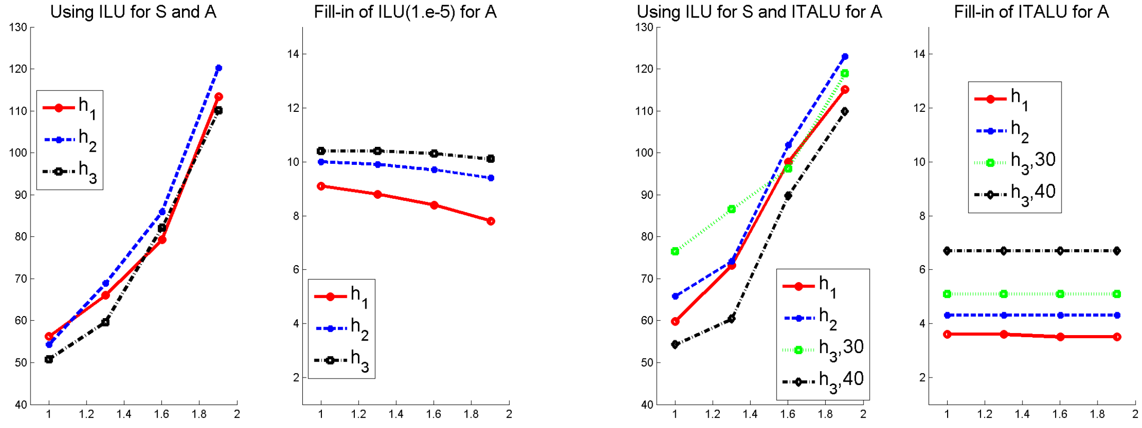

FIG. 6.1. *Mean number of iterations required by FGMRES(30) for the saddle point problem using mass pressure preconditioner for the Schur complement, for different values of h and $Re = 1$. The mean value of the fill-ratio of the preconditioner $\widehat{\mathbf{A}}$ is displayed in each case. On the horizontal axis is $-log_{10}(\Delta t)$. The left plots show performances of the Matlab* `luinc`*(drop) preconditioning and the left plots those of the ITALU procedure.*

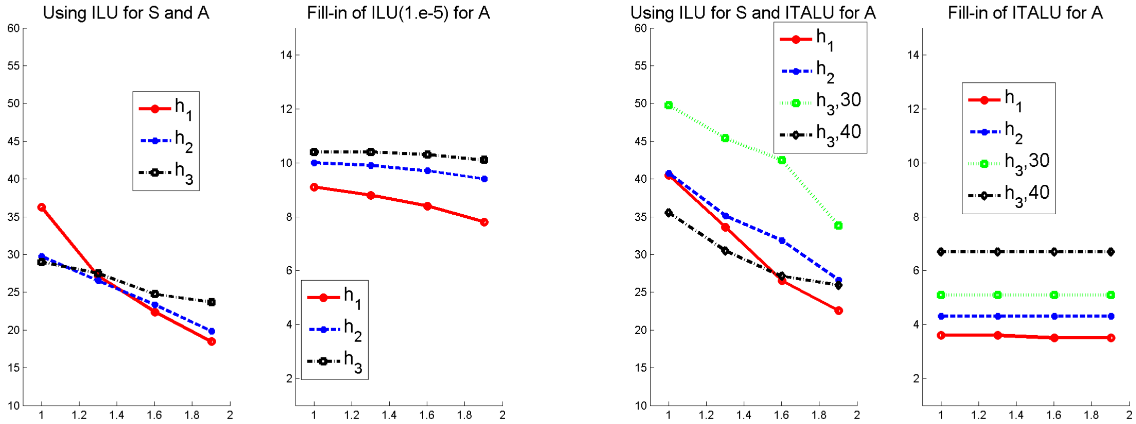

FIG. 6.2. *Mean number of iterations required by FGMRES(30) for the saddle point problem using the Cahouet-Chabard preconditioner for the Schur complement, for different values of h and $Re = 1$. The mean value of the fill-ratio of the preconditioner $\widehat{\mathbf{A}}$ is displayed in each case. On the abscissa, $-log_{10}(\Delta t)$. Comparison between the Matlab ILU(drop) preconditioning on the left plots and the ITALU procedure on the right plots.*

first one is the standard ILU preconditioner of Matlab `luinc`, by setting droptol $= 10^{-5}$ and no pivoting. The results are shown in the left side of Figure 6.1. The second one is the ITALU procedure with the same droptol. It seems obvious that the parameters lfil and itlu must not be fixed, and should vary according to the mesh size h. In particular, we set lfil = 20,25,30 (or 40) and itlu = 3,4,5 respectively, when the mesh size h decreases. Smaller values of lfil or itlu, result in a greater number of FGMRES iterations on average. The results are shown on the right side of Figure 6.1. Near $h = h_3$ we indicate the lfil used. As expected, we can observe that the lfil parameter is very important to obtain a rate of convergence independent of the mesh size.

Using the same values for h, we test the Cahouet-Chabard preconditionner $\widehat{\mathbf{S}}^{-1} = \frac{2\Delta t}{3}\mathbf{M}_p^{-1} + Re\mathbf{L}_p^{-1}$, for the saddle point problem, when $Re = 1$ and $Re = 1000$. This choice is necessary because for higher Reynolds numbers the performances of the mass pressure preconditioner for the Schur complement are not satisfactory.

The results in Figure 6.2 (resp. Figure 6.3) correspond to $Re = 1$ (resp. $Re = 1000$). The results in the left side of these figures correspond to the standard ILUT preconditioner and those in the right side to the

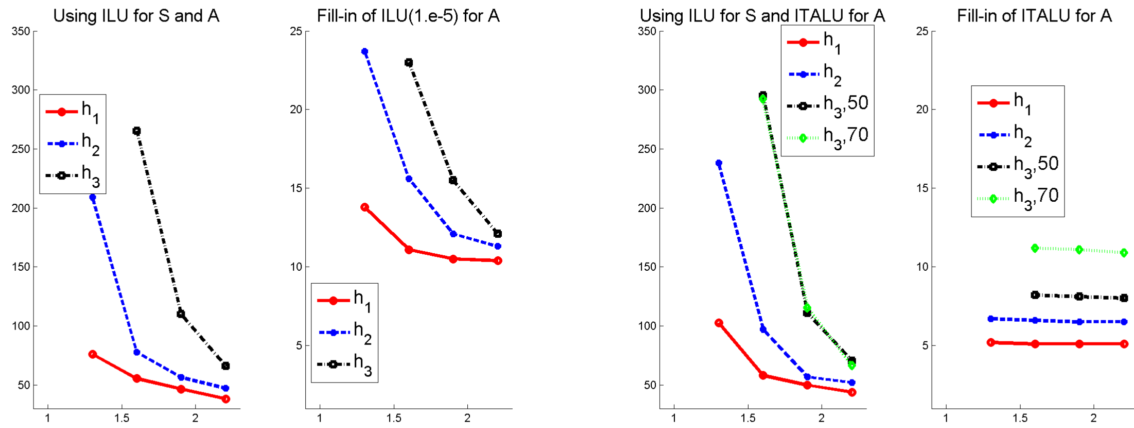

Fig. 6.3. *Mean number of iterations required by FGMRES(30) for the saddle point problem using the Cahouet-Chabard preconditioner for the Schur complement, for different values of h and $Re = 1000$. The mean value of the fill-ratio of the preconditioner $\widehat{\mathbf{A}}$ is displayed in each case. On the abscissa, $-log_{10}(\Delta t)$. Comparison between the Matlab ILU(drop) preconditioning on the left plots and the ITALU procedure on the right plots.*

ITALU preconditioner. If $Re = 1$, we take the same parameters droptol, lfil and itlu as in the previous test case. As in the preceding case, the importance of the lfil parameter can be observed for small values of h. With higher Reynolds numbers, we must increase the parameters lfil and itlu in the ITALU preconditioner. In particular, we set lfil = 30,40,50 (or 70) and itlu = 3,4,5 respectively, when the mesh size h decreases. In this case, choosing higher lfil value does not reduce the mean number of iterations of FGMRES; the only consequence is an increased fill-in of $\widehat{\mathbf{A}}$.

Overall, we stress that the fill-in of $\widehat{\mathbf{A}}$ is much smaller when considering the ITALU procedure presented in Section 4 with respect to the classical approach, in particular when Re is very large. However, to capture correctly the dynamics with higher Reynolds numbers, only small time steps Δt must be chosen and in this case the performances of FGMRES are equivalent.

7. Conclusion.

Though much is known about finding effective preconditioners to solve general sparse linear systems which arise in real-life applications, little has been done so far to address the issue of updating such preconditioners. This paper took an in-depth look at the problem and proposed a few possible practical approaches. The LU factorization and its approximate ILU variants, do not lend themselves easily to updates and this has lead practitioners to simply recompute the preconditioner when a new one is needed, or use an outdated one at the cost of poor convergence. One option is to utilize techniques based on sparse approximate inverses. Though these methods are known not to perform well in practice, they can yield very inexpensive ways to correct LU factors. We have also shown a sparse update technique which can be viewed as a compromise between all these alternatives. Experiments show that the method can perform quite well. From the point of view of efficient practical implementation, some work remains to be done. Here the main ingredient in the construction of ITALU preconditioners is a sparse solve with a sparse right-hand side matrix. Techniques for such tasks have been exploited in the past. It is known for example, that computations of this type often constitute the innermost kernels of sparse direct solvers, see, e.g., [15, 19, 31]. The methods related to approximate inverse techniques require products of sparse matrices with sparse vectors and this too has been exploited in the literature, see, e.g., [14].

REFERENCES

[1] M. Benzi, J. C. Haws, and Tůma, *Preconditioning highly indefinite and nonsymmtric matrices*, SIAM Journal on Scientific Computing, 22 (2000), pp. 1333–1353.

[2] M. Benzi, C. D. Meyer, and M. Tůma, *A sparse approximate inverse preconditioner for the conjugate gradient method*, SIAM Journal on Scientific Computing, 17 (1996), pp. 1135–1149.

[3] M. Benzi and M. Tůma, *A sparse approximate inverse preconditioner for nonsymmetric linear systems*, SIAM Journal on Scientific Computing, 19 (1998), pp. 968–994.

[4] P. Birken, J. D. Tebbens, A. Meister, and M. Tuma, *Preconditioner updates applied to CFD model problems*, Applied Numerical Mathematics, (2008). In press.
[5] M. Bollhöfer, *A robust ILU with pivoting based on monitoring the growth of the inverse factors*, Linear Algebra and its Applications, 338 (2001), pp. 201–213.
[6] J. Cahouet and J.-P. Chabard, *Some fast 3D finite element solvers for the generalized Stokes problem.*, Int. J. Numer. Methods Fluids, 8 (1988), pp. 865–895.
[7] C. Calgaro, E. Creusé, and T. Goudon, *An hybrid finite volume-finite element method for variable density incompressible flows*, J. Comput. Phys., 227 (2008), pp. 4671–4696.
[8] C. Calgaro, P. Deuring, and D. Jennequin, *A preconditioner for generalized saddle-point problems: Application to 3d stationary Navier-Stokes equations*, Numer. Methods Partial Differential Equations, 22 (2006), pp. 1289–1313.
[9] S. M. Chan and V. Brandwajn, *Partial matrix refactorization*, IEEE trans. Power Systems, (1986), pp. 193–199.
[10] J.-P. Chehab, *Matrix differential equations and inverse preconditioners*, Computational and Applied Mathematics, 26 (2007), pp. 95–128.
[11] ———, *Sparse matrix approximation via matrix differential equations*, 2008. Submitted.
[12] J.-P. Chehab and J. Laminie, *Differential equations and solution of linear systems*, Numerical Algorithms, 40 (2005), pp. 103–124.
[13] E. Chow and Y. Saad, *Experimental study of ILU preconditioners for indefinite matrices*, Journal of Computational and Applied Mathematics, 86 (1997), pp. 387–414.
[14] ———, *Approximate inverse preconditioners via sparse-sparse iterations*, SIAM Journal on Scientific Computing, 19 (1998), pp. 995–1023.
[15] T. A. Davis, *Direct methods for sparse linear systems*, SIAM, Philadelphia, PA, 2006.
[16] H. C. Elman, *A stability analysis of incomplete LU factorizations*, Mathematics of Computation, 47 (1986), pp. 191–217.
[17] A. Ern and J.-L. Guermond, *Éléments finis: théorie, applications, mise en œuvre*, vol. 36 of Mathématiques & Applications (Berlin) [Mathematics & Applications], Springer-Verlag, Berlin, 2002.
[18] R. Eymard, T. Gallouët, and R. Herbin, *Finite volume methods*, in Handbook of numerical analysis, Vol. VII, Handb. Numer. Anal., VII, North-Holland, Amsterdam, 2000, pp. 713–1020.
[19] J. R. Gilbert and T. Peierls, *Sparse partial pivoting in time proportional to artithmetic operations*, SIAM Journal on Scientific Computing, 9 (1988), pp. 862–874.
[20] V. Girault and P.-A. Raviart, *Finite element methods for Navier-Stokes equations*, vol. 5 of Springer Series in Computational Mathematics, Springer-Verlag, Berlin, 1986. Theory and algorithms.
[21] M. Grote and H. D. Simon, *Parallel preconditioning and approximate inverses on the connection machine*, in Parallel Processing for Scientific Computing – vol. 2, R. F. Sincovec, D. E. Keyes, L. R. Petzold, and D. A. Reed, eds., SIAM, 1992, pp. 519–523.
[22] M. J. Grote and T. Huckle, *Parallel preconditionings with sparse approximate inverses*, SIAM Journal on Scientific Computing, 18 (1997), pp. 838–853.
[23] J.-L. Guermond and L. Quartapelle, *A projection FEM for variable density incompressible flows*, J. Comput. Phys., 165 (2000), pp. 167–188.
[24] R. J. LeVeque, *Finite volume methods for hyperbolic problems*, Cambridge Texts in Applied Mathematics, Cambridge University Press, Cambridge, 2002.
[25] N. Li, Y. Saad, and E. Chow, *Crout versions of ILU for general sparse matrices*, SIAM Journal on Scientific Computing, 25 (2003), pp. 716–728.
[26] Y. Saad, *ILUT: a dual threshold incomplete ILU factorization*, Numerical Linear Algebra with Applications, 1 (1994), pp. 387–402.
[27] ———, *Iterative Methods for Sparse Linear Systems, 2nd edition*, SIAM, Philadelpha, PA, 2003.
[28] Y. Saad and B. Suchomel, *ARMS: An algebraic recursive multilevel solver for general sparse linear systems*, Numerical Linear Algebra with Applications, 9 (2002).
[29] D. Silvester, H. Elman, D. Kay, and A. Wathen, *Efficient preconditioning of the linearized Navier-Stokes equations for incompressible flow.*, Journal of Computational and Applied Mathematics, 128 (2001), pp. 261–279.
[30] G. Strang, *On the construction and comparison of difference schemes*, SIAM J. Numer. Anal., 5 (1968), pp. 506–517.
[31] W. F. Tinney, V. Brandwajn, and S. M. Chan, *Sparse vector methods*, IEEE trans. Power Apparatus and Systems, (1985), pp. 295–301.
[32] H. A. van der Vorst, *Iterative solution methods for certain sparse linear systems with a non-symmetric matrix arising from PDE-problems*, J. Comp. Phys., 44 (1981), pp. 1–19.