Preconditioning techniques for nonsymmetric indefinite linear systems[*]

Youcef Saad
Center for Supercomputing Research and Development
University of Illinois at Urbana-Champaign
Urbana, Illinois 61801

December 29, 1992

Abstract

The standard preconditioning techniques for conjugate gradient methods often fail for matrices that are indefinite and/or strongly nonsymmetric. The most common alternatives considered for these cases are either to use expensive direct solvers or to resort to one of many techniques based on the normal equations. This paper examines several such alternatives and compares them. In particular an incomplete LQ factorization is proposed and some of its implementation details are described. A number of experiments are reported to compare these methods.

1 Introduction

Despite the tremendous efforts devoted in the last few decades to the development of numerical methods for solving general large sparse linear systems, the current state of the art in iterative methods remains unsatisfactory in many respects. It is often observed that iterative methods are still not as commonly used as direct methods in large production codes despite their appeal for large two or three-dimensional simulations. The main weakness of iterative solvers is their poor robustness, or rather their narrow range of applicability. Often, a particular iterative solver may be found to be very effective for a specific class of problems or for certain conditions on the coefficient matrix. However, when this iterative solver is integrated into a large production or research package, the code can encounter "bad" cases for which the iterative method will either not converge, be excessively slow or simply break down. Sometimes, the failure to converge is not a

[*]Research supported by the National Science Foundation under Grants No. US NSF-MIP-8410110 and US NSF DCR85-09970, the US Department of Energy under Grant No. DOE DE-FG02-85ER25001, by the US Air Force under Contract AFSOR-85-0211, and the IBM donation.

big problem because the iterative solver is a part of another iterative loop such a Newton scheme, and the occasional misbehavior of the solver may not ruin the convergence of the overall process. It may, on the other hand, become a more serious problem if the linear system solver fails systematically as might happen in the case of Newton's method when the Jacobian at the solution point is indefinite, and a CG like scheme is used.

Thus, the main weakness of iterative solvers as compared with direct solvers is that they are not robust enough to handle as wide a class of matrices. The interesting question as to whether there might exist an iterative solver that could handle effectively any linear system of a given type, more effectively than the best available direct solver remains open. Clearly, there are problems where direct methods will always be preferred, such as narrow banded systems or systems whose matrix has a graph that is close to that of a tree [4]. For a large class of problems, namely all those originating from three-dimensional Partial Differential Equations, iterative solvers may be far more effective than direct solvers. However, much work remains to be done to improve the robustness of existing methods and to develop other ones that can handle a broader range of matrices than those than can be treated by currently available techniques.

In this paper we consider a class of linear systems that have been given little attention in the past, namely nonsymmetric and indefinite systems. What is often meant by an indefinite linear system is one whose coefficient matrix A has a symmetric part $\frac{1}{2}(A+A^T)$ that is not positive definite. This class of problems has been identified by many authors as the hardest one to handle with iterative methods. Indefinite problems arise in many areas of scientific computing, one of the best known examples being perhaps the one arising from the Helmhotz equation $\Delta u + k^2(x,y)u = f$, which occurs in different forms in various models of wave propagation [7]. When discretizing this equation we obtain a symmetric linear system that is often not positive definite. Another important example is that of least squares problems with constraints [2]. Most important, is the fact that in practice, many linear systems that arise in the course of a Newton type iteration may occasionally be indefinite to some degree, during the process.

Among the iterative methods available, Krylov subspace methods offer probably the best potential for developing general purpose iterative linear system solvers. These methods may be unacceptably slow if they are used without preconditioning and the question that arises is how to adapt the standard preconditionings for indefinite linear systems. The classical preconditionings will in general fail for strongly indefinite systems not only because they might break down, such as when encountering a zero pivot, but also because the quality of the approximation obtained from the incomplete factorization may be poor. In fact it may be argued that any incomplete factorization for an indefinite problem may face this difficulty. Considering an incomplete factorization of the form

$$A = LU + E \tag{1}$$

where E is the error, then the matrices of the various forms of the preconditioned linear system are similar to

$$L^{-1}AU^{-1} = I + L^{-1}EU^{-1}. \tag{2}$$

When the matrix A is diagonally dominant then L and U are well conditioned, and the eigenvalues of $L^{-1}EU^{-1}$ remain confined within reasonable limits, typically with a nice clustering around the origin. On the other hand, when the original matrix is not diagonally dominant, L^{-1} or U^{-1} may have very large norms, thus causing the error $L^{-1}EU^{-1}$ to have arbitrarily large eigenvalues. This form of unstabilty has been studied by Elman [6] who proposed a detailed analysis of ILU and MILU preconditioners. We also observed experimentally that ILU preconditioners can be very poor when L^{-1} or U^{-1} are large, and that this situation often occurs for indefinite problems, and for problems with large nonsymmetric parts. As a result of this limitation we will not attempt, in this paper, to present a general purpose preconditioner, but only to extend slightly the applicability of the preconditioned conjugate gradient methods by proposing a few alternatives and improvements to the standard ILU preconditioners.

The emphasis of this paper is on implementation and experimentation rather than on theory. We will examine a few preconditioners, show how to implement them for general sparse linear systems, and finally compare them on a few sample problems. The preconditioners considered are an incomplete LQ preconditioner, an incomplete LU factorization with column pivoting, an SSOR preconditioner on the normal equations, and the standard Incomplete Choleski Conjugate Gradient on the normal equations.

2 The Incomplete LQ (ILQ) factorization

2.1 Principle of the method

Consider a general sparse matrix A whose (nonzero) rows are $a_1, a_2, ..., a_N$. The standard complete LQ factorization would consist of finding a lower triangular matrix L and an orthogonal matrix Q such that $A = LQ$. This can be achieved in a number of different ways. George and Heath for example propose to compute the Choleski factorization of $B = AA^T$. If $B = LL^T$ is the Choleski decomposition of B then the lower triangular matrix L is identical with that of the LQ factorization of A. This requires forming the matrix AA^T which may be much denser than A but reordering techniques can be used to reduce the amount of work in forming L. We will refer to this as symmetric squaring.

Another way of proceeding is to use a Gram-Schmidt process. This approach may seem undesirable at first because of its poor numerical properties when it comes to orthogonalizing a large number of vectors. However, because of sparsity, any given row of A will be typically orthogonal to most of the other rows and a result the Gram-Schmidt process is much less prone to numerical difficulties. An advantage of the orthogonalization approach over that of symmetric squaring is its simplicity and its strong similarity with the the LU factorization process: at every step a given row is combined with previous rows and then normalized. To define an *incomplete* factorization we must only include a *dropping* strategy, which will allow to drop elements according to a certain rule, in order to avoid excessive fill-in. In the standard ILU(0) factorization the rule is to drop all fill-in elements immediately after they are introduced at a given elimination step. The resulting ILU factorization has the attractive property that the structure of $L + U$ is the same as that of A. However, relying upon the structure of A only is not safe in cases where the matrix is indefinite. For example, the incomplete LU factorization will break down immediatly if $a_{11} = 0$. Similarly, for the incomplete LQ factorization we need to select a different criterion, which will be based on the magnitude of the elements generated. The simplest idea is to keep the p_L largest elements of a row of L and the p_Q largest elements in a row of Q, where p_L and p_Q are two chosen parameters. The corresponding general procedure would be as follows:

Incomplete LQ factorization
For $i = 2, 3, ..., N$ Do:

1. Compute all *nonzero* inner products $l_{ij} = q_j a_i^T, j = 1, 2, ..., i-1$.
2. Determine the p_L largest elements in $l_{i.}$ and assign a zero value to the others.
3. Compute $\hat{q}_i = a_i - \sum_{j=1, l_{ij} \neq 0}^{j=i} l_{ij} q_j$
4. Determine the p_Q largest elements of $\hat{q}_i$ and assign a zero value to the other elements.
5. Compute $l_{ii} = \|\hat{q}_i\|_2$ and $q_i = \hat{q}_i / l_{ii}$

The first step of the above procedure needs particular attention. If we had to actually compute all the inner products $q_1 a_i^T$ through $q_{i-1} a_i^T$, the total cost of the incomplete factorization would be of the order of N^2 steps and the algorithm would be of little practical value. Note, however, that most of these inner products are equal to zero because of sparsity and the question arises whether or not it is possible to compute these inner products with a much lower cost. The answer is yes. The key observation is that if we call l the column of the $i-1$ inner products l_{ij}, then l is the product of the matrix Q_{i-1} whose rows are $q_1, .., q_{i-1}$ by the vector a_i^T, i.e.,

$$l = Q_{i-1} a_i^T \tag{3}$$

Such a sparse matrix by vector product can be performed in two different ways. The standard way, would be to calculate the inner products $q_j a_i^T$ for $j = 1, ..., i-1$ which is unacceptable as was seen above. The alternative is to compute it as a linear combination of the columns of Q_{i-1}. Let $a_{i,i_1}, a_{i,i_2}, ..., a_{i,i_{k_i}}$ be all the nonzero elements of the $i-th$ row of A. Then the desired column l is given by

$$l = \sum_{j=1}^{k_i} a_{i,i_j} q_{i_j}\prime \tag{4}$$

where $q_j\prime$ represents the $j-$th column of Q_{i-1}. This is far more economical than the first approach because k_i is usually small and the columns of Q_{i-1} are sparse. The total number of multiplications used is equal to the sum of the number of nonzero elements in the columns $i_1, ..., i_{k_i}$.

There is one difficulty with the above implementation of step 1, which is that it requires both the row data structure of Q and of its transpose. A standard way of handling this problem is to build a linked list data structure for the transpose. This avoids the storage of an additional real array for the matrix and simplifies the process of updating the matrix Q, as new rows are obtained. It is important to note that this linked data structure is only used in the preprocessing phase and is discarded once the factors L and Q have been built. For these reasons the scheme is not too uneconomical, although it suffers from not being easily amenable to parallel and vector processing.

After the i-th step is performed we have the following relation

$$\hat{q}_i = l_{ii}q_i + f_i = a_i - \sum_{j=1}^{i-1} l_{ij}q_j \tag{5}$$

where f_i is the row of elements that have been dropped from the row $\hat{q}_i$ in step 4. The above equation translates into

$$A = LQ + E \tag{6}$$

where E is the matrix whose i-th row is f_i, and the notation for L and Q is as before. Typically, the error matrix E is small because of the strategy adopted in dropping elements. One major problem with the above decomposition is that the matrix Q is not orthogonal in general. In fact nothing guarantees that it is even nonsingular unless we make the dropping strategy severe enough as is shown in the next theorem, in which it is assumed that the step 3 of ILQ is replaced by an ideal orthogonalization process in which $\hat{q}_i$ is made orthogonal to the previous q_j's.

Theorem 2.1 *Let Π_i be the orthogonal projector onto the span of the rows $q_1, q_2, ..., q_i$, and assume that at every step of ILQ, the row $\hat{q}_i$ is orthogonal to all previous q_j's. Then the matrix Q_{i+1} is of full rank if and only if $\|\Pi_i f_{i+1}\|_2 < l_{i+1,i+1}$.*

Proof. . A necessary and sufficient condition for $q_1, ..., q_{i+1}$ to be linearly independent, assuming that $q_1, .., q_i$ satisfies this property, is that $(I - \Pi_i)q_{i+1} \neq 0$ or equivalently that $\|\Pi_i q_{i+1}\|_2 < 1$. This is because when $\|(I - \Pi_i)q_{i+1}\|_2 \neq 0$ one can build an orthogonal basis of $span\{Q_{i+1}\}$ from the (nonzero) row $(I - \Pi_i)q_{i+1}$ completed with an orthogonal basis of $span\{Q_i\}$. Conversely, if $q_1, ..., q_{i+1}$ are linearly independent, then we must have $\Pi_i q_{i+1} \neq q_{i+1}$, i.e., $(I - \Pi_i)q_{i+1} \neq 0$.

We observe from (5) that

$$\Pi_i q_{i+1} = \frac{1}{l_{i+1,i+1}} \Pi_i(\hat{q}_{i+1} - f_{i+1}) \tag{7}$$

where $l_{i+1,i+1}$ denotes the norm of $\hat{q}_{i+1} - f_{i+1}$. By our assumption $\hat{q}_{i+1}$ is orthogonal to $q_1, ..., q_i$ and as a result we have

$$\Pi_i q_{i+1} = \frac{-1}{l_{i+1,i+1}} \Pi_i f_{i+1} \tag{8}$$

which yields the result. □

Corollary 2.1 *Assume that at every step $j \leq i$ of ILQ, the row $\hat{q}_j$ is orthogonal to $q_1, \dots, q_{j-1}$ and that the dropped out row at step j is f_j. Then a sufficient condition for the matrix Q_{i+1} to be of full rank, is that $\|f_{i+1}\|_2 < l_{i+1,i+1}$.*

Proof. . The result follows from the theorem by simply observing that $\|\Pi_i f\|_2 \leq \|f\|_2$ for any f. □

These results provide conditions under which the constructed matrix Q_i remains of full rank, and in fact orthogonal because of the assumptions, at every step. One of the weaknesses of the above result is that it assumes that at every step the vector $\hat{q}_i$ is made orthogonal to the previous q's which is not quite achieved by the algorithm. In the form in which it is presented, the algorithm performs a Gram-Schmidt step, assuming that the previous vectors $q's$ are orthogonal which is not the case, in general. Clearly, orthogonalization can be achieved by a more expensive process which would, for example, include as many reorthogonalization steps as necessary to achieve orthogonality. However, this is not practical. In fact the above results are not likely to be useful in practice in view of the following problem. Even if we were to obtain a reasonably simple criterion for ensuring the nonsingularity of the matrix Q, this criterion is likely to conflict with the sparsity requirement: the criterion could be so severe that the only way in which it could be achieved is by making f_i very small, which means allowing more fill-in in the L and Q matrices. The simplest way to handle this difficulty is to use $L^{-1}A$ instead of Q whenever possible, i.e., whenever the matrix Q is not needed explicitly as in preconditioned conjugate gradient methods.

2.2 Implementations in conjugate gradient techniques

There are several ways of exploiting the decomposition (6) as a preconditioner in a conjugate gradient like algorithm. The simplest idea that comes to mind is to form the preconditioned system

$$L^{-1}AQ^Ty = L^{-1}b \tag{9}$$

whose solution y is related to the solution x of the original system by $x = Q^Ty$. The drawback of the above approach is that, as was mentioned above, it is not easy to guarantee that the matrix Q^T is nonsingular. In fact our experiments reveal that this does indeed happen for hard problems.

An alternative is to replace the matrix Q in (9) by its "approximation" $L^{-1}A$, which is known to be nonsingular by construction. This leads to a natural preconditioning of the normal equations $AA^Ty = b$, namely to

$$L^{-1}AA^TL^{-T}y = b \tag{10}$$

The conjugate gradient technique can be applied to the equivalent system (10) and the solution x of the original problem $Ax = b$ is then given by $x = A^TL^{-T}y$. Clearly, there are several other ways of using L to precondition the normal equations $AA^Tx = b$.

For example, an implementation of the preconditioned conjugate gradient method with left preconditioning, applied to the normal equations is as follows.

Algorithm: ILQCG/NE

```
1. Start: Compute $r_0 = b - Ax_0$, $z_0 = Q^{-1}r_0$ $p_0 = A^T r_0$.
2. Iterate: For $i = 0, 2, \ldots,$ until convergence do
    • $y_i = Ap_i$
    • $\alpha_i = (z_i, r_i)/(y_i, y_i)$
    • $x_{i+1} = x_i + \alpha_i p_i$
    • $r_{i+1} = r_i - \alpha_i y_i$
    • $z_{i+1} = (LL^T)^{-1} r_{i+1}$
    • $\beta_i = (z_{i+1}, r_{i+1})/(z_i, r_i)$
    • $p_{i+1} = A^T z_{i+1} + \beta_i p_i$
```

Similarly, one can compute the ILQ factorization of A^T and precondition the linear system $A^T Ax = A^T b$ in a similar way. Ideally, i.e., when the incomplete factorization is mathematically exact then the method will converge in zero step, since the matrix on the left hand side would be the identity matrix.

2.3 Application to least squares problems

An important application of the incomplete LQ factorization is when solving least squares problems of the form:

$$\min_x \|b - Ax\|_2 \tag{11}$$

where A is an $N \times m$ matrix with $m < N$. When A is of full rank the solution to the above problem is the unique solution of the system of normal equations

$$A^T Ax = A^T b. \tag{12}$$

When the matrix A is very large and sparse, it is natural to think of using the conjugate gradient method for solving (12). Moreover, a good preconditioning technique can be provided by the incomplete LQ factorization applied to the matrix A^T. Let $A^T = LQ + E$ the incomplete LQ factorization of A^T. This is nothing but ILQ applied to the columns of A rather than its rows, and it can also be viewed as the incomplete QR factorization of A. Then a preconditioned version of (12) is given by

$$L^{-1}A^T(AL^{-T}y - b) = 0. \tag{13}$$

from which the solution x to (11) can be computed using the relation

$$x = L^{-T}y. \tag{14}$$

In fact there is no obligation to solve the system (13) by a conjugate gradient technique, since we can write the preconditioned version of (11) to which (13) corresponds:

$$\min_{y \in R^m} \|b - AL^{-T}y\|_2 \tag{15}$$

The above problem can be solved by any means to find the unknown y from which x is computed by (14). Clearly, the preconditioned columns AL^{-1} need not be computed explicitly. The effect of post-multiplying A by L^{-1} is to make the columns of the coefficient matrix of the preconditioned least squares problem (15) closer to an orthogonal matrix.

3 Incomplete LU factorization with pivoting

We now briefly discuss an alternative to the incomplete LQ factorization which is a natural extension of the standard incomplete LU factorization. What makes the incomplete LU factorization fail in many cases is the absence of any pivoting strategy. A simple pivoting strategy can be introduced in order to prevent this and the implementation does not require a heavy overhead.

A serious difficulty with any incomplete LU factorization that is based on the structure of A is that at any step we may encounter not only a zero pivot but a whole row of zeros. This can happen in the first step of ILU as is illustrated in the following example

$$\begin{pmatrix} x & x & x & 0 \\ x & 0 & 0 & 0 \\ 0 & 0 & x & 0 \\ x & 0 & 0 & x \end{pmatrix} \tag{16}$$

where x denotes a nonzero element. As is easily seen this matrix is always nonsingular, provided each x is nonzero. If the pivot element in the first step of the is a_{11} then the ILU(0) algorithm produces a second row that is zero because by definition of ILU(0) the fill-in elements created in positions (1,2) and (1,3) are dropped. This break down may be avoided however if more fill-in is allowed. Note that here neither (partial) row or column pivoting helps in any way since both the row and column are zero.

A solution to this difficulty is to drop elements according to their magnitude rather than position. For example a simple strategy is to keep a fixed number, say k_u, of the largest elements (in absolute value) generated in the U part of the row during the elimination. The simplest way to incorporate pivoting in an incomplete factorization code is to perform column (partial) pivoting. In order to keep cost minimal, typical incomplete LU factorizations proceed by rows and don't require the column data structure. As a result interchange of the rows is impractical since it will not only require scanning the column but also exchanging row, i.e., altering the row data structure. With column pivoting this problem does not occur because we only need to keep track of the new ordering of unknowns as they are permuted.

An outline of the main elimination step of our ILU with Pivoting (ILUP) follows. The factorization proceeds row-wise in that one row of L and the same row of U are determined at every step. We use two work vectors w_L and w_U to store the elements of the row of L and U respectively, and initialize them with the corresponding parts of the row a_i. Both w_L and w_U are stored as dense vectors with pointers to determine the column positions of their elements. Fill-ins are easily appended to these work vectors. We will call i_L the number of nonzero elements of a_i that are in the strict lower part of A and i_U the number of those elements located in the upper part. In other words, i_L and i_U represent the initial lengths of w_L and w_U respectively. Two input parameters q_L and q_U determine the number of fill-in elements allowed in every row of L and U respectively. The first part of the elimination process is simply to go through the elements $w_1, ..., w_{i_L}, w_{i_L+1}, ..w_{i_L+q_L}$ and perform the elimination corresponding to these elements. Thus the vectors w_L and w_U will be updated at each elimination step with fill-ins appended to them. Then the second phase is to select the $i_U + q_U$ largest elements in the resulting vector w_U and drop the others. The element of largest magnitude is then exchanged with the diagonal element. Note that the dropping strategy for the elements in L is different from that adopted for U. By the nature of the algorithm adopted, it is the latest fill ins that are dropped from L. This resembles the level of fill-in strategy often used to generalize the ICCG(p) techniques to general matrices [12].

Unfortunately, even with this implementation, the occurrence of zero rows is not uncommon. However, it is our experience that with the above approach and with sufficient fill-in, the zero rows will appear only at the very end of the process, after most of the matrix has already been (incompletely) factored. Therefore, one way of minimizing the difficulty is to quit and use only the available factors as preconditioners, i.e., to complete L and U by identity matrices.

We should mention that the idea of incomplete factorization based on a dropping strategy by magnitude rather than position is not new. For example Harwell's MA28 [4] includes an option to this effect and so does Zlatev's Y12M [13]. These codes incorporate a dropping tolerance strategy whereby elements whose magnitude fall below a certain tolerance factor are dropped as soon as they are generated. The viewpoint taken in these excellent codes however differs substantially from ours: these techniques are primarily direct methods and the drop tolerance is usually so small that a few steps of iterative refinement will suffice to provide a correct answer. Iterative refinement will converge only when the perturbation to A is small and this might require substantially more work than with a limited fill-in type technique such as those considered in iterative methods. Another point is that it is never known beforehand how much storage will be required to get an incomplete factorization with a given drop tolerance.

In summary, the main features of our implementations of our version of incomplete LU factorization with column pivoting are the following.

- Use column pivoting only. A pointer is kept to keep track of the new ordering of the unknowns.

- Drop elements when generating U according to their magnitude instead of the nonzero structure or level of fill-in. Drop elements in L according to their level of fill-in.

- If a zero row appears then complete the L and U matrices by diagonals of unity and quit.

4 CG/SSOR on the normal equations.

4.1 SSOR/NE for general sparse matrices

In this section we will briefly discuss simple implementations of relaxation methods applied to the normal equations of the form

$$A^T A x = A^T b \tag{17}$$

or

$$A A^T y = b. \tag{18}$$

Bjork and Elfving [3] have shown that in order to use relaxation schemes on the normal equations, one only needs to access one column A at a time for (17) and a row at a time for (18). For completeness we now explain how this can be achieved for (18) for example. Starting from an approximation to the solution of (18), a basic relaxation step consists of modifying its components in a certain order by a succession of relaxation steps of the simple form

$$x_{new} = x_{old} + \delta_i e_i \tag{19}$$

where e_i is the $i-th$ column of the identity matrix. The scalar δ_i is chosen so that the $i-th$ component of the residual vector becomes zero. Thus we must have

$$(b - AA^T(x_{old} + \delta_i e_i), e_i) = 0 \tag{20}$$

which yields,

$$\delta_i = \frac{(b, e_i) - (A^T x_{old}, A^T e_i)}{(A^T e_i, A^T e_i)} \tag{21}$$

If we assume for simplicity that the rows of A have been normalized so that

$$\|A^T e_i\|_2 = 1 \tag{22}$$

and denote by f_i the $i-th$ component of f, then a basic relaxation step consists of taking

$$\delta_i = b_i - (A^T x_{old}, A^T e_i) \tag{23}$$

Consider now the implementation of, for example, the Gauss-Seidel procedure based on (19) and (23) for a general sparse matrix. To evaluate δ_i from (23) we need the vector $A^T x_{old}$ and its inner product with the i-th row of A. This inner product is inexpensive to compute because the row $A^T e_i$ is usually sparse. On the other hand the matrix by vector product $A^T x_{old}$ must be computed carefully if we want to avoid performing a matrix by vector multiplication at each relaxation step. This can be achieved by computing the initial $A^T x_{old}$ in the Gauss-Seidel sweep as a full vector z and then each time the approximate solution x_{old} is updated by (19), to update z accordingly. The Gauss-Seidel sweep would therefore be as follows, where a_i is again the i-th row of A.

Algorithm: Gauss-Seidel Sweep for $AA^T x = b$

```
1. Given an initial $x$, compute $z := A^T x$.
2. For $i = 1, 2, .., N$ Do
    (a) Compute $\delta_i = b_i - a_i z$
    (b) Compute $x_i := x_i + \delta_i$
    (c) Compute $z := z + \delta_i a_i^T$
```

All that is needed to implement the above algorithm is the row data structure of A. If we denote by nz_i the number of nonzero elements in the $i - th$ row of A, then each step of the above sweep costs $2nz_i$ operations for (a) , one addition in (b) and another $2nz_i$ operations in (c) bringing the total to $4nz_i + 1$. The total for a whole sweep becomes $4nz + N$ for Gauss-Seidel and twice as much for SSOR($\omega = 1$) where nz represents the total number of number of nonzero elements of A. Storage consists of the right hand side, the vector x and the additional work vector z.

Note that the matrix AA^T can be dense or in general much less sparse than A, yet the cost of the above implementation depends only of the nonzero structure of A. This is not a negligible advantage of relaxation type preconditioners over incomplete factorization preconditioners in the context of Conjugate Gradient methods as is described in the next section.

One question left aside so far concerns the usual acceleration of the above relaxation scheme by under or over-relaxation. If we introduce the usual acceleration parameter ω, then we only have to replace (a) in the above algorithm by

$$\delta_i = \omega(b_i - (z, a_i)) \tag{24}$$

One serious difficulty here is to determine the optimal relaxation factor. If nothing in particular is known about the matrix AA^T all that can be said is that the method will converge for any ω lying strictly between 0 and 2 [11] because the matrix is positive definite. Moreover, another question not addressed here is how can convergence be affected by various reorderings of the rows. When the matrix is issued from an elliptic partial differential equation, one can easily reorder the unknowns so that the matrix A is block tridiagonal. This fact has been exploited by Kamath and Sameh [8] to devise a parallel scheme by grouping rows such that several simultaneous relaxation steps can be performed in parallel, see Section 4.3 for more details.

4.2 Implementations of SSOR/NE Preconditionings

There are several ways of exploiting the relaxation schemes as preconditioners to conjugate gradient methods applied to either (17) or (18). We only consider (18) but the adaptation of the scheme to (17) is straightforward. We need a procedure that delivers an approximation to $(A^TA)^{-1}v$ for any vector v. One such procedure consists of performing one of several steps of SSOR to solve the system $(A^TA)w = v$. If we denote this operator by Q^{-1}, then the usual conjugate gradient method applied to (18), sometimes referred to as Craig's method, with left preconditioning Q can be recast in the form [5].

Algorithm ?: CGNE/SSOR/ Left preconditioning

```
1. Start: Compute $r_0 = b - Ax_0$, $z_0 = Q^{-1}r_0$, $p_0 = A^Tz_0$.
2. Iterate. For $i = 0, ...,$ until convergence do
   (a) $y_i = Ap_i$
   (b) $\alpha_i = (r_i, z_i)/\|p_i\|_2^2$
   (c) $x_{i+1} = x_i + \alpha_i p_i$
   (d) $r_{i+1} = r_i - \alpha_i y_i$
   (e) $z_{i+1} = Q^{-1}r_{i+1}$
   (f) $\beta_{i+1} = (z_{i+1}, r_{i+1})/(z_i, r_i)$
   (g) $p_{i+1} = A^Tz_{i+1} + \beta_i p_i$
```

The equivalent version for the standard system (17) referred to as CGNR is as follows,

CGNR/SSOR Left Preconditioning

```
1. Start: Compute $r_0 = b - Ax_0$, $z_0 = Q^{-1}r_0$, $p_0 = r_0$.
2. Iterate: For $i = 0, ...,$ until convergence do
   (a) $y_i = Ap_i$
   (b) $\alpha_i = (r_i, z_i)/\|y_i\|_2^2$
   (c) $x_{i+1} = x_i + \alpha_i p_i$
   (d) $r_{i+1} = r_i - \alpha_i y_i$
   (e) $\tilde{r}_{i+1} = A^Tr_{i+1}$
   (f) $z_{i+1} = Q^{-1}\tilde{r}_{i+1}$
   (g) $\beta_{i+1} = (z_{i+1}, \tilde{r}_{i+1})/(z_i, \tilde{r}_i)$
   (h) $p_{i+1} = z_{i+1} + \beta_i p_i$
```

The above procedure will not break down if A is nonsingular since then the matrix A^TA is symmetric positive definite, and so is the preconditioning matrix Q. There are many alternatives and variations to the above algorithm. The standard alternatives, based on the same formulation (17) are either to use the preconditioner on the right, solving the system $A^TAQ^{-1}y = b$, or to split the preconditioner into a forward SOR sweep on the left and a backward SOR sweep on the right of the matrix A^TA. The other options are obviously to use the normal equations (18) with again three different ways of preconditioning. Therefore, there are at least obvious six different algorithms. Moreover, one can also implement more robust algorithms such the LSQR technique [9]. We expect to see little differences for problems that are reasonably well conditioned (after preconditionings) between all these different options. For problems that are very poorly conditioned, the LSQR option may perform better [9].

4.3 Parallel implementations

As was mentioned before, it is possible in some well-known cases to reorder the unknowns such that the matrix $B = AA^T$ (resp. $B = A^TA$) has diagonal blocks that are diagonal matrices. This simply amounts to identifying a partition of the set $\{1, 2, ..., N\}$ into subsets $S_1, ..., S_k$ such that the rows (resp. columns) whose indices belong to the same set S_i are orthogonal to each other. As a result of this, when implementing a block SSOR scheme where the blocking is identical with the partition, there will be no linear systems to solve. To be more specific, let us reorder the rows by scanning those in S_1 followed by those in S_2, etc..., and let us denote by A_i the matrix consisting of the rows belonging to the $i-$th block. We also assume that all rows have been normalized so that their 2-norm is unity. Then a block Gauss-Seidel analogous to Algorithm 1 of section 4.1, will be as follows

Algorithm: Block Gauss-Seidel Sweep for $AA^Tx = b$

```
1. Given an initial $x$, compute $z := A^T x$.
2. For $i = 1, 2, .., k$ Do
    (a) Compute $d_i = b_i - A_i z$
    (b) Compute $x_i := x_i + d_i$
    (c) Compute $z := z + A_i^T d_i$
```

Here x_i and b_i are subvectors corresponding to the blocking and d_i is a vector of length the size of the block, instead of being a scalar as in Section 4.1.

The question that arises is how to find the partition S_i. In certain simple cases, such as block-tridiagonal matrices this can be easily done [8]. For general sparse matrices, it is important to think in terms of the symmetric squared matrix AA^T and the problem is to find a reordering of this matrix such that it has diagonal diagonal blocks. This question is identical with that of reordering a general sparse matrix by levels such that during the L-U solve phases in Preconditioned conjugate gradient methods, several unknowns can be solved for simultaneously [1]. The same technique can be used but it relies on an arbitrary choice of the first unknown. The problem of determining the best first element, i.e., the one that yields the smallest number of blocks k is a hard problem. An interesting observation again is that the reordering of the unknowns can be found without explicitly forming the symmetric squared matrix, or rather without having to store it. This is because the level-scheduling procedure only requires one row of the matrix at a time. This row can be generated, used and then discarded. More details and experiments with the technique of blocking in SSOR schemes will appear in another paper.

5 Incomplete Choleski on the normal equations

The incomplete Choleski factorization can be used to precondition the normal equations (17) or (18). This approach is attractive because of the success of incomplete factorization preconditioners for symmetric positive definite problems. We should point out however that the incomplete Choleski factorization is not guaranteed to exist for positive definite matrices. All the results that ensure existence rely on some form of diagonal dominance.

Concerning implementation details, here are a few important points. First, once again there is not nee to explicitly compute the matrix $B = AA^T$, since all that is required is to be able to access one row of B at a time. This row can be computed and used and then discarded. Therefore, the ICC(0) process will have the following structure. Note that initially the row u_1 is defined as the first row of A.

ICC(0) for $B = AA^T$
For i=2,3,...,N Do:

1. Compute all *nonzero* inner products $\beta_{ij} = a_j a_i^T, j = 1, 2, ..., i-1$. Let $NZ(j) \equiv \{i | \beta_{ij} \neq 0\}$.
2. For every $i \in NZ(j)$ Compute $b_j := b_j - \beta_{ij} u_i$ and drop nonzero elements with column numbers not in $NZ(j)$.
3. Define l_j to be the lower part of resulting $\hat{b}_j$, (i.e., $l_{ij} = \beta_{ij}, i < j, i \in NZ(i)$ and $l_{ij} = 0$ otherwise.) Similarly, define u_j to be the upper part of $\hat{b}_j$.

Here again the same technique as in ILQ is used to compute the inner products in step 1. The factor L obtained from this algorithm can be used in exactly the same manner as the matrix L of the ILQ preconditioning of Section 2, to precondition the normal equations and there are just as many possibilities.

Because the matrix L of the *complete* LQ factorization of A is identical with the Choleski factor of B, one might wonder why the IC(0) factorization of B does not always exist while the ILQ factorization virtually always exists. The reason seems is simply that ILQ is not based on the structure of A but on the size of the elements dropped out. It is likely that a similar technique can be used to define a more robust IC factorization of B.

6 Numerical Experiments

In this section we report a number of numerical experiments conducted on an Alliant FX-8, using double precision arithmetic. We tested with two model problems which are discussed in two separate subsections. We compared the following five methods:

Method 1: CGNE/ILQ. Conjugate gradient method applied to the normal equations with ILQ preconditioner. Here we only consider the variant (17) of the normal equations, so all references to the normal equations mean the equation $A^TAx = A^Tf$.

Method 2: CGNE/IC(0). Conjugate gradient method applied to the normal equations with Incomplete Choleski preconditioner.

Method 3: GMRES/ILU(0). GMRES [10] applied to original system with ILU(0) preconditioning from the right, i.e., GMRES applied to $AM^{-1}y = f$. To test the preconditioner rather that the iterative solver (GMRES) we took in all tests the number of directions in GMRES to be 10.

Method 4: GMRES/ILUP. GMRES applied to the original system with ILU preconditioning with column pivoting, from the right.(GMRES/ILUPIV.)

Method 5: CGNR/SSOR applied to the normal equations with SSOR($\omega = 1$) preconditioner.

6.1 Problem 1

We consider the following elliptic partial differential equation in the region $\Omega = (0,1) \times (0,1)$

$$-\Delta u + \gamma(x\frac{\partial u}{\partial x} + y\frac{\partial u}{\partial y}) + \beta u = g, \quad in \quad \Omega \tag{25}$$

with Dirichlet boundary conditions. The above problem is discretized using centered differences for both the second order and first order derivatives. The values of γ and β are varied to make the problem more or less difficult to solve. In our first test we took $\gamma = 10.0, \beta = -100.0$. The grid size for the first test is $h = 1/33$, leading to a problem of size $32 \times 32 = 1024$. The right hand side is chosen once the discrete equations are formed. It is selected so that the solution x to the discrete system is one everywhere. This allows an easy verifications of the result. All the methods start with the same pseudo-random vector and the process is stopped as soon as the actual residual vector is less than $\epsilon \equiv 10^{-7}$. In this difficult problem, all the methods have some difficulties.

Figure 1 plots the residual norm versus the total number of arithmetic operations (in millions) for each of the 5 methods. We have deliberately avoided to show the time instead of the number of operations because our codes are not optimized for the Alliant FX-8. In this first figure

In the following table we indicate the total number of iterations required for convergence for each of the methods.

7 Conclusion

Indefinite and nonsymmetric linear systems can be very hard to solve by iterative methods. We have compared a few techniques that can be effective in some circumstances but we must emphasize that none of these techniques can be viewed yet as a general purpose solver. The ILQ technique is a promising alternative to the standard ILU or to direct solvers. Despite their poor reputation in handling most definite problems, the methods that are based on the normal equations seem to be quite useful for indefinite problems. As was already observed [5, 8], normal equation approaches can be more effective than other approaches based on the direct equations. The best illustration of this fact is when the original matrix is orthogonal. Then its spectrum is ...

References

[1] E. C. Anderson and Y. Saad. Solving sparse triangular systems on parallel computers. *International Journal of High Speed Computing*, 1:73–96, 1989.

[2] A. Bjork. *Least Squares Methods*. Elsevier/North-Holland, New-York, 1988. Handbook for Numerical Analysis Series, P. G. Ciarlet, J. L. Lions eds.

[3] A. Bjork and T. Elfving. Accelerated projection methods for computing pseudo-inverse solutions of systems of linear equations. *BIT*, 19:145–163, 1979.

[4] I. S. Duff, A. M. Erisman, and J. K. Reid. *Direct Methods for Sparse Matrices*. Clarendon Press, Oxford, 1986.

[5] H. C. Elman. *Iterative Methods for Large Sparse Nonsymmetric Systems of Linear Equations*. PhD thesis, Yale University, Computer Science Dept., New Haven, CT., 1982.

[6] H. C. Elman. A stability analysis of incomplete LU factorizations. *Math. Comp.*, 47:191–217, 1986.

[7] C. I. Goldstein and E. Turkel. An iterative method for the Helmholtz equation. *Journal of Computational Physics*, 49:443–457, 1983.

[8] C. Kamath and A. Sameh. A projection method for solving nonsymmetric linear systems on multiprocessors. Technical Report 611, CSRD, university of Illinois, Urbana, IL, 1986.

[9] C. C. Paige and M. A. Saunders. An algorithm for sparse linear equations and sparse least squares. *ACM Trans. Math. Software*, 8:43–71, 1982.

[10] Y. Saad and M. H. Schultz. GMRES: a generalized minimal residual algorithm for solving nonsymmetric linear systems. *SIAM J. Sci. Statist. Comput.*, 7:856–869, 1986.

[11] R. S. Varga. *Matrix Iterative Analysis*. Prentice Hall, Englewood Cliffs, NJ, 1962.

[12] J. W. Watts-III. A conjugate gradient truncated direct method for the iterative solution of the reservoir simulation pressure equation. *Society of Petroleum Engineer Journal*, 21:345–353, 1981.

[13] Z. Zlatev. Use of iterative refinement in the solution of sparse linear systems. *SIAM J. Numer. Anal.*, 19:381–399, 1982.