Journal of Computational and Applied Mathematics 24 (1988) 89–105
North-Holland

Preconditioning techniques for nonsymmetric and indefinite linear systems *

Youcef SAAD
Center for Supercomputing Research and Development, University of Illinois at Urbana-Champaign, Urbana, IL 61801, U.S.A.

Received 8 February 1988
Revised 18 June 1988

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.

Keywords: Indefinite linear systems, preconditioned conjugate gradient, incomplete LQ factorization, normal equations, least squares problems, SSOR preconditioners.

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 remain 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 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

* 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.

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 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 that 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 $\frac{1}{2}(A + A^{\mathrm{T}})$ 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 linear system arising from the Helmholtz equation $\Delta u + k^2(x, y)u = f$, which occurs in different forms in various models of wave propagation [8]. 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 during the process.

Among the iterative methods available, Krylov subspace methods offer a good potential for leading to 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. Consider 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 instability has been studied by Elman [6] who proposed a detailed analysis of ILU and MILU preconditioners. We also observed experimentally that ILU precon-

ditioners 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 conjugate gradient methods by proposing more robust alternatives 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, the standard incomplete LU factorization as well as a variant with column pivoting, an SSOR preconditioner on the normal equations, and the standard Incomplete Cholesky 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 $a_1, a_2, \ldots, a_N$. The complete LQ factorization consists 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 [7], for example, exploit the fact that the Cholesky factor of the maxtrix $B = AA^{\mathrm{T}}$ is identical with the matrix L of the decomposition $A = LQ$. This requires forming the data structure of the matrix AA^{T}, which may be much denser than A, but reordering techniques can be used to reduce the amount of work required to compute 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 the rows remain very sparse in the incomplete LQ factorization to be described shortly, any given row of A will be typically orthogonal to most of the previous rows of Q, and, as a result the Gram–Schmidt process is much less prone to numerical difficulties. From the data structure point of view, Gram–Schmidt is optimal in that it does not require allocating more space than is necessary, as is the case with approaches based on symmetric squaring. Another advantage of the orthogonalization approach over that of symmetric squaring is its simplicity and its strong similarity with the LU factorization process: at every step a given row is combined with previous rows and then normalized. Since we are interested only in an approximate LQ factorization, the Gram–Schmidt process is a perfectly suitable method. To define an *incomplete* factorization we must only incorporate a *dropping* strategy, which will allow us 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 only upon the structure of A is not safe in cases where the matrix is indefinite. For example, the incomplete LU factorization will break down immediately 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 in 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 is as follows:

Algorithm 1. Incomplete LQ factorization
For $i = 1, 2, \ldots, N$ Do:
Step 1. If $i = 1$ define $\hat{q}_1 = a_1$ and goto Step 5.
Step 2. Compute all *nonzero* inner products $l_{ij} = q_j a_i^T$, $j = 1, 2, \ldots, i-1$.
Step 3. Determine the p_L largest l_{ij}'s $j = 1, 2, \ldots, i-1$ and assign a zero value to the others.
Step 4. Compute $\hat{q}_i = a_i - \sum_{j=1, l_{ij} \neq 0}^{j=i-1} l_{ij} q_j$.
Step 5. Determine the p_Q largest elements of $\hat{q}_i$ and assign a zero value to the other elements.
Step 6. Compute $l_{ii} = \| \hat{q}_i \|_2$ and $q_i = \hat{q}_i / l_{ii}$.

The second step of the above procedure needs particular attention. If we actually had to compute all the inner products $q_1 a_i^T$, $q_2 a_i^T, \ldots, q_{i-1} a_i^T$, 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 them 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 $q_1, \ldots, q_{i-1}$ 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 $j = 1, \ldots, i-1$, which is unacceptable as was just argued. The alternative is to compute them as a linear combination of the columns of Q_{i-1}. Let $a_{i,i_1}, a_{i,i_2}, \ldots, a_{i,i_{k_i}}$ be all the nonzero elements of the ith row of A. Then the desired column l is given by

$$l = \sum_{j=1}^{k_i} a_{i,i_j} q'_{i_j} \tag{4}$$

where q'_k represents the kth column of Q_{i-1}. Using the above formula to compute l 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 $i_1, \ldots, i_{k_i}$.

One difficulty with this implementation of Step 2 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 list 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 fairly economical, although it suffers from not being amenable to parallel and vector processing.

After the ith step is performed, the following relation holds:

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

where f_i is the row of elements that have been dropped from the row $\hat{q}_i$ in Step 5. The above equation translates into

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

where E is the matrix whose ith 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 decomposition (6) 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 suggested by the next theorem. We will assume that Step 4 of ILQ is replaced by an ideal orthogonalization process in which $\hat{q}_i$ is forced to be orthogonal to all the previous q_j's.

Theorem 2.1. *Assume that at every step $j \leqslant i$ of ILQ, the row $\hat{q}_j$ is orthogonal to $q_1, \ldots, q_{j-1}$ and let Π_i be the orthogonal projector onto the span of the rows $q_1, q_2, \ldots, q_i$. 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 $q_1, \ldots, q_{i+1}$ to be linearly independent, assuming that $q_1, \ldots, q_i$ satisfy 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 $q_1, \ldots, q_{i+1}$ 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 $q_1, \ldots, q_i$ 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 $j \leqslant i$ of ILQ, the row $\hat{q}_j$ is orthogonal to $q_1, \ldots, q_{j-1}$ and that the row of elements that are dropped 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 $\| \Pi_i f \|_2 \leqslant \| f \|_2$ for any f. □

These results provide conditions under which the constructed matrix Q_i remains of full rank at every step. One of their weaknesses is that they assume that at every step the vector $\hat{q}_i$ is made orthogonal to the previous q's. In the form in which it is presented, the algorithm performs an inaccurate Gram–Schmidt step, because, in general, the previous vectors q_j are not orthogonal to each other. Clearly, orthogonalization can be enforced by a more elaborate 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 additional 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 might be so severe that the only way in which it could be fulfilled 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 such 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-type algorithm. The simplest idea that comes to mind is to form the preconditioned system

$$L^{-1}AQ^{\mathrm{T}}y + L^{-1}b \tag{9}$$

whose solution y is related to the solution x of the original system by $x = Q^{\mathrm{T}}y$. 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 the singularity of Q does indeed occur, especially for poorly conditioned 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 $AA^{\mathrm{T}}y = b$, namely to

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

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 $x = A^{\mathrm{T}}L^{-\mathrm{T}}y$. Ideally, i.e., if the incomplete factorization were mathematically exact, then the conjugate gradient method applied to (10) would converge in one step, since the matrix on the left-hand side would become the identity matrix. There are several other ways of using L to precondition the normal equations $AA^{\mathrm{T}}y = b$. If we define $M \equiv LL^{\mathrm{T}}$, then an implementation of the conjugate gradient method applied to the normal equations (18), sometimes referred to as Craig's method, and preconditioned from the left takes the following form [5].

Algorithm 2. CGNE/ILQ with left preconditioning

Step 1. Start: Compute $r_0 = b - Ax_0$, $z_0 = M^{-1}r_0$, $p_0 = A^{\mathrm{T}}z_0$.

Step 2. Iterate: For $i = 0, 1, \ldots,$ until convergence do

(a) $w_i = Ap_i$,
(b) $\alpha_i = (z_i, r_i)/(p_i, p_i)$,
(c) $x_{i+1} = x_i + \alpha_i p_i$,
(d) $r_{i+1} = r_i - \alpha_i w_i$,
(e) $z_{i+1} = M^{-1}r_{i+1}$,
(f) $\beta_i = (z_{i+1}, r_{i+1})/(z_i, r_i)$,
(g) $p_{i+1} = A^{\mathrm{T}}z_{i+1} + \beta_i p_i$.

Similarly, one can compute the ILQ factorization of A^T and precondition the linear system $A^TAx = A^Tb$. Thus, if $A^T = LQ + E$ is the ILQ decomposition of A^T and if we define $M \equiv LL^T$, the Conjugate Gradient algorithm applied to (17) and preconditioned from the left is as follows.

Algorithm 3. CGNR/ILQ with left preconditioning

Step 1. Start: Compute $r_0 = b - Ax_0$, $\tilde{r}_0 = A^Tr_0$, $z_0 = M^{-1}\tilde{r}_0$, $p_0 = r_0$.

Step 2. Iterate: For $i = 0, \ldots,$ until convergence do

(a) $w_i = Ap_i$,
(b) $\alpha_i = (\tilde{r}_i, z_i)/\| w_i \|_2^2$,
(c) $x_{i+1} = x_i + \alpha_i p_i$,
(d) $r_{i+1} = r_i - \alpha_i w_i$,
(e) $\tilde{r}_{i+1} = A^Tr_{i+1}$,
(f) $z_{i+1} = M^{-1}\tilde{r}_{i+1}$,
(g) $\beta_{i+1} = (z_{i+1}, \tilde{r}_{i+1})/(z_i, \tilde{r}_i)$,
(h) $p_{i+1} = z_{i+1} + \beta_i p_i$.

As is well-known [5], Algorithm 3 attempts to minimize the 2-norm of the error while Algorithm 4 attempts to minimize the residual norm.

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

$$A^TAx = A^Tb. \tag{12}$$

When the matrix A is a 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 $A^T = LQ + E$ be the incomplete LQ factorization of A^T. This is nothing but ILQ applied to the columns of A rather than to 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:

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

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^{-T} need not be computed explicitly.

The effect of post-multiplying A by L^{-T} 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 early in the factorization, 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 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 (2,2) and (2,3) are dropped. This breakdown may be avoided if more fill-in is allowed. Note that here neither partial row or column pivoting helps at the second step since both the row and column are zero.

A solution to this difficulty is to drop elements according to their magnitude rather than their position. For example, a simple strategy is to keep a fixed number, say q_u, of the elements of largest 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 partial pivoting of the columns. In order to keep cost minimal, typical incomplete LU factorizations proceed by rows and don't require the column data structure. As a result interchanging rows is impractical since it involves not only scanning the columns but also 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. As is traditionally done, 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 $w_1, \ldots, w_{i_L}$, $w_{i_L+1}, \ldots w_{i_L+q_L}$ 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 [13].

Unfortunately, even with this implementation, the occurrence of zero rows is not excluded. However, it is our experience that with the above apporach 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 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 [14]. 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 the limited fill-in type techniques 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 implementation of the incomplete LU factorization with column pivoting are as follows.

- Use column pivoting only. A pointer is maintained to keep track of the new ordering of the unknowns.
- Drop elements when generating U according to their magnitude instead of according 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

$$A^{\mathrm{T}}Ax = A^{\mathrm{T}}b \tag{17}$$

or

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

Bjork and Elfving [3] have shown that in order to use relaxation schemes on the normal equations, one only needs to access one column of A at a time for (17) and one 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

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

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

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

which yields,

$$\delta_i = \frac{(b,\ e_i) - \left(A^{\mathrm{T}} y_{\text{old}},\ A^{\mathrm{T}} e_i\right)}{\left(A^{\mathrm{T}} e_i,\ A^{\mathrm{T}} e_i\right)}. \tag{21}$$

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

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

and denote by b_i the ith component of b, then a basic relaxation step consists of taking

$$\delta_i = b_i - \left(A^{\mathrm{T}} y_{old},\ A^{\mathrm{T}} e_i\right). \tag{23}$$

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 $A^{\mathrm{T}} y_{\text{old}}$ and its inner product with the ith row of A. This inner product is inexpensive to compute because $A^{\mathrm{T}} e_i$ is usually sparse. On the other hand the matrix by vector product $A^{\mathrm{T}} y_{\text{old}}$ must be computed carefully if we want to avoid performing a matrix by vector product at each relaxation step. This can be achieved by computing the initial $A^{\mathrm{T}} y_{\text{old}}$ in the Gauss–Seidel sweep as a full vector z and then each time the approximate solution y_{old} is updated by (19), to update z accordingly. The forward Gauss–Seidel sweep would therefore be as follows, where a_i denotes again the ith row of A.

Algorithm 4. Forward Gauss–Seidel Sweep for $AA^{\mathrm{T}} y = b$

Step 1. Given an initial y, compute $z := A^{\mathrm{T}} y$.

Step 2. For $i = 1, 2, \ldots, N$, do

(a) Compute $\delta_i = b_i - a_i z$,

(b) Compute $y_i := y_i + \delta_i$,

(c) Compute $z := z + \delta_i a_i^{\mathrm{T}}$.

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 ith 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 Symmetric Gauss–Seidel (i.e., SSOR with $w = 1$), where nz represents the total number of number of nonzero elements of A. Storage consists of the right-hand side, the vector y and the additional work vector z.

Similarly, if we denote by a'_i the ith column of A and assume that it is normalized so that its 2-norm is unity, then a Gauss–Seidel sweep for (18) is as follows :

Algorithm 5. Forward Gauss–Seidel Sweep for $A^TAx = A^Tb$

Step 1. Given an initial x, compute $z := Ax$.

Step 2. For $i = 1, 2, \ldots, N$ do

(a) Compute $\delta_i = (A^Tb)_i - z^Ta_i'$,

(b) Compute $x_i := x_i + \delta_i$,

(c) Compute $z := z + \delta_i a_i'$.

Note that the matrices AA^T and A^TA can be dense or in general much less sparse than A, yet the cost of the above implementations depend only on the nonzero structure of A. This is a significant 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 acceleration of the above relaxation schemes by under or over-relaxation. If we introduce the usual acceleration parameter ω, then we only have to replace (a) in Algorithm 4 by

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

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 [12] because the matrix is positive definite. Moreover, another question not addressed here is how convergence can be affected by various reorderings of the rows. When the matrix arises from discretizing 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 [9] 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 for conjugate gradient methods applied to either (17) or (18). If we consider (18), for example, then we need a procedure that delivers an approximation to $(AA^T)^{-1}v$ for any vector v. One such procedure consists of performing one step of SSOR to solve the system $(AA^T)w = v$. If we denote the linear operator that transforms v into the vector resulting from this procedure by M^{-1}, then the usual conjugate gradient method applied to (18), can be recast in the same form as Algorithm 2. We will refer to this algorithm as CGNE/SSOR. Similarly, we can obtain an analogue of Algorithm 3 that is associated with the normal equations (17), by defining M^{-1} to be the linear transformation that maps a vector v into a vector w resulting from a forward and a backward sweep of Algorithm 5. We will refer to this algorithm as CGNR/SSOR.

The CGNE/SSOR and CGNR/SSOR algorithms will not break down if A is nonsingular since then the matrices AA^T and A^TA are symmetric positive definite, as are the preconditioning matrices M. There are many alternatives and variations to these algorithms. The standard alternatives based on the same formulation (17) are either to use the preconditioner on the right, solving the system $A^TAM^{-1}y = b$, 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. Similar options can also be written for the normal equations (18) with again three different ways of preconditioning.

Therefore, there are at least six obvious different algorithms. Moreover, one can also implement more robust algorithms such as the LSQR technique [10]. We expect to see little differences for problems that are reasonably well conditioned (after preconditioning) between all these different options. For problems that are very poorly conditioned, the LSQR implementation may perform better.

4.3. Parallel implementations

As was mentioned in section 4.1, it is possible in many cases to find an ordering of the unknowns such that the matrix $B = AA^T$ (resp. $B = A^TA$) associated with this ordering has diagonal blocks that are diagonal (or block diagonal) matrices. This simply amounts to identifying a partition of the set $\{1, 2, \ldots, N\}$ into subsets $S_1, \ldots, S_k$ 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, all of the unknowns belonging to the same set S_j can be updated in parallel. 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 ith block. We assume that all rows of the same set are orthogonal to each other and that they have been normalized so that their 2-norm is unity. Then a block Gauss–Seidel sweep analogous to Algorithm 4 of section 4.1, is as follows.

Algorithm 6. Forward block Gauss–Seidel sweep for $AA^Ty = b$
Step 1. Given an initial y, compute $z := A^Ty$.
Step 2. For $i = 1, 2, \ldots, k$ do
(a) Compute $d_i = b_i - A_iz$.
(b) Compute $y_i := y_i + d_i$.
(c) Compute $z := z + A_i^Td_i$.

Here y_i and b_i are subvectors corresponding to the blocking and d_i is a vector of length the size of the block, which replaces the scalar δ_i of Algorithm 4. Each of the steps (a), (b) and (c) can be performed in parallel.

The question that arises is how to find the partition S_i. In certain simple cases, such as block-tridiagonal matrices, this can easily be done [9]. 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 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 [1] 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 Cholesky on the normal equations

The incomplete Cholesky 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 Cholesky factorization is not guaranteed to exist for an arbitrary positive definite symmetric matrix. All the results that ensure existence rely on some form of diagonal dominance.

Concerning implementation details, there is no need 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.

Algorithm 7. IC(0) for $B = AA^T$

For $i = 2, 3, \ldots, N$ do:

Step 1. Obtain all the *nonzero* inner products $\beta_{ij} = a_j a_i^T$, $j = 1, 2, \ldots, i-1$. Set $NZ(i) \equiv \{ j \mid \beta_{ij} \neq 0 \}$ and $w := (\beta_{ij})_{j=i,N}$.

Step 2. For every $j \in NZ(i)$, compute $w := w - \beta_{ij} u_j$ and drop nonzero elements with column numbers not in $NZ(i)$.

Step 3. Define l_i to be the L-part of resulting w, (i.e., $l_{ij} = w_j$, $j < i$, $j \in NZ(i)$ and $l_{ij} = 0$ otherwise.) Similarly, define u_i to be the U-part of w.

Here the same technique as in ILQ can be used to compute the inner products economically 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 with the same number of possible combinations as in section 4.2.

Because the matrix L of the *complete* LQ factorization of A is identical with the Cholesky 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 is simply that unlike the above IC(0) procedure, 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 two model problems which are discussed in separate subsections and compared the following five methods:

Method 1. CGNR/ILQ: Conjugate gradient method applied to the normal equations with ILQ preconditioner. Here we only considered the left preconditioning variant as described in Section 2, and took $p_L = p_U = 10$.

Method 2. CGNR/IC(0): Conjugate gradient method applied to the normal equations with Incomplete Cholesky preconditioner. Here the preconditioner is split, i.e., the conjugate gradient techinque is applied to the system $L^{-1}A^TAL^{-T}y = L^{-1}b$.

Method 3. GMRES/ILU(0): GMRES [11] applied to original system with ILU(0) preconditioning from the right, i.e., GMRES applied to $AM^{-1}y = b$. To test the preconditioner rather that

the iterative solver (GMRES) we took the number of directions in GMRES to be 10 in all tests.

Method 4. GMRES/ILUP: GMRES(10) applied to the original system with ILU preconditioning with column pivoting, from the right. In all our tests, the parameters q_L, q_U that determine the fill-in elements allowed in L and U are chosen so that $q_L = q_U = 5$.

Method 5. CGNR/SGS: CGNR applied to the normal equations with Symmetric Gauss–Seidel (SSOR with $w = 1$) preconditioner.

6.1. Problem 1

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

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

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 verification of the result. All the methods start with the same pseudo-random vector and the process is stopped as soon the actual residual vector has decreased by a factor of at least $\epsilon \equiv 10^{-7}$.

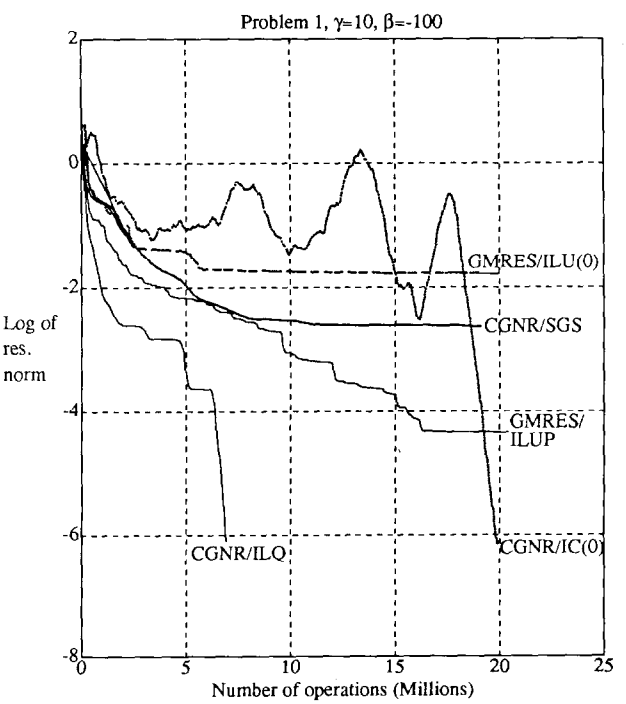

Fig. 1.

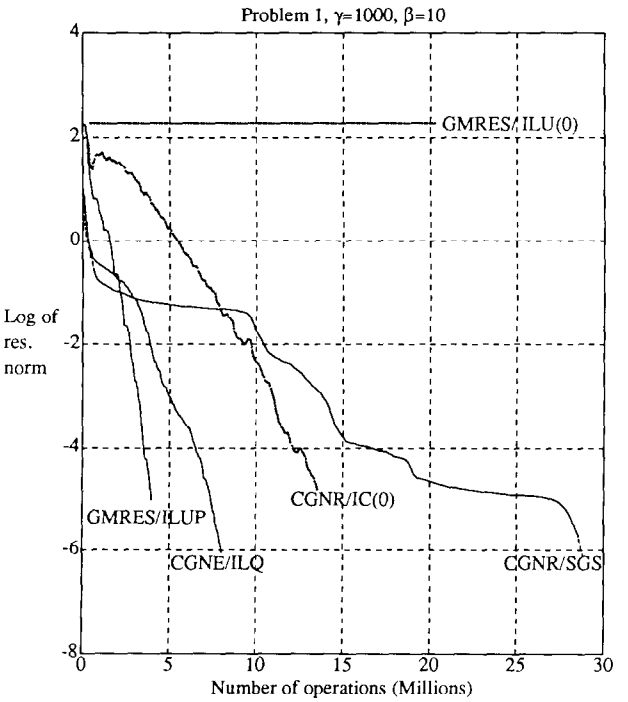

Fig. 2.

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 showing the time instead of the number of operations because our codes are not optimized for the Alliant FX/8. We have also not included the operation count for the preprocessing times, which in general represent a fraction of the operation count of the iteration phase. The only two methods that have succeeded in satisfying the stopping criterion are CGNR/ILQ and CGNR/IC(0), although the second method took much longer to converge. The first method converged in 66 steps while the second one in 310 steps. The other three methods, CGNR/SGS, GMRES(10)/ILU(0) and GMRES(10)/ILUP(5) did not converge within a reasonable time, with ILUP not too far from converging. Note that for both Method 1 and Method 5, the rows of A are scaled so that their 2-norms are equal to unity. As a result, the residual norms shown in Fig. 1 (as well as in the next figures) are scaled differently for these two methods and the other three methods.

In our second test we changed the values of the parameters to $\beta = -10$, $\gamma = 1000.0$, thus making the problem much more nonsymmetric. The conditions and methods tested are identical with those of the previous case. Figure 2 shows the behavior of the 5 methods on this example. Here the only method that has not converged is GMRES(10)/ILU(0). The fastet method here is GMRES/ILUP which took 66 steps to converge, followed by CGNR-ILQ (109 steps), CGNR/IC(0) (198 steps) and finally CGNR/SGS (312 steps).

6.2. *Problem 2*

The second test problem we consider is taken from [9]. It is the following partial differential

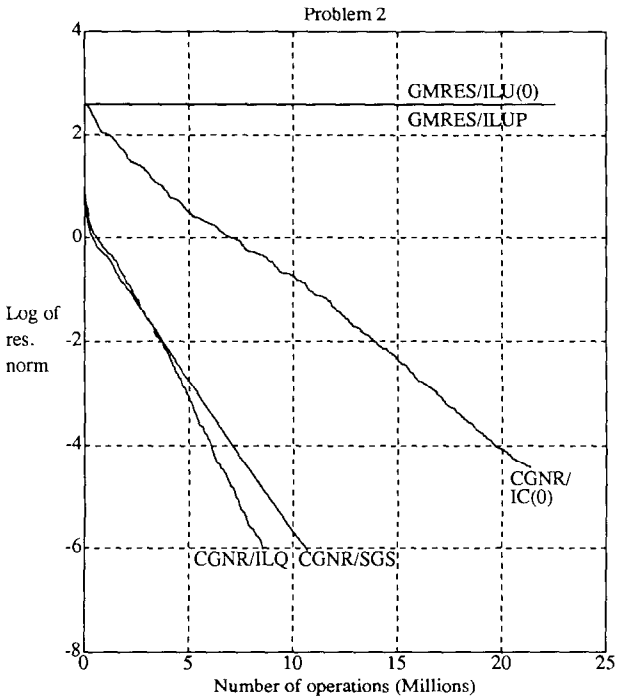

Fig. 3.

equation in the region $\Omega = (0,1) \times (0,1)$

$$-\Delta u + \gamma\left(\frac{\partial(e^{xy}u)}{\partial x} + \frac{\partial(e^{-xy}u)}{\partial y}\right) = g \quad \text{in } \Omega \tag{26}$$

with Dirichlet boundary conditions. The above problem is again discretized using centered differences for both the second order and first order derivatives on a 33×33 grid, leading to a matrix of dimension $N = 1024$. In this test we took as in [9] $\gamma = 1000.0$. The right hand side is chosen as in the previous examples, i.e., it is selected so that the solution x to the discrete system is one everywhere. All the methods start with the same pseudorandom vector and the stopping criterion is as in section 6.1 The results are shown in Fig. 3. Unlike the previous two cases, the SGS approach performs reasonably well on this problem. The number of iterations required for convergence is 127 for CGNR/ILQ, 116 for CGNR/SGS and 316 for CGNR/IC(0). ILU(0) and ILUP(5) did not achieve convergence in a reasonable time.

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 as a general purpose iterative solver. The ILQ technique combined with the normal equation approach is a promising alternative to the standard ILU or to direct solvers both for the cases where the coefficient matrix is indefinite and

for the case where it has large nonsymmetric part. Despite their poor reputation in handling most definite problems, the methods that are based on the normal equations seem to be quite useful in some of the more difficult situations. It has already been observed by many authors, see e.g., [5,9], that normal equation approaches can be more effective than the more standard approaches based on the direct equations. The best illustration of this fact is when the original matrix is orthogonal, with its eigenvalues spread all around the unit circle. Then, the normal equation approach will converge in one step whereas a method such as GMRES is likely to have difficulties because of the circular shape of the spectrum.

The SGS preconditioning seems to be effective for cases where the matrix is positive real, possibly strongly nonsymmetric, but may have some difficulties for strongly indefinite problems. We have not tested any of the enhancements to the symmetric point Gauss–Seidel scheme, such as incorporation of optimal acceleration parameter w, or blocking. With these enhancements, SGS may be a good alternative to ILU preconditioners in these cases because it is easy to use. The experiments reported in [9] seem to indicate that with blocking, this scheme is fairly robust for problems arising from Elliptic Partial Differential Equations. Generally speaking, for other types of problems there are no experiments indicating which of these methods is to be preferred. In fact early experiements with problems from the Harwell/Boeing collection of test matrices reveal that all of these preconditioners will still fail or become too slow for many problems. On the positive side there are many problems that can be solved with the improved preconditioners (ILQ and ILUP) but not with ILU(0).

References

[1] E. Anderson and Y. Saad, Solving sparse triangular systems on parallel processors., Tech. Rep., University of Illinois, CSRD, Urbana, Il, 1988, in preparation.

[2] A. Bjork, Least squares methods, in: P.G. Giarlet and J.L. Lions, Eds., *Handbook of Numerical Analysis* (North-Holland, Amsterdam, to appear).

[3] A. Bjork and T. Elfving, Accelerated projection methods for computing pseudoinverse solutions of systems of linear equations, *BIT* **19** (1979) 145–163.

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

[5] H. Elman, Iterative methods for large sparse nonsymmetric systems of linear equations, PhD thesis, Yale University, Computer Science Dept., New Haven, CT, 1982.

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

[7] J. George and M. Heath, Solution of sparse linear least squares problems using givens rotations, *Lin. Algebra Appl.* **34** (1980) 69–83.

[8] C. Goldstein and E. Turkel, An iterative method for the Helmholtz equation *J. Comput. Phys.* **49** (1983) 443–457.

[9] C. Kamath and A. Sameh, A projection method for solving nonsymmetric linear systems on multiprocessors, Tech. Rep. 611, CSRD, University of Illinois, Urbana, IL, 1986.

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

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

[12] R. Varga, *Matrix Iterative Analysis* (Prentice-Hall, Englewood Cliffs, NJ, 1962).

[13] J.W. Watts-III, A conjugate gradient truncated direct method for the iterative solution of the reservoir simulation pressure equation, *Soc. Petroleum Eng. J.* **21** (1981) 345–353.

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