A class of incomplete orthogonal factorization methods. II: implementation and results[1]

Andreas T. Papadopoulos[2], Iain S. Duff[3] and Andrew J. Wathen[4]

ABSTRACT

We present, implement and test several incomplete orthogonal factorization methods based on Givens rotations for large sparse unsymmetric square and rectangular matrices. The methods are applied to a variety of square systems and their performance as preconditioners is tested against standard incomplete LU factorization techniques. For rectangular matrices corresponding to least-squares problems, the resulting incomplete factorizations are applied as preconditioners for conjugate gradients for the system of normal equations. A comprehensive discussion about the uses, advantages and shortcomings of these preconditioners is given.

Keywords: Preconditioning, sparse linear systems, sparse least-squares, iterative methods, incomplete orthogonal factorizations, Givens rotations.

AMS(MOS) subject classifications: 65F10, 65F25, 65F50.

---

[1]Current reports available by anonymous ftp to ftp.numerical.rl.ac.uk in directory pub/reports. This report is available in compressed postscript as file padwRAL2002019.ps.gz or as the PDF file padwRAL2002019.pdf. The report is also available through URL http://www.numerical.rl.ac.uk/reports/reports.html.
An earlier and more expanded version was published as Technical Report NA-02-07 from OUCL. Submitted to *BIT*.
[2]andp@comlab.ox.ac.uk, Oxford University Computing Laboratory, Wolfson Building, Parks Road, Oxford, OX1 3QD.
[3]i.s.duff@rl.ac.uk, the work of this author was supported in part by the EPSRC Grant GR/R46441.
[4]wathen@comlab.ox.ac.uk, Oxford University Computing Laboratory, Wolfson Building, Parks Road, Oxford, OX1 3QD.

Computational Science and Engineering Department
Atlas Centre
Rutherford Appleton Laboratory
Oxon OX11 0QX
May 29, 2002

Contents

1 Introduction.

This paper is a sequel to the theoretical work on incomplete orthogonalization methods by Bai, Duff and Wathen (2001) which describes and analyses a family of incomplete QR factorizations using Givens rotations. Whereas their paper was concerned only with theoretical aspects, we have efficiently implemented the methods and have provided numerical results.

We consider three specific methods. The first two of these methods perform the reduction using Givens rotations in a column-wise fashion: column-Incomplete Givens Orthogonalization (cIGO-method) drops entries by position only, whereas column-Threshold Incomplete Givens Orthogonalization (cTIGO-method) drops entries dynamically by both their magnitudes and positions. The third method, row-Threshold Incomplete Givens Orthogonalization (rTIGO-method), again drops entries dynamically, but only the magnitude is now taken into account and reduction is performed in a row-wise fashion. We give detailed accounts of how we code these algorithms to ensure efficiency of computation and memory use.

The use of incomplete factorizations as preconditioners is widespread. In particular, since the seminal paper of Meijerink and van der Vorst (1977), which gave sufficient conditions for stability, incomplete LU factorizations have been and continue to be employed in many application areas as a preconditioner for the solution of square and sparse systems. Such factorizations compute a lower triangular matrix L and an upper triangular matrix U which are approximate factors of A in the sense of having a predefined or dynamically defined sparsity based on the size of entries computed. That is, during the factorization process, certain entries in the factors are set to be zero. The triangular factors are usually very sparse and can be computed in a fraction of the time necessary for a complete factorization; most of the computational work is simply by-passed through the use of dropping rules. If the dropping rule is not too strict, one can usually achieve good convergence with such preconditioners (Manteuffel 1980, Saad 1988, Saad 1996). More recently, research has focused on ordering or preprocessing strategies for the system matrix to improve the performance of ILU preconditioners (Benzi, Haws and Tůma 2000, Benzi, Szyld and van Duin 1999).

More recent research has introduced important innovations in the field of incomplete factorization techniques, such as approximate inverse techniques (see for example Benzi, Meyer and Tůma (1996) or Saad (1996), ch. 10.5) and incomplete QR type factorizations. In this paper, we will study a class of incomplete QR methods.

It is well known that a square, nonsingular matrix A can be decomposed into a product of an orthogonal matrix Q and a nonsingular upper triangular matrix R, such that $A = QR$. Obviously, one can use such a decomposition to solve the linear system $Ax = b$ directly. However, complete QR factorizations are even more expensive than their LU counterpart (Golub and Van Loan 1996). On the

other hand, incomplete variants of a QR factorization can be used to precondition iterations in a similar manner to incomplete LU preconditioning.

The question that arises naturally is how one should proceed to obtain this incomplete QR type factorization. The most researched approach employs the Incomplete Modified Gram-Schmidt technique (see Jennings and Ajiz (1984) or Saad (1996), ch. 10.7), which uses a modified Gram-Schmidt orthogonalization process coupled with dropping rules to compute the incomplete factors Q_{inc} and R_{inc}. An important difficulty with this approach is that Q_{inc} is in general not orthogonal and hence difficult to invert; Q_{inc} could even be singular if not enough fill-in is allowed, making the incomplete factorization of uncertain value even as a preconditioner. Also, the incomplete factors computed by this process may be of poor quality in terms of speeding up convergence of the iteration unless a large number of fill-ins (and consequently computational time) are allowed. Despite this, there have been successful attempts to improve the robustness of this method (Wang 1993, Wang, Gallivan and Bramley 1997).

The principal advantage of the use of Givens rotations for incomplete orthogonalization purposes is that Q_{inc} remains orthogonal. The use of Givens rotations in this context is advocated by Bai et al. (2001). This is the starting point of the work presented here, but is not the first attempt to use incomplete QR factorization using Givens rotations for preconditioning purposes.

In his PhD thesis, James (1990) uses such a factorization for conjugate gradients for the system of normal equations. His choice of a static sparsity pattern dropping rule (the sparsity pattern of A^TA) however, produced preconditioners of poor quality. James also identified cases for which the incomplete triangular factor resulting from this method is singular, even when A is not. The implementation of James closely resembles the preconditioning technique of George and Heath (1980), developed for the least-squares solution of rectangular systems.

Zlatev and Nielsen (1988) have also developed an incomplete orthogonalization preconditioner for conjugate gradients. Zlatev (1991) discusses in detail incomplete QR using Givens rotations. He appears to be the first to produce incomplete QR factorizations using Givens rotations that also use some threshold dropping rule. He uses a row-wise approach to the factorization process, that is, the reduction of the system matrix A to upper triangular form is performed row by row (see Section 2.3).

Most of this limited literature preconditions the least-squares problems via the normal equations. The case of square systems is largely ignored. Apparently, this is due to the availability of effective ILU preconditioning techniques for square matrices, whereas for rectangular matrices ILU is not a competitor.

In this paper, we have chosen to present results for both square and rectangular (least-squares) systems, to present comparisons with ILU for the former and to allow for comparison with other techniques which may be suggested for rectangular systems.

The paper is organized as follows: in Section 2, we describe the three algorithms (cIGO, cTIGO and rTIGO) which we have coded and tested. The first two, with slight modifications, were presented by Bai et al. (2001), whereas the third one uses the approach in James (1990) and George and Heath (1980), except that the dropping strategy is different. Some preliminary observations about each of these algorithms are also included, as is a brief discussion of the implementation issues for each algorithm.

In Section 3, we present a number of numerical experiments involving these factorizations as preconditioners for Krylov subspace methods for square systems, and compare their performance to standard incomplete LU type factorizations. Many brief examples are presented, as well as a short discussion on the effect of reordering and preprocessing the system prior to computing the incomplete factorization. Some brief relevant remarks can be found at the end of this section.

In Section 4, we switch our attention to preconditioning least-squares systems where QR factorization seems a more 'natural' choice. After presenting test results, we give a comprehensive discussion of the issues regarding Givens incomplete QR as preconditioner which we encountered. This discussion offers insights into how one could potentially improve the performance of these preconditioners.

Some overall concluding remarks can be found at the end of this paper.

2 Description of algorithms and observations.

A dropping rule (or dropping strategy, or nonzero strategy) tells us in which of the entries of a matrix we allow fill-in during an incomplete factorization process.

We define a set of pairs of integers, which refer to the indices of the entries of a particular matrix in which fill-in is allowed.

For example, the set of integer pairs

$$P_{A,U} := \{\ (i,j) \mid a_{ij} \neq 0\ ,\ i \leq j\ ,\ 1 \leq i,j \leq n,\ i \neq j\ \} \cup \{\ (i,i) \mid 1 \leq i \leq n\ \}$$

denotes the set of indices of the entries of A that are nonzero and above or on its main diagonal. Similarly, $P_{A,L}$ would denote the set of indices of nonzero entries in the lower triangular part of A, and so on. Note that one generally includes all main diagonal entries in the sparsity set, even if their initial value is zero.

The simplest non-trivial dropping rule 'keep entries only in $P_{A,U}$ and $P_{A,L}$ (i.e. in P_A) and drop the rest' is what we implemented in the first of our three algorithms.

We can now present the three different algorithms for computing an incomplete QR factorization using Givens rotations that we have implemented, and briefly discuss some of their properties.

The first two of these algorithms (cIGO and cTIGO) are variants of those presented by Bai et al. (2001). We have somewhat diverged from the actual algorithms presented in that paper because it was more concerned with their

theoretical aspects. However, when it came to implementing them efficiently, we discovered that certain details needed to be changed or omitted. Despite this, the core idea of these methods, that is the order in which we perform rotations, remains the same.

As for the third algorithm (rTIGO), although not presented by Bai et al. (2001), we coded it and include here a short presentation, as it is another way to achieve an incomplete QR factorization using Givens rotations.

No matter how one tries to organise the reduction of A to upper triangular form, two rows of the matrix have to be updated for each entry rotated out. This is a fundamental disadvantage of Givens QR when compared to LU decomposition and its incomplete variants. In the incomplete LU decomposition, only a single row changes for each entry we zero out.

2.1 column-Incomplete Givens Orthogonalization (cIGO) method.

The corresponding algorithm is described in pseudocode in Bai et al. (2001). We include a brief outline here:

For each column j, we successively annihilate, using Givens rotations, from the bottom up to the first sub-diagonal, the nonzero entries located in the strictly lower triangular part of the matrix whose incomplete factorization we are computing. For each such elimination, both the j^{th} row and the row where the nonzero just eliminated was located are updated by the rotation angle but only if the respective entries belong to the predetermined sparsity pattern set P. At the end of this process, the upper triangular part of A will contain the incomplete triangular factor R_{inc} whereas the incomplete orthogonal factor Q_{inc} can be recreated as a product of the Givens matrices corresponding to the rotations we have performed.

We observe that:

- A natural and convenient choice for the sparsity pattern P is the sparsity pattern of the original matrix A, i.e. $P = P_A$. If A has zeros on the main diagonal, these will be included in the sparsity pattern in a preprocessing step before IGO starts. Hence $nz(R_{inc}) = nz(A_U)$, and Q_{inc} is a product of $nz(A_L) - n$ rotation matrices. Another possible choice for P would be to choose it as the sparsity pattern of the matrix A^2, again including all entries on the main diagonal.

- The incomplete orthogonal factor Q_{inc} is never explicitly computed nor are any entries of Q_{inc} explicitly dropped. It is, however, an incomplete factor in the sense that not all the rotations that would be required for a complete Givens QR factorization are performed during the cIGO process; we only rotate out entries that belong to the set $P \cap A_L$.

In Bai et al. (2001) the incomplete orthogonal factor was actually computed and updated explicitly after each rotation and entries could then be dropped. However, we quickly realised that it is both easier, faster and cheaper both in time and storage, to only store Q_{inc} as a product of rotations.

Although incomplete, the factor Q_{inc} is however always orthogonal as it consists of a product of rotation matrices. A procedure for generating Q_{inc} from the Givens rotation matrices it consists of can be found in Bai et al. (2001). When preconditioning with the cIGO factorization, we actually require the computation of matrix-vector products of the form $Q_{inc}^T v$. An algorithm for this operation can be found in Section 3.4 of Papadopoulos, Duff and Wathen (2002) or in Section 5.1 of Golub and Van Loan (1996).

- The j-th row of R_{inc} is generated at step j of the procedure. Denote by $\tilde{a}_{ij}$ an entry of the partially reduced form of the matrix A after a series of rotations. Then, at the beginning of such a loop on j, we have the following situation:
  - If $\tilde{a}_{jj} \neq 0$ then $R_{inc}(j,j) \neq 0$;
  - Else, if $\tilde{a}_{jj} = 0$ but there is an entry $\tilde{a}_{ij} \neq 0$ for some i, $j < i \leq n$, then $R_{inc}(j,j) \neq 0$;
  - Else $R_{inc}(j,j)$ will be 0.

  If, for some j, we encounter the third possibility mentioned above, then of course R_{inc} will be a singular matrix. cIGO type factorizations of nonsingular matrices can give rise to such singular incomplete upper triangular factors. Two such examples can be found in the appendix of Papadopoulos et al. (2002) and also in James (1990). The easy way to remedy such a situation is simply to assign some arbitrary nonzero value to such a diagonal entry of R_{inc}, but this approach usually leads to preconditioners of poor quality convergence-wise.

2.2 column-Threshold IGO (cTIGO).

The cIGO process is very basic but yields a cheap preconditioner. However, in practice, and for the choice of P we consider here, having such a basic preconditioner is seldom effective. We need to produce incomplete factorizations that are closer to the full QR factorization and we can achieve this by allowing some fill-in by means of a threshold dropping strategy.

The cTIGO-Method Algorithm

1. For $j = 1, \ldots, n-1$ Do: *! for each column of A*
2. Define $k_r(j)$:=max$\{i \mid i \geq j \ , \ a_{ij} \neq 0\}$ *! max. row i with nonzero entry in column j of A*

3. If $k_r(j) = j$ Then Cycle *! no entries to be annihilated below diagonal on this column*

4. For $i = k_r(j)$ Down to $j+1$ Do: *! all nonzero subdiagonals in column j are successively annihilated*

5. If $(i,j) \in P$ or $|a_{ij}| > \tau$ Then: *! rotate out if entry in P or is a large enough fill-in*

6. Compute $\rho := \sqrt{a_{jj}^2 + a_{ij}^2}$ *! compute rotation angles*

7. Compute $c := a_{jj}/\rho$

8. Compute $s := a_{ij}/\rho$

9. Set $a_{jj} := \rho$

10. Store rotation data j, i, c, s

11. For $k = j+1, \ldots, n$ Do: *! compute updated i and j row segments*

12. Compute $temp_i := -sa_{jk} + ca_{ik}$

13. Compute $temp_j := ca_{jk} + sa_{ik}$

14. If $(j,k) \in P$ or $|temp_j| > \tau$, Set $a_{jk} := temp_j$

15. Else, Set $a_{jk} := 0$ *! entry too small so drop it*

16. If $(i,k) \in P$ or $|temp_i| > \tau$, Set $a_{ik} := temp_i$

17. Else, Set $a_{ik} := 0$ *! entry too small so drop it*

18. EndDo

19. Keep all entries in P plus the p largest fill-ins in the j-th row a_{j*}

20. Keep all entries in P plus the p largest fill-ins in the i-th row a_{i*}

21. EndDo *! end of secondary loop*

22. For $k = j, \ldots, n$ and $a_{jk} \neq 0$ Do: *! store final j-th row of R*

23. Set $r_{jk} := a_{jk}$

24. EndDo

25. EndDo

26. Set $r_{nn} := a_{nn}$

The same remarks apply as for the cIGO algorithm, but we should also note the following:

- In the case of cTIGO, we still work with a given sparsity pattern P, but, by the use of a thresholding strategy, we allow additional fill-in on top of this in order to generate preconditioners of potentially better quality. Hence, when $P = P_A$, the resulting incomplete factors will have $nz(R_{inc}) \geq nz(A_U)$ and Q_{inc} will be a product of at least $nz(A_L) - n$ rotations.

- The drop tolerance τ for the fill-ins may vary according to the row or column we are working with. For example, the following set of dropping rules can be used, assuming τ is the user input value for the drop tolerance and we are currently reducing column j of the matrix:

  - When determining on which rows to rotate, and $\tilde{a}_{ij}$ is a nonzero entry not in P, choose row i for rotation only if $|\tilde{a}_{ij}| > \tau|\tilde{a}_{jj}|$;
  - When updating the j-th and i-th rows, we keep a fill-in $\tilde{a}_{jk}$ (or $\tilde{a}_{ik}$) that is not in P only if $|\tilde{a}_{jk}| > \tau|\tilde{a}_{jj}|$ (respectively, if $|\tilde{a}_{ik}| > \tau|\tilde{a}_{ii}|$ for the i-th row).

  In fact, for square systems, this is exactly the thresholding strategy used in our code, but other combinations may also provide interesting results. The strategy used for rectangular (least-squares) systems is described in Section 4.2.

- The memory control tolerance p can also be adjusted for each row.

- Note again the combination of both column and row-oriented loops in the cTIGO algorithm. It turns out that because of the order in which we rotate out entries, accommodating and handling arbitrary fill-ins in the lower triangular part of A is quite difficult to program. Hence we use a sparsity pattern P as a basis on top of which we allow the extra fill-in, in the hope that we can obtain preconditioners of good quality without too much of this difficult to handle fill-in (see Sections 2.4 and 4.4).

2.3 Row-wise elimination TIGO (rTIGO).

This seems to be the traditional approach with respect to the order in which we rotate out entries in order to reduce A to upper triangular form (see James (1990), George and Heath (1980)). We will therefore limit ourselves to the following observations:

- The rows i, $i = 1, \ldots, j-1$, used to rotate out entries in row j when reducing it to upper triangular form, already have the required reduced form. Hence, using them will alter their content but only in the columns from i to n and their upper triangular form will be preserved. As for the j-th row, such a rotation will make the entry in position (j, i) zero but will not create any

fill-in in positions $(j,k), k = 1, \ldots, i-1$. Hence, we achieve our objective of reduction to upper triangular form.

- We now drop entries by magnitude only, without caring whether they belong to a sparsity pattern P or not. Hence, much sparser incomplete factorizations than in the case of cTIGO can be obtained. Their quality as preconditioners remains to be seen.

- The threshold drop tolerance τ can again vary in a dynamic fashion. In our implementation for square systems, and assuming we are currently reducing row j to upper triangular form by rotating its nonzeros against main diagonal entries in the previous $j-1$ rows, we have chosen the following combination:

  - To perform a rotation, the entry $\tilde{a}_{ji}$ we want to rotate out must satisfy the criterion $|\tilde{a}_{ji}| > \tau |\tilde{a}_{ii}|$, otherwise we drop it (skip the rotation).
  - For row j which we are currently reducing, we drop entries $\tilde{a}_{jk}$ unless they satisfy $|\tilde{a}_{jk}| > \tau \times \frac{||A(j,:)||_1}{nz(A(j,:))}$. The dropping rule is therefore the weighted 1-norm of the j-th row of the original matrix. Similarly, for row i, we drop entries $\tilde{a}_{ik}$ not satisfying $|\tilde{a}_{ik}| > \tau \times \frac{||A(i,:)||_1}{nz(A(i,:))}$.

  Note that the rule applies to off-diagonal entries only. The entries on the main diagonal of the matrix are always kept regardless of their magnitude in order to increase the potential of R_{inc} to be nonsingular. This is also true for the cIGO and cTIGO algorithms. The strategy used for rectangular (least-squares) systems is described in Section 4.2.

2.4 Computer implementation details.

The methods of the previous section were coded in Fortran 77 using standard sparse matrix and vector data structures, as well as routines for sparse matrix-vector or vector-vector operations (Duff, Erisman and Reid 1986, Saad 1996).

We only briefly mention some of the issues related to coding these methods. For the interested reader, a detailed presentation of the computer implementation can be found in Papadopoulos et al. (2002). The source code is available from the authors.

For the first two of these IGO methods, eliminations for reducing the matrix to upper triangular form are performed in a column-wise fashion, but after each such elimination two row segments of the matrix need to be updated. Hence, we will naturally use Compressed Sparse Row (CSR) structures to accommodate our matrix and any fill-in in a row-wise fashion. However, we have to keep track of the column by column distribution of the original nonzeros and any fill-ins in the lower triangular part of the matrix so that the next rotation in a particular column can be found without requiring prohibitive searching.

For cIGO, where the sparsity pattern is predetermined and no other fill-in is allowed during the reduction process, this is done once at the beginning: based on the CSR structure for the nonzeros of the matrix, we build the associated CSC structure, but with the real array substituted by an integer one containing pointers to the position of the entry (and hence the value) in the CSR structure.

For cTIGO, things are more complicated as fill-in now can appear in a row after a rotation. We have chosen to set up a linked list that keeps track of the column-wise distribution of this fill-in. Any time a fill-in appears, we add the information of its position to the end of the linked list and adjust the necessary links to account for this new entry. This approach has two major drawbacks. Firstly, the length of the linked list increases with each new fill-in and this structure will inevitably develop 'holes' in it ('dead' links corresponding to entries that appeared as fill-ins but were dropped at a subsequent rotation). Since for a linked list we do not have a predetermined amount of space available for each column, we can afford to have such holes. 'Garbage collection' operations to cover these holes were deemed too complex and expensive, although they would limit the size of the list if implemented. The most important drawback however seems to be the fact that information about fill-ins in a particular column is in general scattered over the whole of the linked list, thus eliminating any hope of efficient cacheing when accessing or altering entries in this structure. An alternative to the linked list would be a CSC type structure. This would keep information about each column together but, since the maximum available space for each column has to be predetermined, it would require frequent 'garbage collections' to get rid of 'holes' (fill-ins that were dropped at a subsequent rotation), or we would quickly run out of space.

The need to produce and update the column by column information on fill-ins is the cause of bottlenecks in cTIGO. We can witness this by comparing numerical results for cTIGO and rTIGO, as for rTIGO only row-oriented structures are necessary (see Papadopoulos et al. (2002), Section 3).

Regarding the application of these incomplete factorizations as preconditioners, we briefly note that for non-symmetric Krylov subspace methods (for example, GMRES, Bi-CGSTAB) we apply the preconditioner $M = Q_{inc}R_{inc}$. Right-preconditioning for these methods then effectively consists of the solution of systems of the form $Mz = v$. As Q_{inc} is available as a series of rts rotations and is orthogonal by construction, these can be applied successively on the vector v to compute the matrix-vector product $Q_{inc}^T v$. The resulting vector then acts as a right-hand side to a triangular solve involving R_{inc}. For CGNR, the method we used for rectangular systems, the preconditioner is $M' = R_{inc}^T R_{inc}$. Hence, preconditioning the CGNR iteration involves two successive triangular solves. The total number of floating-point operations required to create the right-hand side of the system $R_{inc}z = Q_{inc}^T v$ is $rts \times$ (4 multiplications $+ 2$ additions/subtractions). A triangular solve involving either R_{inc} or its transpose, requires $nz(R_{inc}) - n$ multiplications and

additions as well as n divisions. This is the total amount of extra work required for preconditioning per iteration.

3 Preconditioning Square Systems.

3.1 Description of experiments.

In this chapter, we shall present a series of numerical results for matrices arising from a variety of applications. The matrices are all available in the public domain, either in the MatrixMarket (MatrixMarket 2000) database or in the collection of sparse matrices at the University of Florida (Davis 2000). In these experiments, we have used the IGO methods as preconditioners for either GMRES or Bi-CGSTAB. We compare the performance of these methods with a variety of ILU type preconditioners. These ILU factorizations were all obtained using Saad's Sparskit2 package (Saad 1994), also written in Fortran77, so there is compatibility of programs used and timings. All the experiments were run on a Sun Ultra 5 workstation.

The incomplete LU type preconditioners we tested are the following: ILU(0) and modified ILU(0), two simple nonparametric factorization techniques that work with the sparsity pattern of A only, ILUt, which uses a threshold dropping rule with no sparsity pattern sets and ILUtp, which is the same as ILUt but allows column pivoting. Extensive description of these methods can be found in Saad (1994) and Saad (1996).

Since ILU methods generate very powerful preconditioners for matrices possessing structure like bandedness or diagonal dominance, as can occur with matrices arising from the discretization of partial differential equations, we have concentrated on more general examples whose sparsity pattern or distribution of values might cause difficulties for ILU methods.

For both iterative methods, the preconditioner is always applied on the right. Again, for both methods, the stopping criterion for the iteration was

$$||b - Ax^{(i)}||_2 \leq ||b - Ax^{(0)}||_2 \times 10^{-9},$$

with the null vector as the initial guess. The right-hand side vector, was taken to be a random vector.

We should finally mention that, especially for ILUtp where column pivoting is allowed, the choice of the pivoting parameter $\tilde{p}$ was 0.01. This is the default value in Sparskit2. Since the ILUtp rule for pivoting between columns i and j at step i is $|\tilde{a}_{ij}| \times \tilde{p} > |\tilde{a}_{ii}|$, this value of $\tilde{p}$ means that we do not pivot often.

Note also that, because of the way we coded Bi-CGSTAB in the experiments, each iteration consists in fact of two matrix-vector products, that is, the Krylov space is twice augmented per Bi-CGSTAB iteration. For GMRES, each iteration corresponds to augmenting our Krylov space by one.

For GMRES, we always choose a restart value of 50. Hence, when we say that GMRES has converged at iteration 123, we mean that it has restarted twice and at the 23^{rd} iteration after the second restart we achieve convergence.

3.2 Examples in brief.

In this section, we will list results for a variety of matrices. These are actually the best results in terms of execution time we could obtain for each method for the threshold parameter values (normally powers of 10) that we tested for each system. The two tables in this section list the attributes and performance of cTIGO (or just cIGO when no fill-in was necessary), or rTIGO respectively, against ILUtp (or ILU(0)), ILUtp apparently being the most powerful and robust of the ILU preconditioners we had at our disposal.

Each table should be read as follows; for each matrix, we have two rows of results, the first shows attributes of the cTIGO (or rTIGO) preconditioner, and the second one shows the results for ILUtp (or ILU(0)) respectively, to facilitate comparison between the observed performance of the best IGO and the best ILU method used.

Note that when, under the column marked 'GMRES its' and next to the number of iterations, a B within brackets ([B]) appears, restarted GMRES was not converging so we had to employ Bi-CGSTAB instead to achieve convergence.

The results in Table 3.1 show that cTIGO is a robust preconditioner, in the sense that if enough fill-in is allowed it can eventually give preconditioners of good quality. The time it takes to compute these preconditioners is usually much higher than for ILUtp, although the density of the two resulting incomplete factorizations is comparable.

Note that throughout the table there are matrices for which cIGO was enough to give a fairly good preconditioner. For these same systems, ILU(0) or MILU(0) did not achieve convergence, and we had to use the threshold method ILUtp. The opposite was observed in the case of matrices arising from the streamline-upwinding Petrov-Galerkin discretization of advection-diffusion problems (see Fischer, Ramage, Silvester and Wathen (1999)), where for many tested cases ILU(0) preconditioning is of good quality but cIGO gives no improvement.

Two matrices for which cIGO (and cTIGO with very little fill-in) performs much better than the ILU methods are the *garon* matrices. These arise in the discretization of 2-D Navier-Stokes problems and, as is usual for such systems, have a small zero block in the bottom right corner. This is probably what causes difficulty with ILU methods. On the other hand, even for cIGO, the zero entries on the main diagonal of this block are covered with values during the incomplete factorization process and the resulting preconditioned iterative method converges.

Going through the results, one also sees that things can easily go terribly wrong for cTIGO. In order to get convergent preconditioners one has to lower the value for τ. But this results in large fill-in in the lower triangular part. This causes cTIGO

to slow down, and eventually, the timings we get do not reflect actual floating-point operations, but rather updating and going through our various structures, particularly the linked list. This issue is addressed in Section 4.4.

We have also identified some examples where we were unable to produce convergent preconditioners using cTIGO. That is, lowering the threshold value enough so that the preconditioner actually makes the method convergent rather than stagnating, causes so much fill-in in the lower triangular part that we run out of memory; the linked list grows with each insertion and quickly reaches the maximum length we have allocated for it. Matrices for which we encountered such problems include *barth*, *barth*4, *bcsstk*24, *can*_1054, $fidap$14, $fidap$35, $finan$512, *graham*, *lns*_3937, *nasa*2910, *plat*1919. Of course, if enough memory (and time) was available we would eventually reach a threshold value for which the cTIGO preconditioner would converge.

The results for rTIGO can be found in Table 3.2. Again, note that rTIGO is almost always slower in computing the preconditioner than the best choice for ILUtp. However, for some of the problems, rTIGO is considerably faster than cTIGO, despite the fact that the preconditioner it computes can be much denser than that from cTIGO. This reinforces our belief that it is the need for keeping and constantly accessing/updating a linked list structure that undermines cTIGO.

On the other hand, rTIGO fails to produce good preconditioners for quite a few problems, especially matrices with zeros on the diagonal, before the limit of available memory is reached. Such examples, apart from the *garon* matrices, are *bcsstk*24, *goodwin*, *gre*1107, *lhr*02, *lns*_3937, *plat*1919, *utm*_5940.

3.3 The effect of preprocessing.

We present one example showing how preprocessing a matrix can improve the overall performance of incomplete factorization preconditioning.

Preprocessing and reordering matrices for ILU has been the focus of some attention recently (see Benzi et al. (2000), Benzi et al. (1999)). Since they only involve computations with integers, reordering methods are usually quite cheap to use. On the other hand, the advantages we get by preconditioning the reordered matrix clearly outweigh the cost of applying a reordering scheme such as reverse Cuthill-McKee or the HSL code MC64. The example we present here clearly shows that reordering the matrix benefits both the IGO and the ILU methods.

We have chosen matrix *bayer*02 arising in chemical kinetics to illustrate this (Figure 3.1). We first ran both cTIGO and ILUtp on the original system, and then reordered it using the HSL 2000 code MC64 (using option 5) (Duff and Koster 2001). cTIGO and ILUtp preconditioning was then applied to the reordered system. The results are shown in Tables 3.3 and 3.4.

MC64 preprocessing of the matrix not only altered the sparsity pattern for the better, in the sense that the reordered matrix diagonal entries are nonzero and most

Table 1: Best results: cTIGO vs ILUtp.

matrix	cTIGO τ / ILUtp τ	$nz(R_{inc})$ / $nz(U)$	*rots* / $nz(L)$	cTIGO time / ILUtp time	GMRES its.	GMRES time
memplus	cIGO	71954	50073	11.6	189 [B]	57.5
	ILU(0)	71954	54197	1.3	200 [B]	51.7
*rdist*1	10^{-3}	353622	146555	52.3	20	5.3
	10^{-8}	161766	480445	9.5	29	5.2
*garon*1	cIGO	46051	39416	1.7	206	17.9
	10^{-3}	162758	306347	26.8	42	8.2
*garon*2	cIGO	202071	173080	7.7	218 [B]	156.1
	10^{-2}	218430	494309	13.9	469 [B]	324.2
*meg*4	cIGO	26378	10462	0.6	2	0.1
	10^{-6}	6202	463	0.2	2	0.1
*add*32	10^{-1}	22030	9058	2.6	10	0.6
	10^{-3}	28216	19364	0.21	4	0.16
b_dyn	10^{-8}	21586	11123	1.6	8	0.15
	10^{-6}	10897	7980	0.1	8	0.1
*bp_*800	10^{-2}	24624	28902	19.2	33	0.9
	10^{-2}	14085	9265	0.2	36	0.4
*dwt_*2680	10^{-2}	183860	194045	160.5	300	49.5
	10^{-3}	275282	298096	19.9	33	4.8
*fidap*3	10^{-8}	254415	85776	51.4	14	2.3
	10^{-8}	90406	61372	1.9	17	1.3
*fidap*24	10^{-3}	223508	118814	69.5	18	2.3
	10^{-2}	69901	91570	2.0	29	1.7
*fidap*37	cIGO	67591	32013	1.9	11	0.7
	0.1	5973	22143	0.2	13	0.5
*gemat*11	0.50	44315	211489	125.1	514	97.2
	10^{-4}	115037	99766	3.6	13	1.0
*gre*1107	10^{-2}	52700	53291	37.2	90	4.3
	10^{-3}	86816	90134	4.1	22	1.0
*lhr*01	10^{-3}	188102	221994	195.5	51	12.7
	10^{-6}	71993	51633	1.8	4	0.2
mahindas	0.1	13615	42360	18.0	341	13.7
	10^{-2}	24414	11419	0.4	44	0.9
*orani*678	10^{10}	45373	95161	83.4	519	55.3
	10^{-2}	330246	135776	41.0	25	4.0
*orsirr_*1	10^{-2}	5926	3158	0.46	30	0.34
	10^{-4}	9515	7642	0.11	9	0.09
*qh*1484	10^{-10}	66531	39189	20.1	69	3.2
	10^{-10}	24026	17857	0.5	62	1.4
poli_large	1.0	33494	1048	5.4	20	2.0
	10^{-2}	44270	26819	0.4	6	0.5
*raefsky*5	10^{10}	86975	81428	8.0	26	3.6
	10^{-2}	57863	62439	0.5	5	0.5
thermal	cIGO	34992	31536	1.1	7	0.9
	10^{-2}	21100	22989	0.2	6	0.25

Table 2: Best results: rTIGO vs ILUtp.

matrix	rTIGO τ	$nz(R_{inc})$	$\#rots$	rTIGO time	GMRES	GMRES
	ILUtp τ	$nz(U)$	$nz(L)$	ILUtp time	its.	time
memplus	10^{-2}	76142	54089	11.3	33 [B]	4.7
	10^{-2}	72619	55321	1.0	14 [B]	3.6
*rdist*1	10^{-4}	331769	127197	28.6	25	4.4
	10^{-8}	161766	480445	9.5	29	5.2
*meg*4	10^{-6}	25226	342	1.0	2	0.1
	10^{-6}	6202	463	0.2	2	0.1
*bp*_800	10^{-3}	44163	14535	4.4	30	0.7
	10^{-2}	14085	9265	0.2	36	0.4
*can*_1054	10^{-2}	196366	116026	56.2	196	21.4
	10^{-2}	248320	250526	37.0	34	5.4
*dwt*_2680	10^{-2}	237770	137368	38.2	247	36.4
	10^{-3}	275282	298096	19.9	33	4.8
*fidap*24	10^{-3}	271764	121090	36.6	15	2.1
	10^{-2}	69901	91570	2.0	29	1.7
*fidap*35	10^{-10}	1251839	616062	20.8	10	2.9
	10^{-9}	564113	516668	3.5	8	1.4
*fidap*37	0.1	28449	2504	0.14	13	0.16
	0.1	5973	22143	0.2	13	0.5
*gemat*11	10^{-3}	300681	153308	39.7	144	28.4
+*mc*64/5	10^{-3}	55897	55983	0.9	32	2.2
mahindas	10^{-2}	23827	10867	1.4	36	0.8
	10^{-2}	24414	11419	0.4	44	0.9
*orani*678	10^{-2}	200882	117335	100.5	286	53.9
	10^{-2}	330246	135776	41.0	25	4.0
*qh*1484	10^{-8}	44066	14082	2.8	16	0.4
	10^{-10}	24026	17857	0.5	62	1.4

Table 3: *bayer*02: cTIGO vs ILUtp.

τ	10^{-10}	10^{-11}	10^{-12}		10^{-8}	10^{-10}	10^{-12}
$nz(R_{inc})$	461108	468881	470406	$nz(U)$	123374	131919	140328
$\#rots$	205364	205842	206374	$nz(L)$	168517	191947	210069
cTIGO time	59.2	59.4	60.0	ILUtp	2.0	2.6	3.2
GMRES(50) its	stagn	3	3		stagn	4	3
GMRES(50) time	n/a	1.3	1.4		n/a	1.2	0.9

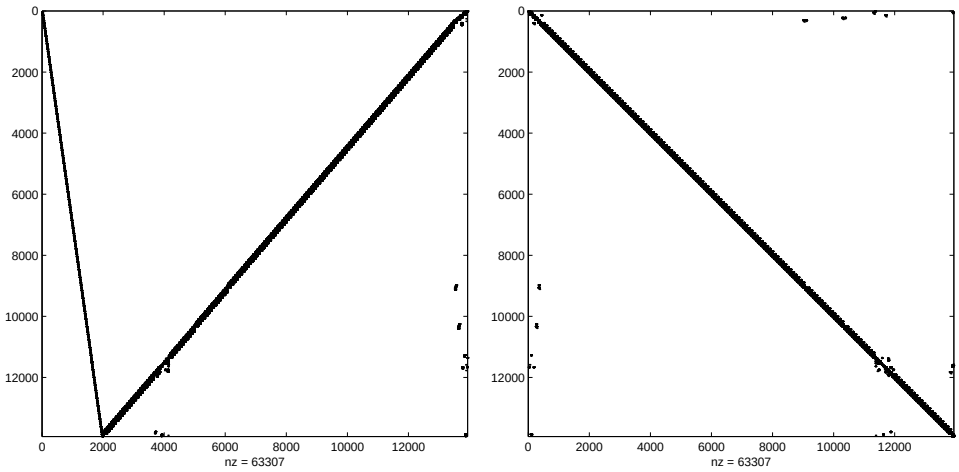

Figure 1: Matrix *bayer*02, original: $n = 13935$, $nz(A_U) = 36518$, $nz(A_L) = 27161$, condest$(A) = 2.3 \times 10^{18}$, and after applying MC64 (JOB=5 option): $n = 13935$, $nz(A_U) = 33676$, $nz(A_L) = 30003$, condest$(A) = 3.6 \times 10^3$.

Table 4: *bayer*02, MC64/5: cTIGO vs ILUtp.

τ	1.0	10^{-1}	10^{-2}		1.0	10^{-1}	10^{-2}
$nz(R_{inc})$	38247	82668	143441	$nz(U)$	13935	33568	52432
$\#rots$	29415	39199	63709	$nz(L)$	5353	20924	37849
cTIGO time	7.1	7.6	14.4	ILUtp	0.2	0.2	0.3
GMRES(50) its	stagn	38	10		stagn	9	5
GMRES(50) time	n/a	6.2	2.2		n/a	0.8	0.5

entries are within a small band around the diagonal, but it also greatly improved the condition number.

This improvement is reflected in the performance of the preconditioners. For the reordered matrix, the incomplete factors, required to achieve convergence in a similar number of iterations as for the original matrix, are sparser and cheaper to compute. This is true for both cTIGO and ILUtp. Hence, although cTIGO benefits from such a preprocessing of the matrix, so does ILUtp, so that the overall ratio by which ILUtp outperforms cTIGO does not change. The same observation applies for rTIGO. Note also that, beneficial as it may be for many problems, preprocessing or reordering is not a panacea. For example, MC64 preprocessing has no discernible effect if applied to the *garon* matrices. The diagonal of the zero block on the bottom right corner gets covered with nonzeros, but the incomplete factorization methods still struggle to produce a preconditioner with good convergence properties.

3.4 Discussion.

The numerical experiments show that IGO methods can provide robust and accurate preconditioners for Krylov subspace iterations for solving large sparse systems of linear equations.

We have tested the performance of these methods against a wide range of standard incomplete LU factorization techniques.

In the case of the non-parametric preconditioners of either type (cIGO with $P = P_A$ and ILU(0)), we have identified cases where cIGO outperforms ILU(0), cases where the opposite happens, as well as cases where either preconditioner will work. Hence, if a simple preconditioner is required, one could use cIGO rather than ILU(0) and achieve better convergence rates at a similar cost for the computation of the preconditioner.

The situation changes, however, if more powerful preconditioning is required, either through thresholding or through extended sparsity pattern sets.

With few exceptions, the time required to compute factorizations of similar preconditioning quality (sparsity of factors, convergence rate of iteration) using either of our methods can be much larger than the corresponding time using one of the available ILU methods.

The programming issues related to why our methods can get swamped during the computation of the incomplete factorization have been addressed. The main issue seems to be the fill-in in the lower triangular part of the matrix and, consequently, the number of rotations performed during the incomplete reduction process. In some cases, one can limit these without affecting the quality of the preconditioner by using a standard reordering or preprocessing scheme applied to the matrix before the incomplete factorization process begins. On the other hand, applying such a reordering scheme to the matrix *a priori*, will also benefit the incomplete LU method. Hence, when we compare the performance of these two classes of incomplete factorizations for the reordered system, the situation will remain unchanged in favour of the ILU methods.

With respect to this problem, we could use different combinations of threshold dropping rules than those used in our earlier experiments. A detailed discussion follows in Section 4.4.

To sum up, it seems that incomplete Givens orthogonalization methods cannot compete with incomplete LU factorization techniques in general. With very few notable exceptions, we have not been able to find examples for which some type of incomplete LU factorization with a good choice of parameters could not perform at least as well as the best of our methods.

On the other hand, the algorithms presented here can be adapted for use with rectangular matrices, and, as we shall see in the next section, it is in this area of preconditioning least-squares problems that they prove more successful.

4 Preconditioning least-squares systems.

Since a QR factorization is well-defined for rectangular systems one can naturally extend the Givens algorithms presented above to produce incomplete factorizations of such systems. More precisely, we will deal with least-squares problems, namely

$$min||Ax - b||_2, \ A \in R^{m \times n}, b \in R^m, x \in R^n, \text{ with } m \geq n.$$

Assuming that an incomplete Givens factorization $Q_{inc}R_{inc}$, $Q_{inc} \in R^{m \times m}$, $R_{inc} \in R^{m \times n}$ has been computed, one can use it to precondition conjugate gradients for the associated system of normal equations $A^T Ax = A^T b$. Since Q_{inc} is orthogonal, the natural preconditioner to use is $M = R_{inc}^T R_{inc}$, which is positive definite if R_{inc} is nonsingular.

Note also that, for this choice of preconditioner, and since by construction all the rows of R_{inc} below the n^{th} are zero, one can disregard them from the start and use the *truncated* incomplete triangular factor $\tilde{R}_{inc} \in R^{n \times n}$. The truncated factor is exactly R_{inc} but with these last zero rows omitted. To simplify notation, from now on R_{inc} will refer to this truncated factor.

Below, we shall address particular issues concerning the construction, efficient implementation and performance of this preconditioner.

4.1 Computer implementation.

Extending our incomplete Givens factorization code to handle such rectangular systems requires only simple alterations, mainly to account for the fact that there is no main diagonal entry for the rows of A after the n^{th}, and hence no natural cut-off point between the upper and lower triangular part of the matrix undergoing Givens reduction to upper triangular form. The setup and data structures required for the IGO variants presented previously remain the same. Some auxiliary routines also had to be modified to handle rectangular matrices.

Despite the fact that we are using the normal equations, there is no need to compute $A^T A$ explicitly, but instead we apply two successive matrix-vector products where necessary within the conjugate gradient algorithm. Indeed, this is implicitly done in the LSQR algorithm (Paige and Saunders 1982).

Similarly, for preconditioning this iteration, there is no need to form $R_{inc}^T R_{inc}$ explicitly, an approach which would be too expensive both in computation and storage. Instead, and since it is legitimate to use the square truncated triangular factors, we apply a lower followed by an upper triangular solve to precondition each CGNR iteration. Hence, there is no need to store the series of rotations that form Q_{inc}, as was necessary in the square case.

4.2 Dropping rules.

The fact that there is no main diagonal entry in the last rows of A and for the rows where there is such an entry its value may well be zero, call for some modifications in the threshold dropping rules.

The memory control dropping rules remain the same as in the square case.

The threshold dropping rules for cTIGO were chosen to be

$$P_A \cup \{a(k,j) \text{ s.t. } a(k,j) > a(j,j) * \tau,\ m \geq k > j\},$$

when determining which entries to rotate out in column j. This is the same dropping rule that we employed in the square case.

When j is a pivot row, the dropping rule now becomes

$$P_A \cup \{a(j,k) \text{ s.t. } a(j,k) > ||A(j,:)||_2 * \tau,\ n \geq k > j\},$$

where the norm notation refers to the 2-norm of the j^{th} row of the original matrix A before any reduction using Givens rotations has begun.

For row i, in which an entry to be rotated out belongs, the dropping rule becomes

$$P_A \cup \{a(i,k) \text{ s.t. } a(i,k) > ||A(i,:)||_2 * \tau,\ n \geq k > j\},\ j+1 \leq i \leq m.$$

Note that, for cTIGO, we have now removed the dependence of the threshold on the diagonal entry when updating the two rows involved in a rotation.

For rTIGO, when determining the next entry to be rotated out in row j, $1 < j \leq m$ that is reduced to upper triangular form (or to zero if $j > n$), we check whether

$$a(j,k) > a(k,k) * \tau,\ k = 1, \ldots, \min(j-1, n),$$

where this k^{th} row has already been reduced to upper triangular form.

After the rotation has been performed, we retain the entries $a(j,k)$ in this row j undergoing reduction if they satisfy

$$a(j,k) > \tau * ||A(j,:)||_2,\ k = 1, \ldots, \min(j-1, n).$$

Similarly, for row i, $i = 1, \ldots, n$ whose main diagonal entry will act as pivot for the rotation performed on row j, after each rotation we will retain entries $a(i,k)$ that satisfy

$$a(i,k) > \tau * ||A(i,:)||_2,\ k = 1, \ldots, n.$$

For both cTIGO and rTIGO, we have removed as much as possible the dependence of the dropping rule on values of main diagonal entries that might well be zero. In doing so, however, the dropping rule becomes less adaptive as it depends on two fixed quantities, namely τ and a norm of a row of the original matrix. In Section 4.4, we discuss why this change actually improves performance.

Table 1: Matrix attributes.

matrix	size	$nz(A)$	initial zero diags.	vanilla its.	CGNR time	cIGO CGNR its.
*illc*_1033	1033×320	5047	315	3470	4.7	6023
*illc*_1850	1850×712	9463	705	2114	6.6	> 3000
*jimenez*_4	385×361	2013	352	9752	6.6	> 10000
*jimenez*_5	574×526	3886	521	22636	26.7	> 50000
*well*_1033	1033×320	5047	315	170	0.22	453
*well*_1850	1850×712	9463	705	447	1.39	960
*pigs_s*1	3140×1988	9525	1015	555	2.63	843
*pigs_s*2	6280×3976	27560	2030	1107	13.3	2168
*pigs_m*1	9397×6119	31131	6118	745	11.5	1464
*pigs_m*2	18794×12238	87275	12236	1553	63.6	3241
*pigs_l*1	28254×17264	92282	17264	1169	59.02	2368
*pigs_l*2	56508×34528	259582	34528	2346	304.6	3574
jordache	24708×23937	218568	23292	> 3000	n/a	> 1000

4.3 Numerical results.

In all the tests presented below we follow the same basic guidelines outlined in the square systems case. Again, we allow enough space for the memory control parameter to be rendered ineffective, so only threshold dropping applies. The right-hand-side is a vector of ones of appropriate size. The initial guess for the iterative method (CGNR) is always the zero vector. Convergence is assumed once the initial residual has been reduced by nine orders of magnitude in the Euclidean norm. This set of results was obtained on a Pentium III at 800 MHz.
The online repositories of sparse matrices that supplied us with test matrices for square systems are curiously lacking in least-squares systems. This is quite possibly due to the fact that such systems, although arising quite frequently in scientific computing, are not at the focus of research in preconditioning methods.

We present here results for most of the least-squares matrices available in Matrix Market and the Rutherford-Boeing collection. Some matrices we left out; the smallest ones from the *jimenez* collection as they were considered too small (dimensions ranging from 100 to 300), as well as the largest one of the *pigs* matrices ($pigs_xl$), as it was too large for our available resources.

The fundamental attributes of our test set of matrices can be found in Table 4.1.

Note that the preconditioner resulting from applying the simple cIGO algorithm is bad in all cases. In fact, it takes approximately twice the number of iterations for CGNR preconditioned with cIGO to achieve the same residual reduction as the

simple CGNR iteration without any preconditioning. On the other hand, and for the larger examples in particular, the need for an efficient preconditioner for CGNR is self-evident.

As in the square case, for each matrix, we performed a number of runs, each time varying the dropping threshold value τ. We present here the result for that value of τ for which the total solution time (incomplete factorization followed by one preconditioned iterative solution of the system) was minimized. We realise that this reflects the basic difficulty of selecting parameters in a parameter dependent method such as cTIGO, rTIGO or ILUt, ILUtp. For each matrix, we present the best such run for cTIGO versus rTIGO in Table 4.2.

As was the case for square systems, fine-tuned rTIGO will be about three to four times faster in generating the preconditioner than its column-based counterpart cTIGO. rTIGO computes a preconditioner that is qualitatively better than cTIGO, although sometimes significantly denser.

In all cases, the preconditioners significantly reduce the number of iterations when compared to simple CGNR, and there are large savings in terms of total solution time despite the extra time necessary to compute the preconditioner or the added work for each iteration due to the triangular solves.

The preconditioners in all cases have one to two times the number of nonzero entries as the system coefficient matrix, so they can be considered very effective in terms of required storage. Note that this does not apply to the intermediate storage space necessary when performing the IGO type reduction; in fact, both cTIGO and rTIGO fail to produce convergent preconditioners for the *jordache* matrix and for the lowest value of the dropping threshold τ we could try before we ran out of memory during the incomplete factorization phase.

4.4 Discussion.

The numerical results presented in the previous section show the fine-tuned performance of the IGO methods, but we feel some further insights and comments should be made.

Variation of performance with τ: In Table 4.2, only the results for the experimentally optimal value for τ for each IGO variant and matrix were listed. They showed the savings one can have over simple CGNR iterations if a TIGO preconditioner is used.

In fact, for each problem, the threshold value τ can vary over a whole interval and still produce a preconditioner that will benefit the iterative solution. We have chosen one of the smaller test matrices, so that τ can be taken very small without encountering memory overflow because of fill-in, to illustrate this (see Figure 4.1 which is plotted on a $log-log$ scale).

We clearly see that a choice of τ between 10^{-5} and 10^{-1} for cTIGO or less

Table 2: Best results: cTIGO vs rTIGO.

matrix	cTIGO τ rTIGO τ	$nz(R_{inc})$ $nz(R_{inc})$	cTIGO time rTIGO time	pCGNR its.	pCGNR time
*illc*_1033	10^{-3}	3919	0.76	80	0.19
	10^{-2}	2550	0.12	198	0.37
*illc*_1850	10^{-2}	9945	1.23	83	0.45
	10^{-2}	13581	0.98	75	0.47
*jimenez*_4	10^{-6}	29157	3.0	16	0.13
	10^{-6}	30241	0.38	11	0.09
*jimenez*_5	10^{-5}	71339	11.07	44	0.8
	10^{-5}	81304	6.87	111	2.2
*well*_1033	0.05	2825	0.27	76	0.15
	10^{-1}	2551	0.06	66	0.12
*well*_1850	0.05	6469	0.41	71	0.33
	0.05	8181	0.30	52	0.27
*pigs_s*1	10^{-1}	8385	0.36	99	0.63
	10^{-1}	11851	0.17	59	0.42
*pigs_s*2	0.05	27374	1.55	114	2.05
	0.1	35644	0.53	113	2.27
*pigs_m*1	10^{-1}	33965	1.45	103	2.43
	10^{-1}	37311	0.59	65	1.56
*pigs_m*2	10^{-1}	82746	5.15	182	11.23
	10^{-1}	112365	1.81	132	8.76
*pigs_l*1	10^{-1}	104457	8.35	121	9.24
	10^{-1}	109915	1.75	78	5.85
*pigs_l*2	0.05	314433	51.6	158	31.5
	0.05	436328	13.50	138	31.1
jordache	10.0	171614	8.83	> 1000	n/a
	0.05	614266	6.18	> 1000	n/a

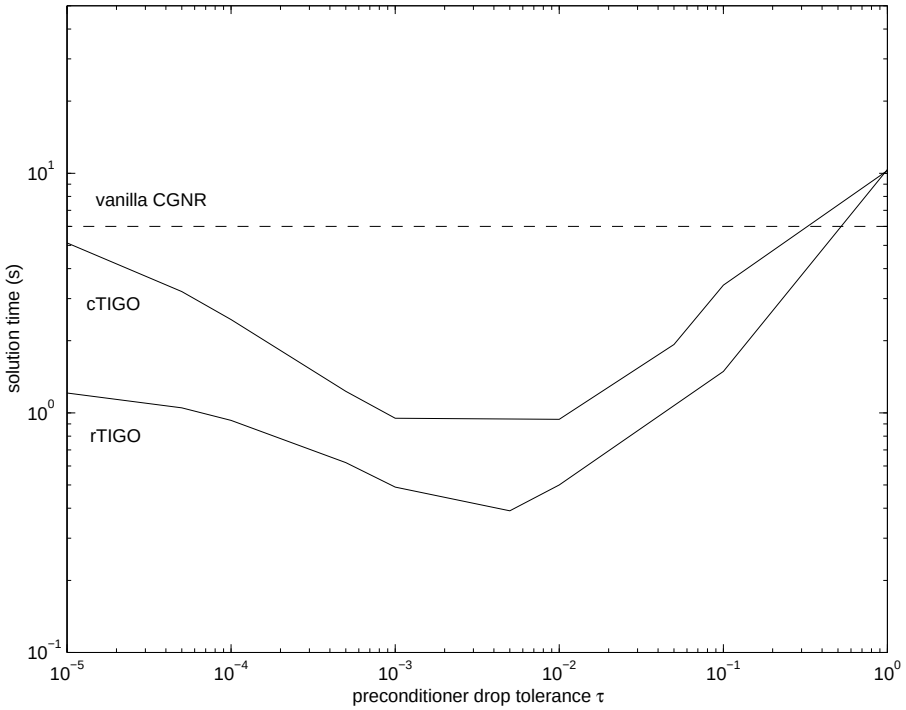

Figure 1: Total solution time (incomplete factorization followed by solution using preconditioned CGNR) for cTIGO & rTIGO CGNR preconditioning for various τ, versus simple CGNR iteration, for the matrix *illc*_1033. *log*−*log* scale.

than 10^{-1} for rTIGO, improves solution time when compared to the simple CGNR iteration.

Since we cannot know *a priori* the best value for τ, we see that a sensible choice can give good savings.

Finally, note the roughly convex shape of the total solution time curves for both methods. The explanation for this behaviour is quite simple. When τ is large, the preconditioner is cheap to compute but qualitatively poor, so the preconditioned iteration converges slowly. Hence, total solution time is high and dominated by the preconditioned iteration. On the other hand, for τ small, the preconditioner is of very good quality, so the preconditioned iteration converges fast, but is expensive to compute. Hence, total solution time is again high, but now dominated by the time it takes to generate the preconditioner. The value(s) of τ for which the total solution time is lowest are exactly those for which a balance between these two extremes is achieved: the preconditioner is relatively cheap to compute and it has a fairly good convergence behaviour.

cTIGO dependence on linked list length: We have already pointed out our observation that the computation of cTIGO is heavily dependent on the length of

the linked list structure which holds column-wise information on the fill-ins in the lower triangular part of the matrix.

We illustrate this observation with the example presented in Figure 4.2 (plotted on a $log-log$ scale). The chosen matrix ($pigs\_s2$) is small, but exhibits this fundamental disadvantage of the linked list.

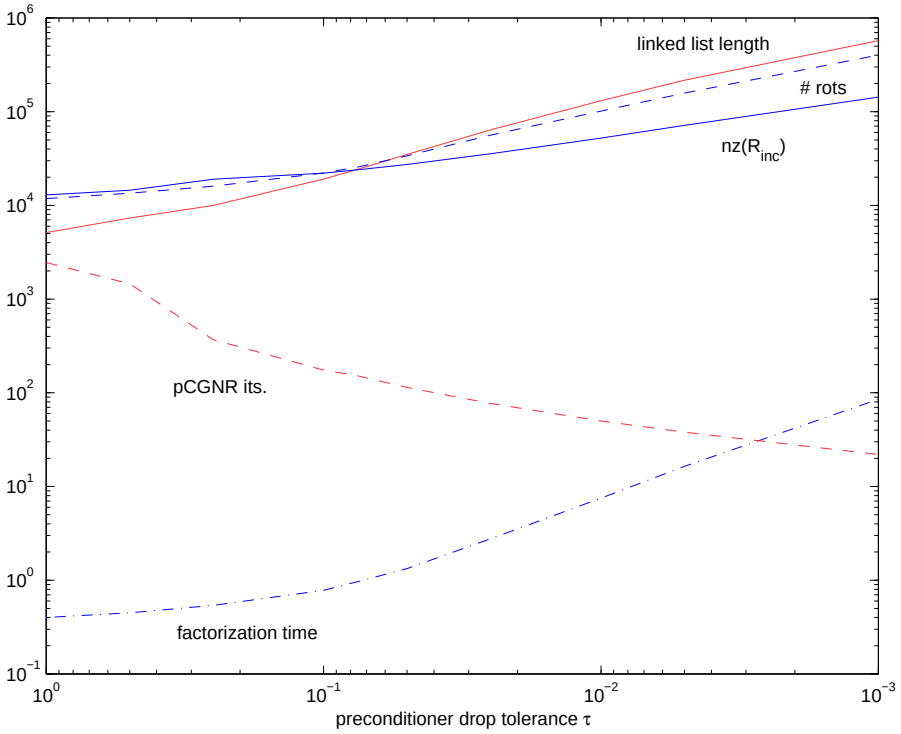

Figure 2: Evolution of preconditioner fundamental data sizes and CGNR iterations over τ, $log-log$ scale. Test problem is $pigs\_s2$.

Note that, for τ less than 10^{-1}, the length of the linked list and the number of rotations performed increase at a higher rate than the number of nonzeros in the incomplete triangular factor. This increased rate is reflected in the time it takes to compute the incomplete factorization. However, the usefulness of this extra amount of work when producing the incomplete factorization is not evident in the number of iterations that pCGNR needs for convergence: they keep decreasing at the same rate. Hence, for τ less than 10^{-1} in this example, cTIGO performs many unnecessary rotations, in the sense that they do not have a compensating impact on the quality of the resulting preconditioner.

This is a vicious circle, as the more such unnecessary rotations we have, the longer the linked list becomes, so not only does it eat up resources but it becomes very time consuming to update. More unnecessary fill-in will appear particularly

in the lower triangular part of the matrix, which in turn generally leads to more rotations being performed to get rid of this fill-in.

The necessity of keeping the size of the linked list short also justifies our choice of always keeping all entries in the original sparsity pattern P_A of the matrix A. We aim to produce qualitatively good preconditioners with only a little of this difficult to accommodate fill-in on top of the nonzeros in the sparsity pattern.

Threshold strategy dependence on main diagonal entry: For systems arising from least-squares problems, most (all) the entries on the main diagonal are usually zero. This means that for a threshold rule that depends on main diagonal values much fill-in will appear, at least for the first rotation involving such a row with a zero initial main diagonal entry. This fill-in will in many cases accumulate during the factorization phase and kill performance (see Papadopoulos et al. (2002), Section 5.4).

It is however our experience that such a preconditioner will only start helping convergence once most (or in some cases, all) the main diagonal entries of R_{inc} have been filled with some value. Just filling those zero diagonal entries with an arbitrary value (in our code we choose 1) is simply not enough.

This is exactly the reason why we choose to keep the dependence on diagonal entries when determining the next rotation but not for updating the two rows involved in the rotation. We want to have enough fill-in to at least create the potential of all the main diagonals becoming nonzero at one stage or another during the factorization phase by the simple expedient of rotating any such fill-in against a zero main diagonal entry.

On the other hand, such a strategy leads to dropping rules that are not adaptive, but rather based on a rigid threshold value determined by the initial matrix and user input. It would be interesting to see whether other choices for the dropping rules, fully adaptive but still independent of the value of main diagonal entries, produce better results.

A more systematic approach would be to try and minimize the number of rotations during the reduction process itself. Algorithms such as those in Duff (1974), Gillespie and Olesky (1995) and Robey and Sulsky (1994) can sometimes achieve a significant reduction in rotations by reordering rows during the factorization process. They are therefore worth considering, given the apparent sensitivity of the IGO methods to the order in which rotations are applied (column-wise reduction in cTIGO versus row-wise reduction for rTIGO).

5 Conclusions.

We have implemented and tested a series of incomplete orthogonal factorization methods for general nonsingular and unsymmetric matrices as well as rectangular overdetermined systems. These factorizations are obtained from the Givens orthogonalization process by dropping fill-in according to either position or

magnitude and by limiting the number of entries according to available storage. The process of reducing the initial matrix to upper triangular form using a succession of Givens rotations was implemented in both column-wise and row-wise fashion.

We have performed extensive comparisons for both square and rectangular (least-squares) systems for these methods. These test results show that, for the square system case, a fine-tuned ILU preconditioner will almost always outperform any of our IGO methods. This is due to the fact that LU factorizations are not only cheaper to compute in terms of floating-point operations but, more significantly perhaps, the sparse data structures required for an ILU factorization are less complex and expensive to use and update than Givens incomplete QR ones.

For the least-squares case, although we did not have any direct competitor as in the square case, the results we presented, coupled with our subsequent observations, show that IGO factorizations can be powerful preconditioning tools. For least-squares problems there does not appear to be software for competing general preconditioning methods currently available. If and when methods are proposed, we hope that they will be compared with the results presented here.

References

Z.-Z. Bai, I. S. Duff, and A. J. Wathen. A class of incomplete orthogonal factorization methods. I: Methods and theories. *BIT*, **41**(1), 53–70, 2001.

M. Benzi, J. C. Haws, and M. Tůma. Preconditioning highly indefinite and nonsymmetric matrices. *SIAM J. Scientific Computing*, **22**(4), 1333–1353, 2000.

M. Benzi, C. D. Meyer, and M. Tůma. A sparse approximate inverse preconditioner for the conjugate gradient method. *SIAM J. Scientific Computing*, **17**(5), 1135–1149, 1996.

M. Benzi, D.B. Szyld, and A. C. N. van Duin. Orderings for incomplete factorization preconditioning of nonsymmetric problems. *SIAM J. Scientific Computing*, **20**(5), 1652–1670, 1999.

T. A. Davis. University of Florida sparse matrix collection. Available from http://www.cise.ufl.edu/research/sparse/matrices, 2000.

I. S. Duff. Pivot selection and row ordering in Givens reduction on sparse matrices. *Computing*, **13**, 239–248, 1974.

I. S. Duff, A. M. Erisman, and J. K. Reid. *Direct Methods for Sparse Matrices.* Oxford University Press, Oxford, England, 1986.

I. S. Duff and J. Koster. On algorithms for permuting large entries to the diagonal of a sparse matrix. *SIAM J. Matrix Analysis and Applications*, **22**(4), 973–996, 2001.

B. Fischer, A. Ramage, D. J. Silvester, and A. J. Wathen. On parameter choice and iterative convergence for stabilised discretisations of advection-diffusion problems. *Comput. Methods in Appl. Mech. Eng.*, **179**(3), 185–202, 1999.

A. George and M. T. Heath. Solution of sparse linear least squares problems using Givens rotations. *Linear Algebra and its Applications*, **34**, 69–83, 1980.

M. I. Gillespie and D. D. Olesky. Ordering Givens rotations for sparse QR factorization. *SIAM J. Matrix Analysis and Applications*, **16**, 1024–1041, 1995.

G. H. Golub and C. F. Van Loan. *Matrix Computations.* The John Hopkins University Press, Baltimore, Maryland, Third Edition, 1996.

D. James. *Appendix: The breakdown of incomplete QR factorizations.* PhD thesis, Department of Mathematics and Computer Science, North Carolina State University, Raleigh, NC, 1990.

A. Jennings and M. A. Ajiz. Incomplete methods for solving $A'Ax = b$. *SIAM J. Scientific and Statistical Computing*, **5**(4), 978–987, 1984.

T. A. Manteuffel. An incomplete factorization technique for positive-definite linear systems. *Mathematics of Computation*, **34**, 473–498, 1980.

MatrixMarket. Matrix Market, NIST, Gaithersburg, MD. Available from http://math.nist.gov/MatrixMarket/data/, 2000.

J. A. Meijerink and H. A. van der Vorst. An iterative solution method for linear systems of which the coefficient matrix is a symmetric M-matrix. *Mathematics of Computation*, **31**(137), 148–162, 1977.

C. C. Paige and M. A. Saunders. LSQR: an algorithm for sparse linear equations and sparse least squares. *ACM Trans. Math. Softw.*, **8**, 43–71, 1982.

A. T. Papadopoulos, I. S. Duff, and A. J. Wathen. Incomplete orthogonal factorization methods using Givens rotations II: Implementation and results. Technical Report NA-02-07, Oxford University Computing Laboratory, Parks Road, Oxford, 2002.

T. H. Robey and D. L. Sulsky. Row ordering for a sparse QR decomposition. *SIAM J. Matrix Analysis and Applications*, **15**, 1208–1225, 1994.

Y. Saad. Preconditioning techniques for nonsymmetric and indefinite linear systems. *J. Comput. Appl. Math.*, **24**, 89–105, 1988.

Y. Saad. SPARSKIT: a basic tool kit for sparse matrix computations. Version 2. Technical report, Computer Science Department, University of Minnesota, 1994.

Y. Saad. *Iterative methods for sparse linear systems.* PWS Publishing, New York, NY, 1996.

X. Wang. *Incomplete factorization preconditioning for least squares problems.* PhD thesis, Department of Mathematics, University of Illinois, Urbana, Illinois, USA, 1993.

X. Wang, K. A. Gallivan, and R. Bramley. CIMGS: an incomplete orthogonal factorization preconditioner. *SIAM J. Scientific Computing*, **18**(2), 516–536, 1997.

Z. Zlatev. *Computational Methods for General Sparse Matrices.* Kluwer Academic Publishers, 1991.

Z. Zlatev and H. B. Nielsen. Solving large and sparse linear least-squares problems by conjugate gradient algorithms. *Computers Math. Applic.*, **15**, 185–202, 1988.