Report no. 02/07

Incomplete Orthogonal Factorization Methods Using Givens Rotations II: Implementation and Results.

Andreas T. Papadopoulos[a] Iain S. Duff[b] Andrew J. Wathen[c]

[a]Oxford University Computing Laboratory, andp@comlab.ox.ac.uk,
[b]Rutherford Appleton Lab., I.S.Duff@rl.ac.uk,
[c]Oxford University Computing Laboratory, wathen@comlab.ox.ac.uk

We present, implement and test a series of incomplete orthogonal factorization methods based on Givens rotations for large sparse unsymmetric matrices. These methods include: column-Incomplete Givens Orthogonalization (cIGO-method), which drops entries by position only; column-Threshold Incomplete Givens Orthogonalization (cTIGO-method) which drops entries dynamically by both their magnitudes and positions and where the reduction via Givens rotations is done in a column-wise fashion; and, row-Threshold Incomplete Givens Orthogonalization (rTIGO-method) which again drops entries dynamically, but only magnitude is now taken into account and reduction is performed in a row-wise fashion. We give comprehensive accounts of how one would code these algorithms using a high level language to ensure efficiency of computation and memory use. The methods are then 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.

Key words and phrases: preconditioning, linear systems, sparse least-squares, incomplete orthogonal factorizations, Givens rotations

Oxford University Computing Laboratory
Numerical Analysis Group
Wolfson Building
Parks Road
Oxford, England OX1 3QD

March, 2002

1 Introduction

This paper is intended as a follow-up to the theoretical work on incomplete Givens orthogonalization methods by Bai, Duff and Wathen [BDW01]. They described and analysed a family of incomplete QR type factorizations using Givens rotations. Whereas their paper was concerned only with theoretical aspects of these methods, we have undertaken the task of efficiently implementing them, as well as providing a variety of numerical results for these methods.

We start by giving a brief background on incomplete factorization techniques for solving linear systems of equations where the coefficient matrix is sparse.

Incomplete LU factorizations: Assuming that we are given a square, sparse, nonsingular matrix $A \in R^{n \times n}$, we can decompose it into a product of a unit lower triangular matrix L, and a nonsingular upper triangular matrix U, such that $A = LU$. This complete factorization can then be used to solve a system of the form $Ax = b$, $x, b \in R^n$ directly. However, these factors can be quite dense matrices. Complete LU factorizations are all too often costly to compute from the point of view of time as well as memory (for storing the dense factors), and hence not practical in applications.

On the other hand, incomplete variants of an LU factorization can be obtained at only a small fraction of the computational cost required for the complete factorization. We can then use this incomplete factorization to precondition an iterative method, such as Krylov subspace methods, to speed up convergence of such an iteration.

The idea of using incomplete factorizations as preconditioners for the iterative solution of linear systems is not a recent one.

The so-called incomplete variants of the LU factorization can be generated via dropping strategies. That is, during the factorization process, we set certain elements in the factors to be zero. The resulting triangular factors are then both sparse and can be computed in a fraction of the time necessary for a complete factorization; most of the computational work is simply bypassed through the use of dropping rules. The resulting incomplete factorization is of course useless for direct solution of the system. What one can do instead is use them to precondition any standard general iterative method. If the dropping rule is not too strict, one can usually achieve good convergence with such preconditioners (see [MAN80], [SAAD88], [SAAD96]). More recently, research has focused on ordering or preprocessing strategies for the system matrix, as a step in improving the performance of ILU preconditioners ([BHT00], [BSVD99]).

Incomplete QR factorizations: Incomplete LU factorizations have received much attention during the last twenty years or so, as LU decomposition is the natural and less complex way one would use to solve a square linear system. However, more recent research has introduced important innovations in the field of incomplete factorization techniques, such as approximate inverse techniques (see [SAAD96], ch. 10.5) and incomplete QR type factorizations for preconditioning iterative methods. It is this latter class of incomplete QR methods that will receive our attention in this paper.

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 aforementioned linear system directly. However, complete QR

factorizations are even more expensive than their LU counterpart. On the other hand, one could use incomplete variants of a QR factorization to precondition iterations, in a manner similar to incomplete LU preconditioning.
The question that arises naturally is how should one proceed to obtain this incomplete QR type factorization. The traditional approach would be the Incomplete Modified Gram-Schmidt technique (see [JA84] or [SAAD96], 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}. This approach has received considerable attention, but this has also brought forward its shortcomings. One of the hardest ones to overcome is the fact that Q_{inc} is in general not orthogonal and hence difficult to invert; Q_{inc} could even be singular if not enough fill is allowed, making the incomplete factorization worthless 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 massive amounts of fill-in (and consequently computational time) are allowed. Despite this, there have been successful attempts to improve the robustness of this method (see [WGB97]).
Incomplete QR via Givens rotations: Wishing to overcome these difficulties, people have investigated other orthogonalization techniques to use in an incomplete QR process. The use of Givens rotations for orthogonalization purposes is advocated in the aforementioned paper by Bai, Duff and Wathen which has been the starting point of the work presented here.
However, this is not the first attempt to use incomplete QR factorization via Givens rotations for preconditioning purposes.
James, in his PhD thesis [JAM90], 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 set 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. The implementation of James closely resembles the preconditioning technique of George and Heath [GH80], developed for the least squares solution of rectangular systems.
Zlatev and Nielsen, in [ZN88], have also developed an incomplete orthogonalization type preconditioner for conjugate gradients.
Zlatev [ZLA91], discusses in detail incomplete QR via Givens rotations. He appears to be the first one to produce incomplete QR factorizations via Givens rotations that also use some threshold dropping rule. He uses the 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.
This small number of papers is all the research relevant to incomplete Givens orthogonalization techniques we were able to locate. Note that most of the authors cited above used their techniques to precondition least-squares problems via the normal equations, and seem to ignore the case of square systems. Apparently, this is due to the availability of effective ILU preconditioning techniques for the square case, whereas this is not true for rectangular systems.
In this paper, we have chosen to present results for both square and rectangular (least-squares) systems, so as to present an exhaustive report on this class of incomplete QR factorizations.

The rest of this paper is organized as follows:
In section 2, we describe the three algorithms (cIGO, cTIGO and rTIGO) we have coded and

tested. The first two, with slight modifications, were presented in [BDW01], whereas the third one uses the approach in [JAM90] and [GH80], except that the dropping strategy is now different. As one can tell from their prefix, in cIGO and cTIGO we perform the reduction of the system matrix A to upper triangular form in a column-wise fashion, as opposed to rTIGO where this is done in a row-wise fashion. Some preliminary observations about each of these algorithms are also included.
Section 3 discusses the implementation issues for each algorithm. In order to be as efficient as possible, and assuming we are working with a high level language, one has to use a variety of sparse matrix and vector structures and associated operations on these; a description of such structures can be found in standard literature such as [DER86] and [SAAD96]. Finally, the issue of using these factors as preconditioners, once they have been computed by any of the three processes, is addressed.
In section 4 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, as well as the problem of encountering ill-conditioned systems. Some brief relevant remarks can be found at the end of this section.
In section 5 we switch our attention to preconditioning least-squares systems, them being the 'natural' choices for one to apply a QR factorization on. After presenting test results, we embark on a comprehensive discussion of all the issues regarding Givens incomplete QR as preconditioner we encountered during our research. 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

We start by introducing the notation used to define sparsity pattern sets and, consequently, dropping rules.
A dropping rule (or dropping strategy, or nonzero strategy) tells us which of the elements of a matrix we allow to be filled-in with nonzero values during, for example, an incomplete factorization process. The easiest way to deal with a dropping rule is to define it as a set of pairs of integers. These refer to the indices of the elements of a particular matrix that are allowed to be filled-in with nonzeros.
For example, the set of integer pairs

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

denotes the set of indices of the elements of A that are both nonzero and above or on its main diagonal.
A dropping rule can then be, for example 'keep elements only in $P_{A,U}$ and $P_{A,L}$ (i.e. in P_A) and drop the rest'; in fact we will impose this exact dropping rule when we implement the first of our three algorithms.

We can now present the three different algorithms for computing an incomplete QR factorization using Givens rotations we implemented, and discuss briefly some of their properties.
The first two of these algorithms (cIGO and cTIGO) are variants of the ones presented in [BDW01]. We have somewhat diverged from the actual algorithms presented in this first paper as 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 in [BDW01], we coded it and include here a short presentation, as it is another way to achieve an incomplete QR factorization using Givens rotations.
A glance at the algorithms should convince us that no matter how one tries to organise the reduction of A to upper triangular form, two rows of the matrix will have to be updated for each element we rotate out. This is a fundamental disadvantage of Givens QR when compared to LU decomposition and its incomplete variants. In the incomplete LU decomposition, we can set the process up so that only a single row changes for each element we zero out.

2.1 column-Incomplete Givens Orthogonalization (cIGO) method

The corresponding algorithm can be described in pseudocode as follows:

The cIGO-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 element in column j of A*
3. If $k_r(j) = j$ Then Cycle *! no elements to be annihilated below diagonal on this column*
4. For $i = k_r(j)$ DownTo $j+1$ and (i,j) $\in P$ Do: *! all nonzero subdiagonals in column j are successively annihilated*
5. Compute $\rho := \sqrt{a_{jj}^2 + a_{ij}^2}$ *! compute rotation angles*
6. Compute $c := a_{jj}/\rho$
7. Compute $s := a_{ij}/\rho$
8. Set $a_{jj} := \rho$
9. Store rotation data j, i, c, s
10. For $k = j+1, \ldots, n$ Do: *! update A without allowing any additional fill-in*
11. If $(i,k) \in P$ And $(j,k) \in P$, Compute $temp := -sa_{jk} + ca_{ik}$

12. ElseIf $(i,k) \in P$ And $(j,k) \in P^c$, Compute $temp := ca_{ik}$
13. If $(j,k) \in P$ And $(i,k) \in P$, Compute $a_{jk} := ca_{jk} + sa_{ik}$
14. ElseIf $(j,k) \in P$ And $(i,k) \in P^c$, Compute $a_{jk} := ca_{jk}$
15. If $(i,k) \in P$, Set $a_{ik} := temp$
16. EndDo
17. EndDo *! end of secondary loop*
18. For $k = j, \ldots, n$ And $(j,k) \in P$ Do: *! fill in R but respect sparsity pattern*
19. Set $r_{jk} := a_{jk}$
20. EndDo
21. EndDo
22. Set $r_{nn} := a_{nn}$

We should make the following notes:

- The incomplete orthogonal factor Q_{inc} is never explicitly computed nor dropping of elements is performed on Q_{inc} explicitly. It is however an incomplete factor in the sense that not all the rotations that would be required for a complete Givens QR factorization have been performed during the cIGO process; we only rotate out elements that belong to the set $P \cup A_L$.
  In the first paper, the incomplete orthogonal factor was actually computed and updated explicitly after each rotation and elements could then be dropped. However, we quickly realised that it is both easier, faster and cheaper from the point of view of computational work and memory requirements, 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 [BDW01]. When preconditioning with the cIGO factorization, we actually require the computation of matrix vector products of the form $Q_{inc}^T v$; we present an algorithm for such an operation in section 3.5.

- The natural and convenient choice for the sparsity pattern set P is the sparsity pattern of the original matrix A, i.e. $P = P_A = \{(i,j) | a_{ij} \neq 0,\ 1 \leq i,j \leq n\}$. Hence $nz(R_{inc}) = nz(A_U)$, and Q_{inc} is a product of $nz(A_U) - n$ rotation matrices.

- The j-th row of R_{inc} is generated at step j of the procedure. Denote by $\tilde{a}_{\cdot,\cdot}$ an element of the partially reduced (after a series of rotations) form of the matrix A. Then, at the beginning of such a loop on j, we can 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 element $\tilde{a}_{ij} \neq 0$ for some i, $n \geq i > j$, 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 and the incomplete factorization is ill-defined. cIGO type factorizations of nonsingular matrices can give rise to such singular incomplete upper triangular factors and we present two such examples in the appendix. 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 a very basic non parametric one yielding cheap preconditioners. However, in practice, having such a basic preconditioner is seldom effective. We need to produce incomplete factorizations that are closer to the full QR type factorization and we can achieve this by allowing extra amounts of fill-in via 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 element in column j of A*
3. If $k_r(j) = j$ Then Cycle *! no elements to be annihilated below diagonal on this column*
4. For $i = k_r(j)$ DownTo $j+1$ Do: *! all nonzero subdiagonals in column j are successively annihilated*
5. If $(i,j) \in P$ Or $|a_{ij}| > \tau$ Then: *! rotate out if element 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 and impose dropping rule*

12. Compute $temp_i := -sa_{jk} + ca_{ik}$
13. Compute $temp_j := ca_{jk} + sa_{ik}$
14. If $(j,k) \in P$ Or $|a_{jk}| > \tau$, Set $a_{jk} := temp_j$
15. Else, Set $a_{jk} := 0$ *! element too small so drop it*
16. If $(i,k) \in P$ Or $|a_{ik}| > \tau$, Set $a_{ik} := temp_i$
17. Else, Set $a_{ik} := 0$ *! element 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 as for the cIGO algorithm apply, but we should also note the following:

- In the case of cTIGO we are still working with a given sparsity pattern set P $(= P_A)$, but, by the use of a thresholding strategy we allow additional fill-ins on top of this in order to receive preconditioners of potentially better quality. Hence, for the incomplete factors we 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 element 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 inputed value for the drop tolerance:
  - When determining on which rows to rotate, and $\tilde{a}_{ij}$ is a fill-in 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, this is exactly the thresholding strategy used in our code, but other combinations may also provide interesting results.

- The memory control tolerance p may 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 this particular order in rotating out elements, accomodating and handling arbitrary fill-ins in the lower triangular part of A is quite difficult from the programming point of view. Hence the use of 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. We shall refer to this matter extensively in section 3.3.

2.3 Row-Wise Elimination TIGO (rTIGO)

This seems to be the traditional approach with respect to the order in which we rotate out elements in order to reduce A to upper triangular form (see [JAM90], GH[80]). For completeness, we present here the threshold incomplete variant of this algorithm we have actually implemented.

The rTIGO-Method Algorithm

1. For $j = 2, \ldots, n$ Do: *! for each row of A*
2. For $i = 1, \ldots, j-1$ And $a_{ji} \neq 0$ *! reduce row j to upper triangular form by rotating against row i*
3. If $|a_{ji}| < \tau$ Then
4. Set $a_{ji} := 0$ *! element too small to be rotated out so drop it*
5. Cycle to Next i
6. EndIf
7. Compute $\rho := \sqrt{a_{ii}^2 + a_{ji}^2}$ *! compute rotation angles*
8. Compute $c := a_{ii}/\rho$
9. Compute $s := a_{ji}/\rho$
10. Set $a_{ii} := \rho$
11. Store rotation data i, j, c, s
12. For $k = i, \ldots, n$ Do: *! compute updated i and j row segments and impose dropping rule*
13. Compute $temp_i := sa_{jk} + ca_{ik}$
14. Compute $temp_j := ca_{jk} - sa_{ik}$

15. If $|a_{ik}| > \tau$, Set $a_{ik} := temp_i$

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

17. If $|a_{jk}| > \tau$, Set $a_{jk} := temp_j$

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

19. EndDo

20. Keep only the p largest entries in the i-th row a_{i*}

21. Keep all nonzero entries in the j-th row a_{j*}

22. EndDo *! end of secondary loop*

23. Keep only the p largest entries in the j-th row a_{j*}

24. EndDo *! j-th row now reduced to upper triangular form*

25. For $i = 1, \ldots, n$

26. For $j = i, \ldots, n$

27. Set $r_{ij} := a_{ij}$ *! R is extracted from reduced form of A*

28. EndDo

29. End Do

The following observations need to be made:

- The rows i, $i = 1, \ldots, j-1$, used to rotate out elements in row j when attempting to reduce 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 zero-in the element in position (j, i) but will not create any fill-in in positions $(j, k), k = 1, \ldots, i-1$. Hence, we achieve our objective of gradual reduction to upper triangular form.

- We now drop elements by magnitude only, without caring whether they belong to a sparsity pattern set 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 we have chosen the following combination:

  - To perform a rotation, the element $\tilde{a}_{ji}$ we want to rotate out must satisfy the criterion $|\tilde{a}_{ji}| > \tau |\tilde{a}_{ii}|$, otherwise we drop it (skip rotation).

– For row j which we are currently reducing, we drop elements $\tilde{a}_{jk}$ 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 elements $\tilde{a}_{ik}$ 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 ensure that R_{inc} is nonsingular. This is also true for the cIGO and cTIGO algorithms.

3 Computer Implementation Details

In this chapter we provide a comprehensive description of how the algorithms presented in the previous chapter can be coded in a high level language (we have used fortran77) using a variety of standard sparse matrix and vector structures. The interest in sparsity arises because its exploitation can lead to enormous computational savings and because most of the large matrix problems that occur in practice are sparse.

3.1 Implementing cIGO

Assume that we are given the matrix A whose cIGO factorization we want to compute; A is available in the ordered compressed sparse row (CSR) format (see [SAAD96], [DER86]). This is not too strong a demand, as matrices in (online) databases are usually stored in some sort of ordered structure, out of which we can derive the ordered CSR format we require here.
Implementing cIGO is then a relatively easy task. The first step is to construct the associated ordered compressed sparse column (CSC) structure of the matrix but with the real CSC array substituted by an integer pointer-to-values-in-CSR array. This is a necessary preparatory step as rotations are performed in a column-wise fashion, so we require some information on the distribution of nonzeros over each column. In fact, we only need this column information for the strictly lower triangular part only, but it is easier to construct it for the whole matrix. If one did not have this CSC structure, one would have to perform an expensive search through the CSR structure in order to pin-point the next rotation in a particular column.
The two structures described above are all we need to perform the cIGO factorization illustrated in section 2.1. For a particular rotation, and since we allow no further fill-in, updating the two rows boils down to just computing new values for the nonzeros in the corresponding positions in the CSR structure.
After the matrix has been reduced, we need to extract R_{inc} out of the reduced form of A and store it in a separate CSR type structure. This is necessary, as the CSR structure for A will contain, at the end of the cIGO process, a number of zeros corresponding to the nonzeros in the lower triangular part of the original matrix. On the other hand, the incomplete factor Q_{inc} will just come out as a list of rotations.
A final thing worth mentioning is the fact that the CSR structure of A needs to contain all the diagonal elements, even if their initial value is zero.

3.2 Implementing cTIGO

Efficient implementation of cTIGO is a much more complex issue. We now have arbitrary fill-in appearing in the lower triangular part (where elements to be rotated out are situated) of the matrix under reduction, and still rotate out elements in a column-wise order. On the other hand, each such rotation requires updating two rows. This combination of column and row oriented actions, on a structure in constant pattern change because of fill-in allowance, calls for special care in coding.
We start by observing that, because of the nature of the algorithm, it is easier to have two separate and independent structures corresponding to the lower and upper triangular parts of the matrix under reduction.
The main structure for keeping each part of the matrix and the values is of a CSR type but with predetermined length for each row.
For the upper triangular part, this large structure will eventually contain the factor R_{inc}.
In this large structure, the space allocated for row j, initially contains the non-zeros of A_U in this row plus a number of zeros which act as space for potential fill-ins during the factorization phase, as in the example below:

$$[\ \ldots\ |\ \text{elms. in } P_{A,U}\ |0\ldots\ \text{space for fill-ins}\ \ldots 0|\ \ldots]$$

So, assuming that we initially have $nz(P_{j,A,U})$ nonzeros in the upper triangular part of the j-th row of A and want a maximum of p nonzeros in the j-th row of R_{inc}, we will allow for at most $p - nz(P_{j,A,U})$ fill-ins due to thresholding on this row.
We require two such arrays; an integer one in which we store the column index, and a real one for an element's value.
Two integer arrays of size n are also employed; one which holds pointers to the position at which each row begins, and a second one which holds the current total number of nonzeros in each row. For a row, the cut-off point between space for elements in $P_{A,U}$ and space for fill-ins is always known, as the elements in $P_{A,U}$ are never thrown away.
Wasteful in terms of memory as it may seem, one has to employ such a large structure so that individual rows can be accessed and modified at any time during the factorization phase at a minimum of effort. The fact that two rows are updated at the same time in this factorization calls for this *a priori* determination of row length. This is of course a restricition on the incomplete factor(s) that does not appear for example when doing ILU type factorizations. But, of course, ILU methods can be organised in such a way that only one row is updated at any given time; this is not true for our class of methods.
Since the space allocated for fill-ins in each row is predetermined, one has to make good use of the memory control parameter p to make sure that a certain balance is maintained. In particular, one easy way to pre-allocate space is the rule $space\ for\ row\ j = min\{n - j + 1, max\{nz(P_{j,A,U}), p\}\}$, for the portion of the row j in the upper triangular part.
A similar structure will be employed for the lower triangular part of the matrix, where the potential elements to be rotated out reside.
Constructing the initial form of these structures is a fairly easy and cheap task, assuming the ordered CSR form of A is available.
We now turn our attention to the task of efficiently determining elements to be rotated out in

a particular column.
A CSC structure of size $nz(A_L)$ with pointers to element values' location in the large structure solves the problem of determining rotations for elements in $P_{A,L}$.
However, we still have for each column a number of rotations that should be performed because elements occurred as thresholding fill-ins in the lower triangular part.
The easy way to deal with these is, for each column, to search through all fill-in elements in the lower triangular part. This approach is of course extremely inefficient; for a $O(10^4)$ sized matrix the time taken for these searches was $60 - 75\%$ (depending on fill-in allowed by the choice of the threshold parameter) of the total time it took to produce the factorization.
The way around this problem is to employ an ordered column linked list structure (see [DER86]) to keep track of fill-ins due to thresholding in the lower triangular part. Linked lists are commonly used in incomplete QR type facorizations. Saad [SAAD96] advocates the necessity of a linked list in the implementation of Incomplete Modified Gram-Schmidt.
When a thresholding fill-in appears in the lower triangular part, we simply add a new entry to the list after the final position and adjust the link for the previous in order of increasing row index element on the same column. Instead of the element's value, we store a pointer to the large CSR structure where the value is held. There is an inefficiency here in the sense that we actually have to search through all entries on the same column with this new addition to find the position of the previous (in ascending row order) element. Of course, this search is only necessary if one wants to preserve the strictly decreasing order of rotations in a particular column, as is done in the original cTIGO algorithm. If one relaxes this requirement and allows rotations in a column to be performed in a random way, the linked list no longer needs to be ordered.
Now assume a thresholding fill-in that appeared during a previous rotation has to be dropped in the current rotation. All we have to do is set the entry for the list's pointer-to-large-CSR-structure for this element to 0, but we do not alter its link. An inefficiency related to this approach appears if for a column the thresholding fill-ins alter frequently. Then, we may end up going through a large number of such 'dead' links at anytime we go through the list. The alternative would be to switch the link in the previous in order entry to point directly to the next and bypass the element altogether, but this operation requires a serarch of its own in determining this previous element. There does not seem to be a clear way to tackle this problem.
Closing this discussion on determining rotations, we note that we have to add an extra integer array to the large CSR structure for the lower triangular part, pointing to the position of each element in the linked list. This array is necessary to update the linked list in the fashion illustrated above.
Having set up the necessary structures, we can now procceed with cTIGO *per se*.
At step j of the process we extract the respective row from the large structure for the upper triangular part and store it as a sparse vector. We then determine the order in which rotations will be performed to zero in the j-th column below the main diagonal by combining the information from the CSC structure for $P_{A,L}$ and the column linked list. For each such rotation, we extract the corresponding i-th row out of the large structures. Note that parts of the i-th row will have to be extracted from both these structures. Updating the two rows by the rotation

angles and imposing the threshold dropping rule is a straightforward operation. Row updating is performed using a modified version of the standard algorithm for adding two sparse vectors (see [DER86]). At the end of this, if any of the rows has more nonzero elements than the space available for them in the large structures, the memory control dropping rule is imposed. This is simply a quick sorting operation on a sparse vector for which a good routine can be found in Y. Saad's Sparskit2 package [SPK99]. The rotation is completed by reinserting the updated segments of the i-th row into the large structures and the updating of the linked list for the part of the i-th row that is in the lower triangular section of the matrix.
After all rotations have been performed on column j, we simply insert the j-th row in the large structure for the upper triangular part.
These operations will require a number of sparse vector structures, but their size is negligible compared to the size of the two large CSR type structures and the linked list.
The structures described above are local. They are only required during the factorization phase. With the cTIGO process complete, R_{inc} can be extracted from the large upper triangular CSR, as a CSR structure with no zeros in it, while Q_{inc} comes out as a list of rotations.

3.3 Implementing rTIGO

In this algorithm all operations are done in a row wise fashion so this relieves us of the burden of having a linked list. On the other hand, since reduction of a row via rotations must be performed in a particular order (from right to left), all the structures have to be and remain ordered at all times to avoid searches for the next element to be rotated out. Note that this was not a necessary requirement for cTIGO. In cTIGO the fill-ins due to thresholding were in general unordered in the large CSR structure, and, as we have explained, the linked list could also be unordered. Note also that for rTIGO we only perform pure thresholding as a dropping rule and no longer care if an element is in P_A.
rTIGO does not require a preparation phase. We do once again set up a large CSR structure in which we will keep elements in the upper triangular part of the matrix under reduction, but apart for determining the length for each row, little else must be done. This structure will be gradually filled up wth nonzeros as elimination progresses.
rTIGO proceeds as follows. At step j we pull out the j-th row of A out of the ordered CSR structure where we store the original matrix and place it in a sparse ordered vector. This is now the row to be reduced and it has not been accessed before. We use the previous rows 1 to $j-1$ to reduce it to upper triangular form. These rows have already been reduced to the desired form and they are stored in the large CSR structure. For each of these rows, their nonzeros are ordered by increasing column index. Hence, if element $(j,i),\ i=1,\ldots,j-1$, has to be rotated out, we extract the corresponding i-th row out of the large CSR structure and place it in an ordered sparse vector. We perform the rotation and subsequent updating of the two rows by the rotation angles. It is feasible to preserve ordering when computing the updated rows without extra work and such an algorithm is given in [DER86]. Thresholding is then performed; this operation also preserves ordering.
We now have to re-insert the updated i-th row back into the large CSR structure. First, if too much fill-in has remained we need to apply the memory control dropping rule. This is now done in two stages. Assuming we have space for at most p_i entries in the i-th row of the

large structure, we first permute the sparse vector structure so that the p_i largest in magnitude entries are in the first p_i positions of this sparse vector. This operation does not preserve order, so we now have to permute these p_i entries so that the vector is again ordered by increasing column index in the part that interests us. This is achieved by applying a *quick_sort* type routine on the integer array of the sparse vector. The updated i-th row is now once again ordered and of such a length that will not interfere with other rows in the large CSR structure, so we can just copy it onto the right position and move on to the next rotation.
After row j has been reduced to upper triangular form, we insert it into the large CSR structure. The memory control dropping rule may have to be enforced first (note that this is the first time we impose this rule on row j; it is not imposed during reduction) but otherwise we are done.
At the end of this process, R_{inc} is extracted from the large structure for which we have no further use.

3.4 Preconditioning with $M = Q_{inc}R_{inc}$ and $M' = R_{inc}^T R_{inc}$

Once the incomplete factors Q_{inc} and R_{inc} have been computed using one of the methods illustrated above we can use them to precondition any standard iterative method.
For square systems, general iterative methods such as GMRES or Bi-CGSTAB can be preconditioned with $M = Q_{inc}R_{inc}$. On the other hand, and this is an advantage of our methods, one could potentially precondition conjugate gradients for the system of normal equations (CGNR) with $M' = R_{inc}^T R_{inc}$.
For square systems, we only give numerical results for the first case. We shall next describe how preconditioning a general iterative method works for this particular form of the preconditioner M. The case of preconditioned CGNR is not as interesting for square systems, although it will be described and applied in section 5 where we consider least-squares systems.
We have implemented the right preconditioned versions of GMRES and Bi-CGSTAB. Description of these algortihms can be found in [SAAD96] and [TEMPL] respectively.
The right-preconditioned versions of these iterations are based on solving

$$AM^{-1}u = b, \;\; u = Mx.$$

The advantage of right preconditioning is that the residual norm at the m-th iteration, i.e. $b - Ax^m$, which one typically uses as a convergence criterion for the iteration, is relative to the initial system.
Going through the algorithms, one sees that the instances where the preconditioner M appears, are computations of the type $w = AM^{-1}v$, where w, v are (dense) vectors. We split this computation into two parts: compute $z = M^{-1}v$ and then compute $w = Az$. The second part is a simple matrix vector product. The first part is equivalent to solving the square system $Mz = v$, given that the preconditioner is nonsingular.
It now remains to see how we can efficiently compute the solution z to $Mz = Q_{inc}R_{inc}z = v$. We first note that Q_{inc}, being a product of rotation matrices, is orthogonal. Hence, our problem is equivalent to solving $R_{inc}z = Q_{inc}^T v$. Assuming that we can compute the modified right hand side $Q_{inc}^T v$, this is now just a system involving an upper triangular matrix. Efficient routines

for solving it, assuming R_{inc} is stored in CSR, are given in [SAAD96].
We shall now see how to compute $r = Q_{inc}^T v$ without explicitly creating Q_{inc}^T.
The IGO algorithms presented so far all store the incomplete orthogonal factor Q_{inc} as a list of rotations. That is, assuming the k-th rotation was performed to zero out the element in position (i, j) of the matrix under reduction, we store the rotation data in two integer and two real arrays as $indx_r(k) = i$, $indx_c(k) = j$, $cos(k) = c$, $sin(k) = s$, where c and s are the respective rotation angles. Note that for each rotation we stored two real variables. In fact, only one real variable (the rotation angle itself rather than the cosine or sine) is necessary, but we have chosen to retain both for clarity.
Assuming that in total rts rotations have been performed during the incomplete factorization process, we have

$$Q_{inc} = G_{rts} \dots G_2 G_1,$$

where G_k is a Givens rotation matrix of the form

$$G_k = G(i, j, \theta) = \begin{pmatrix} \cos\theta & \sin\theta \\ \text{-}\sin\theta & \cos\theta \end{pmatrix}.$$

Hence,

$$Q_{inc}^T = G_1^T G_2^T \dots G_{rts}^T,$$

and so Q_{inc}^T can be computed by going through our list in reverse order and paying attention to the fact that each of the Givens matrices is now transposed.
We are, however, only interested in the effect that Q_{inc}^T has when applied to a vector v. The procedure that gives us $r = Q_{inc}^T v$ is then:

1. Set $r := v$
2. For $k = rts$ DownTo 1 Do
3. Compute $temp := cos(k) \cdot r(indx_r(k)) + sin(k) \cdot r(indx_c(k))$
4. Set $r(indx_c(k)) := -sin(k) \cdot r(indx_r(k)) + cos(k) \cdot r(indx_c(k))$
5. Set $r(indx_r(k)) := temp$
6. EndDo

Thus, 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). The subsequent triangular solve requires $nz(R_{inc}) - n$ multipications and additions as well as n divisions. This is the total amount of extra work required for preconditioning per iteration.

4 Preconditioning Square Systems

4.1 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 on the public domain at either the MatrixMarket [MM00] database or the Davis collection of sparse matrices at the Univeristy of Florida [UFSMC]. In these experiments, we have used the IGO methods as preconditioners for either GMRES or Bi-CGSTAB. As an opposing standard, we are comparing the performance of these methods with a variety of ILU type preconditioners. These ILU factorizations were all obtained using Y. Saad's Sparskit2 [SPK99] package, 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 without considerations of sparsity pattern sets. ILUtp, which is the same as ILUt but we also allow column pivoting. Extensive description of these methods can be found in [SAAD96], [SPK99].
Knowing that ILU methods make for very powerful preconditioners when it comes to matrices possessing structure like bandedness or diagonal dominance, as matrices arising from the discretization of partial differential equations often do, we have concentrated on more general examples whose sparsity pattern or distribution of values will cause ILU methods difficulties.
For both iterative methods, the preconditioners is always applied to 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, since it is not generally available, was created in a random fashion using matlab. However, for some experiments where we wanted to test the accuracy of our iterative solution, we chose a known solution vector, namely $x(i) = i,\ i = 1, \ldots, n$ and created the right hand side appropriately. This approach was also used in [BHT00].
As for the preconditioners, we should mention that for all thresholding methods (cTIGO, rTIGO, ILUt, ILUtp) the space allocated was enough so that the memory control dropping rule is never invoked. The cases where we had such truncation because, otherwise, we would run out of memory when computing the preconditioner, are clearly marked.
We should finally mention that especially for ILUtp where column pivoting is allowed, the choice of the pivoting parameter p (not to be confused with the memory control parameter for cTIGO, rTIGO) was 0.01. Since the ILUtp rule for pivoting between columns i and j at step i is $|\tilde{a}_{ij}| \times p > |\tilde{a}_{ii}|$, the value we chose for p means that we do not pivot often.
Note also that, because of the way we coded Bi-CGSTAB, what is presented in the experiments as 1 iteration consists in fact of two matrix vector products, that is, the Krylov space is twice augmented per Bi-CGSTAB iteration. For GMRES, 1 iteration corresponds to augmenting our Krylov space by one.
For GMRES, we have chosen a restart value of 50 always. Hence, when we say that GMRES

has converged at iteration 123, we mean that it has restarted twice and only then, at the 23^{rd} iteration after the second restart, did we achieve convergence.

4.2 Examples in Brief

In this section we will be briefly listing results for a variety of matrices. These are actually the best results we could obtain for the methods out of the values (normally powers of 10) for the threshold parameters we tested for each system. The two tables in this section list the attributes and performance of cTIGO (or just cIGO when no fill was necessary), respectively rTIGO, against ILUtp (or ILU(0)), the latter 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 one showing attributes of the cTIGO (or rTIGO) preconditioner, and the second one showing the respective results for ILUtp (or ILU(0)), so as to facilitate comparison between the best observed performance of the IGO and the 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 on table 4.1 show that cTIGO is in fact a robust preconditioner, in the sense that if enough fill is allowed it can eventually give preconditioners of good quality. The time it takes to actually compute these preconditioners is usually much higher than for ILUtp, although the density of the two resulting incomplete factorizations is comparable.
Note that spread around the table are matrices for which the non-parametric cIGO preconditioner was enough to give fairly good preconditioners. For these same systems, ILU(0) or MILU(0) were not enough to start convergence, so we had to turn to 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 [STE01]), where for many tested cases ILU(0) preconditioning is of good quality but cIGO gives stagnant iterations.
Two matrices for which cIGO (and cTIGO with very little fill) performs much better than the tested 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 elements on the main diagonal of this block are covered with values during the incomplete factorization process and the resulting preconditioner is convergent.
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 also 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 adressed in detail in section 5.4.

Table 4.1: best results: cTIGO vs ILUTP

matrix	cTIGO τ	$nz(R_{inc})$	# rots	cTIGO time	GMRES	GMRES
	ILUtp τ	$nz(U)$	$nz(L)$	ILUtp time	its.	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	62449	0.5	5	0.5
thermal	cIGO	34992	31536	1.1	7	0.9
	10^{-2}	21100	22989	0.2	6	0.25

Table 4.2: 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

We have also identified some examples where we were unable to produce convergent preconditioners via 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 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 be convergent. To be fair, we did not test ILUtp for these matrices; just the fact that cTIGO 'fails' made them negative examples automatically.

The test results for rTIGO can be found in table 4.2.
Again, note that rTIGO is invariably slower than the best choice for ILUtp in computing the preconditioner. However, for some of the problems, rTIGO is considerably faster than cTIGO, despite the fact that the preconditioner it computes can be quite denser than the cTIGO one. This reinforces our belief that it is actually the need for keeping and constantly

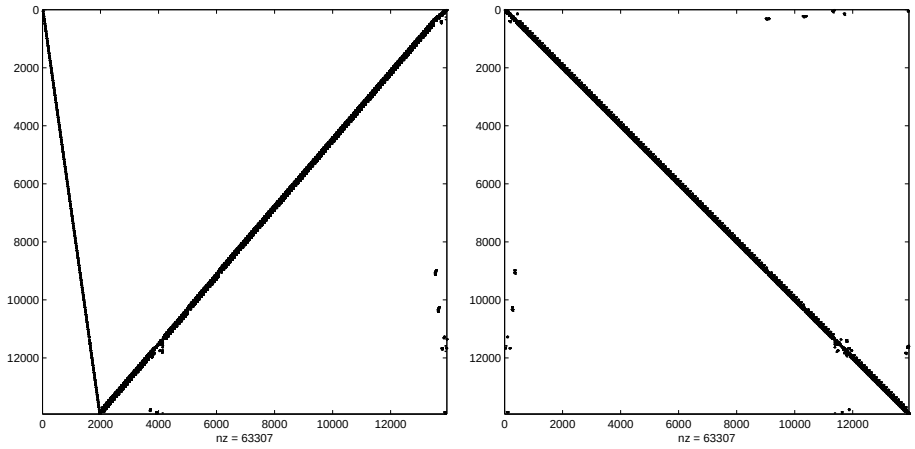

Fig. 4.1: matrix *bayer*02, original: $n = 13935$, $nz(A_U) = 36518$, $nz(A_L) = 27161$, condest$(A) = 2.3 \times 10^{18}$, and after applying mc64, job 5: $n = 13935$, $nz(A_U) = 33676$, $nz(A_L) = 30003$, condest$(A) = 3.6 \times 10^3$

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.

4.3 The Effect of PreProcessing

We now present an 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 [BHT00], [BSVD99]). Since they only involve computations with integer indices, reordering methods tend to be 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 Harwell's MC64. The example we present here clearly shows that reordering the matrix benefits both the IGO as well as the ILU methods.
We have chosen matrix *bayer*02 (Bayer AG, F. Grund, Mar. 1995) arising in chemical kinetics to illustrate this (Fig. 4.1). We first ran both cTIGO and ILUtp on the original system, and then reordered it using the HSL2000 MC64 (option 5) routine (see [HSL00]). cTIGO and ILUtp preconditioning was then applied to the reordered system. The results are shown in the tables below.
MC64 preprocessing of the matrix not only altered the sparsity pattern for the better, in the sense that diagonal elements were filled in and, in the reordered matrix, most elements are witihin a small band, but also greatly improved the condition number.

*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

*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

This improvement is reflected in the performance of the preconditioners. For the processed matrix, sparser and faster to compute incomplete factors are required to achieve convergence in a similar number of iterations as for the original matrix. This is true for both cTIGO and ILUtp. Hence, although cTIGO is benefited by 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 discernable effect if applied to the *garon* matrices.

4.4 Ill-Conditioned Systems

We close this section on numerical results for square systems, with a short reference to ill-conditioned systems.
We often use preconditioned iterative methods and achieve convergence within a finite number of iterations. This means that, assuming we have right preconditioning, we have computed an iterate $x^{(i)}$ for which the residual $||b - Ax^{(i)}||$ is small enough. However, how close is this computed solution to the exact one? It turns out that if the system is ill-conditioned, the computed iterative solution can be very inaccurate, in the sense that the quantity $||x - x^{(i)}||$, where x is the exact solution, is not as small.
In particular for incomplete factorization preconditioners, this is a consequence of the fact that the computed triangular factors (U for ILU methods, and, respectively, R_{inc} for IGO methods) are ill-conditioned themselves or have extremely small entries on the diagonals, making the triangular solves required for preconditioning at each iteration unstable and therefore inaccurate.
We illustrate this by a typical (although a bit extreme) example.
We will precondition the system $Ax = b$ with A being the matrix *nnc*1374 of size n=1374, with $condest(A) = 4.1 \times 10^{15}$, and b such that $x(i) = i,\ i = 1, \ldots, n$.
The results, for ILUtp and rTIGO preconditioning can be found below. The choice of GMRES(50) as the iterative method is of no importance. Qualitatively similar results are obtained with Bi-CGSTAB. rTIGO was used in this example since the incomplete factors are very dense and hence require considerable time to be computed. cTIGO has the same behaviour.

*nnc*1374**: rTIGO results**

τ	10^{-10}	10^{-12}	10^{-14}	10^{-16}	10^{-20}
$nz(R_{inc})$	59296	99586	106778	116083	135236
$\#rots$	27943	41195	45274	47448	48492
GMRES(50) its	stagn	1	1	1	1
$\|\|b - Ax^{(i)}\|\|_2$	n/a	5.4×10^{-5}	5.9×10^{-8}	5×10^{-9}	5.3×10^{-9}
$\|\|x - x^{(i)}\|\|_2$	n/a	155108.9	311.1	74.5	40.2

For the iterative method we used, convergence was achieved once the residual became smaller than 8.76×10^{-3}.
These results show that although the residual of the computed solution is extremely small (of order up to 10^{-9}), the computed solution has not got the accuracy one would expect by looking just at this residual.
Perhaps it is worth mentioning that inability to compute accurate results due to ill-conditioning is not limited to preconditioned iterations. Using matlab's backslash command to solve this system returns a solution vector x_{bslash} for which $||x - x_{bslash}||_2 = 17.4$.

*nnc*1374**: ILUtp results**

τ	10^{-10}	10^{-12}	10^{-14}	10^{-16}	10^{-20}
$nz(U)$	59419	58477	59086	59162	66537
$nz(L)$	19029	20803	21464	21516	24407
GMRES(50) its	stagn	1	1	1	1
$\|\|b - Ax^{(i)}\|\|_2$	n/a	1×10^{-6}	3×10^{-9}	2.8×10^{-9}	4.6×10^{-9}
$\|\|x - x^{(i)}\|\|_2$	n/a	2734.7	14.8	11.8	22.4

This unstable behaviour of the preconditioner is reflected in the matrix residual and error norms. In particular, for an IGO method, one can compute the quantities $||A - Q_{inc}R_{inc}||_F$ and $||I - (Q_{inc}R_{inc})^{-1}A||_F$. These two quantities gauge how fast convergence is achieved (i.e. how fast we obtain residual reduction in the iterative method) and, respectively, how accurate is the computed iterative solution (see [BHT00]).
In particular, for $Q_{inc}R_{inc}$ computed by rTIGO with $\tau = 10^{-12}$, we have $||A - Q_{inc}R_{inc}||_F = 6.6 \times 10^{-4}$, but $||I - (Q_{inc}R_{inc})^{-1}A||_F = 1086.4$. This difference of almost 8 orders of magnitude in the two quantities tells us that we expect the preconditioner to reduce the residual, but the iterative solution will be inaccurate. This observation is indeed reflected in the results presented above. Similar residual and error matrix norm estimates can be computed for the ILU factorization and they should lead to the same conclusions.
The matrix above is by no means an exception. [BHT00] lists a number of ill-conditioned matrices for which inaccurate solutions were obtained even when ordering and preprocessing techniques were applied to them before ILU preconditioning.

4.5 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 these methods' performance against a wide range of standard incomplete *LU*

factorization techniques.
In the case of the non-parametric preconditioners of either type (cIGO 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 sometimes consider using 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 via thresholding or through extended sparsity pattern sets.
With few exceptions, the parametric Givens IQR type methods (cTIGO, rTIGO) are clearly disadvantageous when their performance is compared to standard parametric ILU factorizations (ILUt, ILUtp). The time required to compute factorizations of similar preconditioning quality (sparsity of factors, convergence rate of iteration) via 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 clearly addressed. The main issue seems to be the fill in the lower triangular part of the matrix and, consequently, the number of rotations performed during the incomplete reduction process. One can limit these without affecting the preconditioner's quality by the use of 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.
In relation to this problem, one could also consider the use of different combinations of threshold dropping rules than the ones used for our experiments. A detailed discussion follows in section 5.4.
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 locate examples for which some type of incomplete LU factorization with good choice of parameters could not perform at least as well as any 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.

5 Preconditioning Least-Squares Systems

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

$$Ax = b,\ A \in R^{m,n}, b \in R^m, x \in R^n,\ \text{with } m \geq n.$$

The case of matrices with $m < n$ which arises in linear programming problems will not be considered here.

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 then use it to precondition conjugate gradients for the associated system of normal equations $A^TAx = A^Tb$. Since Q_{inc} is orthogonal, the natural preconditioner to use is $M = R_{inc}^T R_{inc}$, which is also positive definite.
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 indeed 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 thrown out. To simplify notation, from now on R_{inc} will refer to this truncated factor.
In the sections below we shall address particular issues concerning the construction, efficient implementation and performance of this preconditioner.

5.1 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 element 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 in the previous section remain the same. Some auxiliary routines also had to be modified to handle rectangular matrices.
Despite the fact that we are forming the normal equations, there is no need to compute A^TA explicitly, but rather apply two successive matrix-vector products where necessary within the conjugate gradient algorithm.
Similarly, for preconditioning this iteration, there is no need to form $R_{inc}^T R$ explicitly, an approach which would be too expensive both computationally and storage-wise. 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. This approach however requires that we transpose and store in a CSR form R_{inc}^T as well as R_{inc} before starting the preconditioned iteration. Fortunately, this is a one-off operation and not very expensive at that. On the other hand, there is no need to store the series of rotations that form Q_{inc}, as we had to in the square case.

5.2 Dropping Rules

The fact that there is no main diagonal element in the last rows of A and for the rows where there is such an element its value is usually zero, as well as our experiences with square matrices that initially have some zero main diagonal elements, call for some modifications in the threshold dropping rules.
The memory control dropping rules remain the same as in the square case.

Hence, 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 elements to rotate out at column j. This is the same dropping rule we employed in the square case.
Consequently, for this j as pivot row, the dropping rule remains the same as in the square case, namely

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

As for the row i, to which such an element to be rotated out belongs, the dropping rule becomes

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

Note that for cTIGO we have not removed the dependence of the threshold on the diagonal element.

For rTIGO, when determining the next element to be rotated out on row j, $1 < j \leq m$ that is reduced to upper triangular form, 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 elements $a(j,k)$ on this row j undergoing reduction if they satisfy

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

where the norm notation refers to the 2-norm of the j^{th} row of the original matrix A before any reduction via Givens rotations has begun.
Similarly, for the row i, $i = 1, \ldots, n$ whose main diagonal element will act as pivot to the rotation performed on row j, after each rotation we will retain elements $a(i,k)$ that satisfy

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

where again the right hand side refers to the 2-norm of the i^{th} row of the original matrix A.
Note that for rTIGO, we have removed as much as possible the dependence of the dropping rule on values of main diagonal elements that might well be zero. On doing so however, the dropping rule becomes more rigid as it depends on two fixed quantities (τ and a norm of a row of the original matrix). On the other hand, for cTIGO, we still base our threshold dropping rule on the value of the main diagonal element wherever possible. Thus, we still have the potential of a dynamic dropping rule, but will risk the likelihood of too much fill-in because of initial zeros on the main diagonal.

5.3 Numerical Results

In all the tests presented below we comply with 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.
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, as it was too large for our available resources. The fundamental attributes of our test set of matrices can be found in table 5.1.

Table 5.1: matrix attributes

matrix	size	$nz(A)$	initial zero diags.	vanilla its.	CGNR time	cIGO CGNR its.
*illc*_1033	1033×320	5047	315	> 3000	> 10.98	> 3000
*illc*_1850	1850×712	9463	705	2057	14.63	> 3000
*jimenez*_4	385×361	2013	352	> 10000	> 19.1	> 3000
*jimenez*_5	574×526	3886	521	> 10000	> 31.3	> 3000
*well*_1033	1033×320	5047	315	169	0.56	453
*well*_1850	1850×712	9463	705	447	3.27	960
*pigs_s*1	3140×1988	9525	1015	551	6.28	843
*pigs_s*2	6280×3976	27560	2030	1104	29.41	2168
*pigs_m*1	9397×6119	31131	6118	743	28.05	1464
*pigs_m*2	18794×12238	87275	12236	1555	129.12	3241
*pigs_l*1	28254×17264	92282	17264	1169	117.51	2368
*pigs_l*2	56508×34258	259582	34528	2346	438.70	3574
jordache	24708×23937	218568	23292	> 3000	n/a	> 1000

Note that the preconditioner resulting from applying the simple cIGO algorithm is qualitatively bad in all cases. In fact, it takes approximately twice the 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 tested value of τ for which the total solution time (incomplete factorization followed by one preconditioned iterative solution of the system) was minimized. For each matrix, we present the best such run for cTIGO versus rTIGO in table 5.2.
As was the case for square systems, fine-tuned rTIGO will be faster than its column-based counterpart cTIGO by more or less an order of magnitude of time, and compute a preconditioner that is qualitatively better than cTIGO, although quite denser in some cases.

Table 5.2: 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}	5200	2.03	317	1.65
	10^{-2}	2550	0.80	198	0.80
illc_1850	10^{-2}	10492	9.10	137	1.41
	10^{-2}	13581	6.84	75	0.87
jimenez_4	10^{-5}	25327	6.37	17	0.25
	10^{-6}	30241	0.80	11	0.19
jimenez_5	10^{-4}	61075	22.61	204	6.20
	10^{-5}	81316	13.45	117	4.63
well_1033	10^{-2}	4759	1.91	37	0.18
	10^{-1}	2551	0.10	65	0.25
well_1850	0.05	6205	3.04	75	0.66
	0.05	8181	0.54	52	0.50
pigs_s1	10^{-1}	8464	2.25	91	1.68
	10^{-1}	11851	0.50	59	0.83
pigs_s2	0.025	29004	15.12	457	21.8
	0.050	46006	2.39	77	3.14
pigs_m1	10^{-1}	33258	9.18	105	3.40
	10^{-1}	37311	1.30	65	2.90
pigs_m2	0.05	98117	40.3	> 1000	n/a
	10^{-1}	112365	3.31	132	14.50
pigs_l1	10^{-1}	99612	9.83	127	10.06
	10^{-1}	109915	1.90	78	5.93
pigs_l2	0.075	245148	62.50	> 1000	n/a
	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

In all cases, the preconditioners significantly reduce the number of iterations when compared to simple CGNR iterations, 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 preconditioning solves.
The preconditioners in all cases have between 1 or 2 times the number of nonzero elements as the system coefficient matrix, so they can be considered very effective in terms of required storage space. Note that this does not apply to intermediate storage space necessary when performing the IGO type reduction; in fact, cTIGO fails to produce convergent preconditioners for some of the larger test matrices for the lowest value of the dropping threshold τ we could try before we ran out of memory during the incomplete factorization phase. This is also true for rTIGO for the *jordache* matrix.

5.4 Discussion

The numerical results presented in the previous section show us the best our IGO methods can achieve but we feel some further insights and comments should be made.

Variation of performance over τ: In the table presented in the last section, only the results for the experimental optimal value for τ for each IGO method 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 time-wise. 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 5.1). We clearly see that a choice of τ between

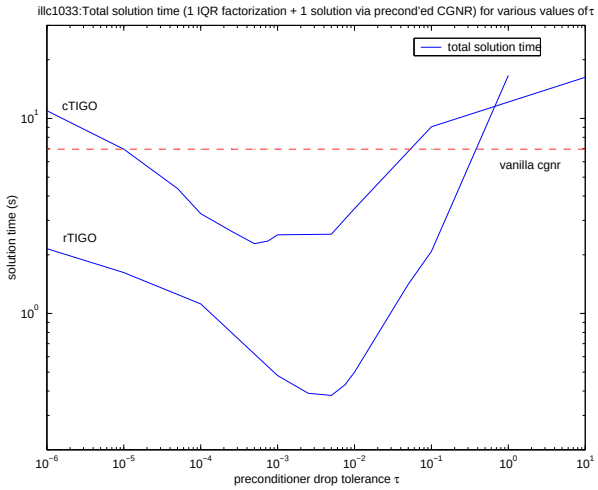

Figure 5.1: *illc_1033*: Total solution time for cTIGO & rTIGO CGNR preconditioning for various τ, versus simple CGNR iteration.

10^{-5} and 10^{-2} for cTIGO or less 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 for it can still give good savings.
Finally, note the roughly cup-like 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 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 total solution time is lowest are exactly those for which a balance between these two extremes has been 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 have seen in various test results how cTIGO effectively stalls as the length of the linked list blows up because of too much fill-in appearing, getting dropped and then reappearing during another rotation.
We illustrate this observation with the example presented in figure 5.2. The chosen matrix (*pigs_s*1) is quite small, but exhibits this fundamental disadvantage of having to use the linked list. Note that the length of the linked list increases linearly over τ except in the transition

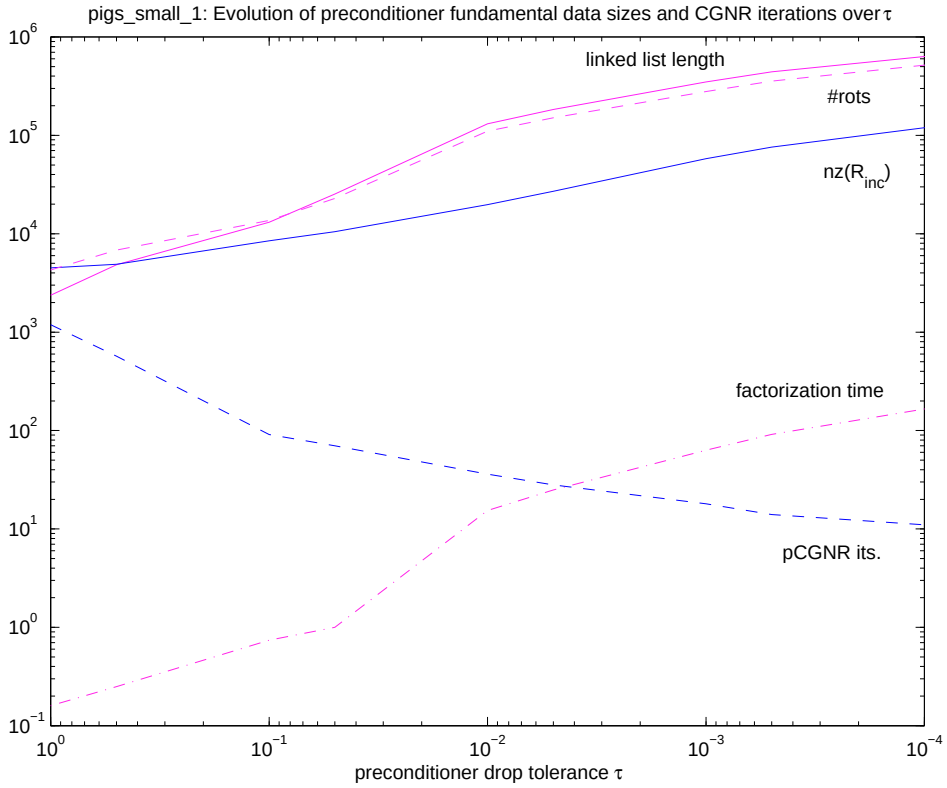

Figure 5.2: *pigs_s*1: Evolution of fundamental preconditioner sizes over τ.

from $\tau = 0.5$ to 0.01, where there is a sudden jump. The same jump is mirrored in the increase in the number of rotations, but more importantly in the time it takes to compute the incomplete factorization. On the other hand, there is no such jump neither in the number of nonzeros in the incomplete triangular factor nor, consequently, in the number of iterations that preconditioned CGNR needs for convergence. Hence, cTIGO performs many unnecessary rotations, in the sense that they do not have any similar 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 it eats up resources but it becomes very time consuming to update it. 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 it.
The necessity to keep the size of the linked list down also justifies our choice to always keep

all elements in the original sparsity pattern P_A of the matrix A. We aim to produce qualitatively good preconditioners with only little of this difficult to accommodate fill-in on top of the nonzeros in the sparsity pattern.

Threshold dependence on main diagonal element: We have already described the dependence of the threshold dropping rules for cTIGO on the value of the main diagonal element, and the relative independence from this for rTIGO. We will here look into this issue in more detail.
For systems arising from least squares problems, most (all) the elements on the main diagonal are usually zeros. 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 element value. This fill-in can in many cases accumulate during the factorization phase and kill performance, as we saw above.
It is however our experience that such a preconditioner will only start being really helpful to convergence once most (or in some cases, all) the main diagonal elements of R_{inc} have been filled with some value. Just filling those zero diagonal elements with an arbitrary value (in our code we chose 1) is simply not enough.
This is exactly the reason why we have chosen to keep the dependence on diagonal elements. 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 out such a fill-in against a zero main diagonal element. We are of course always running the risk of our factorization becoming overflowed with such ineffectual fill-in.
On the other hand, for rTIGO, we have kept this dependence to a minimum; we only employ it when determining the next rotation. But this lead to dropping rules that are not adaptive, but rather based on a rigid threshold value determined by the initial matrix. It would be interesting to see whether other choices for the dropping rules, perhaps with less emphasis on the value of the main diagonal element, but still adaptive, 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 the ones in [GO95] and [RS94] 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).

Performance of IGO methods against other preconditioners for least-squares: Although, because of the apparent lack of least-squares preconditioning software on the public domain, we have not performed any comparative tests, it would be interesting to test the performance of our Givens based incomplete factorizations against other types of preconditioners for such problems. In particular, it would be interesting to produce comparative results against the latest versions of incomplete modified Gram-Schmidt (IMGS), as this approach is also based on incomplete QR factorizations, as well as the preconditioning methods based on A-orthogonalization that are currently being applied to least-squares systems by Benzi and Tuma [BT01]. Still, the results we present in this paper are good enough to show that there is a

future for IGO methods in preconditioning least-squares.

6 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 via a succession of Givens rotations was implemented in both column-wise and row-wise fashion.
We have performed extensive comparative tests 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 much less complex and expensive to use and update than Givens incomplete QR ones. As for the least-squares case, although we did not have any opposing standard as in the square case, the results we presented, coupled with our subsequent observations, show that they can be powerful preconditioning tools, particularly if we consider the fact that for least-squares problems there is a less impressive array of general preconditioning methods currently available.

References

[BDW01] Z.-Z. Bai, I.S. Duff, A.J Wathen, A class of incomplete orthogonal factorization methods. I: methods and theories. BIT 41(1), March 2001.

[BHT00] M. Benzi, J. C. Haws, M. Tuma, 2000. Preconditioning highly indefinite and non-symmetric matrices. SIAM J. Sci. Comp., vol. 22, no. 4, pp. 1333-1353, 2000.

[BSVD99] M. Benzi, D. B. Szyld, A. van Duin, 1999. Orderings for incomplete factorization preconditioning of nonsymmetric problems. SIAM J. Sci. Comp., vol. 20, no. 5, pp. 1652-1670, 1999.

[BT01] M. Benzi, M. Tuma, 2001. A robust incomplete factorization preconditioner for positive definite matrices. Numer. Linear Algebra Appl. 2001; 99:1-20.

[DER86] I. S. Duff, A. M. Erisman, J. K. Reid, 1986. *Direct Methods for Sparse Matrices.* Clarendon Press, Oxford.

[FRSW99] B. Fischer, A. Ramage, D. J. Silvester, A. J. Wathen, 1998. On parameter choice and iterative convergence for stabilised discretisations of advection-diffusion problems. Comput. Methods Appl. Mech. Engrg. 179 (1999).

[GH80] A. George, M. T. Heath, 1980. Solution of sparse linear least squares problems using Givens rotations. Lin. Alg. Appl. 34: 69-83, 1980.

[GO95] M. I. Gillespie, D. D. Olesky, 1995. Ordering Givens rotations for sparse QR factorization. SIAM J. Matrix Anal. Appl., vol. 16, no. 3 , pp. 1024-1041, July 1995.

[HSL00] HSL2000 subroutine package: http://www.cse.clrc.ac.uk/Home/ ; subroutine MC64: 'Permute a sparse matrix to put large entries on the diagonal'; originally developed by I. S. Duff and J. Koster.

[JA84] A. Jennings, M. A. Ajiz, 1984. Incomplete methods for solving $A^T Ax = b$. SIAM J. Sci. Stat. Comp., vol. 5, no. 4, December 1984.

[JAM90] D. James, 1990. PhD Thesis, *Appendix: The breakdown of incomplete QR factorizations*. Research under the supervision of Prof. R. J. Plemmons.

[MAN80] T. A. Manteuffel, 1980. An incomplete factorization technique for positive definite systems. Math. Comp., vol. 34, no. 150, pp. 473-497, April 1980.

[MM00] Matrix Market : http://math.nist.gov/MatrixMarket/data/ : test matrices database maintained by NIST, Gaithersburg, MD.

[RS94] T. H. Robey, D. L. Sulsky, 1994. Row ordering for a sparse QR decomposition. SIAM J. Matrix Anal. Appl., vol. 15, no. 4, pp. 1208-1225, October 1994.

[SAAD88] Y. Saad, 1988. Preconditioning techniques for nonsymmetric and indefinite linear systems. *J. Comput. Appl. Math.*, 24.

[SAAD96] Y. Saad, 1996. *Iterative Methods for Sparse Linear Systems*. PWS Publishing, Boston.

[SPK99] Yousef Saad, SPARSKIT: a basic tool kit for sparse matrix computations, version 2, last updated June 1999. Source code available at: http://www.cs.umn.edu/research/arpa/SPARSKIT/sparskit.html

[STE01] A. Papadopoulos. Incomplete factorization preconditioners for linear systems arising in the SUPG discretization of convection-diffusion problems. Special Topic for Prof. H. Elman's 'Solvers for Incompressible Flow' course, Oxford, Trinity Term 2001.

[TEMPL] R. Barrett, M. Berry, T. F. Chan, J. Demmel, J. M. Donato, J. Dongarra, V. Eijkhout, R. Pozo, C. Romine, H. Van der Vorst. *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods*, online version.

[UFSMC] T. Davis, University of Florida Sparse Matrix Collection, http://www.cise.ufl.edu/research/sparse/matrices, NA Digest, vol. 92, no. 42, October 16, 1994, NA Digest, vol. 96, no. 28, July 23, 1996, and NA Digest, vol. 97, no. 23, June 7, 1997.

[WGB97] X. Wang, K. A. Gallivan, R. Bramley, 1997. CIMGS: An incomplete orthogonal factorization preconditioner. SIAM J. Sci. Comp., vol. 18, no. 2, pp. 516-536, March 1997.

[ZLA91] Z. Zlatev, 1991. *Computational Methods for General Sparse Matrices*, ch. 16. Kluwer Acad. Publ., 1991.

[ZN88] Z. Zlatev, H. B. Nielsen, 1988. Solving large and sparse linear least-squares problems by conjugate gradient algorithms. Comput. Math. Appl., vol. 15, no. 3, pp. 185-202, 1988.

Appendix A Two Examples of cIGO Breakdown

As we have already mentioned, despite the fact that a QR factorization always exists for a general matrix A, and the factor R is nonsingular if A is square and nonsingular, the incomplete variant of QR using Givens rotations can give out an incomplete upper triangular factor R_{inc} that is singular even when A is nonsingular, making the incomplete factorization unusable for preconditioning. Below we identify two situations that might cause such a breakdown.
Consider the nonsingular matrix

$$\mathbf{A} = \begin{pmatrix} 1 & 1 & 0 & 1 \\ 0 & 1 & 1 & 1 \\ 1 & 0 & 3 & 2 \\ 1 & 0 & 0 & 1 \end{pmatrix}.$$

If we apply the cIGO method as described in section 2.1 to A, the first nonzero that we rotate out is the one in position $(4,1)$. Applying the rotation, using element $(1,1)$ as pivot, leads to the matrix

$$\widetilde{\mathbf{A}} = \begin{pmatrix} \sqrt{2} & \frac{1}{\sqrt{2}} & 0 & \sqrt{2} \\ 0 & 1 & 1 & 1 \\ 1 & 0 & 3 & 2 \\ 0 & 0 & 0 & 0 \end{pmatrix},$$

and we can immediately see that the element in position $(4,4)$ will remain zero till the end. Thus, R_{inc} will be singular. This is an example of breakdown caused by sparsity pattern dependencies between two rows. That is, the two rows of the matrix, the first and fourth one, are independent, if viewed as a whole. However, if we look at the segments (columns) of the two rows which are in both row sparsity pattern sets, we see that they are dependent. In a full QR reduction, or if adequate fill-in via thresholding was allowed, this first rotation would create a nonzero in position $(4,2)$, and, when rotating out that element at a subsequent step, the diagonal $(4,4)$ element would reacquire a nonzero value. Note also that, for this case, if rotations were applied the other way around, i.e. first rotate out the $(3,1)$ nonzero and then the $(4,1)$, we would have no breakdown. Of course, one can construct examples where rotations applied in this fashion again lead to breakdown because of dependencies.

Examples of such dependencies are fairly uncommon in practice. They are very unlikely to occur for large systems with nonzero initial diagonals. What usually happens, when a system has a number of zero initial diagonal elements, is that simply not enough rotations are performed to cover up all these zeros on the main diagonal. For example, consider the nonsingular matrix

$$\mathbf{B} = \begin{pmatrix} 1 & 1 & 0 & 1 \\ 0 & 1 & 1 & 1 \\ 1 & 0 & 0 & 2 \\ 1 & 0 & 0 & 3 \end{pmatrix}.$$

Rotating out (and dropping elements) in the usual cIGO fashion first the nonzero in $(4,1)$ and then the one in $(3,1)$, leads to the matrix

$$\widetilde{\mathbf{B}} = \begin{pmatrix} \sqrt{3} & \frac{1}{\sqrt{3}} & 0 & 2\sqrt{3} \\ 0 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & \sqrt{2} \end{pmatrix}.$$

Again, we see that there is no way we can make the third diagonal element nonzero. The sparsity pattern set P_B is simply not large enough for a rotation that will fill the $(3,3)$ position to be performed. There is no easy mechanism to identify how many more elements one needs in the sparsity pattern set to ensure that all diagonals get filled with nonzeros. In fact, even for this small example, it would require all three of elements $(4,2)$, $(3,2)$ and $(4,3)$ to be allowed in the sparsity pattern set in order for $(3,3)$ to become nonzero.

The existence of such examples leads to the question whether there exist classes of matrices with a special property for which, for example, the cIGO factorization is always nonsingular. For example, it can be proven (see e.g. [MAN80]) for ILU type methods that if the matrix A is an M-matrix, then the incomplete upper triangular factor U is also an M-matrix, and hence nonsingular. That is, the M-matrix property of A is retained during the whole process of its incomplete reduction to upper triangular form.
Our question is whether there are such properties that are retained during the reduction via Givens rotations, and, of course, whether we can use them to prove nonsingularity of R_{inc}. Needless to say, the M-matrix property is not one of them.