

# Strengths and limitations of stretching for least-squares problems with some dense rows

JENNIFER SCOTT, STFC Rutherford Appleton Laboratory, UK and University of Reading, UK

MIROSLAV TŮUMA, Charles University, Czech Republic

We recently introduced a sparse stretching strategy for handling dense rows that can arise in large-scale linear least-squares problems and make such problems challenging to solve. Sparse stretching is designed to limit the amount of fill within the stretched normal matrix and hence within the subsequent Cholesky factorization. While preliminary results demonstrated that sparse stretching performs significantly better than standard stretching, it has a number of limitations. In this paper, we discuss and illustrate these limitations and propose new strategies that are designed to overcome them. Numerical experiments on problems arising from practical applications are used to demonstrate the effectiveness of these new ideas. We consider both direct and preconditioned iterative solvers.

CCS Concepts: • **Mathematics of computing** → **Mathematical analysis**; **Numerical analysis**;

Additional Key Words and Phrases: sparse matrices, linear least-squares problems, dense rows, matrix stretching, Cholesky factorization, normal matrix, Schur complement, direct solvers, iterative solvers

## ACM Reference Format:

Jennifer Scott and Miroslav Tůuma. 2020. Strengths and limitations of stretching for least-squares problems with some dense rows. 1, 1 (July 2020), 25 pages. <https://doi.org/xxx>

## 1 INTRODUCTION

Consider the linear least-squares (LS) problem

$$\min_x \|Ax - b\|_2, \quad (1)$$

where the system matrix  $A \in R^{m \times n}$  ( $m \geq n$ ) and the right-hand side vector  $b \in R^m$  are given. The solution  $x$  satisfies the  $n \times n$  normal equations

$$Cx = A^T b, \quad C = A^T A, \quad (2)$$

where, provided  $A$  has full column rank, the *normal matrix*  $C$  is symmetric and positive definite. Our focus is on the case where  $A$  is large and mainly sparse but has some rows that are “dense”. Such rows may be fully dense (all entries are nonzero) or may contain some zeros but nevertheless lead to unacceptable fill in  $C$ . A single dense row is sufficient to cause catastrophic fill in  $C$ . Thus for large-scale problems, a Cholesky or QR factorization of  $C$  is impractical. In particular, the memory demands of a direct solver may be prohibitive (it may not even be possible to form the normal matrix) while the expectation is that using an incomplete factorization of  $C$  as a preconditioner for an iterative solver

---

Authors' addresses: Jennifer Scott, STFC Rutherford Appleton Laboratory, Harwell Campus, Didcot, Oxfordshire, OX11 0QX, UK, University of Reading, Whiteknights, Reading, RG6 6AQ, UK, [jennifer.scott@stfc.ac.uk](mailto:jennifer.scott@stfc.ac.uk); Miroslav Tůuma, Charles University, Prague, Czech Republic, [mirektuma@karlin.mff.cuni.cz](mailto:mirektuma@karlin.mff.cuni.cz).

---

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from [permissions@acm.org](mailto:permissions@acm.org).

© 2020 Association for Computing Machinery.

Manuscript submitted to ACM

Manuscript submitted to ACM

1

such as CGLS [23], LSQR [36] or LSMR [12] applied to (1) is likely to be ineffective because of the large differences between the incomplete and exact factorizations. The presence of dense rows has long been recognised as a fundamental difficulty in the solution of LS problems that are otherwise sparse; see, for example, [2, 5, 9, 13, 42, 43, 45–47]. The analogous problem occurs in interior point methods where the interest is in handling dense columns in  $A$  and the normal matrix is  $AA^T$ ; see, for example, discussions in [4, 15, 16, 31].

In recent years, we have looked at a number of ways to handle dense rows within  $A$ . In particular, we have considered the following:

- A block factorization approach [42] that processes the rows that are identified as dense separately within an iterative solver, using an incomplete factorization preconditioner for the sparse part combined with the factorization of a dense matrix of size equal to the number of dense rows.
- A Schur complement approach [43] that also treats the sparse and dense rows as separate blocks and exploits the block structure within the augmented system formulation of the LS problem. A direct solver or a preconditioned iterative solver for sparse indefinite systems is then employed.
- A sparse stretching strategy [44] that aims to overcome the problems that are observed with standard matrix stretching, in particular, unacceptable amounts of fill in the stretched normal matrix.

We assume the rows of  $A$  that are to be treated as dense are permuted to the end. We also assume conformal partitioning of the vector  $b$  so that (omitting the row permutation matrix for simplicity of notation) we have

$$A = \begin{pmatrix} A_s \\ A_d \end{pmatrix}, \quad A_s \in R^{m_s \times n}, \quad A_d \in R^{m_d \times n}, \quad b = \begin{pmatrix} b_s \\ b_d \end{pmatrix}, \quad b_s \in R^{m_s}, \quad b_d \in R^{m_d}, \quad (3)$$

where  $m_s$  and  $m_d$  denote the number of sparse and dense rows, respectively, with  $m = m_s + m_d$ ,  $m_s \geq n$  and  $m_d \geq 1$  (in general,  $m_s \gg m_d$ ). The LS problem is then

$$\min_x \left\| \begin{pmatrix} A_s \\ A_d \end{pmatrix} x - \begin{pmatrix} b_s \\ b_d \end{pmatrix} \right\|_2. \quad (4)$$

We define  $C_s = A_s^T A_s$  to be the *reduced normal matrix*. When considering the sparse and dense rows within  $A$  separately, the block  $A_s$  of sparse rows may contain null columns and  $C_s$  is then singular with a corresponding number of null rows and columns. In [42, 43], we addressed this by either perturbing the diagonal entries of  $C_s$  or solving a number of related sparse LS problems and combining their solutions to give the solution of the original problem. Both approaches can incur overheads. For the former, an iterative method is needed even if a direct solver is employed to factorize the regularized system, while for the latter, the need to solve more than one LS problem adds an overhead and, for an iterative solver in particular, the increase in the total solution time can be significant.

An advantage of stretching is that, provided  $A$  is of full rank, we are able to handle null columns in the sparse part. However, stretching can result in unacceptable amounts of fill in the stretched normal matrix and its factors. Our sparse stretching strategy [44] aims to limit the fill and our preliminary results demonstrated the potential effectiveness of this idea. Unfortunately, stretching can be costly (in terms of total solution time and the memory requirements) because of the growth in the size of the stretched LS problem. The dimensions of the stretched system can grow rapidly with  $m_d$  so that the stretched LS problem can be considerably larger than the original one. Thus an objective of this study is to propose a novel partial sparse stretching strategy that aims to stretch sufficiently many rows to ensure the sparse row block of the resulting stretched matrix has no null columns and then to employ a block factorization to solve this problem, from which the solution of the original problem can be obtained.

The paper is organised as follows. In Section 2, we describe the algorithm we use to identify the rows of  $A$  that are to be treated as dense. Then, in Section 3, we consider block strategies for solving (4). In particular, we employ a signed block Cholesky factorization which, in exact arithmetic, is mathematically equivalent to the Woodbury formula [49, 50] for the inverse of a rank  $m_d$  modification to the reduced normal matrix. We propose using the block approach to obtain an incomplete factorization preconditioner for use with an iterative solver. We also recall the augmented system and Schur complement approach that can be used in combination with a sparse symmetric indefinite solver. In Section 4, we explain our recent sparse stretching strategy [44] and discuss how it handles null columns in  $A_s$ . In Section 5, we present numerical results for a range of problems, using both direct solvers and a preconditioned iterative solver. Our findings demonstrate that stretching can lead to the need to solve a LS problem that is much larger than the original one and on which iterative solvers can struggle to converge. This motivates us, in Section 6, to propose our partial stretching strategy. Numerical experiments illustrate the advantages that stretching only a small number of rows offers. To try and enhance performance further, in Section 7, we explore other ideas for combining stretching with the construction of incomplete factorization preconditioners. In particular, we propose a variant of sparse stretching that drops small entries from  $A_d$  before stretching is performed. We introduce the concept of weighted sparse stretching that uses the numerical values as well as the pattern of the row blocks  $A_s$  and  $A_d$ . The possibility of developing other preconditioners that exploit the structure of the stretched normal matrix are also briefly discussed. Finally, in Section 8, we summarise our key findings.

### 1.1 Description of the test environment

Our test problems are taken from the SuiteSparse Matrix Collection (<https://sparse.tamu.edu/>) and comprise a subset of those used by Gould and Scott in their study of numerical methods for solving large-scale least-squares problems [17]. In each case, the matrix is “cleaned” (out-of-range entries and explicit zeros are removed along with any null rows or columns). If necessary, the matrix is transposed so that we have an overdetermined system ( $m > n$ ). The  $m_d$  dense rows are identified using Algorithm 1 that is described in Section 2 below.  $n_{ds}$  is the number of null columns in  $A_s$  (if  $n_{ds} > 0$  then the normal matrix  $C_s$  contains null rows and columns). In our experiments, we prescale  $A$  by normalizing each of its columns. That is, we replace  $A$  by  $AD$ , where  $D$  is the diagonal matrix with entries  $D_{ii}$  satisfying  $D_{ii}^2 = 1/\|Ae_i\|_2$  ( $e_i$  denotes the  $i$ -th unit vector). The entries of  $AD$  are at most one in absolute value. In our experiments,  $b$  is set to be the vector of 1’s.

Table 1. Test examples.  $m$  and  $n$  are the row and column dimensions of  $A$ ,  $m_d$  is the number of dense rows,  $n_{ds}$  is the number of null columns in  $A_s$  after the removal of the block  $A_d$  of  $m_d$  dense rows from  $A$ .

Identifier	$m$	$n$	$m_d$	$n_{ds}$
aircraft	7517	3754	17	4
lp_fit2p	13525	3000	25	0
sc205-2r	62423	35213	8	1
scagr7-2r	46679	32847	7	1
scrs8-2r	27691	14364	22	7
scsd8-2r	60550	8650	50	5
sctap1-2b	33858	15390	34	0
south31	36321	18425	381	5
PDE1	271792	270595	1	0
12month1	872622	12471	286	3

The sparse Cholesky solver we employ is HSL\_MA87 [24] from the HSL mathematical software library [27]. It is used with default settings and is combined with a nested dissection ordering to limit fill in the factors. For symmetric

indefinite systems, we use the multifrontal solver HSL\_MA97 [25, 26]) (again, with default settings and nested dissection ordering). Both solvers employ OpenMP for parallelism and the latter uses threshold partial pivoting for numerical stability.

To perform incomplete Cholesky (IC) factorizations, we use the package HSL\_MI35. This implements a limited memory IC algorithm; details are given in [39, 40]. HSL\_MI35 is chosen because the study [17] found that it generally performed well on a range of LS problems. It requires the user to set parameters `lsize` and `rsize` that, respectively, control the number of entries in each column of the IC factor and the memory required to compute it. Increasing either or both of these parameters generally improves the quality of the preconditioner (so that the number of iterations of the preconditioned iterative solver is reduced) but at the cost of more time and memory to compute the factorization; increasing `lsize` also increases the cost of each preconditioner application. In our experiments, we set `lsize` = `rsize`. We found that when stretching was used, results were generally improved by setting the drop tolerances within HSL\_MI35 to zero and so this is used in our reported results.

The characteristics of the machine used to perform our numerical experiments that involve timings are given in Table 2. Unless indicated otherwise, all software is written in Fortran and all reported timings are elapsed times in seconds.

Table 2. Test machine characteristics

CPU	two Intel Core i7-7700 3.6Ghz Quad Core processors
Memory	16 GB
Compiler	gfortran version 7.3.0 with options -O3 -fopenmp
BLAS	Intel MKL

When using an iterative solver, the initial solution guess is taken to be  $x^{(0)} = 0$  and, following Gould and Scott [17], we require the computed residual  $r = b - Ax$  to satisfy either  $\|r\|_2 < \delta_1$  or  $\|A^T r\|_2 / \|r\|_2 < \delta_2 \times \|A^T b\|_2 / \|b\|_2$ . The convergence tolerances  $\delta_1$  and  $\delta_2$  are set to  $10^{-8}$  and  $10^{-6}$ , respectively.

## 2 IDENTIFICATION OF DENSE ROWS

How  $A$  is partitioned into the sparse part  $A_s$  and dense part  $A_d$  determines the efficiency of the overall solution process. In some cases, it is very clear which rows should be treated as dense and consequently a number of papers that recognise the need to handle dense rows do not propose how to identify such rows (see, for example, [2, 5, 22]). In [42], we illustrated the importance of identifying all the dense rows and moving them into  $A_d$ . Following Sun [45] and Vanderbei [47], in [43], we used the following simple definition for a dense row of  $A$ : given a sparsity threshold parameter  $\rho$  ( $0 < \rho \leq 1$ ), row  $i$  of  $A$  is defined to be dense if the percentage of entries in row  $i$  is at least  $\rho$ . We reported on the effect of different choices of  $\rho$ ; the optimum value is problem dependent. While increasing  $\rho$  reduces the number  $m_d$  of rows classified as dense, it can significantly increase the density of the reduced normal matrix  $C_s$ , leading to a poorer quality incomplete factorization preconditioner. In some test cases, we found it was necessary to choose  $\rho$  to be very small to ensure  $C_s$  was sufficiently sparse but this can result in  $m_d$  being large so that the cost of processing the dense part can then dominate the total solution time.

While the simple threshold approach is suitable when the dense and sparse rows are clearly separated by their densities, it does not always work well when there is less distinction (that is, it can fail to identify rows that if moved into  $A_d$  improves performance and  $m_d$  can then be particularly sensitive to the choice of  $\rho$ ). An alternative approach

is given by Meszaros [31] in his paper on interior point methods for linear programmes using the normal equations approach. Meszaros aims to detect rows that, if moved into  $A_d$ , will limit the fill in the reduced normal matrix  $C_s$ . The method starts by removing from  $A$  rows that have more than a chosen number of entries (Meszaros uses 350 in his tests, independent of the problem size); let the remaining matrix be  $\bar{A}$ . The algorithm then looks at the fill in the corresponding normal matrix caused by each row of  $\bar{A}$ . It computes the sparsity of the rows of  $\bar{A}^T \bar{A}$  one-by-one, processing the rows of  $\bar{A}$  in increasing order of their number of entries (row counts) and for each computing how many fill entries it contributes to  $\bar{A}^T \bar{A}$ . When processing row  $k$ , if an entry of  $\bar{A}^T \bar{A}$  is non zero from a row that has already been processed, then it is not counted as a fill entry for row  $k$  (thus only the number of so-called *new fill entries* for row  $k$  is counted). In the second part of the algorithm, Meszaros takes into account the rows with a large number of entries that were initially discarded to compute the approximate number of entries in each column of the normal matrix  $C$ . The rows that lead to significant fill in  $C$  are added into  $A_d$  and the remaining rows form  $A_s$ .

We employ the following modified version of Meszaros' approach. Here  $r_i$  denotes the row count for row  $i$  of  $A$ . The

---

**Algorithm 1** Identification of dense rows.

---

**Require:**  $m \times n$  matrix  $A$ ; parameters  $\rho$  ( $0 < \rho \leq 1$ ),  $\gamma$  ( $0 < \gamma \leq 1$ ),  $\delta > 0$ ,  $mfill > 0$  and  $small \geq 0$ .

**Ensure:**  $A$  partitioned as  $\begin{pmatrix} A_s \\ A_d \end{pmatrix}$ , with  $A_d$  comprising the dense rows.

- 1: Initialization. For  $i = 1, 2, \dots, m$  compute the row count  $r_i$  and set  $flag(i) = 0$ .
  - 2: Permute the rows of  $A$  into increasing order of  $r_i$ . Let the permutation vector be  $q$  and the permuted matrix be  $QA$ .
  - 3: For  $i = 1, 2, \dots, m$  set  $flag(i) = 1$  if  $r_i \geq \rho * n$ .
  - 4: Let  $\bar{m}$  be the number of rows for which  $flag(i) = 0$ . If  $\bar{m} = 0$  terminate with  $A = A_d$ ; otherwise remove the last  $m - \bar{m}$  rows from  $QA$  and let the remaining  $\bar{m} \times n$  matrix be  $\bar{A}$ .
  - 5: For  $i = 1, 2, \dots, \bar{m}$  compute the number  $fill_i$  of new fill entries that row  $i$  contributes to  $\bar{A}^T \bar{A}$ .
  - 6: Set  $fill\_max = \max_{1 \leq k \leq \bar{m}} fill_k$ . If  $fill\_max < mfill$  go to step 9; otherwise set  $count = 0$ .
  - 7: For  $i = 1, 2, \dots, \bar{m}$  if  $fill_i \geq \gamma * fill\_max$  set  $flag(j) = 1$  where  $j = q(i)$ ; otherwise if  $fill_i > small$  increment  $count = count + 1$ .
  - 8: If  $count < \delta$  then for each  $i$  such that  $fill_i > small$  set  $flag(j) = 1$  where  $j = q(i)$ .
  - 9: Move all rows  $i = 1, 2, \dots, m$  of  $A$  such that  $flag(i) = 1$  into  $A_d$ ; the remaining rows form  $A_s$ .
- 

algorithm starts by permuting the rows of  $A$  into increasing order of their row counts. Rows for which  $r_i$  is at least  $\rho * n$  are immediately flagged as dense and removed from  $A$ , leaving a matrix  $\bar{A}$  with  $\bar{m}$  rows. In our experiments, we use  $\rho = 0.05$  or  $0.1$ . Meszaros observes that the sparsity structure of the rows of  $\bar{A}^T \bar{A}$  is easy to build for a sequence  $S_0 \subset S_1 \subset \dots \subset S_{\bar{m}}$ , where  $S_0 = 0$  and  $S_{\bar{m}} = \{1, 2, \dots, \bar{m}\}$ . He sets up a sequence  $S_j$  and builds the sparsity structure of the columns of  $\bar{A}_j^T \bar{A}_j$  one-by-one (where  $\bar{A}_j$  comprises the rows of  $\bar{A}$  indexed by  $S_j$ ), while recording and setting up the fill (which for row  $i$  we denote by  $fill_i$ ). Specifically, Meszaros defines  $S_j$  by appending to  $S_{j-1}$  a row with minimal row count, i.e.

$$S_j = S_{j-1} \cup k, \quad k = \arg \min \{r_l : l \in \{1, 2, \dots, \bar{m}\} \setminus S_{j-1}\}.$$

This is done in step 5 of the algorithm (see [31] for details). Having determined the fill when the rows of  $\bar{A}$  are processed in order of increasing row counts, we compute the maximum fill ( $fill\_max$ ) and use it to determine whether to flag additional rows as dense. If  $fill\_max$  is less than some chosen amount  $mfill$  (in our tests, we set  $mfill = \max(n/100, 100)$ ) then the fill in  $\bar{A}^T \bar{A}$  is considered to be small, no further rows are flagged and the algorithm terminates. Otherwise, row  $i$  of  $\bar{A}$  is flagged as dense if  $fill_i$  is within a factor  $\gamma$  of the maximum fill; in our experiments we use  $\gamma = 0.8$ . The number of unflagged rows for which  $fill_i$  exceeds some threshold  $small$  is also counted. If the total number

of such rows is less than  $\delta$ , we flag them all as dense. Our motivation here is that if a small number *count* of rows contributes to most of the fill, then we can treat all these rows as dense. In our tests, we set *small* = 10 and  $\delta = 0.1 * m$ . Our choice of parameter values is based on experimentation on practical problems.

### 3 BLOCK APPROACHES FOR HANDLING DENSE ROWS

Here we describe block strategies to solve the LS problem (4) and discuss how they can be applied together with either a direct or an iterative solver. While in exact arithmetic some of the approaches are equivalent, this is not the case when they are used as preconditioners: a small change can result in very different behaviour (see, for example, the mathematically equivalent formulas from Lemma 2.3 and 2.4 in [42] that numerically behave very differently).

#### 3.1 A block signed Cholesky factorization

Using the partitioning (3), the normal equations are given by

$$Cx = (C_s + A_d^T A_d)x = c, \quad c = A_s^T b_s + A_d^T b_d. \quad (5)$$

The solution of (5) can be obtained from the equivalent  $(n + m_d) \times (n + m_d)$  system

$$\begin{pmatrix} C_s & A_d^T \\ A_d & -I \end{pmatrix} \begin{pmatrix} x \\ A_d x \end{pmatrix} = \begin{pmatrix} c \\ 0 \end{pmatrix}. \quad (6)$$

Provided  $A_s$  has full column rank,  $C_s$  is symmetric positive definite and, if the partitioning (3) is such that all the rows of  $A_s$  are sparse,  $C_s$  is generally significantly sparser than the original normal matrix  $C$ . Let  $C_s = L_s L_s^T$  be the Cholesky factorization of  $C_s$ . This yields a signed Cholesky factorization

$$\begin{pmatrix} C_s & A_d^T \\ A_d & -I \end{pmatrix} = \begin{pmatrix} L_s & \\ B_d & L_d \end{pmatrix} \begin{pmatrix} I & \\ & -I \end{pmatrix} \begin{pmatrix} L_s^T & B_d^T \\ & L_d^T \end{pmatrix}, \quad (7)$$

where

$$L_s B_d^T = A_d^T \quad (8)$$

and

$$S_d = I + B_d B_d^T = L_d L_d^T. \quad (9)$$

Recall we are assuming  $m_d$  is small so that the Cholesky factorization of the dense  $m_d \times m_d$  (negative) Schur complement  $S_d$  can be computed using dense linear algebra (and is trivial in the not uncommon case  $m_d = 1$ ). Algorithm 2 summarizes the steps to compute the LS solution using (6) once  $L_s$ ,  $B_d$ , and  $L_d$  have been computed. Note that it is not necessary to

---

**Algorithm 2** Block factorization approach for solving sparse-dense LS problems.

---

**Require:**  $B_d$ , the Cholesky factors  $L_s$  and  $L_d$ , and  $c = A_s^T b_s + A_d^T b_d$ .

**Ensure:** LS solution  $x$ .

- 1: Solve  $L_s u_s = c$ .
  - 2: Compute  $w_d = B_d u_s$ .
  - 3: Solve  $L_d u_d = w_d$  and then  $L_d^T y_d = u_d$ .
  - 4: Form  $w_s = u_s - B_d^T y_d$ .
  - 5: Solve  $L_s^T x = w_s$ .
-

compute and store  $B_d$  explicitly; instead (8) can be used in Steps 2 and 4 and in computing  $S_d$ . Note also that  $A_d$  may represent a set of rows (which are not necessarily dense) that are appended to the original matrix  $A = A_s$ . In this case, the algorithm can be viewed as an updating procedure in which the normal matrix factorization remains fixed, thereby limiting the additional work needed to handle the added rows.

### 3.2 Relationship with Woodbury formula

A standard tool for combining the inverse of  $C_s$  with an update based on  $A_d$  is the Woodbury formula (see [20, 49, 50] and the discussion in the review paper of Hager [21]). This formula (which is also sometimes referred to as the Sherman-Morrison-Woodbury formula) allows the inverse of the normal matrix to be written in the form

$$C^{-1} = (C_s + A_d^T A_d)^{-1} = C_s^{-1} - C_s^{-1} A_d^T (I + A_d C_s^{-1} A_d^T)^{-1} A_d C_s^{-1}. \quad (10)$$

The LS solution may then be explicitly expressed as

$$x = x_s + C_s^{-1} A_d^T (I + A_d C_s^{-1} A_d^T)^{-1} (b_d - A_d x_s) \quad \text{with} \quad x_s = C_s^{-1} A_s^T b_s. \quad (11)$$

It is straightforward to show that Algorithm 2 is mathematically equivalent (that is, in exact arithmetic) to the Woodbury formula. From Algorithm 2,

$$\begin{aligned} L_s^T x &= w_s \\ &= u_s - B_d^T y_d \\ &= u_s - B_d^T (L_d L_d^T)^{-1} w_d \\ &= u_s - B_d^T (L_d L_d^T)^{-1} B_d u_s. \end{aligned}$$

Thus using  $C_s = L_s L_s^T$  and (8) and (9),

$$\begin{aligned} C_s x &= L_s u_s - L_s B_d^T (L_d L_d^T)^{-1} B_d u_s \\ &= c - A_d^T (I + B_d B_d^T)^{-1} A_d L_s^{-T} u_s \\ &= c - A_d^T (I + B_d B_d^T)^{-1} A_d C_s^{-1} c \\ &= c - A_d^T (I + A_d L_s^{-T} L_s^{-1} A_d^T)^{-1} A_d C_s^{-1} c. \end{aligned}$$

Recall  $c = A_s^T b_s + A_d^T b_d = C_s x_s + A_d^T b_d$ . Hence

$$\begin{aligned} x &= x_s + C_s^{-1} A_d^T b_d - A_d^T (I + A_d C_s^{-1} A_d^T)^{-1} A_d C_s^{-1} (C_s x_s + A_d^T b_d) \\ &= x_s + C_s^{-1} A_d^T (I + A_d C_s^{-1} A_d^T)^{-1} ((I + A_d C_s^{-1} A_d^T) b_d - A_d (x_s + C_s^{-1} A_d^T b_d)) \\ &= x_s + C_s^{-1} A_d^T (I + A_d C_s^{-1} A_d^T)^{-1} (b_d - A_d x_s), \end{aligned}$$

which is the LS solution (11). Thus Algorithm 2 provides a practical implementation of the Woodbury formula.

### 3.3 Block approach to compute a preconditioner

The block approach can be employed to obtain a preconditioner for use with an iterative solver applied to sparse-dense LS problems. It is straightforward to verify that the following relationships hold

$$C_s + A_d^T A_d = \begin{pmatrix} I & 0 \\ 0 & -I \end{pmatrix} \begin{pmatrix} C_s & A_d^T \\ A_d & -I \end{pmatrix} \begin{pmatrix} I \\ A_d \end{pmatrix}$$

and

$$(C_s + A_d^T A_d)^{-1} = \begin{pmatrix} I & 0 \\ A_d & -I \end{pmatrix} \begin{pmatrix} C_s & A_d^T \\ A_d & -I \end{pmatrix}^{-1} \begin{pmatrix} I \\ 0 \end{pmatrix}. \quad (12)$$

If we compute an incomplete Cholesky factorization  $C_s \approx \tilde{L}_s \tilde{L}_s^T$  then we can obtain an incomplete version of (7), that is,

$$\begin{pmatrix} C_s & A_d^T \\ A_d & -I \end{pmatrix} \approx \begin{pmatrix} \tilde{L}_s & \\ & \tilde{L}_d \end{pmatrix} \begin{pmatrix} I & \\ & -I \end{pmatrix} \begin{pmatrix} \tilde{L}_s^T & \tilde{B}_d^T \\ & \tilde{L}_d^T \end{pmatrix}, \quad (13)$$

with

$$\tilde{L}_s \tilde{B}_d^T = A_d^T \quad \text{and} \quad \tilde{S}_d = I + \tilde{B}_d \tilde{B}_d^T = \tilde{L}_d \tilde{L}_d^T.$$

Combined with (12), the factorization (13) can be used to obtain a symmetric positive definite preconditioner  $M$  for use with an iterative solver such as CGLS, LSQR or LSMR for (4). Observe from (12) that  $y = (C_s + A_d^T A_d)^{-1} z$  can be computed from the solution of the system

$$\begin{pmatrix} C_s & A_d^T \\ A_d & -I \end{pmatrix} \begin{pmatrix} y \\ \hat{y} \end{pmatrix} = \begin{pmatrix} z \\ 0 \end{pmatrix}. \quad (14)$$

Setting  $M$  to be the signed incomplete factorization (13) and using (14), the steps needed to apply the preconditioner are as in Algorithm 2 with  $L_s$  is replaced  $\tilde{L}_s$ .

---

**Algorithm 3** Application of the block incomplete factorization preconditioner.

---

**Require:**  $\tilde{L}_s, \tilde{L}_d, A_d$ , and the vector  $z$ .

**Ensure:**  $y = M^{-1}z$ .

- 1: Solve  $\tilde{L}_s u_s = z$ .
  - 2: Compute  $w_d = A_d \tilde{L}_s^{-T} u_s$ .
  - 3: Solve  $\tilde{L}_d u_d = w_d$  and then  $\tilde{L}_d^T \hat{y} = u_d$ .
  - 4: Form  $w_s = u_s - \tilde{L}_s^{-1} A_d^T \hat{y}$ .
  - 5: Solve  $\tilde{L}_s^T y = w_s$ .
- 

### 3.4 Relationship with our earlier block approach

In [42], we proposed processing dense rows separately within a conjugate gradient method. Specifically, we employed CGLS with an incomplete factorization preconditioner, which we denote here by  $M_{CG}$ . Let  $r$  denote the residual  $r = b - Ax$ . At each CGLS iteration, the preconditioner is applied to the transformed residual vector  $z = A^T r = A_s^T r_s + A_d^T r_d$ , where  $r_s = b_s - A_s x$  and  $r_d = b_d - A_d x$ . From Algorithm 2.7 of [42], in the Cholesky mode the application of the preconditioner  $y = M_{CG}^{-1} z$  is given by

$$y = \tilde{L}_s^{-T} \tilde{L}_s^{-1} (A_s^T r_s - A_d^T \tilde{S}_d^{-1} (A_d \tilde{L}_s^{-T} \tilde{L}_s^{-1} A_s^T r_s - r_d)). \quad (15)$$

Similarly, in Algorithm 3 above,  $y = M^{-1} z$  is given by

$$y = \tilde{L}_s^{-T} \tilde{L}_s^{-1} (z - A_d^T \tilde{S}_d^{-1} A_d \tilde{L}_s^{-T} \tilde{L}_s^{-1} z). \quad (16)$$

Recalling (8) and (9) and using the identity

$$\tilde{S}_d^{-1} A_d \tilde{L}_s^{-T} \tilde{L}_s^{-1} A_d^T = \tilde{S}_d^{-1} B_d B_d^T = I - \tilde{S}_d.$$



we obtain

$$\tilde{S}_d^{-1} A_d \tilde{L}_s^{-T} \tilde{L}_s^{-1} (A_s^T r_s + A_d^T r_d) = r_d + \tilde{S}_d^{-1} (A_d \tilde{L}_s^{-T} \tilde{L}_s^{-1} A_s^T r_s - r_d). \quad (17)$$

Setting  $z = A_s^T r_s + A_d^T r_d$  in (16) and employing (17) yields (15). Thus the preconditioner in Algorithm 3 and in [42] are mathematically equivalent. In practice, their application may differ because the equivalence is in the case of exact arithmetic, which we generally do not have.

### 3.5 Augmented system and Schur complement approaches

Solving (4) is equivalent to solving the  $(m+n) \times (m+n)$  augmented system

$$K \begin{pmatrix} r_s \\ r_d \\ x \end{pmatrix} = \begin{pmatrix} b_s \\ b_d \\ 0 \end{pmatrix}, \quad K = \begin{pmatrix} I & & A_s \\ & I & A_d \\ A_s^T & A_d^T & 0 \end{pmatrix} \quad (18)$$

where

$$r = \begin{pmatrix} r_s \\ r_d \end{pmatrix} = \begin{pmatrix} b_s \\ b_d \end{pmatrix} - \begin{pmatrix} A_s \\ A_d \end{pmatrix} x$$

is the residual vector. The matrix  $K$  is symmetric and indefinite. (18) can be solved using a general-purpose sparse direct solver that ignores the block form and employs numerical pivoting to ensure stability. However, obtaining robust incomplete factorizations preconditioners for such systems (and symmetric indefinite matrices generally) is challenging (see, for example, [8]). It has been considered recently [19, 33, 41] (again, ignoring the existence of the dense rows in  $A_d$ ) but with much less success than the positive definite case.

A modification is to eliminate  $r_s$  and reduce the problem to a smaller 2-block system

$$K_r \begin{pmatrix} x \\ r_d \end{pmatrix} = \begin{pmatrix} -A_s^T b_s \\ b_d \end{pmatrix}, \quad K_r = \begin{pmatrix} -C_s & A_d^T \\ A_d & I \end{pmatrix}. \quad (19)$$

We refer to the  $(n+m_d) \times (n+m_d)$  system (19) as the *reduced augmented system*. The reduced augmented matrix  $K_r$  is symmetric quasi-definite. Again, this system can be solved using a symmetric indefinite solver. This is able to handle the case where  $C_s$  is rank deficient (and not only the case where it has null rows and columns). Alternatively, a block factorization of  $K_r$  can be computed that treats the sparse and dense blocks separately; this is the Schur complement approach [43] (see also [14, 30, 38]). Using the notation of Section 3.1, the block factorization of  $K_r$  is

$$K_r = \begin{pmatrix} L_s & \\ B_d & L_d \end{pmatrix} \begin{pmatrix} -I & \\ & I \end{pmatrix} \begin{pmatrix} L_s^T & B_d^T \\ & L_d^T \end{pmatrix}, \quad (20)$$

where  $C_s = L_s L_s^T$ ,  $B_d$  is given by (8) and  $L_d$  is the Cholesky factor of the (negative) Schur complement  $S_d$  given by (9). If the Cholesky factorization of  $C_s$  is replaced by an IC factorization then we obtain an indefinite preconditioner  $M_S$ . Writing the preconditioning operation as

$$y = \begin{pmatrix} y_s \\ y_d \end{pmatrix} = M_S^{-1} \begin{pmatrix} z_s \\ z_d \end{pmatrix},$$

we have (with the same notation as in Section 3.3 for the incomplete factors)

$$y_s = -\tilde{L}_s^{-T} \tilde{L}_s^{-1} (z_s + A_d^T y_d) \quad \text{and} \quad y_d = \tilde{S}_d^{-1} (z_d - A_d \tilde{L}_s^{-T} \tilde{L}_s^{-1} z_s).$$

That is,

$$y_s = -\tilde{L}_s^{-T} \tilde{L}_s^{-1} (z_s - A_d^T \tilde{S}_d^{-1} (A_d \tilde{L}_s^{-T} \tilde{L}_s^{-1} z_s - z_d)).$$

Comparing this with (15), we see that computing  $y_s$  involves the same operations as applying the preconditioner  $M_{CG}$ . As  $M_S$  is indefinite, it is used with a general non symmetric iterative method such as GMRES. Alternatively, we can obtain a positive-definite preconditioner for use with MINRES [35] by replacing  $M_S$  by

$$|M_S| = \begin{pmatrix} \tilde{L}_s & \\ & \tilde{B}_d \end{pmatrix} \begin{pmatrix} \tilde{L}_s^T & \tilde{B}_d^T \\ & \tilde{L}_d^T \end{pmatrix} \quad (21)$$

see, for example, [37]. Our experience has been that, for our LS problems, MINRES is not competitive with GMRES in terms of iteration counts and times, although it does offer the advantage of requiring less storage [43].

#### 4 A BRIEF INTRODUCTION TO MATRIX STRETCHING FOR LS PROBLEMS

Stretching offers a very different approach to handling dense rows. It aims to split the rows of  $A_d$  into a number of sparser rows and to formulate a (larger) modified problem from which the solution to the original problem can be derived. The idea was proposed by Grear [18] and was subsequently used for solving linear systems by Alvarado [3], Ferris and Horn [11], Aykannat, Pinar and Çatalyürek [6], and Duff and Scott [10]. For LS problems, assume initially that  $A_d$  represents a single dense row, which (following the notation used by Adlers and Björck [2]) we denote by  $f^T$ . The stretching procedure has two steps. In the first, a larger problem is constructed by splitting the dense row  $f^T$  into two parts  $f^T = (f_a^T \ f_b^T)$  such that  $f_a$  and  $f_b$  contain the same number of nonzeros and introducing a linking variable  $s \in R$ . Let us split the sparse block  $A_s$  and the solution vector  $x$  as  $A_s = (A_{sa} \ A_{sb})$ ,  $x = (x_a^T \ x_b^T)^T$  to conform with the splitting of  $f^T$ . It is straightforward to observe [2] that the component  $x$  of the solution of the extended LS problem

$$\min_{(x^T \ s)^T} \left\| \begin{pmatrix} A_{sa} & A_{sb} & 0 \\ f_a^T & f_b^T & 0 \\ f_a^T & -f_b^T & \sqrt{2} \end{pmatrix} \begin{pmatrix} x_a \\ x_b \\ s \end{pmatrix} - \begin{pmatrix} b_s \\ b_d \\ 0 \end{pmatrix} \right\|_2 \quad (22)$$

is the same as the solution  $x$  of the original problem (1). The second step applies an orthogonal transformation to the extended system matrix in (22) to replace  $f_b^T$  in the second block row and  $f_a^T$  in the third block row by zeros (see [2] for details). Orthogonal invariance of the norm leads to the equivalent stretched problem

$$\min_z \|\mathcal{A}z - \hat{b}\|_2$$

with

$$\mathcal{A} = \begin{pmatrix} A_{sa} & A_{sb} & 0 \\ \sqrt{2} f_a^T & 0 & 1 \\ 0 & \sqrt{2} f_b^T & -1 \end{pmatrix}, \quad z = \begin{pmatrix} x_a \\ x_b \\ s \end{pmatrix}, \quad \hat{b} = \begin{pmatrix} b_s \\ b_d/\sqrt{2} \\ b_d/\sqrt{2} \end{pmatrix}.$$

Extending this to splitting  $f^T$  into  $k \geq 2$  parts (with each part containing essentially the same number of nonzeros) gives

$$\mathcal{A} = \begin{pmatrix} A_s \\ F^T \\ s \end{pmatrix}, \quad A_d = f^T = \frac{1}{\sqrt{k}} e^T F^T, \quad F^T \in R^{k \times n}, \quad z = \begin{pmatrix} x \\ s \end{pmatrix}, \quad \hat{b} = \begin{pmatrix} b_s \\ b_d e/\sqrt{k} \end{pmatrix}, \quad (23)$$

where  $e \in R^k$  is the vector of ones,  $s \in R^{k-1}$  and the linking matrix  $S \in R^{k \times (k-1)}$  is given by

$$S = \begin{pmatrix} 1 & & & & & \\ -1 & 1 & & & & \\ & & -1 & \ddots & & \\ & & & \ddots & \ddots & \\ & & & & \ddots & 1 \\ & & & & & -1 \end{pmatrix}. \quad (24)$$

Stretching can be generalised to more than one dense row by stretching one row at a time. A theoretical analysis, accompanied by numerical results for some small sparse problems with a small number of dense rows appended, was given by Adlers and Björck [1, 2].

Standard stretching splits the row indices of the nonzero entries in the dense rows into sets of (almost) equal contiguous segments and then bases the stretching on these sets (but see also a dynamic approach to splitting dense columns in [16]). Unfortunately, this can result in significant fill in the stretched normal matrix and, in particular, in its factor. Simply increasing the number of parts into which the dense rows are split does not necessarily alleviate the problem and can adversely effect the conditioning of the stretched normal matrix. This prompted us to introduce a new sparse stretching strategy that aims to limit the fill in the stretched normal matrix [44]. To motivate this strategy, consider the stretched normal matrix

$$C = \mathcal{A}^T \mathcal{A} = \begin{pmatrix} A_s^T & F \\ & S^T \end{pmatrix} \begin{pmatrix} A_s & \\ F^T & S \end{pmatrix} = \begin{pmatrix} A_s^T A_s + FF^T & FS \\ S^T F^T & S^T S \end{pmatrix}. \quad (25)$$

It is clear that the fill in the principal leading block  $A_s^T A_s + FF^T$  of  $C$  resulting from the dense rows is a minimum if the structure of  $FF^T$  is contained within that of  $A_s^T A_s$ , that is, if

$$\text{Struct}(FF^T) \subseteq \text{Struct}(A_s^T A_s), \quad (26)$$

where for a matrix  $X$ ,  $\text{Struct}(X)$  denotes the set of positions  $(i, j)$  of the non zero entries of  $X$ . A simple example is given in Figure 1. Here the last row  $f^T$  is considered to be dense and the union of the sparsity patterns of rows  $a_j^T$  and  $a_k^T$  covers that of  $f^T$ . If  $f^T$  is split into two rows  $F^T = (f_a^T \ f_b^T)$  as shown then condition (26) is satisfied.

$$\begin{array}{c} \vdots \\ a_k^T \\ a_j^T \\ \vdots \\ f^T \end{array} \begin{pmatrix} & 1 & 2 & 3 & 4 & 5 & 6 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ * & * & & * & * & * \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ * & & & * & * & * \end{pmatrix} \quad \begin{array}{c} \vdots \\ a_k^T \\ a_j^T \\ \vdots \\ f_a^T \\ f_b^T \end{array} \begin{pmatrix} & 1 & 2 & 3 & 4 & 5 & 6 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ * & * & & * & * & * \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ * & & & * & & \\ & & & & * & * \end{pmatrix}$$

Fig. 1. Matrix with a dense row  $f^T$ ; the remaining rows comprise  $A_s$  (left). The rows  $a_j^T$  and  $a_k^T$  in  $A_s$  structurally cover  $f^T$ . Sparse stretching creates the block  $F^T = (f_a^T \ f_b^T)$  such that  $\text{Struct}(FF^T) \subseteq \text{Struct}(A_s^T A_s)$  (right).

The idea behind sparse stretching is to find a splitting of the sparsity pattern  $\text{Struct}(f)$  into  $k$  disjoint non-empty index sets  $t_1, \dots, t_k$  and to construct  $F^T$  so that its  $i$ -th row contains the  $|t_i|$  entries of  $\text{Struct}(f)$  corresponding to the

$i$ -th part of the splitting. Any such splitting that satisfies (26) is said to define a *correct stretching* of row  $f^T$ . Sparse stretching constructs a correct stretching by seeking a (small) set of rows within the sparse part  $A_s$  such that the union of their patterns covers that of the dense row. The number of rows in this set is then the number of parts  $f^T$  is stretched into and  $f^T$  is stretched using the patterns of the rows in this set.

When there is more than one dense row ( $m_d > 1$ ), each such row is stretched separately and the size of the stretched system is thus dependent on the patterns of the dense and sparse rows and on the number of dense rows. If a dense row contains  $\eta > 0$  nonzero entries with column indices that do not occur in any of the rows of  $A_s$  then we simply split the row and put the entries that match the zero columns in  $A_s$  into a separate set. This set is small because it is easily observed that  $\eta \leq m_d$  and  $m_d$  is assumed to be small. This slightly generalizes the concept of sparse stretching that we introduced in [44].

Results presented in [44] for some simple test examples demonstrated that sparse stretching can find a suitable number of parts for the splitting that leads to a significant reduction in the fill in both the stretched normal matrix and its Cholesky factor compared to standard stretching. A key objective of this paper is to explore more generally the effectiveness of sparse stretching for LS problems when used with a direct method and with a preconditioned iterative solver and to propose new ideas and research directions for tackling challenging practical applications.

## 5 NUMERICAL EXPERIMENTS

Having introduced block methods and sparse stretching for LS problems, in this section we present numerical results which illustrate the effectiveness of the methods.

### 5.1 Preliminary stretching results

We start by considering problem sctap1-2b, which has  $m_d = 34$  dense rows. In this experiment, we remove all the dense rows and solve the modified problem and then solve a sequence of problems with  $m_d = 1, 2, \dots, 34$  dense rows added back in. In Table 3, we show how the dimensions of the stretched problem grow as the number of dense rows increases. We also report the effect stretching has on the conditioning of this problem. The condition number estimate

Table 3. The dimensions and condition number estimate for the stretched problem sctap1-2b as the number of dense rows increases.  $m_{str}$  and  $n_{str}$  are the row and column dimensions of the stretched system, and  $cond(C)$  denotes the condition number estimate for the stretched normal matrix  $C$ .

$m_d$	$m_{str}$	$n_{str}$	$cond(C)$
0	33858	15375	$1.7 \times 10^7$
1	34339	15904	$7.3 \times 10^{13}$
5	36399	17260	$4.2 \times 10^{14}$
10	38974	20530	$3.7 \times 10^{15}$
15	40260	21818	$1.4 \times 10^{16}$
20	41550	23096	$1.4 \times 10^{16}$
30	44136	25672	$2.2 \times 10^{16}$
34	45172	26704	$2.3 \times 10^{16}$

of the stretched normal matrix  $cond(C)$  is computed using the MATLAB function `condst`. We see that the size of the stretched system grows quickly and that stretching just one row results in a significant increase in  $cond(C)$  (note that MATLAB may return an overestimate). Experiments with other examples also found that the condition number of the stretched normal matrix is much larger than that of the original normal matrix; this is consistent with the upper bound on the condition number of the stretched system presented by Adlers and Björck [2] (see also [44]).

## 5.2 Results with a direct solver

The stretched normal matrix  $C$  (25) is symmetric positive definite and so a sparse Cholesky solver may be employed; the effectiveness of this is illustrated in Table 4. Here we compare standard and sparse stretching (for the former, the number of parts the dense rows are stretched into is taken to be the same as for sparse stretching so that the stretched matrices have the same dimensions for both approaches); all rows identified as dense using Algorithm 1 are stretched. For completeness, we include results for solving with no stretching (that is, the Cholesky solver is applied to the  $n \times n$  normal equations (2) for the original problem). Despite the increase in the problem dimensions, standard and sparse stretching perform considerably better than simply ignoring the existence of the dense rows. We also observe that for some problems (such as scagr7-2r and scrs8-2r) sparse stretching gives much better results than standard stretching. Note that problem PDE1 has a single dense row but even in this case, sparse stretching is clearly advantageous compared to standard stretching. For problems aircraft and lp\_fit2p, the sparse part  $A_s$  is diagonal and so if the dense row  $j$  has  $nz_j$  entries, it is stretched into  $nz_j$  parts (and both standard and sparse stretching are the same for these examples). We were unable to solve problem 12month1 because, with and without stretching, the Cholesky factorization broke down (a negative pivot was encountered), suggesting the matrix is rank deficient.

Table 4. A comparison of no stretching (“none”) and the standard and sparse stretching strategies when combined with the sparse Cholesky solver HSL\_MA87.  $m$  and  $n$  are the row and column dimensions of  $A$ ,  $m_{ds}$  is the number of dense rows that are stretched,  $m_{str}$  and  $n_{str}$  are the row and column dimensions of the stretched system. For strategy “none”,  $nnz$  is the number of entries in the normal matrix  $C = A^T A$  and for strategies “standard” and “sparse” it denotes the number of entries in the normal matrix for the stretched system  $C$  given by (25) (lower triangle).  $nnz(L)$  and  $nflops$  denote, respectively, the number of entries in the computed Cholesky factor and the flops used to compute it. NS indicates not solved since forming  $C$  requires too much memory. For each problem, the smallest  $nnz(L)$  and  $nflops$  across this Table and Table 5 are in bold.

Identifier	$m$	$n$	strategy	$m_{ds}$	$m_{str}$	$n_{str}$	$nnz$	$nnz(L)$	$nflops$
aircraft	7517	3754	none	0	7517	3754	$1.421 \times 10^6$	$1.421 \times 10^6$	$7.141 \times 10^8$
			standard	17	20267	16504	$5.474 \times 10^4$	<b><math>4.911 \times 10^5</math></b>	<b><math>1.759 \times 10^7</math></b>
			sparse	17	20267	16504	$5.474 \times 10^4$	<b><math>4.911 \times 10^5</math></b>	<b><math>1.759 \times 10^7</math></b>
lp_fit2p	13525	3000	none	0	13525	3000	$4.501 \times 10^6$	$4.501 \times 10^6$	$9.004 \times 10^9$
			standard	25	20284	39759	$1.500 \times 10^5$	<b><math>1.903 \times 10^6</math></b>	<b><math>1.203 \times 10^8</math></b>
			sparse	25	20284	39759	$1.500 \times 10^5$	<b><math>1.903 \times 10^6</math></b>	<b><math>1.203 \times 10^8</math></b>
sc205-2r	62423	35213	none	0	62423	35213	$6.510 \times 10^6$	$8.446 \times 10^7$	$2.314 \times 10^{11}$
			standard	8	75230	48020	$1.408 \times 10^5$	$1.853 \times 10^6$	$8.168 \times 10^7$
			sparse	8	75230	48020	$1.408 \times 10^5$	$1.863 \times 10^6$	$8.232 \times 10^7$
scagr7-2r	46679	32847	none	0	46679	32847	$2.215 \times 10^7$	$9.653 \times 10^7$	$4.067 \times 10^{11}$
			standard	7	59639	45807	$1.979 \times 10^6$	$9.876 \times 10^6$	$1.531 \times 10^{10}$
			sparse	7	59639	45807	$1.841 \times 10^5$	$1.559 \times 10^6$	$9.489 \times 10^7$
scrs8-2r	27691	14364	none	0	27691	14364	$6.217 \times 10^6$	$1.647 \times 10^7$	$2.901 \times 10^{10}$
			standard	22	40507	27180	$8.013 \times 10^5$	$2.983 \times 10^6$	$1.828 \times 10^9$
			sparse	22	40507	27180	$8.968 \times 10^4$	$1.238 \times 10^6$	$7.336 \times 10^7$
scsd8-2r	60550	8650	none	0	60550	8650	$1.957 \times 10^6$	$1.178 \times 10^7$	$1.894 \times 10^{10}$
			standard	50	82190	30290	$6.287 \times 10^5$	$5.182 \times 10^6$	$3.540 \times 10^9$
			sparse	50	82190	30290	$1.794 \times 10^5$	<b><math>1.978 \times 10^6</math></b>	<b><math>1.882 \times 10^8</math></b>
sctap1-2b	33858	15390	none	0	33858	15390	$2.638 \times 10^6$	$1.359 \times 10^7$	$1.645 \times 10^{10}$
			standard	34	45172	26704	$5.389 \times 10^5$	$4.705 \times 10^6$	$2.210 \times 10^9$
			sparse	34	45172	26704	$1.205 \times 10^5$	<b><math>1.544 \times 10^6</math></b>	<b><math>1.126 \times 10^8</math></b>
south31	36321	18425	none	0	36321	18425	$1.536 \times 10^8$	$1.581 \times 10^8$	$1.848 \times 10^{12}$
			standard	381	112398	94502	$3.224 \times 10^5$	$4.366 \times 10^6$	$2.546 \times 10^8$
			sparse	381	112398	94502	$3.224 \times 10^5$	$4.260 \times 10^6$	$2.319 \times 10^8$
PDE1	271792	270595	none	0	271792	270595	NS	NS	NS
			standard	1	362388	361191	$2.332 \times 10^6$	$4.295 \times 10^7$	$2.143 \times 10^{10}$
			sparse	1	362388	361191	$2.332 \times 10^6$	$3.836 \times 10^7$	$1.181 \times 10^{10}$

Table 5. Results for the sparse indefinite solver HSL\_MA97 applied to the augmented system (18) ( $K$ ) and reduced augmented system (19) ( $K_r$ ).  $m$  and  $n$  are the row and column dimensions of  $A$ ,  $nnz$  is the number of entries in  $K_r$  (lower triangle),  $nnz(L)$  and  $nflops$  denote, respectively, the number of entries in the  $L$  factor of  $K$  (and  $K_r$ ) and the flops used to compute it. NS indicates not solved because of insufficient memory for HSL\_MA97. For each problem, the smallest  $nnz(L)$  and  $nflops$  across this Table and Table 4 are in bold.

Identifier	$K$				$K_r$		
	$m$	$n$	$nnz(L)$	$nflops$	$nnz$	$nnz(L)$	$nflops$
aircraft	7517	3754	$4.903 \times 10^6$	$9.353 \times 10^9$	$1.653 \times 10^4$	$1.434 \times 10^6$	$7.245 \times 10^8$
lp_fit2p	13525	3000	$4.345 \times 10^6$	$7.879 \times 10^9$	$3.981 \times 10^4$	$4.116 \times 10^6$	$7.868 \times 10^8$
sc205-2r	62423	35213	$8.815 \times 10^5$	$9.398 \times 10^6$	$1.024 \times 10^5$	<b><math>3.441 \times 10^5</math></b>	<b><math>3.597 \times 10^6</math></b>
scagr7-2r	46679	32847	$8.008 \times 10^5$	$9.889 \times 10^6$	$1.288 \times 10^5$	<b><math>3.819 \times 10^5</math></b>	<b><math>5.077 \times 10^6</math></b>
srs8-2r	27691	14364	$6.415 \times 10^5$	$6.456 \times 10^7$	$4.924 \times 10^4$	<b><math>4.038 \times 10^5</math></b>	<b><math>6.114 \times 10^7</math></b>
scsd8-2r	60550	8650	$1.289 \times 10^6$	$5.204 \times 10^7$	$5.628 \times 10^5$	$3.616 \times 10^6$	$2.749 \times 10^9$
sectap1-2b	33858	15390	$1.864 \times 10^6$	$1.355 \times 10^9$	$8.054 \times 10^4$	$1.557 \times 10^6$	$1.292 \times 10^9$
south31	36321	18425	$3.525 \times 10^5$	$2.498 \times 10^6$	$9.526 \times 10^4$	<b><math>1.092 \times 10^5</math></b>	<b><math>7.671 \times 10^5</math></b>
PDE1	271792	270595	$3.593 \times 10^7$	$1.627 \times 10^{10}$	$1.970 \times 10^6$	<b><math>1.250 \times 10^7</math></b>	<b><math>2.650 \times 10^9</math></b>
12month1	872622	12471	NS	NS	$5.140 \times 10^7$	<b><math>7.300 \times 10^7</math></b>	<b><math>5.688 \times 10^{11}</math></b>

We reiterate that when  $n_{ds} > 1$  (see Table 1), null rows and columns in  $C_s$  lead to the break down of its Cholesky factorization and the block factorization approach of Section 3.1 cannot be used. However, we can employ a sparse indefinite solver to solve the augmented system (18) (or the reduced system (19)); results are presented in Table 5. We see that, in most cases, using the reduced system significantly reduces the size of the factors and the flop count compared to using the augmented system. Problem 12month1 illustrates that the augmented system can lead to the indefinite solver requiring more memory than is available. A key issue with (18) is that the pivot sequence chosen on the basis of the sparsity pattern of  $K$  generally requires significant modification during the factorization to maintain numerical stability and this leads to greater fill in the factors and a higher flop count than is predicted on the basis of the sparsity pattern alone. Comparing Tables 4 and 5, there are some problems (such as aircraft and scsd8-2r) for which sparse stretching leads to sparser factors and lower flop counts than using the reduced augmented system but for others (including south31, PDE1, and 12month1) the converse is true. This illustrates the importance of having more than one approach available. Note also that the indefinite solver HSL\_MA97 incorporates pivoting for numerical stability. This inhibits the use of parallelism compared with the Cholesky solver HSL\_MA87 and thus a comparison of sparsity and flop counts does not give a complete picture of how the solvers behave in practice, which will be dependent on the computing architecture.

**5.2.1 Results with a preconditioned iterative solver.** Having shown that using stretching with a direct solver can lead to significant benefits, we now consider its use combined with a preconditioned iterative solver. A number of freely available implementations of preconditioned LSQR and LSMR<sup>1</sup> are limited to using split preconditioning, that is, they require the preconditioner to be in the factorized form  $M = P^T P$ , where  $P$  is a square non-singular matrix that is chosen by the user so that  $AP^{-1}$  has more favourable spectral properties than  $A$ . The (approximate) inverse operator of Section 3.3 obtained from the block incomplete Cholesky factorization is symmetric positive definite but it cannot be expressed as a product  $P^T P$ . A discussion on how to use non-split preconditioning within this type of iterative solver has been recently given by Orban and Arioli [34] (see also Orban [32] and Benbow [7]). In particular, the preconditioning of LSQR and LSMR can be described as running the Golub-Kahan bidiagonalization in the  $M$  inner product because, if  $M$  is positive definite,  $M^{-1}A^T A$  is self-adjoint for the  $M$  inner product. With this replacement of the Euclidean inner product, the  $i$ -th step of the bidiagonalization becomes

<sup>1</sup>e.g., <http://web.stanford.edu/group/SOL/software/>

$$\begin{aligned} w &= Av_i - \alpha_i u_i; & \beta_{i+1} &= \|w\|; & u_{i+1} &= w/\beta_{i+1} \\ w &= M^{-1}A^T u_{i+1} - \beta_{i+1} v_i; & \alpha_{i+1} &= ((w, w)_M)^{1/2}; & v_{i+1} &= w/\alpha_{i+1}. \end{aligned}$$

Introducing a new vector,  $p = Mv_{i+1}$ , this can be rewritten as

$$\begin{aligned} w &= Av_i - \alpha_i u_i; & \beta_{i+1} &= \|w\|; & u_{i+1} &= w/\beta_{i+1} \\ p &= A^T u_{i+1} - \beta_{i+1} p \\ w &= M^{-1}p; & \alpha_{i+1} &= (w, p)^{1/2}; & v_{i+1} &= w/\alpha_{i+1}; & p &= p/\alpha_{i+1}. \end{aligned}$$

Our reported numerical experiments employ modified LSMR software that incorporates the use of  $M$  inner products.

Results for preconditioned LSMR applied to the stretched normal system (25) are presented in Table 6. Here none indicates no stretching is performed: an IC factorization of the normal matrix  $C$  (2) is computed using HSL\_MI35 and used to precondition LSMR applied to the original problem. When stretching is used, the total time ( $T_{Total}$ ) includes the time to perform the stretching as well as the time to compute the IC factorization preconditioner and to run LSMR. In these tests, when applying HSL\_MI35 to the original  $n \times n$  normal matrix  $C$  we set the parameter `lsize` (the maximum number of entries in each column of its IC factor) to 10. When HSL\_MI35 is applied to the stretched normal matrix  $C$ , we experimented with other settings because we found that for some examples a larger value of `lsize` was needed to obtain a preconditioner that gave convergence in a reasonable number of iterations. We see that while sparse stretching outperforms standard stretching, stretching generally results in more entries in the preconditioner (because the stretched system and `lsize` are larger) and often leads to higher iteration counts and a slower total solution time compared to solving the system without stretching. Problem `lp_fit2p` is omitted because, with values of `lsize` up to 150, we failed to obtain convergence (with no stretching and `lsize` set to 10, we needed 2913 iterations). For problem `sc205-2r`, stretching works well, with a very small iteration count; the slower total solution time is because of the overheads involved in identifying the dense rows and performing the stretching (note that our Fortran code to do the stretching is non trivial and optimising its performance to improve efficiency and reduce the total solution time is outside the scope of our current study). Our results suggest that sparse stretching is much less effective when used with an IC preconditioned iterative solver than it is when used with a direct solver.

## 6 PARTIAL STRETCHING

As already observed, a key issue with sparse stretching is that the stretched system can be much larger than the original system while the block signed Cholesky factorization (Section 3.1) breaks down if the reduced normal matrix  $C_s$  contains one or more null rows and columns. To address both problems, we propose a new stretching strategy that we refer to as *partial stretching*; it is outlined as Algorithm 4. The idea is to select a small subset of the rows of  $A_d$  that cause  $A_s$  to have null columns and to stretch just these rows, adding them to an enlarged sparse row block and moving the remaining dense rows to a block that has fewer rows than  $A_d$ . The result is a partially stretched matrix that is smaller than would result from stretching all the dense rows and, as it retains a number of dense rows, it can be solved using a block signed Cholesky factorization.

In the non trivial case  $m_d > 1$ , there are clearly a number of ways to choose the subset of rows in Step 2 of this algorithm. For example, a simple greedy procedure can be employed, in which the first dense row that has an entry in the first null column of  $A_s$  is selected first, and the process continues, selecting one dense row at a time, until coverage of all the null columns of  $A_s$  is achieved. In our experiments, we employ an alternative approach in which we perform an LU factorization with partial pivoting of  $A_d$  and terminate it after  $n_{ds}$  steps, where  $n_{ds}$  is the number of null columns in  $A_s$ . The column ordering used in this LU factorization is such that columns corresponding to the null columns in  $A_s$  are chosen first. The pivotal rows form the chosen subset. In Table 7, we present results for partial stretching when

Table 6. A comparison of the different stretching strategies when combined with preconditioned LSMR. None indicates no stretching is performed.  $nnz(\tilde{L})$  denotes the number of entries in the IC factor and  $iters$  is the number of LSMR iterations. NC indicates not solved since the number of iterations exceeded 2000; NS indicates not solved because of insufficient memory.  $T_{IC}$  and  $T_{Total}$  are, respectively, the time (in seconds) to compute the IC factorization and the total solution time.

Identifier	strategy	lsize	$nnz(\tilde{L})$	$iters$	$T_{IC}$	$T_{Total}$
aircraft	none	10	$4.124 \times 10^4$	18	0.07	0.12
	standard	50	$3.948 \times 10^5$	158	0.19	0.39
	sparse	50	$3.948 \times 10^5$	158	0.19	0.39
sc205-2r	none	10	$3.873 \times 10^3$	54	0.06	0.35
	standard	10	$5.281 \times 10^5$	5	0.04	0.79
	sparse	10	$5.280 \times 10^5$	7	0.04	0.75
scagr7-2r	none	10	$3.613 \times 10^5$	198	0.17	2.94
	standard	120	$5.535 \times 10^5$	NC	NC	NC
	sparse	120	$4.558 \times 10^5$	5	2.43	3.05
scsd8-2r	none	10	$9.130 \times 10^4$	116	0.05	0.35
	standard	100	$3.053 \times 10^6$	805	1.42	8.94
	sparse	30	$9.381 \times 10^5$	99	0.72	2.06
scrs8-2r	none	10	$1.579 \times 10^3$	368	0.05	0.80
	standard	50	$2.989 \times 10^5$	NC	NC	NC
	sparse	50	$2.989 \times 10^5$	27	0.06	0.45
sctap1-2b	none	10	$1.692 \times 10^3$	496	0.03	0.77
	standard	50	$1.359 \times 10^6$	NC	NC	NC
	sparse	50	$1.356 \times 10^6$	4	0.29	0.31
south31	none	10	$2.020 \times 10^3$	151	1.37	8.70
	standard	50	$4.800 \times 10^6$	1005	2.51	18.5
	sparse	50	$4.812 \times 10^6$	935	2.49	17.7
PDE1	none	10	NS	NS	NS	NS
	standard	100	$2.635 \times 10^6$	61	2.78	4.16
	sparse	100	$2.650 \times 10^6$	53	3.32	4.52
12month1	none	10	$1.367 \times 10^3$	205	15.7	33.1
	standard	100	$2.187 \times 10^6$	321	34.1	50.3
	sparse	100	$2.024 \times 10^6$	926	22.7	69.5

---

**Algorithm 4** Partial sparse stretching for a LS problem with dense rows.

---

**Require:**  $A \in R^{m \times n}$  with  $m \geq n$  split into row blocks  $A_s$  and  $A_d$  where  $A_s \in R^{m_s \times n}$  has one or more null columns and  $A_d \in R^{m_d \times n}$ .

**Ensure:** Partially stretched matrix  $\mathcal{A} \in R^{m\_str \times n\_str}$  with  $m\_str \geq n\_str$  split into row blocks  $\mathcal{A}_s$  and  $\mathcal{A}_d$  where  $\mathcal{A}_s$  has no null columns and the number of rows in  $\mathcal{A}_d$  is less than  $m_d$ .

- 1: Identify the null columns  $j_1, \dots, j_{n_{d_s}}$  in  $A_s$ .
  - 2: Find a small subset of at most  $n_{d_s}$  rows of  $A_d$  such that the block  $A_{d_s}$  comprising these rows has at least one entry in each column  $j_1, \dots, j_{n_{d_s}}$ .
  - 3: Apply sparse stretching to each row of  $A_{d_s}$ .
- 

used with the block approach (Section 3.1). Comparing these with those given in Table 4, we see that, as expected, the dimensions of the partially stretched system are significantly smaller and this, in turn, leads to far fewer entries in the Cholesky factor and a lower flop count. We note however, that it is still possible for the Cholesky factorization of  $C_s$  to break down (and this happens for problem scsd8-2r). This is because, although the sparse block  $\mathcal{A}_s$  of the partially stretched matrix has no null columns, it can still be rank deficient. For example, consider the matrix in Figure 2. It has full column rank, the last three rows comprise  $A_d$  and the first two columns of the sparse part  $A_s$  are null columns ( $n_{d_s} = 2$ ). If we perform two steps of an LU factorization with partial pivoting of  $A_d$  then the first two rows of  $A_d$  are selected and stretched. The resulting partially stretched sparse matrix  $\mathcal{A}_s$  still does not have full column rank. In such





where  $A_{d1}$  comprises the dense rows of  $A$  that are not stretched. The matrix for which an IC factorization computed is then

$$C_s = \mathcal{A}_s^T \mathcal{A}_s = \begin{pmatrix} C_s + FF^T & FS \\ S^T F^T & S^T S \end{pmatrix}.$$

The number of entries in each column of the IC factor is at most  $1s$  size thus, since  $C_s$  contains the smaller matrix  $C_s$ , its IC factor will generally have more entries than the IC factor of  $C_s$ , but more entries will also be dropped, which we may anticipate will lead to a poorer quality preconditioner. The poorer conditioning of the stretched system is also likely to adversely affect the iterative solver. A possible remedy could be to develop an IC factorization strategy that uses different dropping for entries in  $C_s + FF^T$  and  $S^T F^T$ , and is a possible future direction of study.

Table 8. Results for partial stretching combined with the block approach and preconditioned LSMR. Results are also given for the block approach and preconditioned LSMR with no stretching.  $n\text{nz}(\tilde{L})$  denotes the number of entries in the incomplete Cholesky factor of the normal matrix for the stretched system ( $1s$  size = 10).  $\textit{iters}$  is the number of LSMR iterations.

Identifier	Partial stretching with block approach		Block approach without stretching	
	$n\text{nz}(\tilde{L})$	$\textit{iters}$	$n\text{nz}(\tilde{L})$	$\textit{iters}$
aircraft	$1.875 \times 10^4$	1	$3.754 \times 10^3$	1
sc205-2r	$1.344 \times 10^5$	1	$9.602 \times 10^4$	2
scagr7-2r	$4.086 \times 10^5$	7	$1.472 \times 10^5$	3
scrs8-2r	$2.141 \times 10^5$	12	$3.434 \times 10^4$	3
scsd8-2r	$1.189 \times 10^5$	26	$7.432 \times 10^4$	2
south31	$1.884 \times 10^4$	1	$1.842 \times 10^4$	3

## 7 OTHER POSSIBLE STRETCHING APPROACHES

In this section, we propose other variants of sparse stretching with the aim of constructing more effective preconditioners.

### 7.1 Incomplete sparse stretching

We first consider a sparse stretching strategy that is based on dropping small entries from the dense block before applying stretching. The idea is to use only the largest entries in computing an incomplete factorization. Here we use the notation of Section 4.

LEMMA 7.1. *Let  $A = (A_s^T A_d^T)^T$  have one dense row  $A_d = f^T$ . Let  $\tilde{f}$  denote the vector that is obtained by dropping entries  $f_{s_1}, \dots, f_{s_p}$  ( $p \geq 1$ ) of  $f$ . Apply stretching to  $(A_s^T \tilde{f})^T$ , with  $\tilde{f}^T$  split into  $k \geq 2$  parts. Denote the stretched matrix by*

$$\tilde{\mathcal{A}} = \begin{pmatrix} A_s \\ \tilde{F}^T \\ S \end{pmatrix},$$

where  $S$  is the linking matrix. Then a stretched LS problem that is equivalent to the original LS problem is obtained by adding the scaled dropped entries  $\sqrt{k}f_{s_1}, \dots, \sqrt{k}f_{s_p}$  into the rows of  $\tilde{F}^T$  in an arbitrary way.

PROOF.  $\text{Struct}(f - \tilde{f})$  is disjoint from  $\text{Struct}(\tilde{f})$ . Thus, when  $\tilde{f}^T$  is stretched, there are no entries in  $\tilde{F}^T$  with row indices  $s_1, \dots, s_p$  (the indices of the entries that are dropped from  $f$ ). The  $k$  rows  $(\tilde{F}^T)_{q*}$  ( $1 \leq q \leq k$ ) are such that  $\text{Struct}((\tilde{F}^T)_{i*})$  is disjoint from  $\text{Struct}((\tilde{F}^T)_{j*})$  ( $i \neq j$ ). If the dropped entries are added into the rows of  $\tilde{F}^T$  in an arbitrary way, the modified  $\text{Struct}((\tilde{F}^T)_{i*})$  will remain disjoint from the modified  $\text{Struct}((\tilde{F}^T)_{j*})$  and, provided the

dropped entries are scaled in the same way as the entries of  $\tilde{f}^T$  are scaled during stretching, the result will be a correct scaling.  $\square$

The process of sparsification and matrix stretching is illustrated (without showing the scaling) using the following simple example. Here large entries of  $f^T$  are denoted by  $b$  and small entries by  $s$ .

$$\begin{pmatrix} * & * & * & * \\ * & * & & \\ & * & * & \\ b & b & b & s & b & s \end{pmatrix} \rightarrow \begin{pmatrix} * & * & * & * \\ * & * & & \\ & * & * & \\ b & b & b & b \end{pmatrix} \rightarrow \begin{pmatrix} * & * & * & * \\ * & * & & \\ & * & * & \\ b & b & & 1 \\ & b & & -1 \end{pmatrix} \rightarrow \begin{pmatrix} * & * & * & * \\ * & * & & \\ & * & * & \\ b & b & & 1 \\ & b & s & b & s & -1 \end{pmatrix}$$

The first matrix is the original matrix with one dense row and the second is after the small entries have been dropped from the dense row. The third matrix in the sequence is used to compute the preconditioner. The final matrix is a correct stretching of the original matrix (and it is to this matrix that the iterative method is applied).

It is clear that Lemma 7.1 can be generalised to  $m_d > 1$ . The approach, which we term *incomplete sparse stretching*, is described as Algorithm 5.

---

**Algorithm 5** Incomplete sparse stretching to solve a LS problem with dense rows

---

**Require:**  $A \in R^{m \times n}$  with  $m > n$  split into row blocks  $A_s$  and  $A_d$ , with  $A_s \in R^{m_s \times n}$  and  $A_d \in R^{m_d \times n}$ .

**Ensure:** The solution  $x$  of the LS problem.

- 1: Sparsify the dense row block  $A_d$  by dropping small entries. Let the result be  $\tilde{A}_d$ .
  - 2: Apply stretching to  $(A_s^T \tilde{A}_d^T)^T$ . Let the stretched matrix be  $\mathcal{A}$ .
  - 3: Form  $C = \mathcal{A}^T \mathcal{A}$  and compute its incomplete Cholesky factorization.
  - 4: Add the dropped entries into the stretched matrix  $\mathcal{A}^T$  (as in Lemma 7.1).
  - 5: Compute the LS solution by solving the stretched problem using an iterative method with the incomplete Cholesky factorization as the preconditioner.
- 

We illustrate the incomplete sparse stretching approach using test example `lp_fit2p` (which has 25 dense rows). Recall from Section 5.2.1 that this is a tough problem for iterative solvers: using stretching and then employing preconditioned LSMR, we failed to achieve convergence. In our experiments, we take the sparse block  $A_s$  and take  $m_d < 25$  of its dense rows as the block  $A_d$  (the remaining dense rows are discarded thus we are using a modified problem in our tests). Row  $i$  of  $\tilde{A}_d$  is computed by dropping entries in row  $i$  of  $A_d$  that have absolute value less than

$$drops \times \max_j |A_d(i, j)|.$$

Sparse stretching is applied to the resulting sparsified rows. In Table 9, we report on a range of values for the parameter *drops*. Setting *drops* = 0 corresponds to sparse stretching while *drops* = 1 implies only the largest entries in each dense row are retained (aggressive dropping). We see that, if there is a small number of dense rows ( $m_d = 3$  or 5), sparse stretching (*drops* = 0.0) works well but we struggle once the number of dense rows becomes larger and the problem becomes tougher to solve. However, aggressive dropping allows us to compute the solution using a only modest number of iterations and a sparse preconditioner.

Table 9. Results for problem `lp_fit2p`. Incomplete sparse stretching is applied to  $m_d = 3, 5,$  and 10 of dense rows with increasing values of the parameter `drops`. The row and column dimensions of the stretched matrix are denoted by  $m\_str$  and  $n\_str$ . `iters` is the iteration count and  $nnz(\tilde{L})$  is the number of entries in the incomplete Cholesky factor of the stretched normal matrix (`lsize = 5`). NC indicates the limit of 2000 iterations is exceeded.

<i>drops</i>	$m_d = 3$				$m_d = 5$				$m_d = 10$			
	<i>iters</i>	$m\_str$	$n\_str$	$nnz(\tilde{L})$	<i>iters</i>	$m\_str$	$n\_str$	$nnz(\tilde{L})$	<i>iters</i>	$m\_str$	$n\_str$	$nnz(\tilde{L})$
0.0	2	22421	11918	56580	12	26926	16241	98507	NC	36114	25604	153603
0.01	17	21699	11166	52818	263	25830	15325	91930	NC	35018	24508	147027
0.02	23	20802	10299	48499	429	24645	14140	84820	NC	32767	22257	133521
0.1	NC	17721	7218	32787	NC	20376	9871	59094	NC	36114	15822	153603
0.6	1386	16570	6067	32429	NC	18927	8422	49610	NC	22581	12071	72410
0.7	906	16522	6019	33036	151	18912	8407	49884	NC	22564	12054	72308
0.8	118	16508	6005	19678	44	18908	8403	31233	561	22560	12050	70910
0.9	34	16504	6001	15403	19	18906	8401	30482	32	22558	12048	70149
1.0	13	16502	5999	14995	19	18906	8401	30482	32	22558	12048	70149

## 7.2 Weighted sparse stretching

As discussed in Section 4, computing a stretching involves finding index sets so that the sparsity structure of each row of the stretched dense block  $F^T$  is covered by the sparsity structures of one or more rows of  $A_s$ . This is closely related to the problem of minimizing a set cover. In [44], we used a standard greedy approach to obtain such a cover. To motivate our proposed weighted sparse stretching, consider a simple  $6 \times 5$  matrix with full column rank and one dense row

$$A = \begin{pmatrix} A_s \\ A_d \end{pmatrix} = \begin{pmatrix} 1 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 1 & 1 & 0 & 0 & 0 \\ 5 & 6 & 0 & 0 & 0 \\ 0 & 0 & 7 & 8 & 9 \\ 3/\sqrt{2} & 3/\sqrt{2} & 3/\sqrt{2} & 3/\sqrt{2} & 3/\sqrt{2} \end{pmatrix}. \quad (27)$$

The matrix on the left of (28) depicts the result of using the greedy approach to the set cover problem to split the last row. An alternative splitting is given on the right.

$$\begin{pmatrix} A_s \\ F^T \end{pmatrix} = \begin{pmatrix} 1 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 1 & 1 & 0 & 0 & 0 \\ 5 & 6 & 0 & 0 & 0 \\ 0 & 0 & 7 & 8 & 9 \\ 3 & 3 & 3 & 3 & 0 \\ 0 & 0 & 0 & 0 & 3 \end{pmatrix} \quad \text{or} \quad \begin{pmatrix} 1 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 1 & 1 & 0 & 0 & 0 \\ 5 & 6 & 0 & 0 & 0 \\ 0 & 0 & 7 & 8 & 9 \\ 3 & 3 & 0 & 0 & 0 \\ 0 & 0 & 3 & 3 & 3 \end{pmatrix} \quad (28)$$

For these two splittings, the matrix block  $A_s^T A_s + FF^T$  from (25) is as follows

$$A_s^T A_s + FF^T = \begin{pmatrix} 36 & 41 & 10 & 10 & 0 \\ 41 & 47 & 10 & 10 & 0 \\ 10 & 10 & 59 & 66 & 63 \\ 10 & 10 & 66 & 64 & 72 \\ 0 & 0 & 63 & 72 & 91 \end{pmatrix} \quad \text{or} \quad \begin{pmatrix} 36 & 41 & 1 & 1 & 0 \\ 41 & 47 & 1 & 1 & 0 \\ 1 & 1 & 59 & 66 & 72 \\ 1 & 1 & 66 & 74 & 81 \\ 0 & 0 & 72 & 81 & 91 \end{pmatrix}.$$

The right-hand splitting leads to blocks on the diagonal that dominate the off-diagonal blocks and this is likely to lead to higher quality incomplete factorization preconditioners.

Subsets of  $Struct(f)$  such that each row of  $F^T$  is dominated by rows of  $A_s$  can be found using the *bipartite row intersection graph* of  $A_s$  and  $f^T$ , which is defined as follows.

*Definition 7.2.* Denote the  $i$ -th row of  $A_s$  by  $(A_s)_{i*}$  ( $1 \leq i \leq m_s$ ) and let  $Struct(f) = \{j_1, \dots, j_r\}$  where  $r = |Struct(f)|$ . The bipartite graph  $G = (R, B, E)$  with vertex sets  $R = \{1, \dots, m_s\}$  and  $B = \{j_1, \dots, j_r\}$  and edges given by

$$(i, j) \in E \iff j \in Struct((A_s)_{i*}).$$

Thus a standard set cover problem needs to be solved. Once solved, the sets that represent rows of  $F^T$  are made disjoint (see [44]). Based on our observation regarding the potential importance of the magnitudes of the entries, we propose the following approximate algorithm that represents one possible way to solve the weighted set cover problem (see [48]); using the weighted minimum set cover results in our weighted sparse stretching approach.

---

**Algorithm 6** Finding an approximate weighted minimum set cover

---

**Require:**  $A = \{A(i, j)\} \in R^{m \times n}$  with  $m \geq n$  split into row blocks  $A_s$  and  $A_d$ , where  $A_s \in R^{m_s \times n}$  and  $A_d \equiv f^T \in R^{1 \times n}$ .

**Ensure:**  $k$  and an approximate weighted minimum set cover of  $Struct(f)$  characterized by the rows  $r_1, r_2, \dots, r_k$  of  $A_s$  such that  $Struct(f) = \bigcup_{1 \leq i \leq k} Struct(r_i)$ .

- 1: Construct the bipartite row intersection graph  $G = (R, B, E)$  of  $A_s$  and  $f^T$ .
  - 2: Initialise the row weights  $w(r) = \sum_{j \in B} |A(r, j)|$ ,  $r \in R$ .
  - 3: Set  $i = 0$ .
  - 4: **while** there exists  $r_i \in R$  that maximises  $w(l) > 0$ ,  $l \in R$  **do**
  - 5:     Set  $i = i + 1$ .
  - 6:     **for**  $k \in Struct(r_i)$  **do**
  - 7:         **for**  $r \in R$  such that  $k \in Struct(r)$  **do**
  - 8:             Update  $w(r) = w(r) - |A(r, k)|$
  - 9:         **end do**
  - 10:     **end do**
  - 11: **end do**
  - 12: Set  $k = i$ .
- 

To illustrate the potential benefits of weighted sparse stretching, we consider the test example `lp_agg` ( $m = 615$ ,  $n = 488$ ) to which we append up to 35 dense rows with  $n$  entries in each of these rows (this small example is one that we chose to explore in some detail in our original study of sparse stretching [44]). Each appended row is stretched using the weighted and non-weighted (i.e. greedy) set cover algorithms. In Figure 3, iteration counts are given for the stretched problem with the `HSL_MI35` parameter `lsize` that controls the number of entries in each column of the IC factor set to 50 and to 60. As the number of dense rows increases, the problem becomes harder to solve. However, the number of iterations is significantly reduced if the weighted set cover is used, with the relative savings in the iteration counts increasing with the number of stretched rows. Furthermore, increasing the number of entries in the incomplete factorization (that is, increasing `lsize`) can lead to significantly faster convergence.

### 7.3 Other possible preconditioners

Another approach to developing preconditioners for LS problems with some dense rows is to explicitly exploit the saddle-point block structure of the stretched normal matrix  $C$ ; see, for example, the block and constraint preconditioners

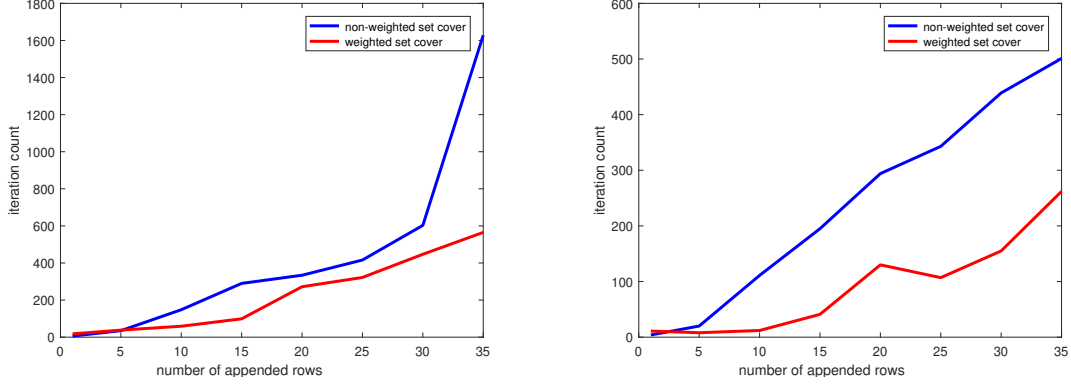


Fig. 3. Iteration counts for problem `lp_agg` as the number of appended dense rows increases. Results are for sparse stretching using the weighted and non-weighted vertex set cover with `lsize = 50` (left) and `lsize = 60` (right).

presented in [28, 29], as well as the survey paper [8]. A detailed study is beyond the scope of this paper; instead we briefly introduce one possible preconditioner and illustrate that it can give improved performance (in terms of the size of the preconditioner and/or the number of iterations of the preconditioned iterative solver required for convergence). This preconditioner results from computing an IC factorization of the following matrix:

$$P_1 = \begin{pmatrix} \text{diag}(C_s) & FS \\ S^T F^T & S^T S \end{pmatrix}.$$

If we compare this with the stretched system (25), we see that the term  $FF^T$  has been dropped from the (1, 1) block. To illustrate the potential of  $P_1$ , we again use problem `lp_agg` to which we append up to 50 dense rows, each with density of 10%. We compare the performance of  $P_1$  with that of  $P_0$ , where  $P_0$  is the preconditioner obtained by computing an IC factorization of the stretched normal matrix  $C$  given in (25). In Figure 4, we plot the iteration counts and preconditioner size as the number of dense rows increases. We see that while  $P_0$  requires (slightly) fewer iterations, it is denser than  $P_1$ , particularly when the number of dense rows is small.

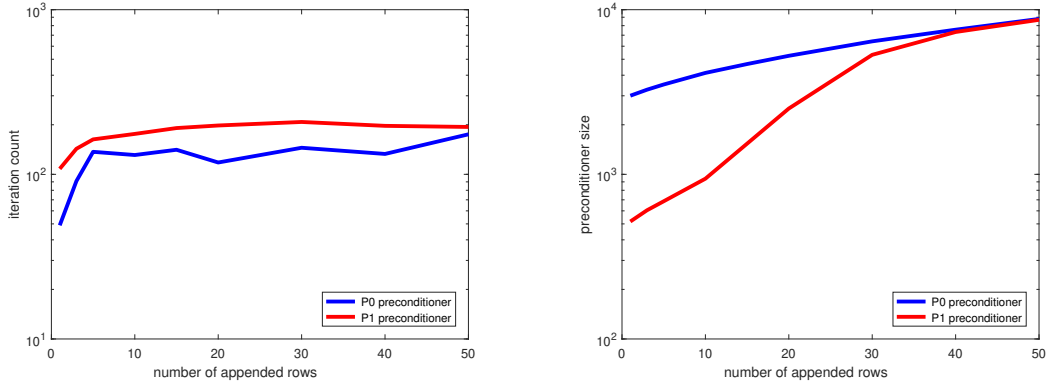


Fig. 4. Iteration counts (left) and preconditioner size (right) for  $P_0$  and  $P_1$  applied to problem `lp_agg` as the number appended dense rows increases. Here `lsize = 5`.

## 8 CONCLUDING REMARKS AND FUTURE DIRECTIONS

It is well known that the presence of one or more dense rows makes solving a LS problem much harder. In an earlier paper [44], we proposed a new matrix stretching strategy (sparse stretching) as a means of sparsifying the dense rows, at the cost of increasing both the row and column dimensions of the LS system. Encouraging preliminary results motivated us to explore further the effectiveness of stretching when used with either a sparse direct solver or a preconditioned iterative solver to efficiently tackle large-scale LS problems that include a small number of dense rows.

Our experiments have shown that sparse stretching can be used with a sparse Cholesky package to significantly reduce the size of the Cholesky factor and the flop count needed to compute it. However, as the number of dense rows (and/or the density of these rows) increases, the growth in the dimensions of the LS problem are substantial and, for very large problems, the memory requirements of the direct solver will ultimately limit the size of the original system that can be solved using sparse stretching. This suggests either limiting the increase in the problem dimensions or employing a preconditioned iterative scheme. To achieve the former, we proposed partial stretching, which only stretches a small number of rows so as to avoid null columns in  $A_s$ .

With regards using an iterative solver, our numerical results suggest that our incomplete factorization preconditioner is not always as effective as we would like for the stretched normal equations. This led us to propose other stretching strategies, namely a so-called incomplete stretching strategy and weighted sparse stretching, both of which take into account not only the sparsity patterns of the row block  $A_s$  and  $A_d$  but also the numerical values. Initial experiments illustrate that these approaches may be beneficial and can lead to either higher quality preconditioners and/or sparser preconditioners. Nevertheless, we feel there is scope to improve performance further and thus an area of future research is the development of other preconditioners to be employed in conjunction with sparse stretching strategies. In particular, we are interested in studying preconditioners that aim to exploit the block structure of the stretched normal matrix. Access to a wider range of large-scale practical problems would be beneficial for this.

Our longer term objective is the development of robust and efficient software for solving large-scale linear LS problems, incorporating direct and iterative methods, that automatically detects and handles dense rows and addresses other issues such as rank deficiency that can make solving these systems difficult. Our study of stretching techniques is one step towards realising this goal.

## ACKNOWLEDGEMENTS

We would like to thank two anonymous reviewers for their careful reading of our manuscript and for their constructive comments that led to important improvements in the presentation of this paper.

## REFERENCES

- [1] M. Adlers. *Topics in Sparse Least Squares Problems*. PhD thesis, Department of Mathematics, Linköpings Universitet, SE-581 83 Linköping, Sweden, available as Linköping Studies in Science and Technology. Dissertations No. 634, 2000.
- [2] M. Adlers and Å. Björck. Matrix stretching for sparse least squares problems. *Numerical Linear Algebra with Applications*, 7(2):51–65, 2000.
- [3] F. L. Alvarado. Matrix enlarging methods and their application. *BIT Numerical Mathematics*, 37(3):473–505, 1997.
- [4] K. D. Andersen. A modified Schur complement method for handling dense columns in interior point methods for linear programming. *ACM Transactions on Mathematical Software*, 22(3):348–356, 1996.
- [5] H. Avron, E. Ng, and S. Toledo. Using perturbed  $QR$  factorizations to solve linear least-squares problems. *SIAM J. on Matrix Analysis and Applications*, 31(2):674–693, 2009.
- [6] C. Aykanat, A. Pinar, and Ü. V. Çatalyürek. Permuting sparse rectangular matrices into singly-bordered block-diagonal form for parallel solution of LP problems. Technical Report BU-CE-0203, Computer Engineering Department, Bilkent Univeristy, Ankara, Turkey, 2002.
- [7] S. J. Benbow. Solving generalized least-squares problems with LSQR. *SIAM J. on Matrix Analysis and Applications*, 21:166–177, 1999.
- [8] M. Benzi, G.H. Golub, and J. Liesen. Numerical solution of saddle point problems. *Acta Numerica*, 14:1–137, 2005.

- [9] Å. Björck. *Numerical Methods for Least Squares Problems*. SIAM, Philadelphia, 1996.
- [10] I. S. Duff and J. A. Scott. Stabilized bordered block diagonal forms for parallel sparse solvers. *Parallel Computing*, 31:275–289, 2005.
- [11] M. C. Ferris and J. D. Horn. Partitioning mathematical programs for parallel solution. *Mathematical Programming*, 80:35–62, 1998.
- [12] D. C.-L. Fong and M. A. Saunders. LSMR: An iterative algorithm for sparse least-squares problems. *SIAM J. on Scientific Computing*, 33(5):2950–2971, 2011.
- [13] A. George and M. T. Heath. Solution of sparse linear least squares problems using Givens rotations. *Linear Algebra and its Applications*, 34:69–83, 1980.
- [14] P. E. Gill, W. Murray, D. B. Ponceleon, and M. A. Saunders. Solving reduced KKT systems in barrier methods for linear and quadratic programming. Technical Report SOL 91-7, Department of Operations Research, Stanford University, 1991.
- [15] D. Goldfarb and K. Scheinberg. A product-form Cholesky factorization method for handling dense columns in interior point methods for linear programming. *Mathematical Programming, Series A*, 99:1–34, 2004.
- [16] J. Gondzio. Splitting dense columns of constraint matrix in interior point methods for large scale linear programming. *Optimization*, 24(3-4):285–297, 1992.
- [17] N. I. M. Gould and J. A. Scott. The state-of-the-art of preconditioners for sparse linear least squares problems. *ACM Transactions on Mathematical Software*, 43(4):36:1–35, 2017.
- [18] J. F. Grcar. Matrix stretching for linear equations. Technical Report SAND90-8723, Sandia National Laboratories, 1990.
- [19] C. Greif, S. He, and P. Liu. SYM-ILDL: C++ package for incomplete factorizations of symmetric indefinite matrices. <https://github.com/inutard/matrix-factor>, 2013.
- [20] L. Guttman. Enlargement methods for computing the inverse matrix. *Annals of Mathematical Statistics*, 17:336–343, 1946.
- [21] W. W. Hager. Updating the inverse of a matrix. *SIAM Review*, 31(2):221–239, 1989.
- [22] M. T. Heath. Some extensions of an algorithm for sparse linear least squares problems. *SIAM J. on Scientific and Statistical Computing*, 3(2):223–237, 1982.
- [23] M. R. Hestenes and E. Stiefel. Methods of conjugate gradients for solving linear systems. *J. of Research of the National Bureau of Standards*, 49:409–435, 1952.
- [24] J. D. Hogg, J. K. Reid, and J. A. Scott. Design of a multicore sparse Cholesky factorization using DAGs. *SIAM J. on Scientific Computing*, 32:3627–3649, 2010.
- [25] J. D. Hogg and J. A. Scott. HSL\_MA97: a bit-compatible multifrontal code for sparse symmetric systems. Technical Report RAL-TR-2011-024, Rutherford Appleton Laboratory, 2011.
- [26] J. D. Hogg and J. A. Scott. New parallel sparse direct solvers for multicore architectures. *Algorithms*, 6:702–725, 2013. Special issue: Algorithms for Multi Core Parallel Computation.
- [27] HSL. A collection of Fortran codes for large-scale scientific computation, 2018. <http://www.hsl.rl.ac.uk>.
- [28] C. Keller, N. I. M. Gould, and A. J. Wathen. Constraint preconditioning for indefinite linear systems. *SIAM J. on Matrix Analysis and Applications*, 21(4):1300–1317, 2000.
- [29] L. Lukšan and J. Vlček. Computational experience with globally convergent descent methods for large sparse systems of nonlinear equations. *Optim. Methods Softw.*, 8(3-4):201–223, 1998.
- [30] A. Marxen. Primal barrier methods for linear programming. Technical Report SOL 89-6, Department of Operations Research, Stanford University, 1989.
- [31] C. Meszaros. Detecting dense columns in interior point methods for linear programs. *Computational Optimization and Applications*, 36:309–320, 2007.
- [32] D. Orban. The projected Golub-Kahan process for constrained least-squares. GERAD Technical Report G-2014-15, 2014.
- [33] D. Orban. Limited-memory  $LDL^T$  factorization of symmetric quasi-definite matrices with application to constrained optimization. *Numerical Algorithms*, 70(1):9–41, 2015.
- [34] D. Orban and M. Arioli. *Iterative solution of symmetric quasi-definite linear systems*, volume 3 of *SIAM Spotlights*. Society for Industrial and Applied Mathematics (SIAM), Philadelphia, PA, 2017.
- [35] C. C. Paige and M. A. Saunders. Solution of sparse indefinite systems of linear equations. *SIAM J. on Numerical Analysis*, 12(4):617–629, 1975.
- [36] C. C. Paige and M. A. Saunders. LSQR: An algorithm for sparse linear equations and sparse least squares. *ACM Transactions on Mathematical Software*, 8(1):43–71, 1982.
- [37] W. Ren and J. Zhao. Iterative methods with preconditioners for indefinite systems. *J. of Computational Mathematics*, 17:89–96, 1999.
- [38] M. A. Saunders. Cholesky-based methods for sparse least squares: The benefits of regularization. Technical Report SOL 95-1, Department of Operations Research, Stanford University, 1995. In L. Adams and J. L. Nazareth (eds.), *Linear and Nonlinear Conjugate Gradient-Related Methods*, SIAM, Philadelphia, 92–100 (1996).
- [39] J. A. Scott and M. Tuma. HSL\_MI28: an efficient and robust limited-memory incomplete Cholesky factorization code. *ACM Transactions on Mathematical Software*, 40(4):24:1–19, 2014.
- [40] J. A. Scott and M. Tuma. On positive semidefinite modification schemes for incomplete Cholesky factorization. *SIAM J. on Scientific Computing*, 36(2):A609–A633, 2014.



- [41] J. A. Scott and M. Tuma. On signed incomplete Cholesky factorization preconditioners for saddle-point systems. *SIAM J. on Scientific Computing*, 36(6):A2984–A3010, 2014.
- [42] J. A. Scott and M. Tuma. Solving mixed sparse-dense linear least-squares problems by preconditioned iterative methods. *SIAM J. on Scientific Computing*, 39(6):A2422–A2437, 2017.
- [43] J. A. Scott and M. Tuma. A Schur complement approach to preconditioning sparse least-squares problems with some dense rows. *Numerical Linear Algebra with Applications*, 79:1147–1168, 2018. DOI: 10.1007/s11075-018-0478-2.
- [44] J. A. Scott and M. Tuma. Sparse stretching for solving sparse-dense linear least-squares problems. *SIAM J. on Scientific Computing*, 41:A1604–A1625, 2019.
- [45] C. Sun. Dealing with dense rows in the solution of sparse linear least squares problems. Research Report CTC95TR227, Advanced Computing Research Institute, Cornell Theory Center, Cornell University, 1995.
- [46] C. Sun. Parallel solution of sparse linear least squares problems on distributed-memory multiprocessors. *Parallel Computing*, 23(13):2075–2093, 1997.
- [47] R. J. Vanderbei. Splitting dense columns in sparse linear systems. *Linear Algebra and its Applications*, 152:107–117, 1991.
- [48] V. V. Vazirani. *Approximation Algorithms*. Springer-Verlag, Berlin, 2001.
- [49] M. A. Woodbury. *The Stability of Out-Input Matrices*. Chicago, Ill., 1949.
- [50] M. A. Woodbury. *Inverting modified matrices*. Statistical Research Group, Memo. Rep. no. 42. Princeton University, Princeton, N. J., 1950.