Dealing with Dense Rows in the Solution of Sparse Linear Least Squares Problems[1]

Chunguang Sun
Advanced Computing Research Institute
Cornell Theory Center
Cornell University
Ithaca, NY 14853-3801
csun@cs.cornell.edu

Tech. Report CTC95TR227, Advanced Computing Research Institute, Cornell Theory Center, Cornell University, Ithaca, NY, May 1995.

Abstract

Sparse linear least squares problems containing a few relatively dense rows occur frequently in practice. Straightforward solution of these problems could cause catastrophic fill and delivers extremely poor performance. This paper studies a scheme for solving such problems efficiently by handling dense rows and sparse rows separately. How a sparse matrix is partitioned into dense rows and sparse rows determines the efficiency of the overall solution process. A new algorithm is proposed to find a partition of a sparse matrix which leads to satisfactory or even optimal performance. Extensive numerical experiments are performed to demonstrate the effectiveness of the proposed scheme. A MATLAB implementation is included.

[1]This work was supported in part by the Cornell Theory Center which receives funding from members of its Corporate Research Institute, the National Science Foundation (NSF), the Advanced Research Projects Agency (ARPA), the National Institutes of Health (NIH), New York State, and IBM corporation.

Dealing with Dense Rows in the Solution of Sparse Linear Least Squares Problems [†]

Chunguang Sun [‡]

October 25, 1995

Abstract

Sparse linear least squares problems containing a few relatively dense rows occur frequently in practice. Straightforward solution of these problems could cause catastrophic fill and delivers extremely poor performance. This paper studies a scheme for solving such problems efficiently by handling dense rows and sparse rows separately. How a sparse matrix is partitioned into dense rows and sparse rows determines the efficiency of the overall solution process. A new algorithm is proposed to find a partition of a sparse matrix which leads to satisfactory or even optimal performance. Extensive numerical experiments are performed to demonstrate the effectiveness of the proposed scheme. A MATLAB implementation is included.

[†]This work was supported in part by the Cornell Theory Center which receives funding from members of its Corporate Research Institute, the National Science Foundation (NSF), the Advanced Research Projects Agency (ARPA), the National Institutes of Health (NIH), New York State, and IBM corporation.

[‡]Advanced Computing Research Institute, Cornell Theory Center, Cornell University, Ithaca, NY 14853-3801(`e-mail:csun@cs.cornell.edu`, `phone:607-254-8863`, `fax:607-254-8888`).

1 Introduction

The numerical solution of linear least squares problems

$$\min_x \|Ax - b\|_2 \tag{1}$$

lies at the heart of many challenging computational problems frequently arising in geodetic survey, photogrammmetry, tomography, structural analysis, surface fitting, numerical optimization, etc. Let A be an $m \times n$ matrix with full column rank. A well-known direct method [8] for solving (1) first computes the QR factorization

$$A = Q \begin{bmatrix} R \\ 0 \end{bmatrix}, \tag{2}$$

where Q is an $m \times m$ orthogonal matrix and R is an $n \times n$ upper triangular matrix. It then solves the triangular system $Rx = c$, where c is a vector containing the first n entries of $Q^T b$. The matrix R is referred to as the *triangular factor* of A. This method is numerically backward stable.

In practical applications, the matrix A is often very large and sparse. In order to solve (1) efficiently, the sparsity of A must be exploited. Extensive research has been done on the direct solution of sparse linear least squares problems [1, 6, 9, 10, 11, 12, 13]. Typically, a direct method for solving (1) involves the following steps:

1. Ordering: Find a permutation matrix P so that AP has a sparse triangular factor R.

2. Symbolic factorization: Determine the symbolic structure of R.

3. Numeric factorization: Compute the numerical values of the nonzeros of R.

4. Triangular solution: Solve the associated triangular system.

Steps 1 and 2 constitute the symbolic phase of the overall process for solving (1), and steps 3 and 4 form the corresponding numeric phase. Since A has full column rank, $A^T A$ is symmetric and positive definite. Let L be the Cholesky factor of $A^T A$. Then L^T is equal to the triangular factor of A, apart from possible sign differences in the rows. Therefore, the ordering algorithms such as minimum degree ordering [7] and nested dissection ordering [3] developed for sparse Cholesky factorization can be applied to $A^T A$ to obtain a sparse triangular factor R. Similarly, the symbolic factorization algorithm for sparse Cholesky factorization described in [5] can be applied to $A^T A$ to predict the symbolic structure of R prior to numeric factorization. This approach is initially proposed in [4] and is now widely used for implementing the symbolic phase of the overall process for solving sparse linear least squares problems.

In practice, a sparse linear least squares problem frequently contains relatively dense rows. Therefore, the corresponding triangular factor R becomes nearly or completely full. For instance, consider a 4×3 matrix with the following sparsity pattern:

$$A = \begin{bmatrix} \times & & \\ & \times & \\ & & \times \\ \times & \times & \times \end{bmatrix}, \tag{3}$$

where a "$\times$" indicates a nonzero entry and a blank space indicates a zero entry. Barring accidental numerical cancellation, $A^T A$ is completely full even though A is very sparse except for one dense row. Therefore, the triangular factor R of A is also completely full. This phenomenon is referred to as *catastrophic fill*. When this happens, we do not get any benefit by employing sparse matrix techniques. Hence the straightforward application of the conventional method is not a viable approach to the solution of sparse linear least squares problems containing relatively dense rows,

In this paper, we propose a new algorithm for solving sparse linear least squares problems containing relatively dense rows. Our approach divides rows of A into sparse rows and dense rows and handles the sparse and dense rows separately. However, whether a row is considered dense or sparse depends on the problem at hand. How to partition A into dense and sparse rows is a crucial and difficult issue which we shall carefully address. We present an efficient algorithm that produces satisfactory partition of a sparse matrix into dense and sparse rows. Extensive numerical experiments show that our overall approach is much faster than the conventional method.

In §2, we review a previously-proposed algorithm for updating the solution to a sparse linear least squares problem. A new algorithm is proposed in §3 for solving sparse linear least squares problems containing relatively dense rows. Experimental results are discussed in §4. Concluding remarks are contained in §5. Finally, a MATLAB implementation of our overall approach is provided in the **Appendix**.

2 A Solution Updating Algorithm

In this section, we briefly describe an algorithm proposed in [4] for updating the solution to a sparse linear least squares problem. Let A_1 be an $m_1 \times n$ sparse matrix with full column rank, and A_2 an $m_2 \times n$ matrix. Suppose that the solution to the following sparse linear least squares problem

$$\min_x \|A_1 x - b_1\|_2 \tag{4}$$

has already been determined. We wish to solve the following augmented problem

$$\min_x \left\| \begin{bmatrix} A_1 \\ A_2 \end{bmatrix} x - \begin{bmatrix} b_1 \\ b_2 \end{bmatrix} \right\|_2 . \tag{5}$$

Let x_0 be the solution to the original problem (4) which is calculated through the QR factorization

$$A_1 P = Q \begin{bmatrix} R \\ 0 \end{bmatrix}, \tag{6}$$

where P is an $n \times n$ permutation matrix, Q is an $m_1 \times m_1$ orthogonal matrix, and R is an $n \times n$ upper triangular matrix. Let x be the solution to the augmented problem (5). Then $z = x - x_0$ can be computed by the following algorithm:

- Solve the sparse triangular systems $R^T E = (A_2 P)^T$.
- Let $\begin{bmatrix} E \\ I \end{bmatrix} = Q_2 R_2$ be the QR factorization of the matrix $\begin{bmatrix} E \\ I \end{bmatrix}$, where I is the $m_2 \times m_2$ identity matrix, Q_2 is an $(n + m_2) \times m_2$ matrix with orthonormal columns, and R_2 is an $m_2 \times m_2$ upper triangular matrix.
- Let u be a vector containing the first n entries of the vector $Q_2(R_2^{-T}(b_2 - A_2 y))$. Solve the sparse triangular system $Rv = u$ and set $z = Pv$.

We refer the reader to [4] for a proof of the correctness of this algorithm.

3 A Partitioning Algorithm

In this section, we consider the solution of a sparse linear least squares problem

$$\min_x \|Ax - b\|_2, \tag{7}$$

where A is an $m \times n$ sparse matrix with full column rank and A may contain some relatively dense rows. The solution updating algorithm described in §2 shows that it is possible to process the sparse rows and the dense rows of A separately. Suppose that A and b are partitioned as

$$\tilde{A} = P_r A = \begin{bmatrix} A_1 \\ A_2 \end{bmatrix} \quad \text{and} \quad \tilde{b} = P_r b = \begin{bmatrix} b_1 \\ b_2 \end{bmatrix}, \tag{8}$$

where P_r is an $m \times m$ permutation matrix, A_1 is a $k \times n (k \geq n)$ matrix containing sparse rows of A, and A_2 is an $(m - k) \times n$ matrix containing dense rows of A. Our objective is to solve

$$\min_x \|Ax - b\|_2 = \min_x \|P_r A x - P_r b\|_2 = \min_x \left\| \begin{bmatrix} A_1 \\ A_2 \end{bmatrix} x - \begin{bmatrix} b_1 \\ b_2 \end{bmatrix} \right\|_2. \tag{9}$$

Note that a partition of A as shown in (8) is valid if and only if A_1 has at least as many rows as columns. Once a valid partition of A is found, the solution updating algorithm discussed in §2 can be used to obtain the solution to (7). In this way, possible catastrophic fill is avoided since the QR factorization of A_1 is computed. However, finding a satisfactory partition for A is by no means trivial. Let x_0 be the solution to $\|A_1x - b_1\|$. As shown in §2, the inverse of the upper triangular matrix R is used repetitively in the update of the solution x_0 by A_2 and b_2. Hence A_1 must have full column rank. Furthermore, if R is badly conditioned, the process of updating the solution x_0 by A_2 and b_2 is numerically unstable. Therefore, the condition number of R, denoted by $\kappa(R)$, cannot be too large. In practice, $\kappa(R)$ is required to satisfy a prescribed tolerance value, denoted by τ.

Let T_1 denote the time for partitioning A and b and for solving $\min_x \|A_1x - b_1\|_2$, T_2 the time for updating the solution x_0 by A_2 and b_2, and T the time for solving the original problem $\min_x \|Ax - b\|_2$ directly. A partition of matrix A is deemed satisfactory if it is a valid partition and meets the following two criteria:

- $T_1 + T_2 < T$.
- $\kappa(R) < \tau$.

A satisfactory partition of A which minimizes $T_1 + T_2$ is termed an optimal partition of A. If A is badly conditioned, it may be impossible to find a satisfactory partition for A since the second criterion $\kappa(R) < \tau$ may not be satisfied for any partition of A. Therefore, we propose a partitioning algorithm that can find a satisfactory partition or even optimal partition for matrices which are not badly conditioned.

In our approach, rows of A are permuted so that they are increasingly dense toward the bottom of the permuted A. More precisely, let l_i be the number of nonzeros in the i^{th} row. Rows of A are permuted so that $l_{i+1} \geq l_i$ for $1 \leq i < m$ in the permuted A, which is denoted by $\tilde{A}$. This row permutation greatly facilitates the partition of $\tilde{A}$ into dense and sparse rows.

For convenience, MATLAB notations are adopted for representing a matrix and its submatrices. For example, $A(i,:)$ is the i^{th} row of A. Let $|A(i,:)|$ denote the number of nonzeros in $A(i,:)$ and $l_{max} = \max_i |A(i,:)|$. After rows of A are permuted as described above, the partition of $\tilde{A}$ is determined by a *sparsity threshold parameter*, denoted by ρ. The value of ρ is between 0 and 1. If $|\tilde{A}(i,:)| \leq \rho n$, then $\tilde{A}(i,:)$ is placed into A_1. If $|\tilde{A}(i,:)| > \rho n$, then $\tilde{A}(i,:)$ is placed into A_2. Clearly, each value of ρ induces a partition of $\tilde{A}$, and two different values of ρ may induce the same partition of $\tilde{A}$. The value of ρ which produces an optimal partition is denoted by ρ_{opt}. We examine the impact of ρ upon the performance of our approach in §4 and determine an optimal or almost optimal value for ρ through empirical results. The overall algorithm for solving sparse linear least squares problems, which may contain relatively dense rows, is shown in Fig. 1 and this algorithm is referred to as algorithm SLS.

The algorithm SLS requires four input parameters A, b, τ and ρ. T_1 represents the running time for the first four steps of the algorithm SLS in Fig. 1, and T_2 represents the running time for the last step of the algorithm SLS in Fig. 1.

1. Initial probe

   if $l_{max} \le \rho n$ **then** solve the sparse LS problem $\min_x \|Ax - b\|_2$ and **return**.

2. Row permutation

   Let P_r be a permutation matrix such that the rows of $P_r A$ are sorted in non-decreasing number of nonzeros per row. Set $\tilde{A} = P_r A$ and $\tilde{b} = P_r b$.

3. Partition

$$A_1 = \begin{bmatrix} \tilde{A}(1:n,:) \\ \tilde{A}(n+1:k,:) \end{bmatrix}, \qquad b_1 = \begin{bmatrix} \tilde{b}(1:n) \\ \tilde{b}(n+1:k) \end{bmatrix},$$

$$A_2 = \tilde{A}(k+1:m,:), \qquad b_2 = \tilde{b}(k+1:m)$$

   such that $|\tilde{A}(k,:)| \le \rho n$, and $|\tilde{A}(k+1,:)| > \rho n$.

4. Solution

   Compute $A_1 P = Q \begin{bmatrix} R \\ 0 \end{bmatrix}$ and form $Q^T b_1 = \begin{bmatrix} c \\ d \end{bmatrix}$, where P is a permutation matrix, Q is an orthogonal matrix which is not computed explicitly, R is an $n \times n$ upper triangular matrix, and c contains the first n entries of $Q^T b_1$.

   if R is nonsingular—i.e., A_1 is nonsingular **then**
       Solve $R\hat{x} = c$ and set $x_0 = P\hat{x}$, where x_0 is the solution to $\min_x \|A_1 x - b_1\|_2$.
       if $A_2 = \emptyset$ **then**
           return
       else
           if $\kappa(R) \ge \tau$ **then** solve the sparse LS problem $\min_x \|Ax - b\|_2$ and **return**.
       end if
   else
       if $A_2 = \emptyset$ **then**
           return a flag indicating "matrix A is singular".
       else
           Solve the sparse LS problem $\min_x \|Ax - b\|_2$ and **return**.
       end if
   end if

5. Solution update — update the solution x_0 computed in step 4 by A_2 and b_2.

Figure 1: Algorithm SLS for solving sparse LS problems containing dense rows

- The first step of SLS checks whether A contains any dense row with respect to the prescribed value of the sparsity threshold parameter ρ. In case that A doesn't contain any dense row, the original linear least squares problem is solved directly.

- In the fourth step, sparse QR factorization of A_1P is computed. This QR factorization dominates the running time of the algorithm SLS. If R is nonsingular, the solution to $\min_x \|A_1x - b_1\|_2$ is calculated. If R is singular or $\kappa(R) \geq \tau$, a seemingly sensible approach would be to move some rows of A_2 to A_1. Then sparse QR factorization of the newly-augmented A_1 is computed. However, there is no guarantee that the condition number of the triangular factor of the augmented A_1 would satisfy the prescribed tolerance value τ. Therefore, a sequence of sparse QR factorizations may be needed. This may lead to extreme poor performance. Our experience indicates that it is almost always better to solve the original problem $\min_x \|Ax - b\|_2$ directly when R is singular or $\kappa(R) \geq \tau$.

4 Numerical Experiments

4.1 Test problems

Our test problems arise from the linear programming setting. From the linear program(LP)

$$\begin{array}{ll} \text{minimize} & \hat{c}^T x \\ \text{subject to} & Mx = \hat{b} \quad \text{and} \quad x \geq 0, \end{array} \tag{10}$$

we construct an extended matrix

$$A = \begin{bmatrix} M & 0 \\ I & 0 \\ 0 & M^T \\ \hat{c}^T & \hat{b}^T \end{bmatrix}, \tag{11}$$

where I is an identity matrix, M, $\hat{c}$ and $\hat{b}$ are as in (10). It is possible to achieve optimality to (10) by solving a sequence of least squares problems involving matrices of the form (11) [2].

In our experiments, the LP problems from Netlib are used and they can be obtained by anonymous ftp to `ftp.netlib.org`(directory: `/lp/data`). In these problems, M is sparse, and vectors $\hat{c}$ and $\hat{b}$ are relatively dense. Therefore, A is a sparse matrix with at least one relatively dense row. The characteristics of the test matrices used in our numerical experiments are shown in Table 1, where m is the number of rows, n the number of columns, $|A|$ the number of nonzeros in A, density$(A) = |A|/(m * n)$, and $\rho_{max} = l_{max}/n$. In order to assess the accuracy of the computed solution $\overline{x}$ to $\min_x \|Ax - b\|_2$, the right-hand side vector b is set to the product of A and a prescribed solution vector x with $x_i = 2 + (i-1)/1000$ for $1 \leq i \leq n$. All performance results are obtained on a Sun SPARCstation IPX. Running times are in seconds.

Table 1: Characteristics of the test problems

No.	Problem	m	n	$\|A\|$	density(A)	ρ_{max}
1	agg	1719	1103	6902	0.0036	0.5104
2	agg2	2033	1274	10941	0.0042	0.5518
3	agg3	2033	1274	10968	0.0042	0.5479
4	bandm	1250	777	5743	0.0059	0.3642
5	boeing1	1804	1077	8906	0.0046	0.4884
6	capri	1264	767	4585	0.0047	0.2073
7	degen2	1959	1201	9873	0.0042	0.5945
8	etamacro	2033	1216	5994	0.0024	0.0855
9	fffff800	2581	1552	14042	0.0035	0.1617
10	finnis	2626	1561	7104	0.0017	0.3331
11	fit1d	2123	1073	28929	0.0127	0.9571
12	forplan	1146	653	10128	0.0135	0.5636
13	gfrd-pnc	2937	1776	7208	0.0014	0.6520
14	maros	4779	2812	22674	0.0017	0.1543
15	perold	3814	2219	16451	0.0019	0.1005
16	pilot4	2833	1621	16064	0.0035	0.1043
17	scagr25	1814	1142	4775	0.0023	0.5727
18	scfxm1	1531	930	6203	0.0044	0.1495
19	scfxm2	3061	1860	12421	0.0022	0.1522
20	scfxm3	4591	2790	18639	0.0015	0.1530
21	scorpion	1321	854	3892	0.0034	0.4192
22	scrs8	3041	1765	8775	0.0016	0.5235
23	scsd1	1598	837	6297	0.0047	0.9092
24	seba	2588	1551	10287	0.0026	0.3424

4.2 SPLS — A sparse LS problem solver

The algorithm SLS in Fig. 1 is implemented in MATLAB 4.2c. Steps 1, 2, 3 and 5 of SLS can be implemented in MATLAB in a straightforward manner. The major component of our implementation is the solution of $\min_x \|A_1 x - b_1\|_2$. In MATLAB 4.2c, the only available mechanism for solving sparse linear least squares problems is the backslash operator "\". For example, $x_0 = A_1 \backslash b_1$ would produce the required solution to $\min_x \|A_1 x - b_1\|_2$. However, our experience with MATLAB 4.2c shows that using the backslash operator for solving sparse linear least squares problems is very slow and unsatisfactory. Therefore, we have designed and implemented our own sparse linear least squares problem solver `SPLS` in C. Our solver is based on the multifrontal scheme [12] and it can be used in MATLAB through the Mex-File facility. The discussion on the algorithmic and implementation details of this solver will be provided elsewhere [12]. The functionality of `SPLS` required by `SLS` can be described as a MATLAB function:

```
X = spls(A, B),
[X, R, p] = spls(A, B).
```
(12)

The input and output arguments in (12) satisfy `A(:,p) = QR` and `X` is equal to `A\B` provided that `A` has full column rank.

To demonstrate the efficiency of `SPLS`, we compare the performance of `SPLS` with that of the backslash operator "\" in MATLAB 4.2c. Four problems from the Harwell-Boeing collection of sparse matrices are used and their characteristics are shown in Table 2. The performance results for `x = spls(A, b)` and `X = A\B` obtained on a Sun SPARCstation IPX are shown in Table 3, where "time" is the runing time in seconds and "#mflops" is the total number of megaflops performed. The true solution x is given as $x_i = 2 + (i-1)/1000$ $(1 \le i \le n)$ and the right hand side b is set to the product of A and x. The "rel_error" in Table 3 is $\|x - \overline{x}\|_2 / \|x\|_2$, where $\overline{x}$ is the computed solution.

Table 2: Characteristics of four Harwell-Boeing problems

problem	m	n	$\|A\|$	density(A)	ρ_{max}
WELL1033	1033	320	4732	0.0143	0.0156
ILLC1033	1033	320	4719	0.0143	0.0156
WELL1850	1850	712	8755	0.0066	0.0070
ILLC1850	1850	712	8636	0.0066	0.0070

The MATLAB code for `SLS` is provided in the **Appendix**. Let `B` $= [b_1, b_2, \cdots, b_r]$ and `X` $= [x_1, x_2, \cdots, x_r]$. The main function is defined as `X = SLS(A, B)` for solving a sequence of sparse linear least squares problems $\|Ax_j - b_j\|_2$ for $1 \le j \le r$. If `A` contains no dense rows with respect to the prescribed value of the sparsity threshold parameter ρ, then `SLS` calls `SPLS` directly. Otherwise,

Table 3: Performance comparison of `SPLS` and "\" in MATLAB 4.2c on a SPARCstation IPX

test problem	`x = spls(A, b)`			`x = A\b`		
	time	#mflops	rel_error	time	#mflops	rel_error
WELL1033	0.850	0.406	1.114e-15	14.117	8.141	1.731e-15
ILLC1033	0.850	0.405	8.026e-14	12.317	6.785	2.135e-13
WELL1850	2.033	1.395	1.123e-15	63.717	44.382	1.046e-15
ILLC1850	2.033	1.338	1.165e-14	64.583	45.865	3.742e-14

`SLS` first calls the function `SLSS`, and then calls the function `SLSU`. The function `SLSS` performs steps 2, 3 and 4 of the algorithm SLS and the function `SLSU` performs step 5 of the algorithm SLS. Implementation details for functions `SLS`, `SLSS` and `SLSU` are provided in the **Appendix**. If a user intends to use the MATLAB function `SLS`, then he/she must supply a MATLAB function whose functionality is equivalent to that of `SPLS` described above.

4.3 Impact of the sparsity threshold parameter

In our numerical experiments, the tolerance value τ for $\kappa(R)$ is set to $\epsilon^{-1/2}$, where ϵ is the machine precision. For each of our test problems, we obtain running times of the algorithm SLS for a sequence of values of ρ between 0 and ρ_{max}. The running times for four test problems 'agg3', 'seba', 'capri', and 'degen2' are shown in Fig. 2, Fig. 3, Fig. 4 and Fig. 5, respectively, where

$$\rho = 0.01, 0.02, 0.03, \cdots, 0.14, 0.15, 0.20, 0.25, \cdots, \rho_{max}$$

are used. According to the magnitude of the condition number $\kappa(R)$, the test problems can be divided into three classes as follows:

- Class I: Regardless how $\tilde{A}$ is partitioned, the condition number of the triangular factor R of A_1 always satisfies the tolerance value τ as long as the partition is valid. Both 'agg3' and 'seba' shown in Fig. 2 and Fig. 3, respectively, are such problems. As ρ increases, R is increasingly dense. Consequently, the sparse QR factorization of A_1 dominates the overall solution process and becomes increasingly expensive. When ρ is small, A_2 may be large and the solution updating part of the algorithm SLS may be expensive. Therefore, the running time curve may have a steep drop when ρ is small. The problem 'agg3' shown in Fig. 2 demonstrates this scenario.

- Class II: When ρ is small, the condition number of the triangular factor R of A_1 does not satisfy the tolerance value τ. When ρ is reasonably large, the condition number of the triangular factor R of A_1 always satisfies the tolerance value τ. The problem 'capri' shown in Fig. 4 belongs to this class. If ρ is small, $\kappa(R) > \tau$ and two sparse QR factorizations are performed according to the algorithm SLS described in Fig. 1. The running time curve has a steep drop for a small value of ρ. After ρ achieves its optimal value, the performance of SLS deteriorates steadily as ρ increases since R becomes increasingly dense.

- Class III: The problem 'degen2' shown in Fig. 5 is a typical problem in this class. Regardless how $\tilde{A}$ is partitioned, the condition number of the triangular factor R of A_1 does not satisfy the tolerance value τ. For each $\rho < \rho_{max}$, two sparse QR factorizations are performed. Obviously, only one sparse QR factorization is required for $\rho \geq \rho_{max}$, and the best performance is achieved with $\rho \geq \rho_{max}$.

Our results show that an optimal or almost optimal value for the sparsity threshold parameter is in the neighborhood of 0.05. The running time curve for a problem in Class I or II may have a steep drop for a small value of ρ which is typically less than 0.03. For all problems except those in class III, the running time curves are relatively flat in the neighborhood of 0.05 and the performance results corresponding to $\rho = 0.03, 0.04, 0.05, 0.06$ are almost indistinguishable. In order to avoid the unpredictable region between 0 and 0.03, we choose 0.05 to be the optimal value for the sparsity threshold parameter ρ.

Some performance results of the algorithm SLS are shown in Table 4, where T_{opt}, $T_{0.05}$ and T_{max} denote the running times of the algorithm SLS with $\rho = \rho_{opt}, 0.05, \rho_{max}$, respectively, and

$$\delta = \frac{(T_{0.05} - T_{opt})}{T_{opt}} * 100.$$

If a problem belongs to class X, then it is marked X in the column under "Class".

Obviously, for problems belonging to class III, the algorithm SLS cannot compete with the straightforward approach which solves sparse linear least squares problems directly. In our test suite, class III contains three problems 'degen2', 'fffff800' and 'scorpion'. As expected, $\rho_{opt} \geq \rho_{max}$ for these problems. For problems 'agg', 'gfrd-pnc' and 'scrs8', $\rho_{opt} > 0.15$. But T_{opt} and $T_{0.05}$ are practically the same for these problems. Among our test problems except those in class III, there are only four problems with $\delta > 20\%$. With a few exceptions, $T_{0.05} \ll T_{max}$ for problems in classes I and II. This demonstrates that our approach is superior to the straightforward approach for problems which are not badly conditioned.

In Table 5, the results under "SLS" correspond to the algorithm SLS with $\rho = \rho_{opt}$, the results under "Straightforward" correspond to the straightforward approach, and $\kappa(R)$ is the condition number of the triangular factor R of $\tilde{A}$. Clearly, the numerical accuracy of SLS is at least as good as that of the straightforward approach. Note that the solution to a problem in class III is not accurate because the corresponding matrix A is almost rank deficient.

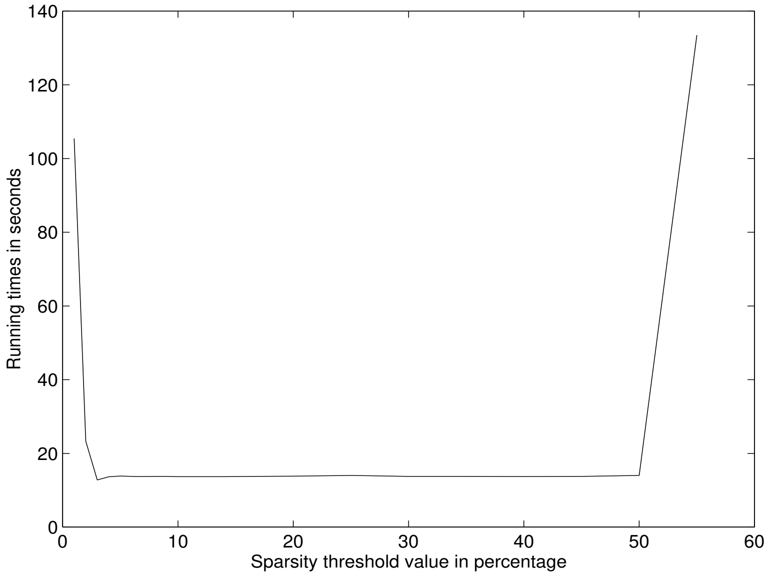

Figure 2: Running times obtained on problem 'agg3'

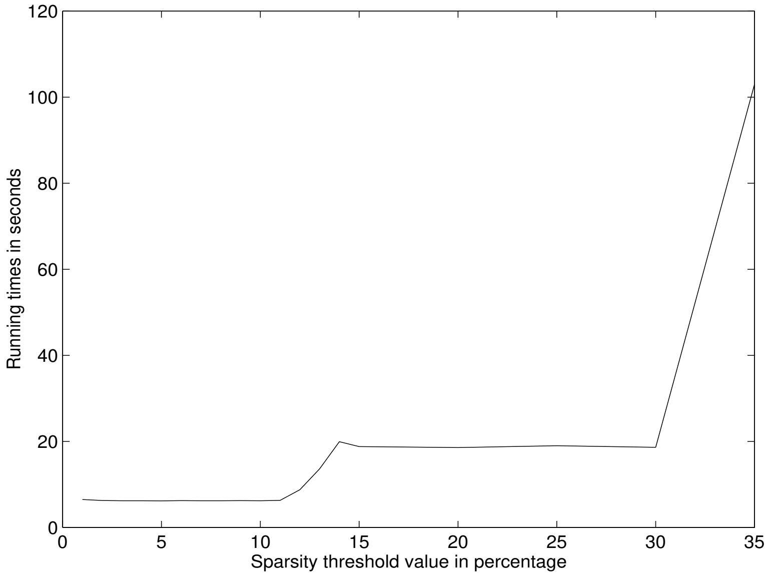

Figure 3: Running times obtained on problem 'seba'

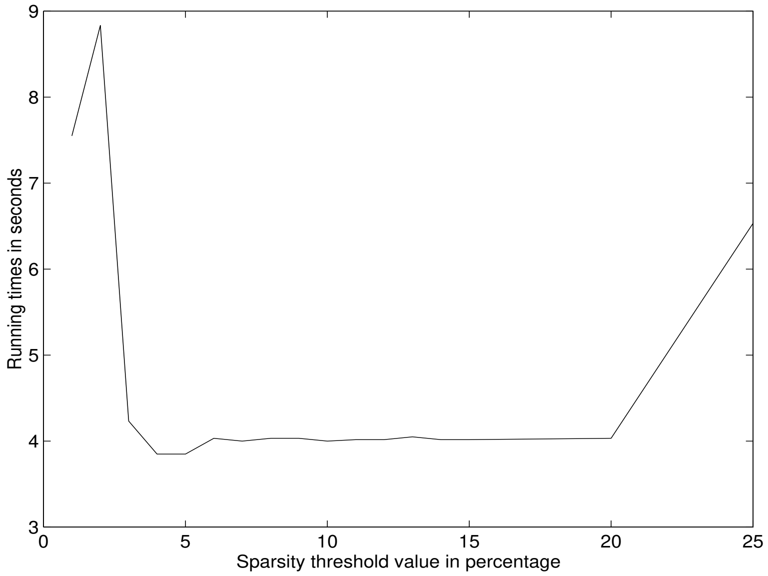

Figure 4: Running times obtained on problem 'capri'

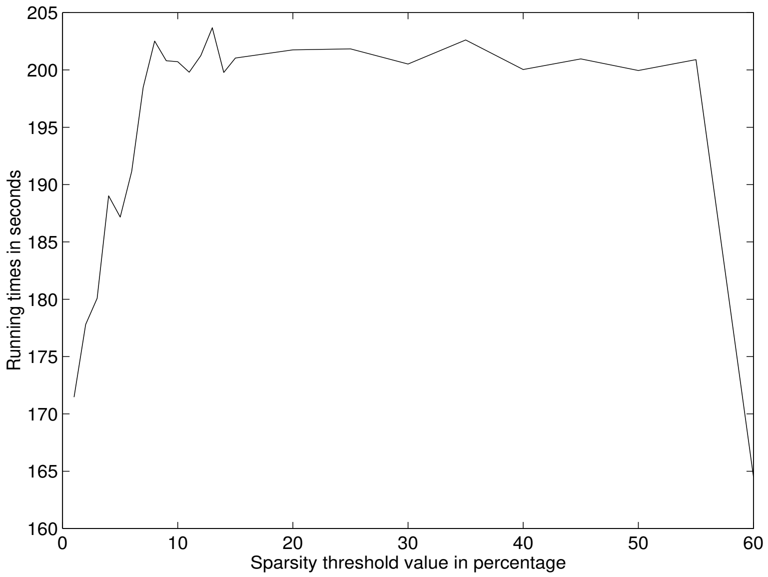

Figure 5: Running times obtained on problem 'degen2'

Table 4: Some performance results of the algorithm SLS

No.	Problem	ρ_{opt}	T_{opt}	$T_{0.05}$	$\delta(\%)$	T_{max}	Class
1	agg	0.35	5.983	6.017	0.557	52.933	I
2	agg2	0.03	12.817	13.617	6.242	140.200	I
3	agg3	0.03	12.800	13.850	8.203	133.517	I
4	bandm	0.03	4.500	4.883	8.519	19.817	II
5	boeing1	0.02	7.417	11.017	48.539	101.383	I
6	capri	0.04	3.850	3.850	0.000	6.533	II
7	degen2	0.60	164.500	187.167	13.779	164.500	III
8	etamacro	0.01	13.617	23.750	74.419	29.283	I
9	fffff800	0.20	58.750	80.033	36.227	58.750	III
10	finnis	0.02	8.433	10.767	27.668	90.567	I
11	fit1d	0.06	6.183	6.200	0.270	170.383	I
12	forplan	0.08	5.917	6.433	8.732	24.350	II
13	gfrd-pnc	0.55	3.967	3.967	0.000	614.383	I
14	maros	0.04	47.217	47.700	1.024	101.500	I
15	perold	0.09	64.900	65.067	0.257	75.900	I
16	pilot4	0.04	35.000	35.283	0.810	41.083	II
17	scagr25	0.11	2.883	2.917	1.156	152.533	I
18	scfxm1	0.03	4.517	4.783	5.904	7.117	I
19	scfxm2	0.03	9.900	10.900	10.101	30.867	I
20	scfxm3	0.02	15.300	16.450	7.516	129.333	I
21	scorpion	0.45	26.467	27.067	2.267	26.467	III
22	scrs8	0.35	9.367	9.417	0.534	148.483	I
23	scsd1	0.04	6.467	8.550	32.216	67.383	I
24	seba	0.05	6.200	6.200	0.000	102.967	I

Table 5: Numerical accuracy of SLS and the straightforward approach

No.	Problem	SLS		Straightforward		$\kappa(R)$
		$\|b - A\overline{x}\|_2$	$\|x - \overline{x}\|_2$	$\|b - A\overline{x}\|_2$	$\|x - \overline{x}\|_2$	
1	agg	4.3228e-12	1.4866e-12	6.8163e-08	3.3068e-08	4.0890e+07
2	agg2	2.9802e-08	1.7689e-12	1.4903e-08	2.2719e-10	7.6811e+06
3	agg3	2.9802e-08	2.1335e-12	1.4924e-08	6.1422e-10	3.6855e+06
4	bandm	1.3694e-12	8.7918e-13	1.8206e-12	1.2687e-12	1.7016e+04
5	boeing1	1.4287e-10	3.9916e-11	8.7222e-11	5.2701e-11	2.5059e+05
6	capri	2.9200e-11	4.1333e-12	1.4735e-11	1.4728e-12	1.1595e+05
7	degen2	2.4774e-12	5.1952e+01	2.4774e-12	5.1952e+01	6.8589e+17
8	etamacro	2.4594e-11	6.5480e-12	2.2372e-11	5.4808e-12	6.1642e+05
9	fffff800	3.4892e-10	1.0472e-07	3.4892e-10	1.0472e-07	4.2754e+10
10	finnis	2.0373e-10	2.7474e-13	6.2005e-11	2.0895e-11	8.2727e+04
11	fit1d	5.7747e-10	2.0205e-12	1.6963e-10	2.4273e-11	8.1921e+04
12	forplan	3.7253e-09	5.0974e-12	1.2375e-08	7.9428e-09	5.4469e+08
13	gfrd-pnc	4.4703e-08	1.4655e-11	7.6262e-09	1.4363e-09	1.0656e+08
14	maros	4.8221e-10	4.2367e-10	4.7356e-10	2.3192e-10	3.7604e+06
15	perold	3.0356e-10	7.3694e-11	2.9760e-10	1.8365e-10	2.0753e+06
16	pilot4	2.0838e-10	1.3168e-10	1.7572e-10	5.2509e-11	6.2518e+05
17	scagr25	2.3283e-09	8.7811e-14	4.9687e-10	1.6317e-10	1.0245e+05
18	scfxm1	1.4784e-11	3.2839e-12	9.5647e-12	5.9858e-12	8.2838e+04
19	scfxm2	2.9772e-11	8.4262e-12	2.8704e-11	2.8787e-11	8.7417e+04
20	scfxm3	2.3298e-10	1.9102e-11	2.0988e-11	2.8108e-11	9.0295e+04
21	scorpion	1.8271e-12	8.0139e+02	1.8271e-12	8.0139e+02	1.7133e+20
22	scrs8	3.1779e-11	5.8256e-12	2.3787e-10	3.2581e-11	1.3599e+06
23	scsd1	3.6386e-12	6.1120e-14	1.0613e-12	1.0217e-12	4.0511e+02
24	seba	1.7337e-11	1.5345e-12	1.6248e-11	2.6788e-12	1.5645e+04

5 Concluding Remarks

We have described an efficient algorithm SLS for solving sparse linear least squares problems containing relatively dense rows. A MATLAB implementation of our algorithm is provided in the **Appendix**. We assume that a sparse linear LS problem solver whose functionality is equivalent to that of `SPLS` described in §4.2 will be provided by the user. Our approach handles sparse rows and dense rows separately. Our major contribution includes the design and implementation of an efficient algorithm which finds satisfactory or even optimal partition for a sparse matrix containing relatively dense rows. A crucial parameter that dictates how a sparse matrix is partitioned is the sparsity threshold parameter ρ. Through extensive empirical results, we conclude that 0.05 is an optimal or almost optimal value for the the sparsity threshold parameter ρ. The efficiency of our approach is clearly demonstrated by the difference between running times of the algorithm SLS with $\rho = 0.05$ and those of the the straightforward approach in Table 4. Furthermore, SLS is at least as accurate as the straightforward approach.

Acknowledgements. I would like to thank Professor Thomas F. Coleman for many discussions relating to this work and for his helpful comments and suggestions on the manuscript.

References

[1] Å. Björck, *Stability analysis of the method of seminormal equations for linear least squares problems*, Linear Algebra Appl., 88/89 (1987), pp. 31–48.

[2] T. F. Coleman, *private communication.* 1995.

[3] J. A. George, *Nested dissection of a regular finite element mesh*, SIAM J. Numer. Anal., 10 (1973), pp. 345–363.

[4] J. A. George and M. T. Heath, *Solution of sparse linear least squares problems using Givens rotations*, Linear Algebra and its Appl., 34 (1980), pp. 69–83.

[5] J. A. George and J. W. H. Liu, *Computer Solution of Large Sparse Positive Definite Systems*, Prentice Hall, Englewood Cliffs, NJ, 1981.

[6] ——, *Householder reflections versus Givens rotations in sparse orthogonal decomposition*, Linear Algebra and its Appl., 88/89 (1987), pp. 223–238.

[7] ——, *The evolution of the minimum degree algorithm*, SIAM Review, 31 (1989), pp. 1–19.

[8] G. Golub, *Numerical methods for solving linear least squares problems*, Numer. Math., 7 (1965), pp. 206–216.

[9] J. W. H. Liu, *On general row merging schemes for sparse Givens transformations*, SIAM J. Sci. Stat. Comput., 7 (1986), pp. 1190–1211.

[10] P. Matstoms, *Sparse QR Factorization with Applications to Linear Least Squares Problems*, PhD thesis, Linköping University, Sweden, 1994.

[11] C. Sun, *Parallel sparse orthogonal factorization on distributed-memory multiprocessors*, Tech. Report CTC93 TR162, Advanced Computing Research Institute, Center for Theory and Simulation in Science and Engineering, Cornell University, Ithaca, NY, Dec. 1993. (Revised, To appear in SIAM Journal on Scientific Computing).

[12] ——, *Improved algorithms for multifrontal QR factorization.* Work in preparation, 1995.

[13] ——, *Parallel solution of sparse linear least squares problems on distributed-memory multiprocessors*, Tech. Report CTC95TR212, Advanced Computing Research Institute, Center for Theory and Simulation in Science and Engineering, Cornell University, Ithaca, NY, May 1995.

Appendix: A Sparse Linear Least Squares Problem Solver

```
function X = sls(A, B)
%
% A   an m-by-n (m>=n) sparse matrix
% B   an m-by-r dense matrix
%
% X   an n-by-r dense matrix, which is the solution to min ||AX - B||.
%
% Written by Chunguang Sun, June 1995
%

if (nargin ~= 2) error('SLS requires two input arguments.'); end
if (nargout ~= 1) error('SLS requires one output argument.'); end

[m, n] = size(A);

rho = 0.05;
tau = 1.0/sqrt(eps);

row_len_bound = ceil(rho*n);

row_len = zeros(m, 1);
AS = spones(A);
row_len = full(sum(AS'));
max_row_size = max(row_len);

if (max_row_size <= row_len_bound)

    X = spls(A, B);     % 'spls' is to be supplied by the user.

else

    [R, p, Y, A2, B2] = slss(A, B, rho, tau, row_len);
    X = slsu(R, p, Y, A2, B2);

end
```

```
function [R, p, Y, A2, B2] = slss(A, B, rho, tau, row_len)
%
% A         an m-by-n (m>=n) sparse matrix
% B         an m-by-r dense matrix
% rho       sparsity threshold parameter
% tau       tolerance value for the condition number of R
% row_len   an m-by-1 dense matrix, row_len(i) is the number of
%           nonzeros in row i of A.
%
% A2   a k-by-n submatrix of A, where k<=m-n.
% B2   a k-by-r submatrix of B, where k<=m-n.
%
%      Let A1 be the submatrix of A formed by removing rows in A2 from A.
%      Let B1 be the submatrix of B formed by removing rows in B2 from B.
% R    an n-by-n sparse upper triangular matrix so that
%      A1(:,p) = Q | R |
%                  | 0 |,
%      where Q is an orthogonal matrix and it is not computed.
% Y    an n-by-r dense matrix, which is the solution to min ||A1 Y-B1||.
%
% Written by Chunguang Sun, June 1995
%

if (nargin ~= 5) error('SLSS requires five input arguments.'); end
if (nargout ~= 5) error('SLSS requires five output arguments.'); end

[m, n] = size(A);
if (m<1 | n<1) error('Input argument one must be non-empty.'); end
if (m < n) error('Input argument one does not have enough rows.'); end

row_len_bound = ceil(rho*n);

[new_row_len, row_perm] = sort(row_len);

H = A(row_perm,:);
C = B(row_perm,:);
```

```
k = n;
for i = n+1:m
    if (new_row_len(i) <= row_len_bound)
        k = i;
    else
        break;
    end
end

[Y, R, p] = spls(H(1:k,:), C(1:k,:));  % 'spls' is to be supplied by the user.

if (k < m)
    if (condest(R) >= tau)

        [Y, R, p] = spls(H, C);  % 'spls' is to be supplied by the user.

    end
end

A2 = H(k+1:m,:);
B2 = C(k+1:m,:);
```

```
function X = slsu(R, p, Y, A2, B2);
%
% Written by Chunguang Sun, June 1995
%

if (nargin ~= 5) error('SLSU requires five input arguments.'); end
if (nargout ~= 1) error('SLSU requires one output argument.'); end
[m2, n2] = size(A2);

if (m2 > 0)
    [m, n] = size(R);
    [b2m, b2n] = size(B2);

    Y1 = Y(p,:);
    E = R'\A2(:,p)';

    I = eye(m2);
    F = [full(E)
           I    ];

    [U, T] = qr(F, 0);
    L = T';

    D = B2 - A2(:,p)*Y1;
    S1 = L\D;

    W = U*S1;
    V = W(1:n,:);

    Z = R\V;
    X1 = Y1 + Z;

    X = zeros(n, b2n);
    X(p,:) = X1;
else
    X = Y;
end
```