`tsnnls`: A solver for large sparse least squares problems with non-negative variables

Jason Cantarella*

Department of Mathematics, University of Georgia, Athens, GA 30602

Michael Piatek†

*Department of Mathematics and Computer Science,
Duquesne University, Pittsburgh, PA 15282*

The solution of large, sparse constrained least-squares problems is a staple in scientific and engineering applications. However, currently available codes for such problems are proprietary or based on `MATLAB`. We announce a freely available C implementation of the fast block pivoting algorithm of Portugal, Judice, and Vicente. Our version is several times faster than Matstoms' `MATLAB` implementation of the same algorithm. Further, our code matches the accuracy of `MATLAB`'s built-in `lsqnonneg` function.

Keywords: Non-negative least-squares problems, NNLS, sparse Cholesky factorization, sparse matrices, LSQR

1. INTRODUCTION

The authors were recently faced with the challenge of finding a fast solver for the sparse non-negative least-squares problem (NNLS) to embed in a much larger scientific application. The problem is given by

$$\min_x \frac{1}{2}||Ax-b||_2^2 \quad \text{with } x \geq 0 \tag{1}$$

where A is an $m \times n$ matrix, $x \in \mathbb{R}^n$, $b \in \mathbb{R}^m$ and $m > n$, and we assume A has full column rank. This is a standard problem in numerical linear algebra ([1, 4]) which is handled by a number of commercial libraries ([2, 9, 10]) and by the `MATLAB`-based Sparse Matrix Toolbox of [5]. While these methods work well, their users must incur the overhead of a large math package or the expense and license restrictions of commercial libraries. There does not seem to be a freely available solver for this problem without these disadvantages. This motivated the development of `tsnnls`, a lightweight ANSI C implementation of the block principal pivoting algorithm of [7] which matches the accuracy of the `MATLAB`-based codes and is considerably faster. The code can be obtained at `http://www.cs.duq.edu/~piatek/tsnnls/` or `http://ada.math.uga.edu/research/software/tsnnls/`. Users may redistribute the library under the terms of the GNU GPL.

*Email: cantarel@math.uga.edu

†Email: piatek@mathcs.duq.edu

2. ALGORITHMS

The following is a summary of our main algorithm as described in [7]. The fundamental observation underlying the block principal pivoting algorithm is that Equation 1 can be rewritten (using the definition of the L^2 norm) as a quadratic program:

$$\min_x -(A^T b)^T x + \frac{1}{2} x^T A^T A x, \qquad \text{with } x \geq 0. \tag{2}$$

Since A has full rank, $A^T A$ is positive-definite, and this is a convex program which can be rewritten as a linear complementarity problem:

$$y = A^T A x - A^T b, \quad y \geq 0, x \geq 0, \langle x, y \rangle = 0. \tag{3}$$

The last condition means that the nonzero entries of x and y occupy complementary variables: any given position must vanish in x or y (or both). In fact, the nonzero entries in y represent variables in x which would decrease the residual $Ax - b$ still further by becoming negative, and so are set to zero in the solution to the constrained problem.

Suppose we have a division of the n indices of the variables in x into complementary sets F and G, and let x_F and y_G denote pairs of vectors with the indices of their nonzero entries in these sets. Then we say that the pair (x_F, y_G) is a *complementary basic solution* of Equation 3 if x_F is a solution of the unconstrained least squares problem

$$\min_{x_F \in \mathbb{R}^{|F|}} \frac{1}{2} ||A_F x_F - b||_2^2, \tag{4}$$

where A_F is formed from A by selecting the columns indexed by F, and y_G is obtained by

$$y_G = A_G^T \left(A_F x_F - b \right). \tag{5}$$

If $x_F \geq 0$ and $y_G \geq 0$, then the solution is *feasible*. Otherwise it is *infeasible*, and we refer to the negative entries of x_F and y_G as *infeasible variables*. The idea of the algorithm is to proceed through infeasible complementary basic solutions of (3) to the unique feasible solution by exchanging infeasible variables between F and G and updating x_F and y_G by (4) and (5). To minimize the number of solutions of the least-squares problem in (4), it is desirable to exchange variables in large groups if possible. In rare cases, this may cause the algorithm to cycle. Therefore, we fall back on exchanging variables one at a time if no progress is made for a certain number of iterations with the larger exchanges.

The original block-principal pivoting algorithm works very well for what we call "numerically nondegenerate" problems, where each of the variables in F and G have values distinguishable from zero by the unconstrained solver in the feasible solution. If this is not the case, a variable with solution value close to zero may be passed back and forth between F and G, each time reported as slightly negative due to error in the unconstrained solver. We work around this problem by zeroing variables in the unconstrained solution that are within 10^{-12} of zero. Although this strategy works well in practice, we have not developed its theoretical basis. Indeed, this seems to be an unexplored area: [7] do not discuss the issue in their original development of the algorithm and Matstoms' `snnls` implementation fails in this case.

The details are summarized below.

Block principal pivoting algorithm (modified for numerically degenerate problems)

```
Let $F = \emptyset$, $G = \{1, \dots, n\}$, $x = 0$, $y = -A^T b$, and $p = 3$.
Set $N = \infty$.
**while** $(x_F, y_G)$ is an infeasible solution {
   Set $n$ to the number of negative entries in $x_F$ and $y_G$.
   **if** $n < N$ (the number of infeasibles has decreased) {
      Set $N = n$ and $p = 3$.
      Exchange all infeasible variables between $F$ and $G$.
   } **else** {
      **if** $p > 0$ {
         Set $p = p - 1$.
         Exchange all infeasible variables between $F$ and $G$.
      } **else** {
         Exchange only the infeasible variable with largest index.
      }
   }
   Update $x_F$ and $y_G$ by Equations 4 and 5.
   Set variables in $x_F < 10^{-12}$ and $y_G < 10^{-12}$ to zero.
}
```

The normal equations solver.

Solving Equation 4 requires an unconstrained least-squares solver. We will often be able to do this by the method of normal equations. Since some of our software design choices depend on the details of this standard method, we review them here. To solve a least-squares problem $Ax = b$ using the normal equations, one solves

$$A^T A x = A^T b \tag{6}$$

using a Cholesky factorization of the symmetric matrix $A^T A$. This is extremely fast. For an $m \times n$ dense matrix A, the matrix multiplication required to form $A^T A$ requires $n^2 m$ flops, which is more expensive than the standard Cholesky algorithm which is known to take $\frac{1}{3}n^3 + O(n^2)$ flops. For our sparse matrix problems, we found a comparable relationship between the time required for a sparse matrix-multiply and the TAUCS sparse Cholesky algorithm.

The numerical performance of this method can be a problem. The condition number of $A^T A$ is the square of the condition number κ of A. For this reason, we must expect a relative error of about $c\kappa^2\epsilon$, where ϵ is the machine epsilon ($\simeq 10^{-16}$ in our double-precision code), and c is not large. As [3] points out, the Cholesky decomposition may fail entirely when $\kappa^2\epsilon \geq 1$, so we cannot expect this method to handle matrices with $\kappa > 10^8$. Our tests indicate that this simple analysis predicts the error in the normal equations solver very well (see Section 4), so we can anticipate the accuracy of the solver by estimating the condition number of $A^T A$.

3. SOFTWARE ARCHITECTURE

Our primary design goal in the development of tsnnls was to create the most efficient solver which met the user's accuracy requirements and did not depend on commercial software or restricted libraries. It is clear that the heart of the algorithm is the solution of the least-squares problem in Equation 4 for the new x_F. But the way these solutions are used is quite interesting. In the intermediate stages of the calculation, we only use x_F and y_G to search for infeasible variables to shift between F and G. So we need only calculate correct *signs* for all the variables— beyond this the numerical quality of these solutions is unimportant. But the last solution of Equations 4 and 5 is the result of the algorithm, so this solution must meet the user's full accuracy needs. Our implementation takes advantage of this situation by using the method of normal equations for the intermediate solutions of Equation 4 and then recomputing the final solution using the more accurate LSQR solver of [6].

The method of normal equations is already fast. But two of our implementation ideas improve its speed still further in our solver. As we mentioned in Section 2, computing A^TA is the most expensive step in the normal equations solver. A first observation is that we need not form A^T explicitly in order to perform this matrix multiplication, since A^TA_{ij} is just the dot product of the i^{th} and j^{th} columns of A. This provides some speedup. More importantly, we observe that each least-squares problem in tsnnls is based on a submatrix A_F of the same matrix A. Since $A_F^TA_F$ is a submatrix of A^TA, we can precompute the full A^TA and pass submatrices to the normal equations solver as required. This is a significant speed increase. We make use of the TAUCS library of [8] for highly optimized computation of the sparse Cholesky factorization needed for the method of normal equations.

We can estimate the relative error $\kappa^2\epsilon$ of each normal equations solution by computing the condition number κ^2 of $A_F^TA_F$ with the LAPACK function dpocon. Since we have already computed the Cholesky factorization of A^TA as part of the solution, this takes very little additional time in the computation. This is used to determine when a switch to a final step with LSQR is necessary for error control.

In order to simplify its' use in other applications, our library incorporates simplified forms of the TAUCS and LSQR distributions. These are compiled directly into our library, so there is no need for the user to obtain and link with these codes separately.

4. SOFTWARE TESTING

We tested our implementation using problems produced by the LSQR test generator which generates arbitrarily sized matrices with specified condition number and solution (see [6] for details on how the generator works). We report on the relative error of our method with the problems of type $P(80, 70, 4, x)$, which were typical of our test results. Here 80 and 70 are the dimensions of the matrix, each singular value is repeated 4 times, and x is a parameter which controls the condition number of the problem. For these matrices the exact solution was known in advance, so we could measure the relative error of our solutions as a function of condition number.

The results of this test are shown in Figure 1. The line of datapoints indicated by $\times$ shows the error in tsnnls using only our normal equations solver. As expected, it fits very well to about $\frac{1}{10}\kappa^2\varepsilon$ where κ is the condition number of the matrix and ϵ is machine epsilon. The second set of data points (denoted by $\triangleright$) shows that we usually improve our relative error by 2 or 3 orders of magnitude by recomputing the final solution with LSQR. The third set of data points (denoted by $\circ$) plots the error from the MATLAB function lsqnonneg on these problems. For condition numbers up to 10^6, we see that tsnnls and lsqnonneg have comparable accuracy. But surprisingly, our method seems to be more stable than lsqnonneg for very ill-conditioned problems.

We also tested the performance of our software against that of lsqnonneg and that of the snnls code of [5]. All of our timing tests were performed on a dual 2.0 GHz Power Macintosh G5 running Mac OS X 10.3, compiling with gcc 3.3 and -O3, and linking with Apple's optimized

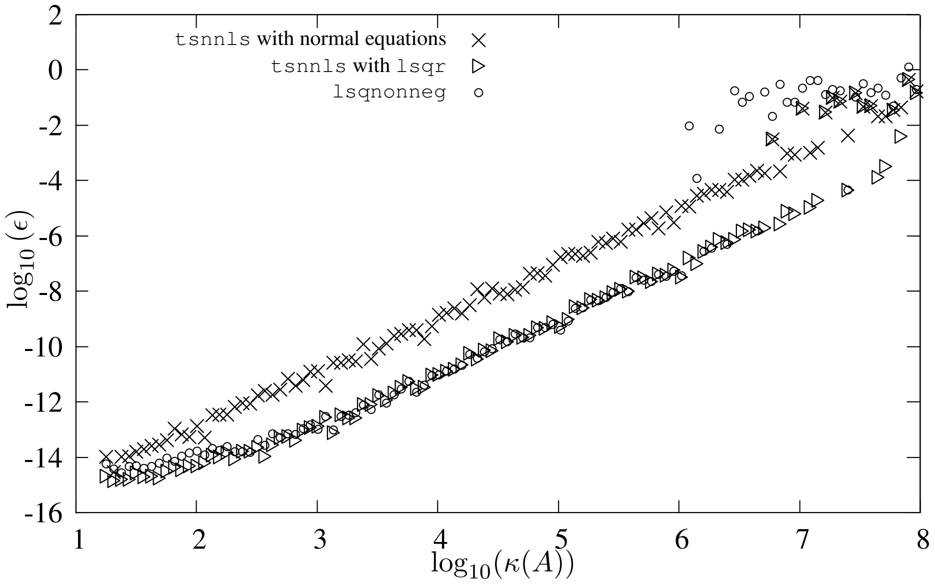

FIG. 1: This plot shows the relative error in the solution of a selection of 80×70 test problems generated by the LSQR test generator with inputs $P(80, 70, 4, x)$ and condition numbers varying from 10^1 to 10^8. The logarithm (base 10) of this error ϵ is plotted against the logarithm of condition number for three codes: tsnnls restricted to use only the normal equations solver, the final version of tsnnls, which recomputes the final solution with LSQR, and the MATLAB function lsqnonneg.

versions of LAPACK and BLAS. We ran snnls under MATLAB 7 with argument -nojvm.

We were required to make two modifications to the snnls code to complete our tests. First, the snnls code uses the column minimum degree permutation (colmmd) before performing sparse Cholesky decompositions. However, as this ordering is deprecated in MATLAB 7 in favor of absolute minimum degree ordering, we tested against a modified snnls using colamd. This was a strict performance improvement for our test cases. We also made the same workaround to handle degenerate problems that we discussed for tsnnls in Section 2.

Our performance results are shown in Figure 2. We tested runtimes for randomly generated, well-conditioned matrices from MATLAB's sprandn function. The matrices were of size $n \times (n - 10)$. The plot shows runtime results for a set of density 1 matrices and a set of density 0.01 matrices, intended to represent general dense and sparse matrices. Each data point represents the average runtime for 10 different matrices of the same size and density.

We can see that the runtime of our implementation is approximately proportional to that of snnls, and that for dense problems it is several times faster. We were surprised to note that the constant of proportionality decreases for sparse matrices and that our method is almost 10 times faster than snnls for matrices of density 0.01.

The runtime of each code is controlled by three computations: the matrix-multiply used to form $A^T A$, the Cholesky decomposition of that matrix, and the final recalculation of the solution (if performed). We expected to be several times faster than snnls since our caching strategy for $A^T A$ eliminates a matrix-multiply operation for each pivot. The number of pivots, however, does not seem to vary with the density of our random test matrices and so does not explain our additional speed increase for sparse problems.

We explored this phenomenon by profiling both our code and snnls. For our random test

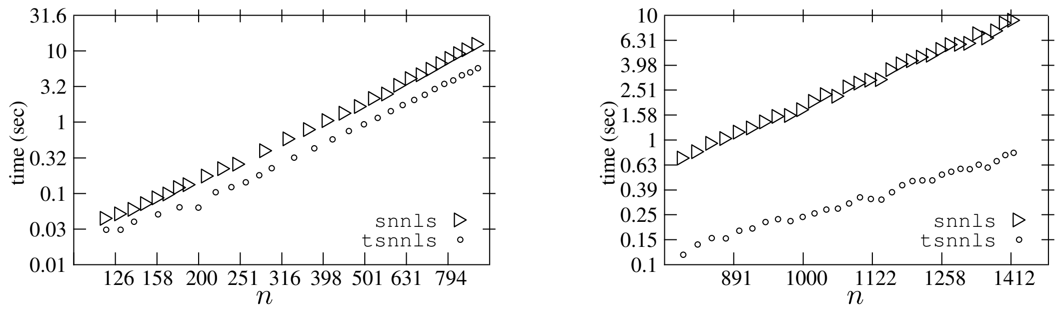

FIG. 2: These log-log scaled plots show the runtime of tsnnls and snnls on density 1.0 (left) and 0.01 (right) matrices of size $n \times (n - 10)$ on a 2.0 Ghz Apple PowerMac G5. We can see that the runtime of tsnnls is basically proportional to that of snnls, but that the constant of proportionality depends on the density of the test matrices. This effect is explained below. All runtimes were calculated by repeating the test problems until the total time measured was several seconds or more.

problems at density 0.01, the final unconstrained solution in snnls (computed using the the MATLAB \ operation) consumes almost 50% of the total runtime. On the other hand, in tsnnls the final unconstrained solution (using LSQR) consumes only 5% of runtime. Since the Cholesky decompositions take comparable time, this would seem to explain the runtime disparity.

We did not show performance data for MATLAB's built-in lsqnonneg because it was so much slower than both tsnnls and snnls. For sparse matrices, this is in part because lsqnonneg is a dense-matrix code. Yet, even on dense matrices, both methods outperformed lsqnonneg by an overwhelming amount. For instance, for a 500×490 dense matrix, lsqrnonneg takes over 100 seconds to complete while snnls and tsnnls both finish in less than one second. We take this as a confirmation of the suggestion in MathWorks' documentation of lsqnonneg that it is not appropriate for large problems.

5. ACKNOWLEDGEMENTS

This work was funded by the National Science Foundation through the University of Georgia VIGRE grant DMS-00-8992 and DMS-02-04826 (to Cantarella and Fu). Piatek acknowledges support through DMS-0311010 to Eric Rawdon. We would like to thank our colleagues for many helpful discussions.

[1] Björck, Å. 1996. *Numerical methods for least squares problems.* Society for Industrial and Applied Mathematics (SIAM), Philadelphia, PA.

[2] Byrd, R. H., Hribar, M. E., and Nocedal, J. 1999. An interior point algorithm for large-scale nonlinear programming. *SIAM J. Optim. 9,* 4, 877–900 (electronic). Dedicated to John E. Dennis, Jr., on his 60th birthday.

[3] Foster, L. 1991. Modifications of the normal equations method that are numerically stable. In *Numerical linear algebra, digital signal processing and parallel algorithms (Leuven, 1988).* NATO Adv. Sci. Inst. Ser. F Comput. Systems Sci., vol. 70. Springer, Berlin, 501–512.

[4] Lawson, C. L. and Hanson, R. J. 1995. *Solving least squares problems.* Classics in Applied

Mathematics, vol. 15. Society for Industrial and Applied Mathematics (SIAM), Philadelphia, PA. Revised reprint of the 1974 original.

[5] MATSTOMS, P. 2004. SLS: A MATLAB toolbox for sparse least squares problems.

[6] PAIGE, C. C. AND SAUNDERS, M. A. 1982. Lsqr: An algorithm for sparse linear equations and sparse least squares. *ACM Trans. Math. Softw. 8,* 1, 43–71.

[7] PORTUGAL, L. F., JÚDICE, J. J., AND VICENTE, L. N. 1994. A comparison of block pivoting and interior-point algorithms for linear least squares problems with nonnegative variables. *Math. Comp. 63,* 208, 625–643.

[8] TOLEDO, S., ROTKIN, V., AND CHEN, D. 2003. TAUCS: A library of sparse linear solvers. Version 2.2.

[9] TOMLAB OPTIMIZATION, INC. 2004. TOMLAB optimization environment.

[10] WALTZ, R. A. AND NOCEDAL, J. 2003. KNITRO user's manual. Technical Report OTC 2003/05.