Iterative Solution of Sparse Linear Least Squares using LU Factorization

Gary Howell, Marc Baboulin

Iterative Solution of Sparse Linear Least Squares using LU Factorization

Gary W. Howell* Marc Baboulin†

Abstract

In this paper, we are interested in computing the solution of an overdetermined sparse linear least squares problem $Ax \simeq b$ via the normal equations method. Transforming the normal equations using the L factor from a rectangular LU decomposition of A usually leads to a better conditioned problem. Here we explore a further preconditioning by L_1^{-1} where L_1 is the $n \times n$ upper part of the lower trapezoidal $m \times n$ factor L. Since the condition number of the iteration matrix can be easily bounded, we can determine whether the iteration will be effective, and whether further preconditioning is required. Numerical experiments are performed with the Julia programming language. When the upper triangular matrix U has no near zero diagonal elements, the algorithm is observed to be reliable. When A has only a few more rows than columns, convergence requires relatively few iterations and the algorithm usually requires less storage than the Cholesky factor of $A^T A$ or the R factor of the QR factorization of A.

Keywords: Sparse linear least squares, iterative methods, preconditioning, conjugate gradient algorithm, `lsqr` algorithm, LU factorization.

1 Introduction

Linear least squares (LLS) problems arise in many high-performance computing (HPC) applications when the number of linear equations is not equal to the number of unknown parameters. We consider the overdetermined full rank LLS problem

$$\min_{x \in \mathbb{R}^n} \|Ax - b\|_2, \tag{1}$$

with $A \in \mathbb{R}^{m \times n}, m \geq n$ and $b \in \mathbb{R}^m$.

LLS solvers can be based on either *direct* methods (e.g., Cholesky factorization of the normal equations, sparse LDL^T factorization of the augmented system, or QR factorization of A), or *iterative* methods (e.g., preconditioned

*North Carolina State University, USA, `gwhowell@ncsu.edu`
†Université Paris-Sud, France, `marc.baboulin@lri.fr`

Krylov subspace methods). In this paper, we focus on the iterative conjugate gradient (CG) solution of the normal equations

$$A^T Ax = A^T b. \tag{2}$$

For the CG method, the linear rate of convergence is bounded by the quantity

$$\frac{\kappa - 1}{\kappa + 1},$$

where $\kappa = cond_2(A) = \sqrt{cond_2(A^T A)}$ is the 2-norm condition number of A (see [3, p. 289] for more details). Then having a better conditioning of the normal equations matrix will influence positively the convergence of the solution of Equation (2) via the CG algorithm.

As with linear systems, a variety of preconditioners have been proposed to overcome the potential ill-conditioning of the normal equations (see [12] for a comprehensive overview of these preconditioners). One possibility is to precondition the normal equations by an incomplete Cholesky decomposition of $A^T A$ (see, e.g., the RIF preconditioner [2] or its new left-looking variant in [23]).

There are also approaches based on LU factorization preconditioning [1, 4, 5, 21]. LU preconditioning has been also studied in [14] with the objective of exploiting the recent progress in sparse LU factorization. For example, both `Matlab` and `Octave` use fast sparse LU factorizations built on the UMFPACK package [6]. Other direct sparse solvers (e.g., [16, 17, 19, 22]) offer scalable sparse LU factorizations for large problems and are commonly used in HPC applications.

For the overdetermined case $m > n$, we can perform a rectangular LU factorization $PAQ = LU$ of the $m \times n$ matrix A, where L is a lower trapezoidal $m \times n$ matrix, U is an upper triangular $n \times n$ matrix, P is a permutation matrix obtained with threshold pivoting (swap of rows so that the lower trapezoidal L has $1's$ on the diagonal) and Q is a permutation matrix obtained from column pivoting to avoid fill-in in the L and U factors. As L tends to be better conditioned than A [24, p. 231], it was explored in [14] transforming Equation (2) to

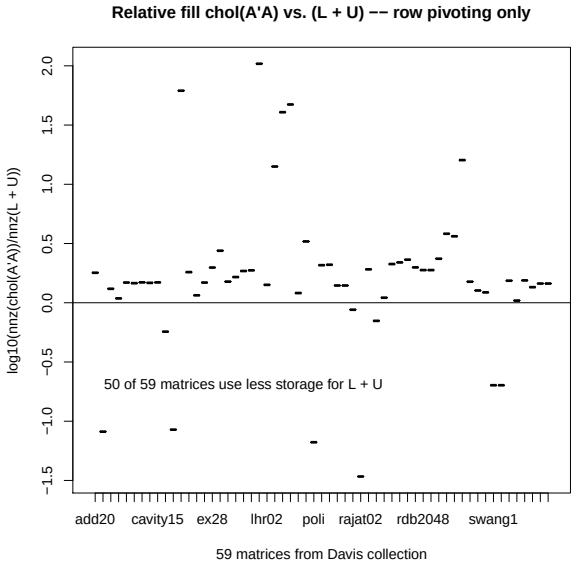
$$L^T Ly = L^T c, \tag{3}$$

with $y = UQ^T x$, and $c = Pb$, when U is nonsingular. As illustrated in [14], L is usually better conditioned than A and fewer iterations are required for the L iteration.

Regarding the storage issues, we plot in Figure 1 the storage required for an LU factorization of A (with partial pivoting) and for the Cholesky factorization of $A^T A$ for matrices from the Davis collection[1] (modified such that most of these matrices had ten percent more rows than columns). We observe that the storage of L and U for a nearly square rectangular matrix is usually less than that of the Cholesky factor of $A^T A$ (or equivalently, in exact arithmetic, the R factor from the QR factorization of A). In fact, if every set of k columns of A

[1] `http://www.cise.ufl.edu/research/sparse/matrices/`

has nonzero entries in at least $k+1$ rows, for all $1 \leq k \leq n-1$ (i.e. A is "strong Hall" [6, pp. 83-84]), the storage required by U is always bounded by that of R (see [8, 9, 10]). Moreover, if the "strong Hall" property holds, the storage required for L is bounded by that needed for an orthogonal Householder derived Q, $QR = AP$ [9, 10].



..

Figure 1: Storage for LU factorization of A vs Cholesky factorization of $A^T A$.

In this paper, we consider a partition of the lower trapezoidal matrix L as

$$L = \left[\frac{L_1}{L_2}\right]$$

where L_1 is $n \times n$ lower triangular and L_2 is $(m-n) \times n$. We explore iterating with the normal equations in terms of

$$F = LL_1^{-1} = \left[\frac{I_n}{L_2 L_1^{-1}}\right].$$

Note that a partition of $A = \left[\frac{A_1}{A_2}\right]$ (instead of L here) is considered in [5] where an LU factorization of A_1 is used to precondition the normal equations. In [1], a partition $A = \left[\frac{A_1}{A_2}\right]$ is also used (where A_1 is a set of basis obtained with an LU factorization), and the augmented system is transformed into an equivalent symmetric quasi-definite system.

We will show that iterating on LL_1^{-1} results in faster convergence than with L, due to a better conditioning of LL_1^{-1} and does not require additional storage.

Moreover, since a bound on the condition number of LL_1^{-1} can be explicitly computed, this can indicate if further preconditioning is required.

The plan of this paper is as follows. Section 2 motivates the use of LL_1^{-1} preconditioning. Section 3 describes the experimental framework including the test matrices and some implementation issues. Section 4 presents numerical results on the test matrices. Section 5 concludes the paper.

2 LL_1^{-1} preconditioning

In this section we motivate our choice to precondition the conjugate gradient solution of the normal equations by using the matrix $F = LL_1^{-1}$ obtained from the partition of the rectangular matrix L as

$$L = \left[\frac{L_1}{L_2}\right]$$

where L_1 is square and lower triangular and L_2 is $(m-n) \times n$ so that

$$LL_1^{-1} = \left[\frac{I_n}{L_2 L_1^{-1}}\right] = \left[\frac{I_n}{C}\right]$$

with $C = L_2 L_1^{-1}$. Then Equation (3) becomes

$$[I_n | C^T] \left[\frac{I_n}{C}\right] z = [I_n | C^T] c,$$

or equivalently,

$$F^T F z = F^T c, \tag{4}$$

with $z = L_1 U Q^T x$, and $c = Pb$. Since L_1 and U are respectively lower and upper triangular, x is computed from z using forward and backward substitutions. For matrices with only a few more rows than columns, explicit computation of $C = L_2 L_1^{-1}$ may be worthwhile, being easily performed in parallel, and allowing easy parallelization of the iteration. If we do not explicitly compute C, there is no additional storage beyond that required for L.

In this paper, we solve Equation (4) using the `lsqr` algorithm [20], with the iteration matrix $F = LL_1^{-1}$. The iteration requires multiplying both by F and F^T. We compute

$$v = Fu = \left[\frac{I_n}{L_2 L_1^{-1}}\right] u = \left[\frac{u}{L_2 L_1^{-1} u}\right]$$

entailing a forward triangular solve with L_1. Similarly, we compute

$$F^T v = [I_n | L_1^{-T} L_2^T] v = v + L_1^{-T}(L_2^T v),$$

which requires a triangular backsolve. The multiplications by F and F^T take almost the same number of flops as multiplications by L, but the backward and

forward solves are likely to be harder to parallelize than multiplication by L. In the remainder of this paper, the `lsqr` algorithm where we iterate with

$$F = LL_1^{-1} = \begin{bmatrix} I \\ \hline C \end{bmatrix}$$

instead of A is referred to as `lsqrLinvL` and is given in Algorithm 1. The adapted `lsqr` algorithm which is called in `lsqrLinvL` is detailed in Algorithm 2.

Algorithm 1 `lsqrLinvL`

function $[x, its] = lsqrLinvL(A, b, tol, maxit)$
% A is an input matrix of m rows and n columns
% b is an input m-vector
% tol (convergence tolerance) is a positive input scalar
% $maxit$ is the maximal number of iterations
% output x is an n vector to minimize $\|Ax - b\|_2$
% its is the actual number of iterations performed
[L,U,prow,qcol] ← lu(A,'vector');
r ← permute(b,prow);
[x,its] ← lsqr(L,r,tol,maxit);
if ($its \geq maxit$),"maxits exceeded", end
x ← U \ x;
x ← ipermute(qcol,x);
End

Figure 2 compares the condition numbers of A, L, and C for some rectangular matrices small enough that we can explicitly compute the SVD in `Matlab` and thus the condition number as the ratio of largest and smallest singular values. These matrices were created in a similar fashion to those discussed in the next section, with the number of columns at most 5,000. Most had ten per cent more rows than columns.

We have observed that $F = LL_1^{-1}$ is usually better conditioned than L. Moreover we can easily estimate $cond_2(F)$ even for large matrices. Consider that

$$F^T F = [I_n | C^T] \begin{bmatrix} I_n \\ \hline C \end{bmatrix} = I_n + C^T C,$$

then we have

$$\sigma_i^2(F) = \lambda_i(F^T F) = 1 + \lambda_i(C^T C) = 1 + \sigma_i^2(C),$$

where $\sigma_i(\cdot)$ and $\lambda_i(\cdot)$ are the singular values and eigenvalues of the considered matrices. If λ_{max}, λ_{min} denote the largest and smallest eigenvalues of $C^T C$, and since $C^T C$ is real and symmetric, then we have $0 \leq \lambda_{min} \leq \lambda_{max}$. The condition number of $F = LL_1^{-1}$ can be expressed as

$$cond_2(F) = \sqrt{\frac{\lambda_{max} + 1}{\lambda_{min} + 1}}.$$

Algorithm 2 `lsqr`: adapted from Paige and Saunders, 1982

function $[x, its] = lsqr(L, b, tol, maxit)$
% output x is an n vector to minimize $\|Ax - b\|_2$
% its is the actual number of iterations performed

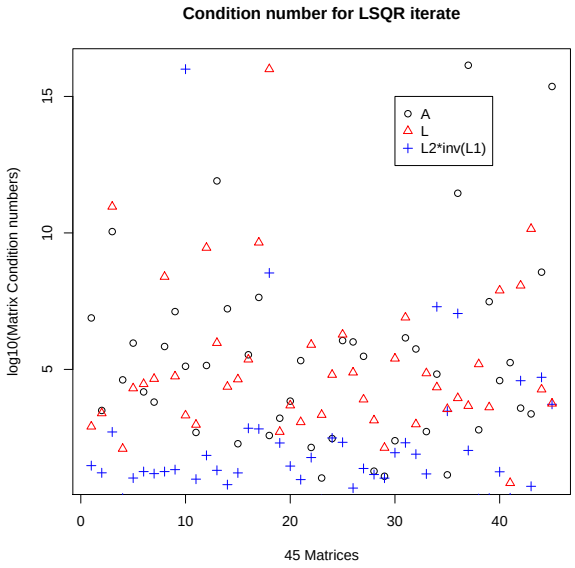
% Initialization
$\beta \leftarrow \|b\|_2; \phi_b \leftarrow \beta; b \leftarrow b/\beta;$
$x \leftarrow 0_n; \delta \leftarrow 1; its \leftarrow 0;$
$[m, n] \leftarrow$size(L);
$L_1 \leftarrow L$[1:n,1:n];
$L_2 \leftarrow L$[n+1:m,1:n];
$v_1 \leftarrow L_2' u$[n+1:m];
$v_1 \leftarrow L_1' \backslash u2$;
$v \leftarrow u$[1:n]$+v_1$;
$\nu \leftarrow \|v\|_2$;
$v \leftarrow v/\nu$;
$w \leftarrow v$;
$\phi_a \leftarrow 1$;
% Lanczos iteration
While $((|\delta| > tol)$ and $(its < maxit))$
% Continue the bidiagonalization
$u_1 \leftarrow L_1 \backslash v$;
$u \leftarrow [v; L_2 u_1] - \nu u$;
$\beta \leftarrow \|u\|_2$;
$u \leftarrow u/\beta$;
$v_1 \leftarrow L_2' u$[n+1:m];
$v_1 \leftarrow L_1' \backslash v_1$;
$v_1 \leftarrow u$[1:n]$+v_1$;
$v \leftarrow v_1 - \beta v$;
$\nu \leftarrow \|v\|_2$;
$v \leftarrow v/\nu$;
% Construct and apply next orthog. transformation
$\rho \leftarrow \sqrt{\rho_b^2 + \beta^2}$
$c \leftarrow \rho_b/\rho$;
$s \leftarrow \beta/\rho$;
$\theta \leftarrow s\nu$;
$\rho_b \leftarrow -c\nu$;
$\phi \leftarrow c\phi_b$;
$\phi_b \leftarrow s\phi_b$;
% update x, w
$\delta \leftarrow \phi/\rho$;
$\phi_a = \sqrt{|\phi\delta|}$;
$x \leftarrow x + \delta w$;
$w \leftarrow v - (\theta/\rho)w$;
$its \leftarrow its + 1$;
End
$x \leftarrow L_1 \backslash x$;
End

Then we get

$$cond_2(F) \leq \sqrt{\lambda_{max} + 1}.$$

In the case where $m - n < n$, we have $\lambda_{min} = 0$, so we get an equality

$$cond_2(F) = \sqrt{\lambda_{max} + 1}.$$



..

Figure 2: Comparison of the conditioning of various `lsqr` iteration matrices.

We can estimate λ_{max} with a few iterations of the symmetric power method [11, p. 406] applied to $C^T C$. A few iterations suffice for a $cond_2(F)$ estimate. So having computed L, we can easily decide whether further preconditioning is needed. For the LL_1^{-1} preconditioning, the above analysis shows that we can estimate $cond_2(F)$ for the partial pivoting case $|l_{ij}| \leq 1, i > j$, and also for the threshold pivoting case where $|l_{ij}|$ is not bounded by 1. Inexpensive estimation of $cond_2(F)$ does need L_1 to be lower triangular and nonsingular (so that the symmetric power method can use forward and backward solves on each iteration).

3 Experimental framework

In this section, we define the test set for our experiments and the implementation choices based on the Julia programming language [15].

The numerical experiments are carried out with the `lsqr` algorithm where we iterate with

$$F = LL_1^{-1} = \begin{bmatrix} I \\ \hline C \end{bmatrix},$$

using Algorithms 1 and 2. In general, neither C nor L_1^{-1} are explicitly computed (except in some cases described in Section 4). The $LU = PAQ$ factorization is performed in full precision and with both column and row permutations, using the UMFPACK routine [7]. UMFPACK chooses the Q column permutations to reduce the matrix fill-in and the P row permutations to reduce the Gaussian multipliers size (threshold pivoting). For our numerical experiments, we get L such that the l_{ij} entries satisfy

$$l_{ii} = 1 \text{ for } i = j,\ l_{ij} = 0 \text{ for } i < j,\ |l_{ij}| < 10 \text{ for } i > j$$

To construct our test problems, we took 235 matrices from the Davis collection, with most of the matrices having less than a maximum of twenty thousand rows or columns. For matrices with more columns than rows, we transposed. Since most matrices were square, we augmented the matrices with one hundred additional rows (or, for square matrices with fewer than $1,000$ columns, we increased the number of rows by ten per cent). The additional rows were perturbed copies of randomly selected rows of the original square matrix. The randomly perturbed entries were the original entries times a factor in the range [0.9, 1.1], i.e.

$$a_{ij_{new}} = (1 + 0.1\tau_{ij})a_{ij},$$

where τ_{ij} was randomly selected from a uniform distribution on [-1,1]. The minimum number of columns n was 25, the minimum of $m + n$ was 182, the maximum of $m + n$ was 39532. The average number of columns was around 5600. The average number of rows was around 7230.

We performed numerical experiments with the `Julia` language which is a high-level language for numerical computing and provides good performance (close to that of a C code). `Julia` codes require fairly minimal changes from `Matlab` or `Octave` codes, while being much faster, making `Julia` very suitable for prototyping HPC algorithms. Wherever there are loops, `Julia` runs much faster[2]. For example, for sparse matrix multiplications [18], a sparse matrix-vector multiplication Ax in C (`gcc` in Linux) and `Julia` (compiled with the same `gcc`), the C version was less than ten per cent faster, with `Octave` much slower. Moreover converting code from `Octave` to `Julia` is much easier than converting to C or Fortran.

Some of the issues we encountered are as follows: In some instances, we had to be aware that in the statement $A = B$ for `Julia` arrays, the copy of B is "shallow" i.e., no new copy of A is produced, so changes in A also change B. Though `Matlab`, `Octave` and `Julia` all use sparse LU and QR factorizations based on SuiteSparse [7], the `Julia` `lufact` and `qrfact` functions do not offer user options. The `lufact` wrapper that we used for UMFPACK is the default in `Julia`, using a threshold pivoting of 10, and both row and column pivoting. To disable row scaling, we had to find what line to change in the `lufact` wrapper for UMFPACK, else we would have been solving a weighted least squares problem. Also, using `Octave`, we could obtain a factorization of the form $QR = \begin{bmatrix} I_n \\ C_{drop} \end{bmatrix} P$

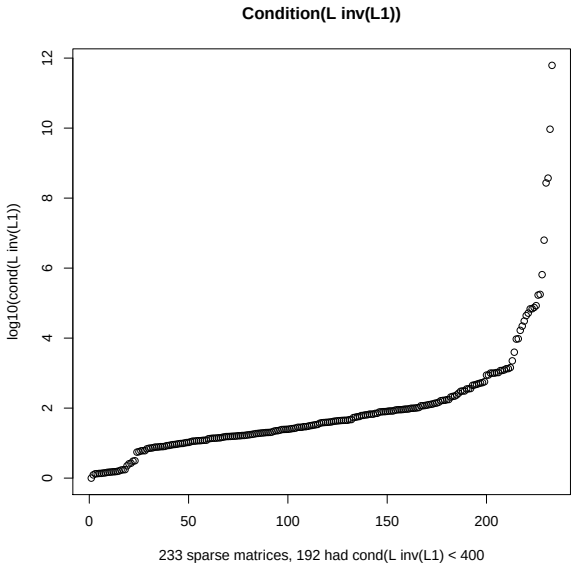
[2] See `https://julialang.org/benchmarks/` for Julia benchmarks

(C_{drop} is the matrix C on which we apply a drop tolerance, see Section 4), where we could explicitly obtain the orthogonal matrix Q, upper triangular matrix R, and column permutation P. Alas the `qrfact` function in `Julia` currently returns a least squares solution, but does not offer options to return R, Q, or P. In exact arithmetic, $R^TR = R^TQ^TQR$, so in this sense the storage for R from the Cholesky factorization (`cholmod` routine from SuiteSparse, as wrapped in `Julia`) is the same as the storage for R from the QR factorization. We used this equivalence as a work-around for the `Julia qrfact`, so long as $\left[\frac{I_n}{C_{drop}}\right]$ was not so poorly conditioned that the Cholesky factorization failed.

4 Numerical experiments

We performed `julia lufact` $LU = PAQ$ decompositions for the 235 matrices of the test set defined in Section 3.

We plot in Figure 3 the condition number of LL_1^{-1} for these 235 matrices. We observe that LL_1^{-1} tends to be acceptably well conditioned (192 of 235 matrices had $cond_2(LL_1^{-1}) < 400$). As noted in Section 2, the condition number is easily computed, using the symmetric power method. Of the 43 more poorly conditioned matrices, 29 were among the 47 matrices with more than one hundred rows in C (i.e. correspond to "tall" A matrices), indicating that LL_1^{-1} preconditioning is appropriate for near square matrices.

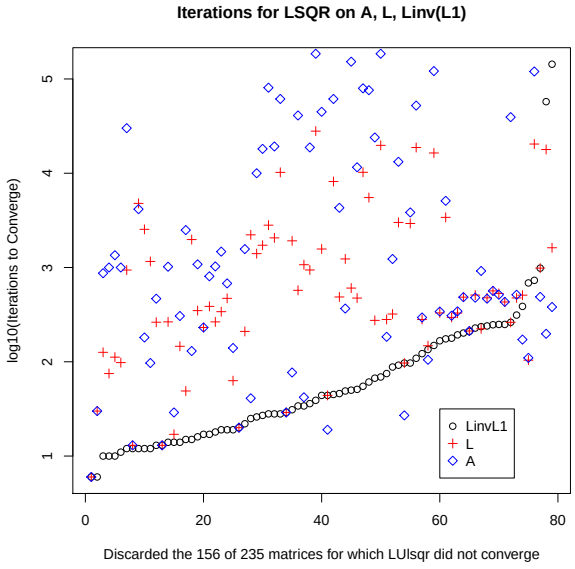


..

Figure 3: Condition number of LL_1^{-1} for the 235 test matrices.

We plot in Figure 4 the iteration count obtained using test matrices for which

`lsqrLU` converged in n iterations (iteration matrix L). For these matrices, we observe that the convergence is significantly accelerated by iterating on LL_1^{-1}.



..

Figure 4: Convergence for 79 matrices for which `lsqr` iteration with L converged in n iterations.

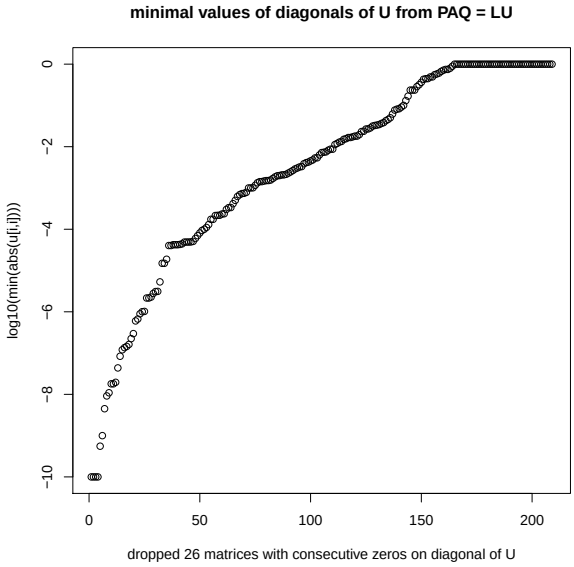
After the `lsqrLinvL` algorithm has converged, a final solve with U is needed. However we can observe in Figure 5 that for some matrices U may have near zero diagonal elements. If we consider the normalized (by column norm) size of the diagonal elements, 30 of the 235 matrices had diagonal values u_{ii} such that $|u_{ii}|/max_i|u_{ij}| < 10^{-10}$, so for these matrices neither L nor LL_1^{-1} preconditioning was judged to be feasible.

We discard 30 matrices because $|u_{ii}|/max_i|u_{ij}| < 10^{-10}$. For the remaining 205 of the matrices U, we obtain, as explained in Section 2, an upper bound $K = \sqrt{\lambda_{max} + 1}$ of $cond_2(LL_1^{-1})$, where λ_{max} is the largest eigenvalue of C^TC (computed using the symmetric power method.) Then we select the `lsqr` iteration matrix by the size of K, as follows:

- If $K < 400$, we use `lsqrLinvL` (`lsqr` algorithm iterating with LL_1^{-1}). This situation concerns 173 of the 205 matrices.

- If $400 < K < 10^8$, we explicity compute C and we perform a drop tolerance on C by zeroing elements c_{ij} such that $|c_{ij}| < |c_i|/K^{.25}$, where c_i denotes the maximal entry of the ith column of C, leading to the matrix C_{drop}[3].

[3] Taking $K^{.25}$ means for example that if $K = 10^4$, then all entries less than 1/10 of the column max in absolute value are discarded. Taking a constant less than .25 would reduce fill, but might retard convergence.



..

Figure 5: Size of the diagonal elements of U.

We then compute the Cholesky factor R satisfying

$$R^T R = [I_n | C_{drop}^T] \begin{bmatrix} I_n \\ \hline C_{drop} \end{bmatrix},$$

and we use `lsqr` iteration with $LL_1^{-1}R^{-1}$ (28 of 205 matrices).

- If $K > 10^8$, we have $cond_2(F^T F) > 10^{16}$. Then the Cholesky decomposition in double precision arithmetic gives a floating point error. In this case, we try `lsqr` iteration with L (4 of 205 matrices).

The algorithm where we choose, as mentioned above, the iteration matrix according to the size of K will be referred to as the "hybrid" `lsqrLinvL` algorithm in the remainder of this paper. In our experiments, we iterated with a stopping criterion of $|\delta| < 10^{-10}$ in Algorithm 2. The results[4] are summarized in Table 1. In this table, the number of iterations to convergence is expressed as a multiple of n (number of columns of A) and the storage required is expressed as a multiple of the nonzero entries of A. As expected, the case $400 < K < 10^8$ requires more storage ($\times 13$ in average) due to the use of R but decreases the number of iterations by a factor 2.3 in average. The decreased storage for the largest condition numbers is the average storage for $A + L + U$, which is the same storage required for $K < 400$, but with the average computed over a much smaller set.

[4]The results are stored in a comma separated file with 235 rows and 19 numbers per line, loadable as an R language data frame.

As we can see in Figure 6, when using a drop tolerance on C and then computing $R^T R = F_{drop}^T F_{drop}$ (where $F_{drop} = \left[\frac{I_n}{C_{drop}}\right]$), the resulting R factor is relatively dense matrix (R has relatively few entries when A had only 25 columns and many rows).

Condition Number	Iteration Matrix	Cases of 205	Iterations Multiple of n	Nonzeros Multiple $nnz(A)$
$K < 400$	LL_1^{-1}	173	.32 n	12.6
$400 < K < 10^8$	$LL_1^{-1}R^{-1}$	28	.138 n	165
$10^8 < K$	L	4	.6 n	6.52

Table 1: Storage and iterations for 205 matrices using the hybrid `lsqrLinvL` algorithm. The last two columns give averages over 173, 28, and 4 test matrices, respectively.

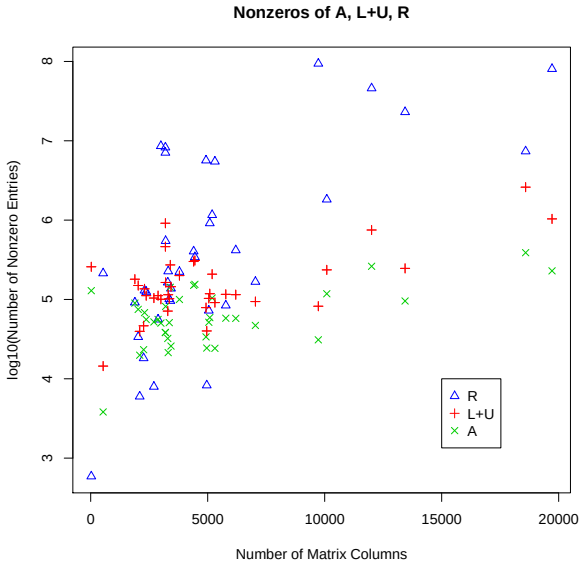


Figure 6: Fill-in for the Cholesky factorization of $F_{drop}^T F_{drop}$.

For each of the 205 matrices, the hybrid `lsqrLinvL` algorithm converged to a solution. For 152 of 205 matrices, we have

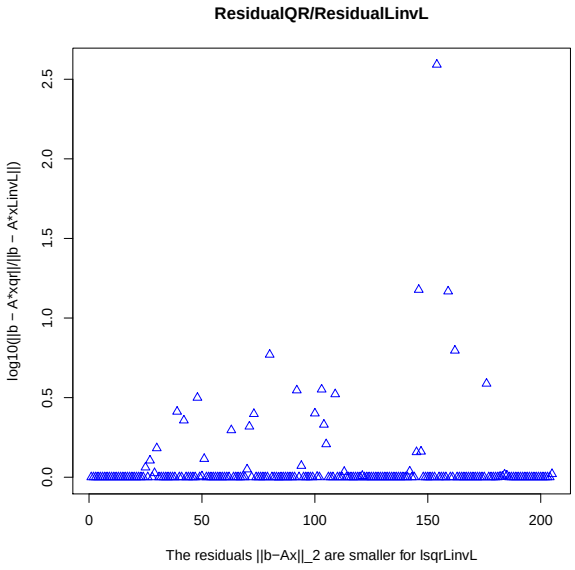
$$\frac{\|x_{LinvL} - x_{qr}\|_2}{\|x_{LinvL}\|_2} < 10^{-8},$$

where x_{LinvL} and x_{qr} are the solution computed using the hybrid `lsqrLinvL`

and `SparseQR` algorithms, respectively. When the solutions differ, which of `SparseQR` and `lsqrLinvL` is better in terms of finding x with a lower LLS residual? Figure 7 plots the log of

$$\frac{\|b - Ax_{qr}\|_2}{\|b - Ax_{LinvL}\|_2}.$$

This graph shows that in most cases, the x values do not differ significantly, so the ratio is 1 and its log is zero. But when x differs (i.e. the log of the ratio is larger than zero), then residuals are smaller for the hybrid `lsqrLinvL` algorithm.



..

Figure 7: Comparison between LLS residuals for x_{qr} and x_{LinvL}.

Before claiming victory of `lsqrLinvL` over `SparseQR`, we should note that for the 53 solutions that differed significantly, $\|x\|_2$ tended to be larger for the hybrid `lsqrLinvL` method than for `SparseQR` (42 of 53 cases). Then users concerned with $\|x\|_2$ might well prefer `SparseQR`.

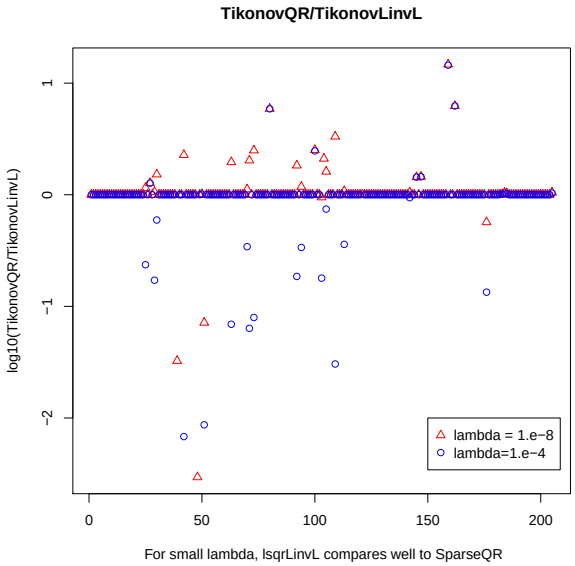
Balancing concerns for size of the residual vs. size of the solution can be thought of as "regularization". For example, the Tikhonov regularization (see, e.g., [13, p. 193]) chooses x to minimize the function

$$T(x) = \sqrt{\|Ax - b\|_2^2 + \lambda^2 \|x\|_2^2}.$$

Figure 8 plots the log of the ratio

$$\sqrt{\frac{\|b - Ax_{qr}\|_2^2 + \lambda^2 \|x_{qr}\|_2^2}{\|b - Ax_{LinvL}\|_2^2 + \lambda^2 \|x_{LinvL}\|_2^2}},$$

with the Tikhonov regularization parameters $\lambda = 10^{-8}$ and $\lambda = 10^{-4}$. We recall that for most matrices, $\|x_{qr} - x_{LinvL}\|$ is quite small. In these cases, residuals and solution norms match and the ratio is close to one. In some cases, residuals and norms do not match, then the ratios $T(x_{qr})/T(x_{LinvL})$ are larger than one (points greater than zero on the plot) indicating that $T(x)$ is larger for `SparseQR` than for `lsqrLinvL` solutions. For $\lambda = 10^{-8}$, `lsqrLinvL` usually gives a solution with a smaller $T(x)$. For $\lambda = 10^{-4}$, the $T(x_{qr})$ quantity is usually smaller.



..

Figure 8: Comparison between Tikhonov regularizations of x_{qr} and x_{LinvL}.

5 Conclusion and future work

When the number of equations is not much larger than the number of variables, LU factorization usually allows iterative least squares solves with less storage than QR factorization or Cholesky factorization of the normal equations. Moreover, when the number of equations is not much more than the number of unknowns, iterating in the `lsqr` algorithm using LL_1^{-1} usually requires relatively few iterations for the solution of a least squares problem. A main limitation is that the method fails when U has near zero diagonal elements, i.e., when the problem is numerically not overdetermined. The near singularity of U is typically not discovered until the relatively expensive LU factorization has been attempted.

As future work, a possible way to further reduce the required storage is to perform the original LU factorization in a lower precision (or to use an incomplete factorization of A). The iteration is then with $AU^{-1}L_1^{-1}$, possibly with a

preliminary cheaper iteration with $\left[\frac{I}{L_2 L_1^{-1}}\right]$. That iteration can be compared to the RIF preconditioner [2], which is a sparse Cholesky-like factorization of $A^T A$. Other future tasks will be to address larger problems using the distributed computation possibilities provided by `Julia`, and to analyze the resulting execution time and speedup.

Acknowledgments

We would like to thank Iain Duff, Keichi Morikuni, Dominique Orban, and Michael Saunders for advice and encouragement. We also thank the `Julia` developper Andreas Noack for his help. Thanks also to North Carolina State University for use of the Henry2 HPC cluster.

References

[1] M. Arioli and I S. Duff. Preconditioning linear least-squares problems by identifying a basis matrix. *SIAM J. Scientific Computing*, 37(5):S544–S561, 2015.

[2] M. Benzi and M. Tuma. A robust incomplete factorization preconditioner for positive definite matrices. *Numerical Linear Algebra with Applications*, 10:385–400, 2003.

[3] Å. Björck. *Numerical Methods for Least Squares Problems*. SIAM, Philadelphia, 1996.

[4] A. Björck and I. S. Duff. A direct method for the solution of sparse linear least squares problems. *Linear Algebra Appl.*, 34:43–67, 1980.

[5] A. Björck and J.Y. Yuan. Preconditioners for least squares problems by *LU* factorization. *Electronic Transactions on Numerical Analysis*, 8:26–35, 1999.

[6] T. Davis. *Direct Methods for Sparse Linear Systems*. SIAM, Philadelphia, 2006.

[7] T. A. Davis. UMFPACK user guide. http://www.suitesparse.com, 2016. Version 5.7.6.

[8] A. George and E. Ng. On row and column orderings for sparse least squares problems. *SIAM, J. Numer. Anal.*, 20:326–344, 1983.

[9] J.R. Gilbert. Predicting structure in sparse matrix computations. *SIAM, J. Matrix Anal. Appl.*, 15:62–79, 1994.

[10] J.R Gilbert and E.G. Ng. Predicting structure in nonsymmetric sparse matrix factorizatons. In *Graph Theory and Sparse Matrix Computations, IMA Vol. Math. Appl.*, pages 107–139, New York, 1993. Sringer Verlag.

[11] G. H. Golub and C. F. Van Loan. *Matrix Computations*. The Johns Hopkins University Press, Baltimore, 1996. Third edition.

[12] N. Gould and J. Scott. The state-of-the-art of preconditioners for sparse linear least-squares problems. *ACM Transactions on Mathematical Software*, 43(4):36:1–36:35, 2017.

[13] P. C. Hansen, V. Pereyra, and G. Scherer. *Least Squares Data Fitting with Applications*. The Johns Hopkins University Press, Baltimore, 2013.

[14] G. W. Howell and M. Baboulin. LU preconditioning for overdetermined sparse least squares problems. In *Parallel Processing and Applied Mathematics - 11th International Conference, PPAM 2015, Krakow, Poland, September 6-9, 2015. Revised Selected Papers, Part I*, pages 128–137, Heidelberg, 2015. Springer.

[15] S. Karpinski J. Bezanson, A. Edelman and V. B. Shah. Julia: A fresh approach to numerical computing. *SIAM Review*, 59:65–98, 2017.

[16] X. S. Li. An overview of SuperLU: Algorithms, implementation, and user interface. *ACM Transactions on Mathematical Software*, 31(3):302–325, 2005.

[17] X. S. Li and J. W. Demmel. SuperLU_DIST: a scalable distributed-memory sparse direct solver for unsymmetric linear systems. *ACM Transactions on Mathematical Software*, 29(9):110–140, 2003.

[18] D. Orban. Private communication, 2017.

[19] J.-Y. L'Excellent P. R. Amestoy, A. Guermouche and S. Pralet. Hybrid scheduling for the parallel solution of linear systems. *Parallel Computing*, 32(2):136–156, 2006.

[20] C. Paige and M. Saunders. An algorithm for sparse linear equations and sparse least squares. *ACM Trans. on Math. Software*, 8(1):43–71, 1982.

[21] G. Peters and J. H. Wilkinson. The least squares problem and pseudo-inverses. *Computing J.*, 13:309–316, 1970.

[22] O. Schenk and K. Gärtner. PARDISO User Guide. http://www.pardiso-project.org/manual/manual.pdf, 2014.

[23] J. Scott and M. Tuma. Preconditioning of linear least squares by robust incomplete factorization for implicitly held normal equations. *SIAM J. Scientific Computing*, 38(6):C603–C623, 2016.

[24] G. W. Stewart. *Matrix Algorithms - Volume I: Basic Decompositions*. SIAM, Philadelphia, 1998.