# LU Preconditioning for Overdetermined Sparse Least Squares Problems

Gary W. Howell[1] and Marc Baboulin[2]

[1] North Carolina State University, USA
gwhowell@ncsu.edu
[2] Université Paris-Sud and Inria, France
marc.baboulin@lri.fr

**Abstract.** We investigate how to use an $LU$ factorization with the classical `lsqr` routine for solving overdetermined sparse least squares problems. Usually $L$ is much better conditioned than $A$ and iterating with $L$ instead of $A$ results in faster convergence. When a runtime test indicates that $L$ is not sufficiently well-conditioned, a partial orthogonalization of $L$ accelerates the convergence. Numerical experiments illustrate the good behavior of our algorithm in terms of storage and convergence.

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

## 1 Introduction to *LU* preconditioning for least squares

Linear least squares (LLS) problems arise when the number of linear equations is not equal to the number of unknown parameters. For example LLS problems occur in many parameter estimation and constrained optimization problems [2, 22]. Commonly, nonlinear least squares problems are solved via algorithms which solve sparse linear least squares problems at each step [14]. Levenberg-Marquardt algorithms [21] are one example.

Here 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 $b \in \mathbb{R}^m$. When $A$ is sparse, direct methods based on $QR$ factorization or the normal equations are not always suitable because the $R$ factor or $A^TA$ can be dense. A common iterative method to find the least squares solution $x$ is to solve the normal equations

$$A^TAx = A^Tb, \tag{2}$$

by applying the conjugate gradient (CG) algorithm to $A^TA$. In this case the matrix $A^TA$ does not need to be explicitly formed, avoiding possible fill-in in the formation of $A^TA$. As with other sparse linear systems, preconditioning techniques based on incomplete factorizations can improve convergence. One method to precondition the normal equations (2) is to perform an incomplete Cholesky decomposition of $A^TA$ (e.g., RIF preconditioner [5]).

When $A^TA$ and its Cholesky factorization are denser than $A$, it is natural to wonder if the $LU$ factorization of $A$ can be used in solving the least squares problem. In this paper we use an $LU$ factorization of the rectangular matrix $A = \begin{pmatrix} A_1 \\ A_2 \end{pmatrix}$ where $L$ is unit lower trapezoidal and $U$ is upper triangular. Following [11, p. 102], such a factorization exists when $A_1$ is non singular, which for $A$ full rank, can be obtained by permuting rows of $A$. For the nonpivoting case, the normal equations (2) become

$$L^TLy = c, \tag{3}$$

with $c = L^Tb, Ux = y$, and we can apply CG iterations on (3).

Least squares solution using $LU$ factorization has been explored by several authors. Peters and Wilkinson [24] and Björck and Duff [7] give direct methods. This work follows Björck and Yuan [8] using conjugate gradient methods based on $LU$ factorization, an approach worth revisiting because of the recent progress in sparse $LU$ factorization. In contrast to the SP1, SP2, and SP3 algorithms of [8], the lsqrLU algorithm presented here uses a lower trapezoidal $L$ returned from a direct solver package, easing implementation. Where Saunders, cited in [8], iterated with $U^{-1}A$, we iterate directly with $L$ from $LU = PA$, an approach amenable to parallel implementation and to further preconditioning, if necessary. Here we use the `Matlab` and `Octave` fast sparse $LU$ factorizations built on Davis' UMFPACK package [10]. Because other direct solver packages (e.g., [1, 19, 20, 26]) offer scalable sparse $LU$ factorizations, it appears likely that the algorithm used here can also be used to solve larger problems.

The rate of linear convergence for CG iterations on the normal equations is (see [6, p. 289])

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

where $\kappa = \sqrt{\text{cond}(A^TA)} = \text{cond}(A)$ and $\text{cond}(A)$ denotes the 2-norm condition number of $A$ (ratio of largest and smallest singular values of $A$) . CG methods work acceptably well when $\text{cond}(A) = O(100)$, but converge very slowly, if at all, when $\text{cond}(A) > O(1000)$.

In our experiments, $L$ is often much better conditioned than $A$, so convergence of the CG method is relatively rapid. Moreover, the total number of nonzeros in $L$ and $U$ is usually less than in the sparse Cholesky factorization of $A^TA$. Successful iteration with $L$ depends on the good conditioning of $L$, often requiring partial pivoting in the factorization. For parallel computations, pivoting for the $LU$ decomposition may be expensive or not available, so we may also need to further precondition the problem. In the test problems here, partial orthogonalization of $L$ (described in the next paragraph) is an effective strategy.

If a condition estimate indicates that $L$ is not sufficiently well conditioned for fast convergence, it is natural to use a drop tolerance on $L$ to get $L_{drop}$. If $R$ denotes the $R$-factor in the QR decomposition $QR = L_{drop}$, then we expect $LR^{-1}$ to be better conditioned than $L$, allowing faster convergence for the conjugate gradient iteration. A partial orthogonalization using a drop tolerance for $A$ was proposed in [17] and developed as a multilevel algorithm by Li and Saad [18].

For the test set of 51 full rank matrices described in the next section, iteration with $L$ was sufficient to get convergence (using at most $n$ iterations, relative error less than $10^{-6}$) in all but three cases. In all 51 cases, partial orthogonalization of $L$ gave convergence, typically in fewer iterations.

An inexpensive estimate of the conditioning [13, 15] of square triangular matrices allows the condition estimate to be used at runtime to determine whether partial orthogonalization is needed. If partial orthogonalization is chosen, the condition estimate is used to control the drop tolerance. Computing a drop tolerance for $L$ from the estimated $cond(L(1:n,1:n))$ gives a more reliable algorithm than computing a drop tolerance for $A$ from the estimated $cond(A(1:n,1:n))$ (which also requires an $LU$ factorization). Numerical experiments show the robustness of the least squares algorithm combining $LU$ factorization and possibly (depending on a runtime estimate of $L$ conditioning) partial orthogonalization of $L$

The remainder of this paper is organized as follows. Section 2 describes a set of test matrices, the $LU$ factorization used, the observed conditioning of $L$, and "fill" for $L$ and $U$. Section 3 presents an algorithm to precondition least squares using the $LU$ factorization of $A$, and shows convergence results on the test matrices, comparing the `lsqr` algorithm to the `lsqr` algorithm preconditioned by $LU$. Section 4 shows a way to use partial orthogonalization of $L$ to improve convergence and presents results of numeric experiments on the same set of matrices. Section 5 has concluding remarks.

## 2 *LU* decomposition on rectangular matrices and a set of test matrices

In this paper, the $LU$ decomposition is computed using `Octave` [4] and `Matlab`. Both frameworks call UMFPACK [10]. While both partial (row) pivoting $L_pU_p = PA$ and row and column pivoting algorithms $L_qU_q = PAQ$ are implemented, row pivoting is more commonly available in parallel packages, [19], [12] and thus a logical choice for test problems. $L$ was computed with maximal element row pivoting, so that $L$ is unit lower trapezoidal.

As a numerical test bed, we considered a set of 59 matrices adapted from the University of Florida collection [9]. We took most unsymmetric matrices larger than 500 and of maximal dimension at most 5000, small enough that we could explicitly compute the singular values, and could use a $QR$ algorithm to compare solutions of the LLS problem. We wanted rectangular matrices with more rows than columns, so transposed if necessary. The "more column than row" matrices are mainly from linear programming problems, for which the transposed (dual) problem requires a least squares solution. So for these matrices the least squares problem has practical interest. For square matrices, we randomly selected 10 percent of rows, duplicating them and appending them to the end of the matrix, randomly perturbing each nonzero entry by at most ten per cent.

The 51 full rank matrices were derived from add20, bwm2000, cage9, cavity11, cavity12, cavity13, cavity14, cavity15, Chebyshev3, circuit_2, crew1, ex24, ex26, ex27, ex28, ex29, ex31, heart1, heart3, lhr02, olm5000, orsreg_1, piston, poli, psmigr_2, psmigr_3, raefsky1, raefsky2, rajat02, rajat04, rajat05, rajat11, rajat12, rajat14, rajat19, rbsa480, rdb2048, rdb2048_noL, rdb5000, rdist1, rdist2, sherman5, shermanACa, swang1,swang2, thermal, tols4000, utm3060, viscoplastic1, wang1, and wang2. By limiting matrix size to around 5000, we were able to compute singular values and thus the condition number as the ratio of largest to smallest singular values. Eight matrices, derived from adder_dcop, extr1, Kohonen, lhr04, lns_3937, raefsky6, SciMet, and sherman3, were rank deficient, with computed condition numbers greater than $1.e16$. For the eight rank deficient matrices, `Matlab` QR could not compute least squares solutions. Dropping the rank deficient matrices from the set of 59 resulted in the set of 51 matrices used as a test collection for comparing convergence.

For the collection of 59 matrices (see Figure 1), we computed the 2-norm condition number of $L_p$ from $PA = L_pU_p$ as the ratio of largest to smallest singular value. By rounding larger condition numbers down to $1.e16$, and computing the multiplicative mean by averaging the logarithms of $cond(L_p)/cond(A)$, we observed that the condition numbers $L_p$ are on average 4000 times smaller than the condition numbers of $A$. In Figure 1, the better conditioning of $L_p$ is indicated by the closer clustering of maximal and minimal singular values to 1.

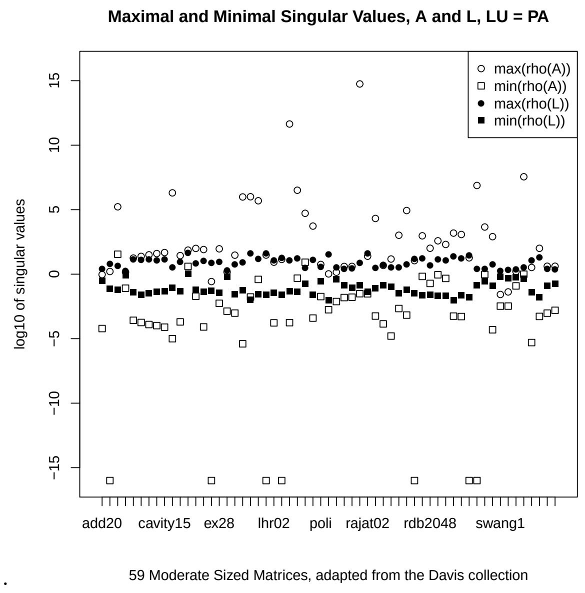
The results for the $PAQ = L_qU_q$ case (row and column pivoting with multipliers bounded by 10) were similar, $L_q$ in most cases being better conditioned than $L_p$. For 38 of the 59 matrices, the total number of nonzeros in $L_p$ and $U$ was less than the number of nonzeros in $chol(A)$. For most of the 59 matrices, $L_q$ is sparser than $L_p$.

## 3 Iterative algorithms with $L$ and experiments on convergence

CGNE, CGNR, (see for example [25]) and `lsqr` [23] are all methods of solving the normal equations by a conjugate gradient algorithm. CGNE is appropriate for minimizing the solution $x$ for an underdetermined system. CGNR and `lsqr` minimize $\|r\|_2^2 = \|Ax - b\|_2^2$. We found that the decline of $\|r\|_2$ is more monotonic for `lsqr` than for CGNR and that when many iterations are required, `lsqr` convergence is more likely. The `lsqrLU` algorithm is given in Algorithm 3.1 (see [23] for more details).

If $L$ is denser than $U$, multiplications by $U^{-1}A$ in `lsqr` could replace multiplications by $L$. Since $U$ is upper triangular, a solve $Uw = y$ efficiently replaces $w = U^{-1}y$ (and similarly for multiplications in `lsqr` by $L^T = U^{-T}A^T$). If storage is comparable, multiplication by $L$ and $L^T$ is likely to execute more efficiently, particularly for parallel computations.

Figure 2 compares convergence results obtained by iterating with `lsqr` on $L_p$ and $A$, where $L_p$ is the $L$ factor obtained using the $LU$ factorization of $A$ with partial pivoting ($PA = L_pU_p$). The graph plots the relative error taking

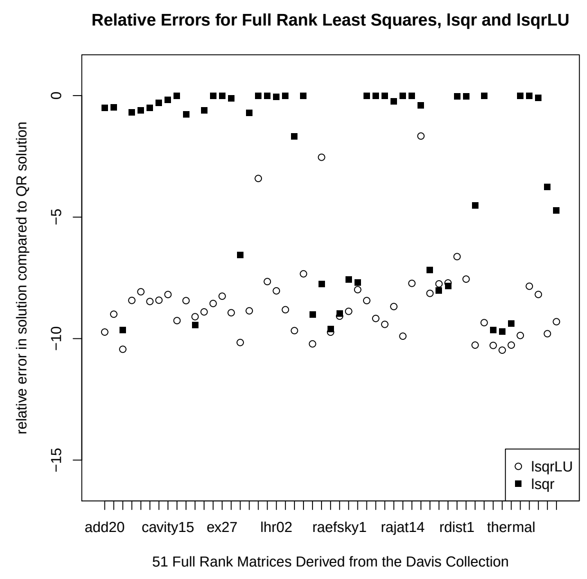


..

Fig. 1: Maximal and minimal singular values of $L_p$ are closer to one than for $A$: $L_p$ is better conditioned than $A$.

**Algorithm 3.1** `lsqrLU`

```
function $[x, its]$ = lsqrLU($A, b, tol, maxit$)
  % $A$ is an input matrix of $m$ rows and $n$ columns
  % $b$ is an input $m$-vector
  % $tol$ is an input scalar larger than zero, convergence tolerance.
  % $maxit$ 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] $\leftarrow$ lu($A$,“vector”);
     r $\leftarrow$ permute($b$,prow);
     $[x, its] \leftarrow$ lsqr(L,r,$tol$, $maxit$);
     $x \leftarrow$ U $\setminus$ $x$;
     if ($its \geq maxit$),“maxits exceeded, did not converge”
end
```



..

Fig. 2: Relative difference from QR solution, `lsqr` and `lsqrLU`. `lsqrLU` has better convergence because $cond(L_p) << cond(A)$

the exact solution as that obtained using a $QR$ factorization algorithm. There is a maximum of $2n$ `lsqr` iterations (or convergence with a tolerance of $1.e$-10). The relative errors for `lsqrLU` are plotted at convergence with tolerance $1.e$-10 (or after at most $n$ iterations). The set of 51 matrices was described in Section 2. The vector $x$ minimizes $\|Ax - b\|_2$ where $b$ is randomly generated, so that the typical problem is overdetermined, with nonzero residual.

Though we allow twice as many `lsqr` iterations compared to
tt lsqrLU, many more matrices converge with iterations on $L$ than with iterations on $A$. For `lsqrLU`, only three matrices (those that had not converged after $n =$ maxit iterations) had relative error larger than $1.e$-6. For this tol $= 1.e$-10, the relative error obtained from iterating with $L$ is smaller than that from iterating on $A$. For more data on this numeric experiment see the R data frame ([16]).

## 4 Partial orthogonalization

Iteration with $L$ from partial pivoting is often satisfactory. For larger problems, partial pivoting may not be available (e.g., for distributed memory SuperLU [20]) or may be much slower than an LU factorization without pivoting (as can happen in MUMPS [1]). Since the speed of convergence depends on the condition of $L$, a natural idea is to improve the conditioning of $L$. Similarly, Jennings and Ajiz [17] improved the conditioning of $A$. When the estimated condition of $L_p$ is larger than $10^2$, we construct $QR = L_{drop}$ and the `lsqr` iteration matrix is taken as with $L_p R^{-1}$. To avoid having $R$ dense, we would like to drop many entries of $L_p$. $L_{drop}$ is obtained from $L_p$ by zeroing all column entries with absolute value less than $colmax/condest(L_p)^{\alpha}$, where $\alpha = 0.25$ and $colmax$ is the maximal column entry for each column. For example, if $condest(L_p) = 10^4$, the drop tolerance is $colmax/10$, and for $condest(L_p) = 10^8$, the drop tolerance is $colmax/100$. Here, $L_p$ was obtained by partial pivoting with $colmax = 1$. Using a larger $\alpha$ would give a lower drop tolerance, better conditoning and faster convergence, but also more nonzero entries in $R$.

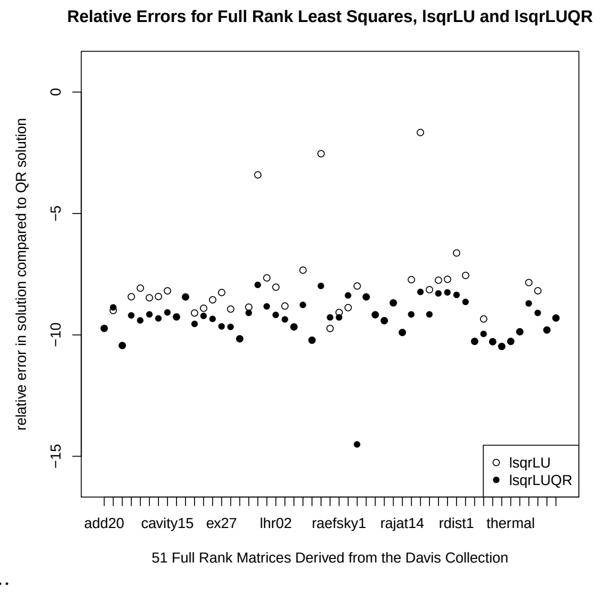


Fig. 3: Relative difference from QR solution, at convergence or max of $n$ iterations ($n$ is the number of matrix columns). `lsqr` using $L$ and preconditioning with partial orthogonalization `lsqrLUQR` converges for all members of this set of matrices.

We denoted `lsqr` using $L$ from $LU$ decomposition as `lsqrLU` (see Section 3). When we use partial orthogonalization of $L$, then we obtain the `lsqrLUQR` algorithm described in Algorithm 4.1.

For Figure 3, the `lsqrLUQR` data points take $C_{max} = 100$ and $\alpha = .25$, i.e used `lsqrLU` for $condest(L(1:n,1:n)) < 100$, else also use partial orthogonalization with $\alpha = .25$. As the `Matlab` or `Octave` function *condest* [13, 15] gets a fast estimate of the 1-norm $cond(L)$ (from the square submatrix $L(1:n,1:n)$ of $L$), it can be used for a runtime algorithm decision.

**Algorithm 4.1** lsqrLUQR: LU preconditioning with partial orthogonalization

```
function $[x, its]$ = lsqrLUQR($A, b, tol, maxit$)
  % $A, b, tol, maxit, x, its$ same as for lsqrLU
    [L,U,prow] $\leftarrow$ lu($A$,'vector');
    r $\leftarrow$ permute($b$,prow);
    $C_{max} \leftarrow 100$
    $\alpha \leftarrow .25$
    $C_{est} \leftarrow$ condest(L(1:$n$,1:$n$))
    if $C_{est} > C_{max}$,
      $\beta \leftarrow 1/C_{est}^{\alpha}$
      $L_{drop} \leftarrow$ drop(L,$\beta$)
        % if $|l_{i,j}| < \beta$,   $l_{drop,i,j} \leftarrow 0$   else   $l_{drop,i,j} \leftarrow l_{i,j}$
      $R_{drop} \leftarrow$ qr($L_{drop}$) % $Q_{drop}$ $R_{drop}$ = $L_{drop}$, $Q_{drop}$ not stored
      $[x, its] \leftarrow$ lsqr($LR_{drop}^{-1}, r, tol, maxit$);
        % L $R_{drop}^{-1}$ not computed,
        % lsqr solves with $R_{drop}$, $R_{drop}^T$, multiplies by L, $L^T$.
    else
      $[x, its] \leftarrow$ lsqr(L,$r, tol, maxit$);
    end
    x $\leftarrow$ U \ x;
    if ($its \geq maxit$),"maxits exceeded, did not converge"
end
```

For a given convergence tolerance, choosing $C_{max}$ larger means that `lsqrLU` is more likely to be used, entailing less storage and more iterations. Decreasing $\alpha$ (i.e making the drop tolerance larger) also decreases storage and increases the number of required iterations.

Figure 3 represents the convergence on $L$ preconditioned by partial orthogonalization. The figure plots the quantity

$$\log_{10} \frac{\|x_{QR} - x_{alg}\|_2}{\|x_{QR}\|_2},$$

to compare the solutions $x_{alg}$ computed with the algorithms `lsqr` with $LU$ preconditioning `lsqrLU` as in Figure 1 (open circles), and `lsqrLUQR` as described above (solid circles) with the solution $x_{QR}$ obtained using a sparse QR factorization.

In Figure 3, `lsqr` with $L$ and partial orthogonalization converges for each member of this set of matrices. `lsqrLUQR` had relative error less than 1.$e$-6 for all 51 matrices. If $R = Q^T L_{drop}$ is computed, then the total storage is on avarage about the same as required for $chol(A^T A)$. If only $L$ and $U$ are needed, storage requirements are usually less than for $chol(A^T A)$. Data ( R data frames) for the assertions of this section and additional plots can be downloaded from [16]. Data includes required storage, arithmetic operations, and iterations for each of five solution methods for each matrix in the data set.

For comparison, we also applied partial orthogonalization directly to the sparse matrix. Using the first $n$ rows of $A$ to estimate condition number and then dropping entries $a_{kj}$ such that

$$|a_{kj}| < \frac{max_i |a_{ij}|}{condest(A)^{.25}},$$

we computed $QR = A_{drop}$ and iterated with `lsqr` on $AR^{-1}$. Similarly to the procedure with $L$, we estimated the condition of $A$ by $condest(A(1{:}n,1{:}n))$.

Then only 11 of the 51 matrices gave relative error (compared to QR solution) less than 1.$e$-6. A problem may be that the condition estimate based on the first $n$ rows is less useful for $A$ than it is for the triangular matrices $L_p$. An acceptable partial orthogonalization $AR^{-1}$ would require fewer $A$ entries to be dropped and hence increased storage.

## 5 Conclusions and future work

The `lsqrLU` and `lsqrLUQR` preconditioned versions of `lsqr` were reliable in our numerical experiments. `lsqrLU` requires on average less storage than computing the Cholesky factorization of $A^T A$. If an additional partial preconditioning step is considered, the total storage is on average about the same as for the Cholesky factorization.

Condition estimates allow a runtime decision on whether to use partial pivoting, and what drop tolerance can be used. To address larger problems, it would be interesting to investigate how to use fast statistical methods to estimate the condition number of $L_p$, similarly to [3].

Since $LU$ factorization with partial pivoting is available in distributed packages for sparse direct solvers [1, 12], we expect `lsqrLU` to be scalable to larger problems. Scalable versions of `lsqrLUQR` may use `MIQR` [18] for partial orthogonalization.

## References

1. P. R. Amestoy, A. Guermouche, J.-Y. L'Excellent, and S. Pralet, *MUMPS: a MUltifrontal Parallel sparse direct Solver*, 2014, http://mumps.enseeiht,fr/index.php?page=home.
2. M. Baboulin, L. Giraud, S. Gratton, and J. Langou, *Parallel tools for solving incremental dense least squares problems. Application to space geodesy*, Journal of Algorithms and Computational Technology **3** (2009), no. 1, 117–133.
3. M. Baboulin, S. Gratton, R. Lacroix, and A. J. Laub, *Statistical estimates for the conditioning of linear least squares problems*, 10th International Conference on Parallel Processing and Applied Mathematics (PPAM 2013) (Heidelberg) (Roman Wyrzykowski et. al., ed.), Lecture Notes in Computer Science, vol. 8384, Springer-Verlag, 2014, pp. 124–133.
4. D. Bateman and A. Adler, *Sparse matrix implementation in octave*, 2006, Available from arxiv.org/pdf/cs/0604006.pdf.
5. M. Benzi and M. Tuma, *A robust incomplete factorization preconditioner for positive definite matrices*, Numerical Linear Algebra with Applications **10** (2003), 385–400.

6. Å. Björck, *Numerical methods for least squares problems*, SIAM, Philadelphia, 1996.
7. A. Björck and I. S. Duff, *A direct method for the solution of sparse linear least squares problems*, Linear Algebra Appl. **34** (1980), 43–67.
8. A. Björck and J.Y. Yuan, *Preconditioners for least squares problems by LU factorization*, Electronic Transactions on Numerical Analysis **8** (1999), 26–35.
9. T. Davis, *The University of Florida sparse matrix collection*, Available from http://www.cise.ufl.edu/research/sparse/matrices/.
10. ———, *Direct methods for sparse linear systems*, SIAM, 2006.
11. G. H. Golub and C. F. Van Loan, *Matrix computations*, The Johns Hopkins University Press, Baltimore, 1996, Third edition.
12. A. Gupta, *Improved symbolic and numerical factorization algorithms for unsymmetric sparse matrices*, SIAM J. Matrix Anal. Appl. **24** (2002), 529–552.
13. W. W. Hager, *Condition estimates*, SIAM J. Sci. Stat. Computing **5** (1984), 311–316.
14. M. T. Heath, *Numerical methods for large sparse linear squares problems*, SIAM J. Sci. Stat. Computing **5** (1984), no. 4, 497–513.
15. N. J. Higham and F. Tisseur, *A block algorithm for matrix 1-norm estimation with an application to 1-norm pseudospectra*, SIAM J. Matrix Anal. Appl. **21** (2000), 1185–1201.
16. G. Howell and M. Baboulin, *Data and plots for lsqr, lsqrLU and lsqrLUQR*, 2015, http://ncsu.edu/hpc/Documets/Publications/gary_howell/contents.html.
17. A. Jennings and M.A. Ajiz, *Incomplete methods for solving $A^TAx = b$*, SIAM J. Sci. Stat. Comput. **5** (1984), no. 4, 978–987.
18. N. Li and Y. Saad, *MIQR: a multilevel incomplete QR preconditioner for large sparse least-squares problems*, SIAM J. Matrix Anal. and Appl. **28** (2006), no. 2, 524–550.
19. X. S. Li, *An overview of SuperLU: Algorithms, implementation, and user interface*, ACM Transactions on Mathematical Software **31** (2005), no. 3, 302–325.
20. 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** (2003), no. 9, 110–140.
21. M. Lourakis, *Sparse non-linear least squares optimization for geometric vision*, European Conference on Computer Vision **2** (2010), 43–56.
22. J. Nocedal and S. J. Wright, *Numerical optimization*, Springer, 1999.
23. C. Paige and M. Saunders, *An algorithm for sparse linear equations and sparse least squares*, ACM Trans. on Math. Software **8** (1982), no. 1, 43–71.
24. G. Peters and J. H. Wilkinson, *The least squares problem and pseudo-inverses*, Computing J. **13** (1970), 309–316.
25. Y. Saad, *Iterative methods for sparse linear systems*, SIAM, Philadelphia, 2000, Second edition.
26. Olaf Schenk and Klaus Gärtner, *PARDISO User Guide*, http://www.pardiso-project.org/manual/manual.pdf, 2014.