Stanford University, Management Science & Engineering (and ICME)

MS&E 318 (CME 338) Large-Scale Numerical Optimization

Instructor: Michael Saunders Spring 2018

Notes 7: **LUSOL: a Basis Factorization Package**

1 Origins

LUSOL is a set of procedures for computing and updating LU factors of a general sparse matrix A. The design aims follow:

- Allow A to be square or rectangular with arbitrary rank.
- Factorize $A = LU$ directly, finding suitable row and column orderings.
- Replace a column or row of A.
- Add or delete a column or a row (thus altering the size of A).
- Perform a general rank-one update $A \leftarrow A + vw^T$ for sparse v and w.
- Balance stability and sparsity throughout, keeping L well-conditioned.

The primary application of LUSOL has been for square basis factorizations $B = LU$ (and column replacement) as part of the optimization packages MINOS [14, 15, 16], SQOPT and SNOPT [7, 9], and the nonlinear complementarity packages MILES [20], PATH [3, 5], and PATHNLP [17].

In the optimization packages it is also applied to rectangular matrices $\begin{pmatrix} B & S \end{pmatrix}$ (transposed) to find a column ordering that makes B better conditioned [7].

The original LUSOL procedures are described by Gill, Murray, Saunders and Wright [8]. The main factorization uses a traditional Markowitz strategy with Threshold Partial Pivoting similar to other early sparse LU packages—notably Y12M [24], LA05 [18], LA15 [19], MA28 [4] and MOPS [22]. The Bartels-Golub update for column-replacement follows the sparse implementation of Reid [18]. This involves a "forward sweep" of eliminations. The other updates are implemented similarly (some of them requiring a backward sweep).

LUSOL has continued to evolve with the addition of rank-revealing LU factorizations for sparse matrices, using either Threshold Rook Pivoting or Threshold Complete Pivoting. A further option is intended for symmetric quasi-definite matrices [23].

The online source code [21] contains two versions (f77 and f90). The Fortran 77 version is suitable for f77, f90, and later compilers and convertible to C by the f2c translator. A Matlab interface is provided [13]. In 2004, a Fortran 77 → Pascal → C translation was created by Kjell Eikland for use within the open source LP/MILP system lp_solve [12]. Since 2005 it has been lp_solve's default basis factorization package (BFP). Since 2012, an f90 version of LUSOL has been running within a new f90 version of SQOPT, and it has a new Matlab interface [10]. In both f77 and f90 versions, the efficiency of rook pivoting was significantly improved in 2013.

2 Purpose

LUSOL maintains a sparse factorization $A = LU$ and permutations P, Q such that PUQ is a sparse upper triangular (or upper trapezoidal) matrix, stored explicitly as a set of *sparse rows*. Depending on input parameters, L tends to be well-conditioned, while the condition and rank of U reflect the condition and rank of A.

The main functions of LUSOL follow:

Factor Determine L, U, P, Q directly from A. The initial $L = L_0$ is such that PL_0P^T is a sparse lower triangular matrix with unit diagonals and bounded off-diagonals, stored explicitly as a set of *sparse columns*. The initial U_0 is stored explicitly as a set of *sparse rows*.

Solve Given a dense vector y, use the current factors to solve one of the following systems:
$Lx = y, \quad L^Tx = y, \quad Ux = y, \quad U^Tx = y, \quad Ax = y, \quad A^Tx = y.$

Multiply Given a dense vector y, use the factors to form one of the following products:
$x = Ly, \quad x = L^Ty, \quad x = Uy, \quad x = U^Ty, \quad x = Ay, \quad x = A^Ty.$

Update Modify L, U, P, Q to reflect one of the following changes to A:

Add a column,	Replace a column,	Delete a column,
Add a row,	Replace a row,	Delete a row,
Add a rank-one matrix σvw^T.		

The initial L_0 is not altered but updates are accumulated in a product form $L = L_0M_1M_2\dots M_\ell$ for a sequence of stabilized elementary matrices M_j. Updated U factors are maintained explicitly as sparse rows.

3 Factor

The factorization procedure `lu1fac` receives A as a list of triples (i, j, A_{ij}), where each A_{ij} is typically nonzero. (Any zero or tiny elements are deleted.) The nonzeros are sorted into a *column list* containing pointers to the start of each column and the corresponding column lengths (the number of entries). Contiguous storage is used for the entries in any given column, but if new entries arise during factorization, the whole column may be moved to another part of storage.

A similar *row list* is constructed to store the sparsity structure of A by rows. To save storage the row list does not contain the numerical values themselves (although they would improve the efficiency of Threshold Rook Pivoting as mentioned below).

Additional data structures are used to sort the columns and rows in order of increasing length.

Each stage of the LU factorization computes l and u, the next column of L and the next row of U, using a nonzero A_{ij} as "pivot element":

$$l = A_{.j}/A_{ij}, \qquad u^T = A_{i.}, \qquad L \leftarrow \begin{bmatrix} L & l \end{bmatrix}, \qquad U \leftarrow \begin{bmatrix} U \\ u^T \end{bmatrix}, \qquad A \leftarrow A - lu^T.$$

Note that the ith row and jth column of $A - lu^T$ are empty (zero). The data structure holding A has a decreasing number of rows and columns, but a certain amount of *fill* (new nonzeros) may be generated by the rank-one term lu^T.

3.1 Data structures

The main LU factorization routine looks like this in Fortran 90:

```
subroutine lu1fac( m    , n    , nelem, lena , luparm, parmlu, &
                   a    , indc , indr , p    , q     ,         &
                   lenc , lenr , locc , locr ,                 &
                   iploc, iqloc, ipinv, iqinv, w     , inform )

  integer(ip),   intent(in)    :: m, n, nelem, lena
  integer(ip),   intent(inout) :: luparm(30)
  integer(ip),   intent(out)   :: inform
  real(rp),      intent(inout) :: parmlu(30), a(lena), w(n)
  integer(ip),   intent(inout) :: indc(lena), indr(lena), &
                                  p(m)      , q(n)      , &
                                  lenc(n)   , lenr(m)   , &
                                  iploc(n)  , iqloc(m)  , &
                                  ipinv(m)  , iqinv(n)  , &
                                  locc(n)   , locr(m)
```

The nonzeros of A are input via parallel arrays `a`, `indc`, `indr` containing `nelem` triples (`aij, i, j`) in any order. `luparm` and `parmlu` contain integer and double precision input parameters, and receive certain output values.

During the LU factorization, the sparsity pattern of the remaining (modified) A is stored twice, in a *column list* and a *row list.*

The column list is (`a`, `indc`, `locc`, `lenc`), where

`a(*)`	holds the nonzeros,
`indc(*)`	holds the indices for the column list,
`locc(j)`	points to the start of column `j` in `a(*)` and `indc(*)`,
`lenc(j)`	is the number of nonzeros in column `j`.

The row list is (`indr`, `locr`, `lenr`), where

`indr(*)`	holds the indices for the row list,
`locr(i)`	points to the start of row `i` in `indr(*)`,
`lenr(i)`	is the number of nonzeros in row `i`.

At all stages, `p` and `q` contain complete row and column permutations P, Q.

At the start of stage `k`, `p(1)`, . . . , `p(k-1)` are the first `k-1` rows of the final P. The remaining rows are stored in an *ordered list* `(p, iploc, ipinv)`, where

`iploc(nz)`	points to the start in `p(*)` of the set of rows that currently contain `nz` nonzeros,
`ipinv(i)`	points to the position of row `i` in `p(*)`.
`iploc(1) = k`	(and this is where rows of length 1 begin),
`iploc(2) = k+r`	if there are `r` rows of length 1 (and this is where rows of length 2 begin), and so on up to `iploc(n)`.

There is a similar ordered list (`q`, `iqloc`, `iqinv`) for the column permutation Q.

At the end, the final factors are repacked so that L is stored columnwise counting backward from the end of (`a`, `indc`, `indr`), and U is stored rowwise counting forward from the beginning of (`a`, `indc`, `indr`). The array `w` has `w(j)` ≤ 0 if column `j` seems to be dependent on the other columns. This is how we report singularity.

3.2 Pivot strategies

To preserve sparsity, a *Markowitz strategy* is used to select potential pivots A_{ij}. Pivots should have a low *Markowitz merit function* $M_{ij} \equiv (r_i - 1)(c_j - 1)$, where r_i and c_j are the lengths of row i and column j of the current "A", because M_{ij} bounds the fill that could be created by lu^T. The sparsest columns and rows are searched in turn (columns of length 1, rows of length 1, then columns of length 2, rows of length 2, and so on). The lowest M_{ij} is recorded for pivots that satisfy a specified stability test. Typically only 5 or 10 of the shortest columns and rows are searched, but some of the stability tests may require more extensive searching.

To preserve stability, one of the following *threshold pivoting strategies* is used. Let `Ltol` be a given number, typically 10 or 5 or perhaps nearer 1. (It controls the size of the off-diagonals of L and possibly those of U.) Further, let $A_{\max}$ be the largest remaining nonzero and $D_{\max}$ the largest remaining diagonal (both absolute values).

Strategy	Name	Stability Test
Threshold Partial Pivoting	TPP	$\|l\|_\infty \le$ `Ltol`
Threshold Rook Pivoting	TRP	$\|l\|_\infty$ *and* $\|u/A_{ij}\|_\infty \le$ `Ltol`
Threshold Complete Pivoting	TCP	$A_{\max} \le$ `Ltol` $\times \lvert A_{ij}\rvert$
Threshold Symmetric Pivoting	TSP	TRP for symmetric A
Threshold Diagonal Pivoting	TDP	$D_{\max} \le$ `Ltol` $\times \lvert A_{ii}\rvert$

In practice, TPP is used most often. It preserves sparsity well and is usually sufficiently stable with `Ltol` = 100 or 10. Smaller values are needed if A is singular or ill-conditioned and *rank-revealing* properties are desired. Ideally, any rank-deficiency should be identified by the correct number of small diagonals in U, but this is not guaranteed by TPP. The other options are needed for greater reliability in determining rank—for example, in the Simplex Method when the first rather arbitrary B is factorized.

TPP requires A_{ij} to be sufficiently large compared to other elements in its own column. It can be implemented efficiently, and every column contains at least one entry that satisfies the test, so relatively few columns need be searched. It is convenient to store the largest element in each column in a known place (as the first entry in the column).

TRP is symmetric in that A_{ij} must be sufficiently large compared to other elements in its own column *and* its own row. It costs more than TPP (may require more searching and produce less sparse factors), but it has more definite rank-revealing properties and seems acceptably efficient with `Ltol` as low as 2 or even 1.1.

TCP needs us to keep track of $A_{\max}$, the largest element in the current A. A *heap* structure is used to store the largest element in each column, and then $A_{\max}$ is always at the top of the heap. TCP satisfies the TRP test but in a more restrictive way. If `Ltol` is too close to 1, the TCP factors are likely to be unacceptably dense.

(Note that pathological examples are known for which TRP or even TCP fail to reveal rank correctly.)

TSP and TDP are intended for symmetric matrices that are either definite or *quasi-definite*. (In exact arithmetic, symmetric factorizations $PAP^T = LDL^T$ exist for such matrices for any permutation P.) TSP is implemented by pivoting on diagonals only and requiring $\|l\|_\infty \le$ `Ltol` as for TPP.

TDP needs $D_{\max}$, the largest remaining diagonal. It has not been implemented yet (and may not be needed). Another heap structure would be needed to store the diagonals $|A_{ii}|$, so that $D_{\max}$ is available at the top of the heap.

The main benefit of TCP and TDP is that they concentrate singularities at the end of P and Q, so that PUQ will be *upper trapezoidal.* The other strategies may give small diagonals in any part of U (if A is singular). This often indicates singularity correctly, but the following matrix illustrates that TPP may be misleading:

$$A = \begin{pmatrix} \delta & 1 & 1 & 1 \\ & \delta & 1 & 1 \\ & & \delta & 1 \\ & & & \delta \end{pmatrix}, \qquad \delta = 10^{-4} \text{ or } 10^{-11}, \text{ say.}$$

TPP would accept all diagonals and return $L = I$, $U = A$. LUSOL's singularity check would then report that all diagonals of U are big enough compared to other entries in their own column ($\operatorname{rank}(A) = 4$), or that all are too small ($\operatorname{rank}(A) = 0\,!$).

In contrast, the rook pivoting strategy would reject all δ entries because they are too small compared to the other elements in their own column *and* row. Instead, TRP is likely to choose pivots from the first superdiagonal (effectively permuting the first column to the end). The result for both values of δ is

$$PUQ \approx \begin{pmatrix} 1 & 1 & 1 & \delta \\ & 1 & 1 & -\delta^2 \\ & & 1 & \delta^3 \\ & & & -\delta^4 \end{pmatrix}, \qquad \operatorname{rank}(A) \approx 3.$$

3.3 Backward and forward triangles

The strategy of searching columns of length 1 and then rows of length 1 automatically reveals the following structure in a typical sparse A. For some preliminary permutations P_1 and Q_1, the permuted matrix P_1AQ_1 appears as in Figure 1, where $\begin{bmatrix} U_1 & V \end{bmatrix}$ is called the

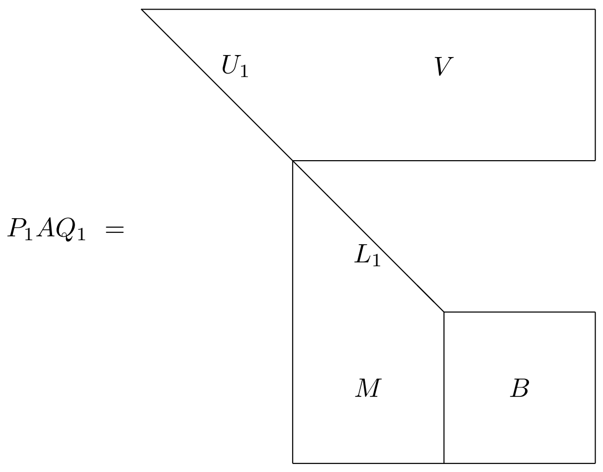

Figure 1: The backward and forward triangles in a typical sparse matrix

backward triangle (the top rows of U) and $\begin{bmatrix} L_1 \\ M \end{bmatrix}$ is the forward triangle (the first columns of L). No elimination has occurred yet. It remains to compute LU factors of block B.

3.4 Sparse elimination

A typical step in the LU factorization of block B is illustrated here. The Markowitz strategy has selected a pivot ① in a reasonably sparse column and row, and this is regarded as large enough under the TPP stability test when `Ltol` ≥ 2.0. The merit function M_{11} predicts that at most $2 \times 3 = 6$ new entries will be created in rows 2–3 and columns 2–4 (an over-estimate because 16, 1 and 5 are already nonzero). Note that all other rows and columns will be unaltered.

TPP can pivot on	①
TRP can pivot on	4
TCP can pivot on	16

$$\begin{array}{|c|ccc|cccccc|}
\hline
\textcircled{1} & 4 & 2 & 1 & & & & & & \\
\hline
2 & & 16 & & \times & & \times & & & \times \\
1 & 1 & & 5 & & \times & \times & & \times & \\
\hline
 & & \times & & & & & \times & & \\
 & & & \times & & & \times & & \times & \\
 & & & & \times & & 45 & \times & & \times \\
 & 6 & & & & \times & & & & \\
 & & \times & & & & & & & \times \\
 & & & & \times & & & \times & & \\
 & & & \times & & & & & \times & \\
\hline
\end{array}$$

Most of the complexity of `lu1fac` arises from updating the lists that hold the nonzeros. Recall that they are implemented as arrays (`a`, `locc`, `lenc`, `indc`) for the column list and (`locr`, `lenr`, `indr`) for the row list (sparsity structure only).

When the pivot row is deleted from the column list (to become the first row of U), there is room in the modified columns for *one* new entry. In this example, *one* is sufficient and columns 2–4 can be updated *in place.*

Similarly, when the pivot column is deleted from the row list, row 3 can be updated in place. However, row 2 has *two* new entries. All of row 2 must be moved to the beginning of the row list's free storage, and its previous space marked as unused. Thus, columns and rows can migrate around memory within their own lists, leaving gaps that are periodically compressed ("garbage collection").

After most of the LU factors have been computed, the remaining rows and columns of the updated A will be sufficiently dense (say 50%) to warrant a switch to dense processing.

4 Solve

The current LUSOL procedures for solving $Lx = y$, $Ux = y$, $\dots$ treat y as a dense vector. Solves with L_0 and L_0^T are the most efficient because the columns are processed in natural order (forwards or backwards) and never altered. The product-form updates to L are handled reasonably efficiently (but each involves two arbitrary indices).

Ideally, new options should be implemented in LUSOL to take advantage of sparse right-hand sides y. For example, each iteration of Primal Simplex requires solution of $Bv = a_s$, where a_s is a column of A (with an average of only 5 or 10 entries regardless of the size of A). A significant cost in some simplex implementations lies in the most trivial operation: setting a dense vector to zero before unpacking a_s. This is true even for $m = 10,000$, and certainly for $m = 1$ million.

Gilbert and Peierls [6] show how to solve a triangular system $Lx = y$ in $O(p)$ operations, where p is the number of *nonzero multiplications* needed to form the product Lx. When y is sparse, p may be very small *regardless of problem size.* The true benefits of this technique have been realized in CPLEX; see Bixby [1].

The GP algorithm requires L to be stored by columns. In LUSOL, this means $L_0x = y$, $U_0^Tx = y$ and updated $U^Tx = y$ could be solved efficiently. LA05 and LA15 maintain both the row and the column structure of U (and probably of L_0 also). They do take advantage of sparse y.

5 Multiply

The Multiply procedures may be needed to recover parts of A from L and U if the original data has been over-written. Less trivially, products with L and L^T are needed if an iterative solver (such as LSQR or LSMR) is applied to a least-squares problem in the following way. Suppose A is rectangular ($m > n$) with $\operatorname{rank}(A) = n$. A sparse factorization $A = LU$ is typically cheaper than a QR factorization. To solve

$$\min \|Ax - b\|_2$$

we may solve the equivalent problem $\min \|Ly - b\|_2$, $Ux = y$ using the same iterative solver. Although L is typically less sparse than A, the pivoting strategies keep it well-conditioned even if A is not. Hence the solver may converge sufficiently quickly.

Note that U is being used as a right-preconditioner, and we assumed $A = LU$ accurately. If U is exact or just *approximate*, it may be more efficient to apply the iterative solver to the equivalent problem $\min \|AU^{-1}y - b\|$, $Ux = y$.

6 Update

The main work for all updates comes from a forward or backward sweep of stabilized eliminations to triangularize a modified U.

Forward sweeps tend to alter only a few rows of U. The most common update (Bartels-Golub-Reid column replacement) requires a sparse column to be *inserted* into U (possibly altering several rows) and then a forward sweep to eliminate the sparse row spike that appears, as mentioned in Notes 5. A multiple of one row of U can eliminate the first nonzero in the row spike. A test first determines whether the row of U should be swapped with the row spike (to become the new row spike). This is the stability/sparsity row interchange associated with Bartels, Golub, and Reid.

Backward sweeps may alter many rows of U. They eliminate a *column spike* and generate a row spike that grows in length and must be added to several subsequent rows, with occasional row interchanges. The final row spike must be eliminated by a forward sweep.

The more general updates are needed for *symmetric* matrices (such as KKT systems in active-set methods). They have not received much use, as block-LU updates have prevailed.

7 Basis repair

A vital use of LUSOL within MINOS, SQOPT, and SNOPT is to ensure that the current basis B is not too ill-conditioned. If necessary, certain columns of B can be replaced by unit vectors (associated with slack variables). The row and column permutations P and Q define which unit vectors should be introduced and which columns they should replace.

BR factorization refers to the use of TRP (or TCP) when the current B appears to be ill-conditioned (e.g., unexpected growth of $\|x_B\|$ or $\|y\|$ has occurred) or when it is important to ensure that B is reasonably well-conditioned (e.g., for the very first basis factorization, or at the start of each major iteration on problems with nonlinear constraints). The name BR indicates the Rank-Revealing requirement. The stability tolerance `Ltol` needs to be somewhat strict (say 2.5) and may need to be reduced in stages toward 1.0 if singularity persists.

BS factorization is sometimes invoked to select a better B from the current basic and superbasic columns $\begin{pmatrix} B & S \end{pmatrix}$. Since the condition of L is always controlled, the factors

$$\begin{pmatrix} B^T \\ S^T \end{pmatrix} = LU, \qquad PLP^T = \begin{pmatrix} L_1 & \\ L_2 & I \end{pmatrix}, \qquad PUQ = \begin{pmatrix} U_1 \\ 0 \end{pmatrix} \tag{1}$$

are relevant. The first m rows of P point to the rows of (1), i.e., the columns of $\begin{pmatrix} B & S \end{pmatrix}$, that define a possibly better-conditioned basis $\bar{B}$, to the extent that one exists:

$$\begin{pmatrix} \bar{B} & \bar{S} \end{pmatrix} = \begin{pmatrix} B & S \end{pmatrix} P^T.$$

On some regularly-structured problems, TPP with `Ltol` ≥ 2.0 is not reliable. We therefore apply BS factorization with TPP and `Ltol` $= 1.9$.

Since $\begin{pmatrix} B & S \end{pmatrix}$ may be significantly bigger than B, we use LUSOL's option to compute the permutations P and Q without storing L and U. Thus after BS factorization points to a suitable $\bar{B}$, a normal factorization of $\bar{B}$ must be computed. If necessary, BR factorization is used to repair $\bar{B}$.

8 A well-conditioned nullspace operator Z

Given a rectangular matrix A (say $m \times n$ with $m < n$ and $\text{rank}(A) = m$), we may need to find a matrix operator Z that spans the nullspace of A (thus $AZ = 0$), is reasonably well-conditioned, and permits efficient computation of $p = Zv$ or $q = Z^T w$ from given vectors v and w. As with BS factorization in (1), we use TPP or TRP to factorize A^T with `Ltol` < 2.0 to keep L well-conditioned:

$$A^T = LU, \qquad PLP^T = \begin{pmatrix} L_1 & \\ L_2 & I \end{pmatrix}, \qquad PUQ = \begin{pmatrix} U_1 \\ 0 \end{pmatrix}.$$

We see that

$$L^{-1}A^T = U \quad \Rightarrow \quad PL^{-1}A^T = PU = \begin{pmatrix} U_1 \\ 0 \end{pmatrix} Q^T \quad \Rightarrow \quad \begin{pmatrix} 0 & I \end{pmatrix} PL^{-1}A^T = 0.$$

Hence $AZ = 0$ with $Z \equiv L^{-T}P^T \begin{pmatrix} 0 \\ I \end{pmatrix}$. Solving $L^T w = P^T \begin{pmatrix} 0 \\ v \end{pmatrix}$ gives us $w = Zv$, and solving $Lw = t$ and setting $s = \begin{pmatrix} 0 & I \end{pmatrix} Pw$ gives us $s = Z^T t$. This Z should be well-conditioned because L and hence L^{-T} are, and because Z is just some of the columns of L^{-T}.

The lusolZ package [11] provides MATLAB software for computing the nullspace Z of the transpose of a sparse matrix S (so that $S^T Z = 0$). It uses sparse QR or LU factors of either S or S^T computed by SuiteSparseQR [2] or LUSOL [10]. MATLAB routines are provided to compute products of the form $w = Zv$ and $s = Z^T t$ for given vectors v and t.

References

[1] R. E. Bixby. Solving real-world linear programs: a decade and more of progress. *Operations Research*, 50(1):3–15, 2002.

[2] T. A. Davis. Algorithm 915, SuiteSparseQR: Multifrontal multithreaded rank-revealing sparse QR factorization. *ACM Trans. Math. Software*, 38(1):8:1–22, 2011.

[3] S. Dirkse, M. C. Ferris, and T. S. Munson. PATH nonlinear complementarity solver. `http://pages.cs.wisc.edu/~ferris/path.html`, accessed May 2017.

[4] I. S. Duff. MA28: A set of Fortran subroutines for sparse unsymmetric linear equations. Report R8730, AERE Harwell, Oxfordshire, England, 1977.

[5] M. C. Ferris and T. S. Munson. PATH nonlinear complementarity solver. `https://www.gams.com/latest/docs/solvers/path/`, accessed May 2017.

[6] J. R. Gilbert and T. Peierls. Sparse partial pivoting in time proportional to arithmetic operations. *SIAM J. Sci. and Statist. Comput.*, 9(5):862–874, 1988.

[7] P. E. Gill, W. Murray, and M. A. Saunders. SNOPT: An SQP algorithm for large-scale constrained optimization. *SIAM Review*, 47(1):99–131, 2005. SIGEST article.

[8] P. E. Gill, W. Murray, M. A. Saunders, and M. H. Wright. Maintaining LU factors of a general sparse matrix. *Linear Algebra and its Applications*, 88/89:239–270, 1987.

[9] Philip E. Gill, Walter Murray, Michael A. Saunders, and Elizabeth Wong. User's Guide for SNOPT 7.7: Software for Large-Scale Nonlinear Programming. Center for Computational Mathematics Report CCoM 18-1, Department of Mathematics, University of California, San Diego, La Jolla, CA, 2018.

[10] N. W. Henderson. Matlab interface to LUSOL. `https://github.com/nwh/lusol`, 2013.

[11] N. W. Henderson, S. Kim, D. Ma, M. A. Saunders, Y. Sun, R. M. T. Fleming, and I. Thiele. lusolZ: Nullspace of sparse matrix via SPQR or LUSOL. `http://stanford.edu/group/SOL/software/lusolZ/`.

[12] lp_solve open source LP and MILP solver. `http://groups.yahoo.com/group/lp_solve/`.

[13] LUSOL mex interface (Nick Henderson, ICME, Stanford University). `https://github.com/nwh/lusol_mex`, 2011.

[14] B. A. Murtagh and M. A. Saunders. Large-scale linearly constrained optimization. *Math. Program.*, 14:41–72, 1978.

[15] B. A. Murtagh and M. A. Saunders. A projected Lagrangian algorithm and its implementation for sparse nonlinear constraints. *Math. Program. Study*, 16:84–117, 1982.

[16] B. A. Murtagh and M. A. Saunders. MINOS nonlinear optimization solver. `https://www.gams.com/latest/docs/solvers/minos/`, accessed May 2017.

[17] PATH nonlinear complementarity solver. `https://www.gams.com/latest/docs/solvers/path/`, accessed May 2017.

[18] J. K. Reid. A sparsity-exploiting variant of the Bartels-Golub decomposition for linear programming bases. *Math. Program.*, 24:55–69, 1982.

[19] J. K. Reid. LA15 basis factorization package (thread-safe version of LA05). HSL Catalogue, `http://www.hsl.rl.ac.uk/catalogue/la15.xml`, 2013 (original version 2001; version 1.3.0 dated 9 April 2013).

[20] T. F. Rutherford. MILES nonlinear complementarity solver. `https://gams.com/latest/docs/solvers/miles/`, accessed May 2017.

[21] SOL downloadable software. `http://stanford.edu/group/SOL/software.html`.

[22] U. Suhl and L. Suhl. Computing sparse LU-factorizations for large-scale linear programming bases. *ORSA J. of Computing*, 2:325–335, 1990.

[23] R. J. Vanderbei. Symmetric quasi-definite matrices. *SIAM J. Optim.*, 5:100–113, 1995.

[24] Z. Zlatev, J. Wasniewski, and K. Schaumburg. *Y12M: Solution of Large and Sparse Systems of Linear Algebraic Equations.* Lecture Notes in Computer Science 121. Springer Verlag, Berlin, Heidelberg, New York, 1981.