

# TRI DIAGONAL L-D-L<sup>T</sup> FACTORIZATION WITH PIVOTING

WOLFGANG M. HARTMANN<sup>†</sup> AND ROBERT E. HARTWIG<sup>‡</sup>

**Abstract.** A pivoting algorithm is developed for a positive semidefinite tridiagonal matrix, yielding the permuted L-D-L<sup>T</sup> factorization. This factorization is then used to (i) obtain the solution to a positive definite system with pivoting, (ii) compute the Moore-Penrose inverse of a singular positive semidefinite matrix, and (iii) obtain least squares solutions for systems of this type.

**Key words.** positive semidefinite tridiagonal, Cholesky, minimum norm solution, Moore-Penrose Inverse.

**AMS Subject Classifications.** 15A09

**1. Introduction.** The Cholesky factorization is one of the most useful and best known algorithms in numerical linear algebra [2], [6], [9], [10]. Although a tridiagonal version of the Cholesky decomposition is documented in the Linpack and Lapack packages [6], [2], no pivoting is included, and the semi-definite cases are avoided. That pivoting may sometimes be desirable is best illustrated by the examples

$$\mathbf{A} = \begin{bmatrix} \epsilon & 1 \\ 1 & \epsilon + 1/\epsilon \end{bmatrix} \quad \text{and} \quad \mathbf{B} = \begin{bmatrix} \epsilon & 1 \\ 1 & 1/\epsilon \end{bmatrix} \quad ,$$

which are positive (semi) definite and have a small leading diagonal entry. In this paper we propose to derive a stable pivoting technique for the tridiagonal positive semi-definite matrix  $\mathbf{A}$ , which will yield the permuted “Cholesky-like” factorization  $\mathbf{PAP}^T = \mathbf{LDL}^T$ . This factorization will then be applied to the invertible case to give a more stable solution method for the linear tridiagonal system  $\mathbf{Ax} = \underline{b}$ . The factorization provides a fast algorithm for the minimum norm solution of linear systems with large dense positive semi-definite coefficient matrices. As a by-product we obtain a fast algorithm for the Moore-Penrose inverse of a large dense positive semi-definite matrix. Even though our algorithm will work for the invertible case, we are not trying to compete with the numerous fast algorithms, such as Bunch-Kaufman [4] algorithm, Gaussian elimination, rank-1 factorization etc. Our main interest is in the (very) singular systems for which none of those fast methods applies.

Throughout this paper all our matrices will be real and we assume familiarity with the notation on generalized inverses as given in [3]. In particular, the Moore-Penrose (*MP*) inverse will be denoted by  $\mathbf{A}^\dagger$ . As always, rank will be denoted by  $r(\cdot)$ , nullity by  $d(\cdot)$ ,  $\underline{e}_i$  stands for the  $i$ -th unit vector, and  $\mathbf{E}_{ij} = \underline{e}_i \underline{e}_j^T$ . We abbreviate positive (semi)-definite to  $p(s)d$  and we define the  $k \times k$  “cyclic” matrix -  $k$  by  $[\underline{e}_2, \underline{e}_3, \dots, \underline{e}_k, \underline{e}_1]$ .

When counting operations we shall not distinguish between additions and subtractions, but we shall differentiate between multiplications and divisions on account of new kinds of chips (RISC or Pentium), for which division is much slower than multiplication. We shall abbreviate the terms minimal norm least squares solution and operations count to *mnlss* and *OC* respectively, and we denote the number of multiplications, additions, and divisions that are used by the triple  $OC = (M, A, D)$ .

---

<sup>†</sup>SAS Institute, Inc., SAS Campus Drive, Cary, NC 27513, saswmh@unx.sas.com

<sup>‡</sup>Dept. of Mathematics, North Carolina State University, Raleigh, N.C. 27695-8205, hartwig@math.ncsu.edu



In double precision  $\mathbf{A}$  has two hidden zero eigenvalues in addition to eigenvalue  $1+2.10^{-8}$ , in spite of the fact that  $\epsilon$  is not small. It shall be the job of our pivoting algorithm to squeeze the hidden eigenvalues out of the irreducible blocks, and as such determine the true rank of the blocks. For example, our numerical rank determination is capable of finding the nullity of 200 for dense matrices of size  $1000 \times 1000$ . It is in fact comparable to that obtained via the symmetric eigenvalue and *SVD* calculations, at a quarter of the time. We shall present several tables and graphs, showing a comparison of computer time between our method and the eigenvalue and singular value decomposition methods.

Turning to one of the blocks, we may without loss of generality assume that  $b_i \neq 0$  for all  $i$ , and hence that *all*  $a_i > 0$ .

Let us begin with the case where *no* pivoting is needed to illustrate our procedure. Suppose we pivot on  $a_1$ , i.e. use  $\mathbf{R}_1 = \mathbf{M}_1(\epsilon_1 \underline{e}_1)$ , with  $\epsilon_1 = -b_1/a_1$ . Then

$$(2.3) \quad \mathbf{R}_1 \mathbf{A} \mathbf{R}_1^T = \begin{bmatrix} a_1 & 0 \\ 0 & \mathbf{A}' \end{bmatrix},$$

where  $\mathbf{A}' = \mathbf{T}[(\zeta_1, a_3, \dots, a_n), (b_2, \dots, b_{n-1})]$  and  $\zeta_1 = a_2 - b_1^2/a_1$  is the Schur complement of the leading  $2 \times 2$  block. Since  $b_2 \neq 0$ ,  $\zeta_1$  cannot vanish and as such must be positive. We now repeat with  $\mathbf{A}'$ , provided  $\zeta_1$  is not too small. Assuming that we do not have to pivot diagonally, we arrive at

$$(2.4) \quad \mathbf{R} \mathbf{A} \mathbf{R}^T = (\mathbf{R}_{n-1} \mathbf{R}_{n-2} \dots \mathbf{R}_2 \mathbf{R}_1) \mathbf{A} (\mathbf{R}_{n-1} \mathbf{R}_{n-2} \dots \mathbf{R}_2 \mathbf{R}_1)^T = \mathbf{D},$$

where  $\mathbf{R}_i = \mathbf{M}_i(\epsilon_i \underline{e}_1)$ . This yields  $\mathbf{A} = \mathbf{L} \mathbf{D} \mathbf{L}^T$  where

$$(2.5) \quad \mathbf{L} = \mathbf{R}^{-1} = \mathbf{R}_1^{-1} \mathbf{R}_2^{-1} \dots \mathbf{R}_{n-1}^{-1} = \begin{bmatrix} 1 & & & & & \\ -\epsilon_1 & 1 & & & & \\ & -\epsilon_2 & 1 & & & \\ & & & \ddots & & \\ & & & & -\epsilon_{n-1} & 1 \end{bmatrix}$$

is bi-diagonal and  $\mathbf{D} = \text{diag}(a_1, \zeta_1, \zeta_2, \dots, \zeta_{n-1})$ .

When the diagonal entry becomes too small we proceed to the next phase, in which we use diagonal pivoting. This consists of the following three phases. In the first phase we select a suitable pivot, while the second phase contains the actual pivoting procedure. In the final phase we show that the permutations that are used can be passed *through* the row-sweeps that have been performed, allowing a compact form of the solution.

### 3. Tridiagonal Factorization.

**3.1. Pivot Selection.** It goes without saying that *any* algorithm where one uses row and column sweeps, the pivot selection is of considerable importance. In the *singular* (semi-definite tri-diagonal) case, this selection becomes even more crucial and indeed of cardinal and cataclysmic importance. It is tempting to use an *agressive* and even *greedy* pivot selection at each stage. In fact, for an  $n \times n$  tridiagonal matrix the following methods of pivot selection should be considered

- (i) pivot on  $a_k$ , where  $k = \min\{r; a_r = \max\{a_1, \dots, a_n\}\}$
- (ii) perform a ‘‘Burn at both ends’’ (BATBE) test. That is, if  $|b_1| \leq a_1$  then pivot on  $a_1$  otherwise test  $|b_{n-1}| \leq a_n$ . If the latter holds we pivot on  $a_n$ , while otherwise we apply our favorite backup pivot selection, such as (i) or proceed to selection procedure (iii).
- (iii) pivot on  $a_k$  where  $k = \min\{r; |b_r| \leq a_r\}$ . Such  $k$  always exists and cannot exceed  $n-1$ . Indeed, if  $a_i < |b_i|$  for  $i = 1, \dots, n-1$ , then since  $a_n a_{n-1} \geq b_{n-1}^2$ , we know that  $(a_n/b_{n-1})(a_{n-1}/b_{n-1}) \geq 1$ . Thus  $a_n > |b_{n-1}|$ , and we could pivot on  $a_n$ . We shall refer to this method as the ‘‘Burn on one side’’ (BOOS) procedure.

For a dense matrix, the BATBE strategy makes sense since generally the last few entries will be changed at each step. In the tridiagonal case, however, the last entries  $a_n$  and  $b_{n-1}$  will usually not be changed by the earlier pivoting steps, and as such testing  $a_n > |b_{n-1}|$  will serve no purpose.

We have found in practice however, that when dealing with specially constructed matrices that have a large number (say 200) of zero eigenvalues, these aggressive methods **fail** and are not able to recover the rank of the matrix. In fact we run into two kinds of problems. First, more rounding errors are observed than if a more modest (i.e. on averaging) strategy is employed. The reason being that operations between numbers of very different scales, introduces large rounding errors. Second, at the closing stages, it becomes very difficult to distinguish between zero and non-zero diagonal entries. As such it becomes virtually impossible to determine the rank of the matrix, and thus the Moore-Penrose inverse of the matrix.

To avoid these obvious problems we employ the pivot strategy of Frane [7] which scales the present diagonal entries (which are Schur complements) against the original (permuted) diagonal entries. It is the only strategy that has allowed us to detect an exact nullity of several hundred in magnitude.

In order to apply this strategy, we carry the extra vector  $\underline{\alpha}^{(r)} = (\alpha_1^{(r)}, \alpha_2^{(r)}, \dots, \alpha_n^{(r)})$ , which contains the original diagonal entries of  $\mathbf{A}^{(0)} = \mathbf{A}$ , permuted according to the permutations  $\mathbf{P}_1, \mathbf{P}_2, \dots$  that have been used in our pivoting procedure. That is, if  $\underline{\alpha}^{(r)} = (\alpha_1^{(r)}, \dots, \alpha_n^{(r)})$ , then  $\underline{\alpha}^{(r+1)} = \underline{\alpha}^{(r)} \mathbf{P}_r = (\alpha_1^{(r+1)}, \dots, \alpha_n^{(r+1)})$ . Now if at the r-th stage

$$(3.1) \quad \mathbf{A}^{(r)} = \mathbf{D}^{(r)} \oplus \mathbf{T}^{(r)},$$

where  $\mathbf{D}^{(r)} = \text{diag}(d_1, \dots, d_r)$ ,  $\mathbf{D}^{(0)}$  is absent and

$$(3.2) \quad \mathbf{T}^{(r)} = \mathbf{T}[(a_1^{(r)}, \dots, a_{n-r}^{(r)}), (b_1^{(r)}, \dots, b_{n-r-1}^{(r)})],$$

then we select the next pivot index  $m_{r+1}$  for  $\mathbf{T}^{(r)}$  from

$$(3.3) \quad m_1 = \min\{k; a_k = \max\{a_1, \dots, a_n\}\}$$

$$(3.4) \quad m_{r+1} = \min\left\{k; \frac{a_k^{(r)}}{\alpha_k^{(r)}} = \max_{1 \leq i \leq n-r} \left[ \frac{a_i^{(r)}}{\alpha_i^{(r)}} \right] \right\}, \quad r = 1, 2, \dots$$

That is, we form the ratios of the diagonal entries in  $\mathbf{T}^{(r)}$  divided by the correspondingly permuted entries of the original matrix  $\mathbf{A}$ . This strategy is more modest in that it uses *relative* pivot sizes rather than absolute pivot sizes.

**3.2. Fundamental Pivoting Step.** Needless to say, the  $1 \times 1$  case is special and is easily disposed of. So let us assume that  $n > 1$ . Suppose that at the r-th stage,  $r = 2, \dots, n-1$ , we have obtained a matrix of the form

$$\mathbf{A}^{(r)} = \mathbf{D}^{(r)} \oplus \mathbf{T}^{(r)},$$

as in (3.1). Here  $\mathbf{A}^{(0)} = \mathbf{A} = \mathbf{T}[(a_1, \dots, a_n)(b_1, \dots, b_{n-1})]$  and  $\mathbf{D}^{(0)} = \emptyset$ . Assume further that we have decided on a new pivot  $a_k^{(r)}$ ,  $k = m_{r+1}$ , for some  $1 \leq k \leq n-r$ . Our aim is to reduce  $\mathbf{T}^{(r)}$  to  $\text{diag}(d_{r+1}, \mathbf{T}^{(r+1)})$  and  $\mathbf{A}^{(r)}$  to

$$\mathbf{A}^{(r+1)} = \mathbf{D}^{(r+1)} \oplus \mathbf{T}^{(r+1)},$$

where  $\mathbf{D}^{(r+1)} = \text{diag}(d_1, \dots, d_{r+1})$  and

$$(3.5) \quad \mathbf{T}^{(r+1)} = \mathbf{T}[(a_1^{(r+1)}, \dots, a_{n-r}^{(r+1)}), (b_1^{(r+1)}, \dots, b_{n-r-1}^{(r+1)})],$$

which is of size  $(n-r) \times (n-r-1)$ . We shall drop the superscripts on the entries in this discussion, since  $r$  will remain fixed. Referring to the  $(1,1)$  entry in  $\mathbf{T}^{(r)}$ , i.e. the  $(r+1, r+1)$  entry in the complete  $n \times n$  matrix, and setting  $m_{r+1} = k$ , we can produce the desired reduction by taking the following three steps:

- (a) Permute rows and columns 1 through  $k$  cyclically followed by the corresponding permutation for columns 1 through  $k$ . This corresponds to the following action on  $\mathbf{T}^{(r)}$ ,

$$(3.6) \quad \mathbf{T}^{(r)} = \left[ \begin{array}{cccc|c|cccc} a_1 & b_1 & & & 0 & 0 & \cdots & \cdots & 0 \\ & b_1 & a_2 & & \vdots & \vdots & & & \vdots \\ & & & \ddots & 0 & \vdots & & & \vdots \\ & & & & a_{k-1} & b_{k-1} & 0 & \cdots & \cdots & 0 \\ \hline 0 & \cdots & 0 & b_{k-1} & a_k & b_k & 0 & \cdots & 0 \\ \hline 0 & \cdots & \cdots & 0 & b_k & a_{k+1} & & & & \\ \vdots & & & \vdots & 0 & & \ddots & & & \\ \vdots & & & \vdots & \vdots & & & \ddots & & \\ 0 & \cdots & \cdots & 0 & 0 & & & & & a_{n-r} \end{array} \right] \longrightarrow$$

$$\left[ \begin{array}{c|cccc|c|cccc} a_k & 0 & \cdots & \cdots & 0 & b_{k-1} & b_k & 0 & \cdots & 0 \\ \hline 0 & a_1 & b_1 & & & 0 & 0 & \cdots & \cdots & 0 \\ \vdots & b_1 & a_2 & & & \vdots & \vdots & & & \vdots \\ \vdots & & & \ddots & & 0 & \vdots & & & \vdots \\ 0 & & & & a_{k-2} & b_{k-2} & 0 & \cdots & \cdots & 0 \\ \hline b_{k-1} & 0 & \cdots & 0 & b_{k-2} & a_{k-1} & 0 & \cdots & \cdots & 0 \\ \hline b_k & 0 & \cdots & \cdots & 0 & 0 & a_{k+1} & b_{k+1} & & \\ 0 & \vdots & & & \vdots & \vdots & b_{k+1} & a_{k+2} & & \\ \vdots & \vdots & & & \vdots & \vdots & & & \ddots & \\ 0 & 0 & \cdots & \cdots & 0 & 0 & & & & a_{n-r} \end{array} \right]$$

- (b) use the pivot  $a_k$  in the  $(1,1)$  position to sweep out  $b_{k-1}$  using  $\gamma_1 = -b_{k-1}/a_k$  and  $b_k$  using the multiplier  $\gamma_2 = -b_k/a_k$ . This gives a new matrix of the form

$$(3.7) \quad a_k \oplus \mathbf{T}^{(r+1)} = \left[ \begin{array}{c|cccc|c|cccc} a_k & 0 & \cdots & \cdots & 0 & 0 & 0 & \cdots & \cdots & 0 \\ \hline 0 & a_1 & b_1 & & & 0 & 0 & \cdots & \cdots & 0 \\ \vdots & b_1 & a_2 & & & \vdots & \vdots & & & \vdots \\ \vdots & & & \ddots & & 0 & \vdots & & & \vdots \\ 0 & & & & a_{k-2} & b_{k-2} & 0 & \cdots & \cdots & 0 \\ \hline 0 & 0 & \cdots & 0 & b_{k-2} & a'_{k-1} & b'_{k-1} & 0 & \cdots & 0 \\ \hline 0 & 0 & \cdots & \cdots & 0 & b'_{k-1} & a'_{k+1} & b_{k+1} & & \\ \vdots & \vdots & & & \vdots & 0 & b_{k+1} & a_{k+2} & & \\ \vdots & \vdots & & & \vdots & \vdots & & & \ddots & \\ 0 & 0 & \cdots & \cdots & 0 & 0 & & & & a_{n-r} \end{array} \right]$$

where

$$(3.8) \quad a'_{k-1} = a_{k-1} + \gamma b_{k-1} = a_{k-1} - b_{k-1}^2/a_k \geq 0$$



**3.3. Pass Through Procedure.** We have seen what the fundamental pivoting step looks like. Let us now address the case where we pivot at every step. Our aim is to show that we can indeed pass the later permutations past the earlier sweeps so that the number of off-diagonal entries *does not grow*, and that, by good bookkeeping we can keep track of the positions of the (at most) two “taillights”, i.e. the (at most) two non-zero off-diagonal entries in each of the columns of  $\mathbf{L}$ . These positions will then be used to solve the tri-sparse triangular systems  $\mathbf{L}\underline{x} = \underline{c}$  and  $\underline{y}^T\mathbf{L} = \underline{d}^T$  in  $0(n)$  steps, which are needed in our final stage.

Suppose we have performed  $(\mathbf{R}_1\mathbf{P}_1)\mathbf{A}(\mathbf{R}_1\mathbf{P}_1)^T = \text{diag}(a_{m_1}, \mathbf{T}^{(1)})$  and have applied  $r$  pivoting steps in succession to  $\mathbf{T}^{(0)}, \mathbf{T}^{(1)}, \dots$  giving

$$(\mathbf{Q}_r \dots \mathbf{Q}_2\mathbf{Q}_1)\mathbf{A}(\mathbf{Q}_r \dots \mathbf{Q}_2\mathbf{Q}_1)^T = \text{diag}(\mathbf{D}^{(r)}, \mathbf{T}^{(r)}),$$

where  $\mathbf{D}^{(r)}$  is diagonal and the pivot indices are  $(k_1, k_2, \dots) = (k, m, s, \dots)$ . Consider the product  $\mathbf{R}_2\mathbf{P}_2(\mathbf{R}_1\mathbf{P}_1)$  and focus on  $\mathbf{P}_2\mathbf{R}_1$ , where  $\mathbf{P}_j$  and  $\mathbf{R}_j$  are given as in (3.12). We now state:

*Lemma 1*

$$\text{diag}[1, -m, \mathbf{I}_l][\mathbf{I} + (\alpha\underline{e}_k + \beta\underline{e}_{k+1})\underline{e}_1^T] = [\mathbf{I} + (\gamma\underline{e}_r + \delta\underline{e}_s)\underline{e}_1^T] \cdot \text{diag}(1, -k, \mathbf{I}_l),$$

where for

$$\begin{aligned} m+1 < k &: r = k, s = k+1, \gamma = \alpha, \delta = \beta \\ m+1 = k &: r = 2, s = k+1, \gamma = \beta, \delta = \alpha \\ m+1 > k &: r = m-k+2, s = m-k+3, \gamma = \beta, \delta = \alpha. \end{aligned}$$

*Proof;* Direct verification.

We note that this says  $\mathbf{P}'_2\mathbf{R}_1 = \mathbf{R}'_1\mathbf{P}'_2$ . Repeating this, we see that  $\mathbf{R}_3\mathbf{P}_3(\mathbf{R}_2\mathbf{P}_2)(\mathbf{R}_1\mathbf{P}_1) = [\mathbf{R}_3\mathbf{R}_2^{(1)}\mathbf{R}_1^{(2)}]\mathbf{P}_3\mathbf{P}_2\mathbf{P}_1$ .

It is now clear (by induction) what happens in general. Indeed,  $\mathbf{R}_r^{(\cdot)}$  only has two non-zero off-diagonal entries in column  $r$ , while  $\mathbf{P}_r = \text{diag}(\mathbf{I}_{r-1}, -m_r, \mathbf{I})$ . Combining the operations  $\mathbf{Q}_r \dots \mathbf{Q}_1$  we arrive at

$$(3.14) \quad \mathbf{R}\mathbf{P}\mathbf{A}\mathbf{P}^T\mathbf{R}^T = \mathbf{M} \quad \text{or} \quad \mathbf{P}\mathbf{A}\mathbf{P}^T = \mathbf{L}\mathbf{M}\mathbf{L}^T,$$

where  $\mathbf{M} = \text{diag}(\mathbf{D}^{(r)}, \mathbf{T}^{(r)})$  and  $\mathbf{L} = [\mathbf{R}_1^{(\cdot)}]^{-1}[\mathbf{R}_2^{(\cdot)}]^{-1} \dots [\mathbf{R}_r^{(\cdot)}]^{-1}$ . Considering these factors as column operations we see that  $\mathbf{L}$  is lower triangular with *at most two* non-zero off-diagonal entries in each column.

Let us now turn to the question of how to compute the matrices  $\mathbf{L}$ ,  $\mathbf{P}$  and  $\mathbf{D}$ , which will be needed for our efficient computation of the  $MP$  inverse.

**3.4. Computation of the Three Matrices.** On account of their sparsity, the computation of the three matrices  $\mathbf{D}$ ,  $\mathbf{P}$  and  $\mathbf{L}$  can most economically be done using a set of row vectors. Of these the first two computations are trivial, unlike the computation of  $\mathbf{L}$ , which is far from trivial. It requires simultaneously keeping track of the values as well as the exact locations of the two “taillights” in each column of  $\mathbf{L}$ , under a stream of permutations. We shall in fact need eight strings to completely keep track of all action (our implementation takes  $5n$  double and  $3n$  integer words).

Suppose we have reached stage  $r$ ,  $1 \leq r \leq n-2$ , where we obtained

$$(3.15) \quad \text{diag}(d_1, \dots, d_r, \mathbf{T}^{(r)})$$

with  $\mathbf{T}^{(r)} = \mathbf{T}[(a_1^{(r)}, a_2^{(r)}, \dots, a_{n-r}^{(r)})(b_1^{(r)}, b_2^{(r)}, \dots, b_{n-r-1}^{(r)})]$ . The associated sequences that we have created are the following:

<b>notation</b>	<b>indicating</b>
$f^{(r)} = [k_1, k_2, \dots, k_r, 0, \dots, 0]$	the pivot indices
$d^{(r)} = [d_1, d_2, \dots, d_r, 0, \dots, 0]$	the diagonal entries
$a^{(r)} = [a_1^{(r)}, a_2^{(r)}, \dots, a_{n-r}^{(r)}]$	the diagonal of $\mathbf{T}^{(r)}$
$b^{(r)} = [b_1^{(r)}, b_2^{(r)}, \dots, b_{n-r-1}^{(r)}]$	the subdiagonal of $\mathbf{T}^{(r)}$
$p^{(r)} = [p_1^{(r)}, p_2^{(r)}, \dots, p_{n-r}^{(r)}]$	the product $\mathbf{P}_r \dots \mathbf{P}_1$

We also have two position sequences  $(\tau^{(r)})$  and  $(\mu^{(r)})$  for the taillights, in addition to two “value” sequences  $(g^{(r)})$  and  $(h^{(r)})$ , which respectively indicate the positions and values of the two taillights. We shall return to these shortly.

At the first stage we compute the pivot index  $m = k_1$  from

$$m = \min\{s; a_s = \max\{a_1, \dots, a_n\}\}$$

while at the later stages we find the pivot indices  $m = k_{r+1}$  from

$$m = \min\left\{s; \frac{a_s^{(r)}}{\alpha_s^{(r)}} = \max_{1 \leq i \leq n-r} \left[ \frac{a_i^{(r)}}{\alpha_i^{(r)}} \right] \right\}, \quad r = 2, 3, \dots$$

From this we get the next diagonal entry  $d_{r+1} = a_m^{(r)}$  as well as the next pivot index  $k_{r+1} = m$ , giving the new strings

$$f^{(r+1)} = [k_1, k_2, \dots, k_{r+1}, 0, \dots, 0] \quad \text{and} \quad d^{(r+1)} = [d_1, d_2, \dots, d_{r+1}, 0, \dots, 0].$$

We next compute the entries in  $\mathbf{T}^{(r+1)}$  from

$$(3.16) \quad \begin{aligned} a_i^{(r+1)} &= a_i^{(r)} & i &= 1, \dots, m-2 \\ a_{m-1}^{(r+1)} &= a_{m-1}^{(r)} - [b_{m-1}^{(r)}]^2 / a_m^{(r)} \geq 0 \\ a_m^{(r+1)} &= a_{m+1}^{(r)} - [b_m^{(r)}]^2 / a_m^{(r)} \geq 0 \\ a_i^{(r+1)} &= a_{i+1}^{(r)} & i &= m+1, \dots, n-r-1 \end{aligned}$$

and we drop  $a_{n-r}^{(r)}$  from our list. Likewise we have

$$(3.17) \quad \begin{aligned} b_i^{(r+1)} &= b_{i+1}^{(r)} & i &= 1, \dots, m-2 \\ b_{m-1}^{(r+1)} &= -b_m^{(r)} b_{m-1}^{(r)} / a_m^{(r)} & \text{and} \\ b_i^{(r+1)} &= b_{i+1}^{(r)} & i &= m, \dots, n-r-2 \end{aligned}$$

and we drop  $b_{n-r-1}^{(r)}$ . We thus have

$$a^{(r+1)} = [a_1^{(r+1)}, \dots, a_{n-r-1}^{(r+1)}] \quad \text{and} \quad b^{(r+1)} = [b_1^{(r+1)}, \dots, b_{n-r-2}^{(r+1)}].$$

Let us now turn to the permutation problem, which involves not only the accumulation of the permutation matrices, but also their action on the two taillights. It is clear from Lemma 1 that the values of the two taillights do not change after their first introduction, but that their position does change due to the stream of permutations  $\mathbf{P}_r, \mathbf{P}_{r+1}, \dots$

Now once we have the next pivot index  $k_{r+1}$  then we also know the next permutation matrix needed i.e.,

$$(3.18) \quad \mathbf{P}_{r+1} = \text{diag}(\mathbf{I}_r, -k_{r+1}, \mathbf{I}) \quad .$$

This must applied to all previous permutation matrices ( $\mathbf{P}_r \dots \mathbf{P}_1$ ). In order to do this efficiently, let us first make some pertinent remarks concerning permutations.

We shall keep track of permutations as permutation matrices, however we shall do our bookkeeping by using rows. Indeed, if  $\mathbf{P} = [\underline{e}_{i_1}, \underline{e}_{i_2}, \dots, \underline{e}_{i_n}]$ , then  $\mathbf{P}\underline{e}_k = \underline{e}_{i_k}$ . That is,  $\mathbf{P}$  maps input integer “ $k$ ” into output integer “ $i_k$ ”. Associated with  $\mathbf{P}$  is the permutation map  $\pi = \begin{bmatrix} 1 & 2 & 3 \dots & n \\ i_1 & i_2 & \dots & i_n \end{bmatrix}$  which we shorten to  $\pi = (i_1, \dots, i_n)$ . Clearly  $\pi(k) = i_k$  is the  $k$ -th entry in  $\pi$ , and  $\mathbf{P}\underline{e}_k = \underline{e}_l$  exactly when  $\pi(k) = l$ . Needless to say the string  $(i_1, \dots, i_n)$  is made up of the *row* indices of the unit vectors in  $\mathbf{P}$ .

Returning to  $\mathbf{P}_{r+1}$ , we have

$$(3.19) \quad \pi_{r+1} = (1, 2, \dots, r | r+2, r+3, \dots, k_{r+1} + r, r+1 | k_{r+1} + r + 1, \dots, n) \quad ,$$

which acts on the string  $p^{(r)}$  to give

$$(3.20) \quad p^{(r+1)} = [\pi_{r+1}(p_1^{(r)}), \pi_{r+1}(p_2^{(r)}), \dots, \dots, \pi_{r+1}(p_n^{(r)})] \quad .$$

Next consider the two taillights. For each we shall create a pair of sequences; one to keep track of its size and one to keep track of its position. To avoid confusion all indices shall be relative to the top of the complete  $n \times 1$  vector.

Consider first the position question. At stage  $r$  the matrix  $\mathbf{R}_{r+1}$  has the form  $\text{diag}$

$$(3.21) \quad \mathbf{R}_{r+1} = [\underline{e}_1, \dots, \underline{e}_r, \underline{y}_{r+1}, \underline{e}_{r+2}, \dots, \underline{e}_n] \quad ,$$

where  $\underline{y}_{r+1} = \alpha_{r+1}\underline{e}_{k_r+r} + \beta_{r+1}\underline{e}_{k_r+r+1}$ , so that the taillights in column  $(r+1)$  have starting vectors of  $(\underline{e}_{k_r+r}, \underline{e}_{k_r+r+1})$  and we have to compute the product

$$(3.22) \quad \mathbf{P}_{n-1} \dots \mathbf{P}_{r+1}(\underline{e}_{k_r+r}, \underline{e}_{k_r+r+1}) \quad , \quad r = 1, 2, \dots, n-2 \quad ,$$

to obtain their final positions. In order to do this we introduce taillight position sequences  $\tau^{(r)}$  and  $\mu^{(r)}$ , for  $\alpha_r$  and  $\beta_r$  respectively. In particular  $\tau^{(1)} = (k_1, 0, \dots, 0)$  and  $\mu^{(1)} = (k_1 + 1, 0, \dots, 0)$ , to which we apply  $\pi_2$  and add the new taillight positions in column 2, giving  $\tau^{(2)} = (\pi_2(k_1), k_2 + 1, \dots, 0)$  and  $\mu^{(2)} = (\pi_2(k_1 + 1), k_2 + 2, \dots, 0)$ . Repeating this we get at stage  $r$  that

$$(3.23) \quad \tau^{(r)} = [\tau_1^{(r)}, \tau_2^{(r)}, \dots, \tau_{r-1}^{(r)}; \quad k_r + r - 1, 0, \dots, 0] \quad \text{and}$$

$$(3.24) \quad \mu^{(r)} = [\mu_1^{(r)}, \mu_2^{(r)}, \dots, \mu_{r-1}^{(r)}; \quad k_r + r, 0, \dots, 0] \quad ,$$

which gives the position vector of the taillights in the first  $r$  columns of  $\mathbf{L}$  at a stage  $r$ . If we now apply  $\pi_{r+1}$  from (3.19) then we arrive at the next position vectors:

$$\tau^{(r+1)} = [\pi_{r+1}(\tau_1^{(r)}), \pi_{r+1}(\tau_2^{(r)}), \dots, \pi_{r+1}(\tau_{r-1}^{(r)}), \pi_{r+1}(k_r + r - 1), k_r + r, 0, \dots, 0]$$

and

$$\mu^{(r+1)} = [\pi_{r+1}(\mu_1^{(r)}), \pi_{r+1}(\mu_2^{(r)}), \dots, \pi_{r+1}(\mu_{r-1}^{(r)}), \pi_{r+1}(k_r + r), k_r + r + 1, 0, \dots, 0] \quad .$$

We note that in all cases  $\tau_r^{(r)} = k_r + r - 1$ ,  $\mu_r^{(r)} = k_r + r$  and that  $\tau_i^{(r)} = 0$  for  $i > r$ , while  $\tau_{r-1}^{(r)} = \pi_r(k_{r-1} + r - 2)$ . We repeat the above for  $r = 1, 2, \dots, n-2$ , and treat the case  $r = n-1$  separately, since then we only have *one* off-diagonal entry.

Lastly, we turn to the computation of the actual values of the taillights  $\alpha_i$  and  $\beta_i$ . These are computed once and then stored. If at stage  $r$  we have the value strings

$$g^{(r)} = [g_1, g_2, \dots, g_r, 0, \dots, 0] \quad \text{and} \quad h^{(r)} = [h_1, h_2, \dots, h_r, 0, \dots, 0]$$

then we compute the next values from

$$g_{r+1} = -b_{m-1}^{(r)}/a_m^{(r)} \quad \text{and} \quad h_{r+1} = b_m^{(r)}/a_m^{(r)} \quad ,$$

where again  $m = k_{r+1}$ . We conclude this section with the observation that the number of non-zero entries in  $\mathbf{L}$  is bounded above by  $3n - 3$ .

Having obtained the permuted L-D-L<sup>T</sup> factorization let us now turn to its applications.

#### 4. Tridiagonal Solution.

**4.1. Positive Definite Case.** Needless to say the above algorithm also works in the case where  $\mathbf{A}$  is tri-diagonal and positive definite real symmetric. We claim that our algorithm is more stable since it uses optimal pivoting at *all* stages.

Suppose we have found  $\mathbf{PAP}^T = \mathbf{LDL}^T$ , where  $\mathbf{A}$  and  $\mathbf{D}$  are invertible and we want to solve  $\mathbf{Ax} = \underline{b}$ . Then the unique solution becomes  $\underline{x} = \mathbf{A}^{-1}\underline{b} = \mathbf{P}^T\mathbf{L}^{-T}\mathbf{D}^{-1}\mathbf{L}^{-1}\mathbf{P}\underline{b}$ , which we compute in several stages. First we compute  $\underline{y} = \mathbf{L}^{-1}\mathbf{P}\underline{b}$  by solving  $\mathbf{Ly} = \mathbf{P}\underline{b}$ . Then we form  $\mathbf{D}^{-1}\underline{y}$  and solve  $\mathbf{L}^T\underline{z} = \mathbf{D}^{-1}\underline{y}$  for  $\underline{z}$ . A final premultiplication  $\mathbf{P}^T\underline{z}$  yields the desired answer  $\underline{x}$ . The only extra feature that we need is a tri-sparse solver for the systems  $\mathbf{L}\underline{x} = \underline{d}$  and  $\mathbf{L}^T\underline{x} = \underline{d}$ , which is more efficient than using the sparse matrix solver of [9]. This we now undertake.

**4.2. Tri-Sparse Solver.** Suppose  $\mathbf{L}$  is a column-tri-sparse unit-diagonal, lower triangular  $n \times n$  matrix. That is each column in  $\mathbf{L}$  has at most three non-zero entries. Let  $\begin{bmatrix} \tau_i \\ \mu_i \end{bmatrix}$  and  $\begin{bmatrix} g_i \\ h_i \end{bmatrix}$ ,  $i = 1, \dots, n$ , denote the position vectors and the values strings of the two taillights in each column. (For  $i > t$  they are absent.) In the computation of the *MP* inverse we shall have to solve the systems  $\mathbf{L}\underline{x} = \underline{b}$  and  $\underline{y}^T\mathbf{L} = \underline{d}^T$  by forward and backward substitution. Our aim is to show that in either case the operations count is linear.

Let us begin with the simpler case  $\underline{y}^T\mathbf{L} = \underline{d}^T$ , which we transpose to give  $\mathbf{L}^T\underline{y} = \underline{d}$ , where  $\mathbf{L}^T$  is row-tri-sparse, with taillights  $(\mathbf{L}^T)_{i\tau_i} = g_i$  and  $(\mathbf{L}^T)_{i\mu_i} = h_i$  in row  $i$ . Back substituting we obtain

$$(4.1) \quad \begin{aligned} y_n &= d_n \\ y_r &= d_r - g_r y_{\tau_r} - h_r y_{\mu_r} \quad , \quad r = n-1, \dots, 1. \end{aligned}$$

From this we at once see that  $OC = (2n - 3, 2n - 3, 0)$ . In order to solve the system  $\mathbf{L}\underline{x} = \underline{b}$ , let us first estimate the operations count. To this effect we shall need the following trivial result.

*Lemma 2* If  $\mathbf{M}$  is sparse  $m \times n$  with  $\#(\mathbf{M})$  non-zero entries, then the product  $\mathbf{M}\underline{b}$  uses

$$OC = (\#(\mathbf{M}), \#(\mathbf{M}) - m, 0).$$

Now partition  $\mathbf{L}$  as  $\mathbf{L} = \begin{bmatrix} \mathbf{I}_r & \mathbf{0} \\ \mathbf{C}_1 & \mathbf{L}_2 \end{bmatrix}$ , where the first row of  $\mathbf{C}_1$  is non-zero, and  $\underline{x}$  and  $\underline{b}$  conformably as  $\underline{x} = \begin{bmatrix} \underline{x}_1 \\ \underline{x}_2 \end{bmatrix}$  and  $\underline{b} = \begin{bmatrix} b_1 \\ b_2 \end{bmatrix}$ . Then  $\underline{x}_1 = \underline{b}_1$  and  $\mathbf{L}_2\underline{x}_2 = \underline{b}_2 - \mathbf{C}_1\underline{b}_1$ . From Lemma 2 we see that the computation  $\mathbf{C}_1\underline{b}_1$  uses  $\#(\mathbf{C}_1)$  multiplications as well as  $\#(\mathbf{C}_1) - (m - r)$  additions and hence the operations count for  $\underline{b}_2 - \mathbf{C}_1\underline{b}_1$  is  $(\#(\mathbf{C}_1), \#(\mathbf{C}_1), 0)$ . Now repeat with  $\mathbf{L}_2 = \begin{bmatrix} \mathbf{I} & \mathbf{0} \\ \mathbf{C}_2 & \mathbf{L}_3 \end{bmatrix}$ ,  $\begin{bmatrix} \mathbf{I} & \mathbf{0} \\ \mathbf{C}_3 & \mathbf{L}_4 \end{bmatrix}$  etc. Since the taillights present in  $\mathbf{C}_i$  are disjoint, the total operations count (by induction) is

$$(4.2) \quad OC = \left( \sum_i (\#(\mathbf{C}_i)), \sum_i (\#(\mathbf{C}_i)), 0 \right) = (\#(\mathbf{L}) - n, \#(\mathbf{L}) - n, 0) \quad ,$$

in which

$$\#(\mathbf{L}) \leq 3n - 3.$$

This is the same as for the first case. To actually implement this algorithm, we test the entries in  $(\tau_i)$  and  $(\mu_i)$  to see which equal  $r = 1, 2, \dots$ . Subsequently, by forward substitution we arrive at  $x_1 = b_1$  and for  $r = 2, \dots, n$

$$(4.3) \quad x_r = b_r - \sum_{\tau_i=r} g_i x_i - \sum_{\mu_j=r} h_j x_j \quad .$$

Instead of performing  $n^2$  comparisons we could alternatively sort the strings  $(\tau_i)$  and  $(\mu_i)$  in increasing order, as

$$\tau_{i_1} = \dots = \tau_{i_r} < \tau_{j_1} = \dots = \tau_{j_s} < \dots < \tau_{k_1} = \dots = \tau_{k_t}$$

etc., after which no more comparisons are needed in the back substitution. We note that we actually computed the conjugate strings for the sequences  $(\tau_i)$  and  $(\mu_i)$ . We are now ready for *MP* inversion.

**4.3. MP Inversion.** When the matrix  $\mathbf{A}$  is singular we may use the permuted L-D-L<sup>T</sup> factorization to find the *MP* inverse of  $\mathbf{A}$  and, if desired compute the *mlss*  $\mathbf{A}^\dagger \underline{b}$  to  $\mathbf{A} \underline{x} = \underline{b}$ . Indeed, we first tri-diagonalize  $\mathbf{A}$  using Householder transformations in  $O(2n^3/3)$  flops after which we apply our algorithm.<sup>1</sup>

We begin by recalling a simple result [3], [12] which will be needed in what follows.

*Lemma 3*

$$(a) \quad \begin{bmatrix} \mathbf{I} \\ \mathbf{F} \end{bmatrix} = (\mathbf{I} + \mathbf{F}^T \mathbf{F})^{-1} [\mathbf{I}, \mathbf{F}^T]$$

$$(b) \quad \text{if } \mathbf{H}\mathbf{H}^{-1} = \mathbf{I} \quad \text{then} \quad \left[ \begin{bmatrix} \mathbf{I} \\ \mathbf{F} \end{bmatrix} \mathbf{H} \right]^\dagger = \mathbf{H}^\dagger \begin{bmatrix} \mathbf{I} \\ \mathbf{F} \end{bmatrix}^\dagger$$

*Proof:*

(a) if  $\mathbf{M}^{-1}\mathbf{M} = \mathbf{I}$  then it is well known that  $\mathbf{M}^\dagger = (\mathbf{M}^T \mathbf{M})^{-1} \mathbf{M}^T$ .

(b) For a full-rank-factorization  $\mathbf{M} = \mathbf{B}\mathbf{C}$  it follows that  $\mathbf{M}^\dagger = \mathbf{C}^\dagger \mathbf{B}^\dagger$ .

Consider the final stage after  $t$  iterations where we have

$$\mathbf{P}\mathbf{A}\mathbf{P}^T = \mathbf{L}\mathbf{M}\mathbf{L}^T \quad \text{with} \quad \mathbf{M} = \begin{bmatrix} \mathbf{E} & \mathbf{0} \\ \mathbf{0} & \mathbf{A}^{(t)} \end{bmatrix}$$

and  $\mathbf{E} \in \mathcal{R}^{t \times t}$  positive diagonal. Recall that  $\mathbf{A}^{(t)}$  was set equal to zero on account of the failure of the Frane test, and yielded the number of “hidden” zeros in the tridiagonal block say  $s = n - t$ . Needless to say in most cases of interest we may assume that  $s \ll n$ . Let us now partition  $\mathbf{L}$  as

$$\mathbf{L} = [\mathbf{L}_1, \mathbf{L}_2] = \begin{bmatrix} \mathbf{K} & \mathbf{0} \\ \mathbf{C} & \mathbf{I}_s \end{bmatrix} = \mathbf{R}^{-1},$$

where  $\mathbf{K} \in \mathcal{R}^{t \times t}$  is trispars, and lower triangular. If we next consider

$$(4.4) \quad \mathbf{L}\mathbf{M}\mathbf{L}^T = \begin{bmatrix} \mathbf{K} & \mathbf{0} \\ \mathbf{C} & \mathbf{I} \end{bmatrix} \begin{bmatrix} \mathbf{E} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} \end{bmatrix} \begin{bmatrix} \mathbf{K}^T & \mathbf{C}^T \\ \mathbf{0} & \mathbf{I} \end{bmatrix}.$$

<sup>1</sup>Here, as in the following flop counts, only the cubic terms are counted. We use the older definition of *flop* for one floating point multiplication or division including an addition as used in the first edition of [10].

then  $\mathbf{LML}^T = \mathbf{L}_1\mathbf{E}\mathbf{L}_1^T$  and hence  $(\mathbf{LML}^T)^\dagger = (\mathbf{L}_1\mathbf{E}\mathbf{L}_1^T)^\dagger = \mathbf{L}_1^{T\dagger}\mathbf{E}^{-1}\mathbf{L}_1^\dagger$ . Consequently all we need is to find  $\mathbf{L}_1^\dagger$ . Suppose that  $\mathbf{R} = \begin{bmatrix} \mathbf{S} & \mathbf{0} \\ \mathbf{U} & \mathbf{I} \end{bmatrix} = \mathbf{L}^{-1}$ . Then  $\mathbf{S} = \mathbf{K}^{-1}$ ,  $\mathbf{U} = -\mathbf{C}\mathbf{K}^{-1} \in \mathcal{R}^{s \times t}$  and  $\mathbf{C} = -\mathbf{U}\mathbf{K}^{-1} \in \mathcal{R}^{s \times t}$ . Now  $\mathbf{L}_1 = \begin{bmatrix} \mathbf{K} \\ \mathbf{C} \end{bmatrix} = \begin{bmatrix} \mathbf{I} \\ \mathbf{C}\mathbf{K}^{-1} \end{bmatrix} \mathbf{K} = \begin{bmatrix} \mathbf{I} \\ -\mathbf{U} \end{bmatrix} \mathbf{K}$ , and hence by Lemma 3, we get  $\mathbf{L}_1^\dagger = \mathbf{K}^{-1} \begin{bmatrix} \mathbf{I} \\ -\mathbf{U} \end{bmatrix}^\dagger = \mathbf{S}(\mathbf{I}_t + \mathbf{U}^T\mathbf{U})^{-1}[\mathbf{I}, -\mathbf{U}]$ . Now since  $(\mathbf{I}_t + \mathbf{U}^T\mathbf{U})^{-1} = \mathbf{I} - \mathbf{U}^T(\mathbf{I}_s + \mathbf{U}\mathbf{U}^T)^{-1}\mathbf{U}$ , we only need to invert an  $s \times s$  matrix  $\mathbf{W} = \mathbf{I}_s + \mathbf{U}\mathbf{U}^T$ . Thus for each block we have

$$(4.5) \quad \mathbf{A}^\dagger = \mathbf{P}^T \mathbf{L}_1^{T\dagger} \mathbf{E}^{-1} \mathbf{L}_1^\dagger \mathbf{P} = \mathbf{P}^T \begin{bmatrix} \mathbf{I} - \mathbf{U}^T \mathbf{W}^{-1} \mathbf{U} \\ -\mathbf{W}^{-1} \mathbf{U} \end{bmatrix} \mathbf{S}^T \mathbf{E} \mathbf{S} [\mathbf{I} - \mathbf{U}^T \mathbf{W}^{-1} \mathbf{U}, -\mathbf{U}^T \mathbf{W}^{-1}] \mathbf{P} \quad ,$$

which is symmetric. To compute  $\mathbf{W}^{-1} = (\mathbf{I}_s + \mathbf{U}\mathbf{U}^T)^{-1}$  we either compute the factor - of  $\mathbf{W}$  and then find  $\mathbf{W}^{-1}$  by forward and backward substitution, or we may consider it a Cholesky update of the identity.

To compute the *mnlss*  $\mathbf{A}^\dagger \underline{b}$  we perform the following steps:

- (a) Find column tri-sparse  $\mathbf{K}$  and  $\mathbf{C}$  from the first  $t$  columns of  $\mathbf{L}$ .
- (b) Compute  $\mathbf{U}$  from  $\mathbf{U}\mathbf{K} = -\mathbf{C}$  using the row tri-sparse solver. Since  $\mathbf{K}$  is of order  $t$ , we obtain an operations count of  $OC = (s(2t-3), s(2t-3), 0)$ . It goes without saying that this calculation can only be performed once we have found the matrices  $\mathbf{K}$  and  $\mathbf{C}$ , that is at the end of the first phase.
- (c) Form  $\mathbf{U}\mathbf{U}^T$  and  $\mathbf{W} = \mathbf{I}_s + \mathbf{U}\mathbf{U}^T$ . This uses  $OC = (.5s(s+1), .5s(s-1)(t-1) + s, 0)$ .
- (d) Factor  $\mathbf{W} = - -^T$  using Cholesky updates of  $\mathbf{I}$ . This takes  $OC = (2\tilde{t}s(s+1), \tilde{t}s^2, 0)$  plus  $\tilde{t}s$  square roots, where  $\tilde{t} \leq t$  is the number of nonzero columns of  $\mathbf{U}$ .
- (e) The following is a loop to be executed for each right-hand vector  $\underline{b}$ .

(i) Form  $\mathbf{P}\underline{b} = \begin{bmatrix} \underline{w} \\ \underline{\eta} \end{bmatrix}$  where  $\underline{w} \in \mathcal{R}^{t \times 1}$  and  $\underline{\eta} \in \mathcal{R}^{s \times 1}$  (this only uses interchanges).

(ii) Compute  $[\mathbf{I}_t - \mathbf{U}^T \mathbf{W}^{-1} \mathbf{U}, -\mathbf{U}^T \mathbf{W}^{-1}] \begin{bmatrix} \underline{w} \\ \underline{\eta} \end{bmatrix} = \underline{w} - \mathbf{U}^T \mathbf{W}^{-1} (\underline{\eta} + \mathbf{U}\underline{w}) = \underline{v} \in \mathcal{R}^{t \times 1}$ . This

requires the following computations:

(a) $\mathbf{U}\underline{w}$	$OC = (st, s(t-1), 0)$
(b) $\mathbf{U}\underline{w} + \underline{\eta} = \underline{h}$	$OC = (0, s, 0)$
(c) $\mathbf{U}^T \mathbf{W}^{-1} \underline{h} = \mathbf{U}^T - -^T - -^1 \underline{h}$ : $\underline{x}_1 = -^T \underline{x}_2$ , $\underline{h} = - \underline{x}_1$ $\mathbf{U}^T \underline{x}_2$	$OC = (s(s-1), s(s-1), s)$ $OC = (ts, t(s-1), 0)$
(d) $\underline{v} = \underline{w} - \mathbf{U}^T \underline{x}_2 \in \mathcal{R}^{t \times 1}$	$OC = (0, t, 0)$
(e) $\mathbf{K}\underline{y} = \underline{v}$ $\underline{y} = \mathbf{K}^{-1} \underline{v} = \mathbf{S}\underline{v}$ .	column-tri-sparse solver: $OC = (2t-3, 2t-3, 0)$
(f) $\mathbf{E}^{-1} \underline{y}$	$OC = (0, 0, t)$
(g) $\mathbf{K}^T \underline{f} = \mathbf{E}^{-1} \underline{y}$ $\underline{f} = \mathbf{S}^T \mathbf{E}^{-1} \underline{y} \in \mathcal{R}^{t \times 1}$	row-tri-sparse solver: $OC = (2t-3, 2t-3, 0)$
(h) $\mathbf{U}\underline{f} \in \mathcal{R}^{s \times 1}$	$OC = (st, s(t-1), 0)$
(i) $\underline{g}_2 = -\mathbf{W}^{-1} \mathbf{U}\underline{f}$ : $- \underline{y}_1 = -\mathbf{U}\underline{f}$ , $-^T \underline{y}_2 = \underline{y}_1$	$OC = (s(s-1), s(s-1), 2s)$
(j) $\underline{f}_1 = \mathbf{U}^T \underline{g}_2 \in \mathcal{R}^{t \times 1}$	$OC = (ts, t(s-1), 0)$
(k) $\underline{g}_1 = \underline{f} + \underline{f}_1 \in \mathcal{R}^{t \times 1}$	$OC = (0, t, 0)$
(l) $\mathbf{P} \begin{bmatrix} \underline{g}_1 \\ \underline{g}_2 \end{bmatrix}$	uses only permutations

Adding the above we see that the total operations count is

$$(4.6) \quad \begin{aligned} \mathbf{M} &= \frac{1}{2}s^2t + 6.5st + 4t + \frac{1}{6}s^3 + 2s^2 - 5s - 6 \\ \mathbf{A} &= \frac{1}{2}s^2t + 5.5st + 4t + \frac{1}{6}s^3 + s^2 - \frac{5}{6}s - 6 \\ \mathbf{D} &= \frac{1}{2}s^2 + 4s + t. \end{aligned}$$

When  $s = 1$  and  $t = n - 1$  this collapses to

$$OC = (11n - 20, 10n - 2, n + 7.5).$$

#### Remarks

Besides the positive definite case, in the actual implementation, we also separated out the cases where we had a small or large number of right-hand columns  $\underline{b}$ . In the latter case we first compute  $\mathbf{L}^\dagger = \mathbf{K}^{-1} \begin{bmatrix} \mathbf{I} \\ -\mathbf{U} \end{bmatrix}^\dagger$  by forward substitution and then form  $\mathbf{E}^{-1}\mathbf{K}^{-1} \begin{bmatrix} \mathbf{I} \\ -\mathbf{U} \end{bmatrix}^\dagger$  followed by  $\mathbf{L}_1^{T\dagger}(\mathbf{E}^{-1}\mathbf{L}_1^\dagger\mathbf{P})$ . This is then multiplied by each of the columns  $\underline{b}_i$ . For a large number of right-hand columns  $\underline{b}_i$  this may be faster than the columnwise approach.

#### 5. Numerical Experiments.

- The final section of this paper
1. describes some important details of the implementation of our algorithm,
  2. compares the computation time of our algorithm with that of some other known methods for the minimum norm solution of problems with semidefinite tridiagonal matrices,
  3. and shows how the computer times of some traditional methods for computing minimum norm solutions for dense semidefinite matrices compare to a 2-stage combination of our algorithm with Householder tridiagonalization.

In a succeeding paper we describe a (nontrivial) way to combine our method with Aasen's method for the minimum norm solution of problems with large sparse or dense semidefinite matrices. The same approach can be used to combine this tridiagonal method with the Bunch-Kaufman method for the minimum norm solution of rankdeficient linear systems. All computer code referred to in this section consists of carefully coded standard C versions (only single indexed arrays and no F2C translations) of subroutines in BLAS, LINPACK, and LAPACK except for blockwise extensions. We decided not to use the blockwise versions of LAPACK, since our code was also not written in blocked form and we wanted to compare software relatively to its algorithms and not the strength of its implementation for a specific platform (hardware).

**5.1. The Implementation.** A set of subroutines was written in C language for the pivoted decomposition and the solution of singular and nonsingular systems  $\mathbf{A}\mathbf{X} = \mathbf{B}$ , of tridiagonal *psd* matrices  $\mathbf{A} \in \mathcal{R}^{n \times n}$ , with  $m$  right-hand side columns  $\mathbf{B} = [\underline{b}_1, \dots, \underline{b}_m] \in \mathcal{R}^{n \times m}$ .

1. During an outer cycle, the driver routine checks the remaining part of  $\mathbf{A}$  for small diagonal ( $a_i$ ) or off-diagonal ( $|b_i|$ ) elements ([8] p. 295)

$$(5.1) \quad |b_i| \leq \max(\text{tol}, \text{tol}(a_{i+1} + a_i)) \quad \text{or} \quad a_i \leq \text{sing},$$

trying to separate the input matrix  $\mathbf{A} \in \mathcal{R}^{n \times n}$  into a series of smaller blockdiagonal matrices,  $\mathbf{A} = \text{diag}(\mathbf{A}_\sigma, \sigma = 1, 2, \dots)$  consisting of rows and columns of  $\mathbf{A}$  inbetween indices  $i_{\sigma_l} < i_{\sigma_u}$ . Each tridiagonal submatrix  $\mathbf{A}_\sigma$  is then treated separately. For convenience we have used

$$\text{sing} = \text{tol} = \epsilon n C \|\mathbf{A}\|_F \quad \text{with} \quad C = \begin{cases} 100 & \text{for } n \leq 200 \\ 1000 & \text{otherwise,} \end{cases}$$

where  $\| \cdot \|_F$  stands for the Frobenius norm and  $\epsilon$  is the machine precision. Similar techniques are used for detecting the block structure in tridiagonal matrices by subroutines for the computation of eigenvalues (see [2]).

2. For the factorization of the tridiagonal submatrix we experimented with three different forms of pivoting. Assume, we are in stage  $r$  and decide for pivot index  $m$ ,  $m \geq r$ :

**MaxP:** The traditional approach is to check all remaining diagonal entries and to find a feasible pivot  $a_k$ , where

$$k = \min\{j; a_j = \max\{a_1, \dots, a_{n-r}\}\} \quad (\text{see page 3})$$

**BoosP:** For large dense problems the BATBE (“Burn at both ends”) algorithm is used since less operations are needed when a pivot is used at the begin or the end of the chain of remaining diagonal entries. Since for tridiagonal matrices only a small neighbourhood of the pivot entry is changed in each step, testing the (rarely ever changed) end of the chain does not make much sense and we modified the BATBE to the BOOS (“Burn on one side”) algorithm. That means, we pivot on  $a_k$ , where

$$k = \min\{j; |b_j| \leq a_j, j = 1, \dots, n - r\} \quad (\text{see page 3})$$

**FraneP:** Frane’s ([7]) method pivots on  $a_k$ , where

$$k_1 = \min\{j; a_j = \max\{a_1, \dots, a_n\}\} \quad (\text{see page 4})$$

$$k_{r+1} = \min\left\{j; \frac{a_j^{(r)}}{\alpha_j^{(r)}} = \max_{1 \leq i \leq n-r} \left[ \frac{a_i^{(r)}}{\alpha_i^{(r)}} \right] \right\}, \quad r = 1, 2, \dots$$

The numeric simulation shows that there is not much difference between the computer times of the different pivoting schemes (see table 1 on page 15), however, there are some differences in the numerical stability of detecting the correct number of nullities in the presence of rounding errors. Our implementation tries to avoid unnecessary cache memory swaps when moving the three or four vectors (remaining diagonal  $\underline{a}$ , subdiagonal  $\underline{b}$ , pivot indices  $\underline{p}$ , and for Frane’s criterion the original diagonal  $\underline{\alpha}$ ).

3. After factorizing  $\mathbf{A}_\sigma$ , the driver routine calls one of the following subroutines depending on the nullity  $s$  of the submatrix to solve the system  $\mathbf{A}_\sigma \mathbf{X}_\sigma = \mathbf{B}_\sigma$ .

**s=0** Solve the system for nonsingular  $\mathbf{A}_\sigma$ . For each right-hand side  $\underline{b}_\sigma$  we solve

$$\mathbf{P}^T \mathbf{L}^{-T} \mathbf{D}^{-1} \mathbf{L}^{-1} \mathbf{P} \underline{b}_\sigma$$

by tri-sparse forward and backward substitution.

**s=1** Solve the system when  $\mathbf{A}_\sigma$  only has nullity one; since  $\mathbf{C}$  becomes a  $1 \times (n - 1)$  vector, only  $O(n)$  memory is needed. In this case  $\mathbf{W} \in \mathcal{R}^{s \times s}$  is scalar which greatly simplifies the *MP* computation developed in section 4.3.

**s>1** For a nullity larger than one,  $\mathbf{C}$  is  $s \times t$  and  $O(s^2)$  memory is needed (in general  $O(n^2)$ ) to obtain the *mnlss*. We also have to use a linear solver to solve for  $\mathbf{U}^T \mathbf{W}^{-1}$  for  $\mathbf{W} \in \mathcal{R}^{s \times s}$ . After the system  $\mathbf{U} = -\mathbf{C} \mathbf{K}^{-1}$  is solved for  $\mathbf{U}$ , the Cholesky factor of  $\mathbf{W} = \mathbf{I}_s + \mathbf{U} \mathbf{U}^T$  is computed by rank-1 Cholesky factor update similar to [6], chapter 10. We implemented two modules, one for ‘small’  $m$ , where we perform the solution for each right-hand side column separately,  $\underline{x} = \mathbf{L}_1^{T\dagger} (\mathbf{E}^{-1} (\mathbf{L}_1^\dagger \mathbf{P} \underline{b}_\sigma))$ , and one for ‘large’  $m$ , where we compute first the matrix product on the left and then multiply with all columns of  $\mathbf{B}_\sigma$ ,  $\mathbf{X} = (\mathbf{L}_1^{T\dagger} \mathbf{E}^{-1} \mathbf{L}_1^\dagger) (\mathbf{P} \mathbf{B}_\sigma)$ . This is possible only since for  $s > 1$  we have already  $O(n^2)$  memory available for solving  $\mathbf{U}^T \mathbf{W}^{-1}$ .

**5.2. Solving Systems with Tridiagonal psd Matrices.** The main purpose of our algorithm is the (numerically stable) minimum norm solution of singular (*psd*) tridiagonal linear systems. It was not easy to find public domain code for this problem. We know of only two other methods which can solve the tridiagonal (or sparse) *mnls* problem,

1. a typical orthogonal diagonalization algorithm like that of an implicit tridiagonal QL algorithm which we found in EISPACK [8] and which is basically unchanged in LAPACK.
2. a rank revealing factorization method which would be more general, i.e. solve unsymmetric and even indefinite systems, but in general would also create much more fillin than our more specific method.

Unfortunately we were not able to obtain any rank revealing factorization code even for test purpose only. We also included the C version of the LAPACK subroutine DPTSV in our comparison to have a feeling for the speed loss of our algorithm compared to the very easy algorithm for nonsingular tridiagonal systems. The content of table 1 shows the CPU times of the following algorithms:

**MaxP,BoosP,FraneP** : Three versions of our tridiagonal *psd* L-D-L<sup>T</sup> algorithm, using the three different pivoting strategies MaxP, BoosP, and FraneP.

**DPTSV** : The LAPACK [2] subroutine DPTSV, which can be used only to solve nonsingular systems. (A slightly modified version was used which takes only one decomposition for the solutions of linear systems with multiple right hand-sides.)

**IMTQL** : An implicit tridiagonal QL algorithm (see EISPACK [8]) which diagonalizes a symmetric tridiagonal matrix using a sequence of orthogonal rotations,  $\mathbf{VAV}^T = \mathbf{D}$ , and which solves the rank deficient linear system  $\mathbf{Ax} = \mathbf{b}$  by  $\mathbf{x} = \mathbf{V}^T \mathbf{D}^\dagger \mathbf{Vb}$ .

For different  $n$  and rank  $r(\mathbf{A})$  we construct coefficient matrices  $\mathbf{A}$  in the following way:

1. The  $2n - 1$  elements of a lower bidiagonal matrix  $\tilde{\mathbf{A}}$  are randomly (uniformly distributed in  $[0,10]$ ) generated and an identity matrix is added. (Without adding the identity matrix we obtained sometimes extremely bad conditioned matrices.)
2.  $d = n - r$  diagonal and corresponding off-diagonal elements in  $\tilde{\mathbf{A}}$  are set to zero on equally distant diagonal locations, i.e. matrix  $\mathbf{A}$  is splitted off into  $d(\mathbf{A}) + 1$  submatrices  $\mathbf{A}_\sigma$  of equal dimensions.
3. The symmetric tridiagonal matrix  $\mathbf{A} = \tilde{\mathbf{A}}\tilde{\mathbf{A}}^T$  is computed.

Table 1 shows the CPU time in seconds on a Pentium Pro 200 PC when  $n$  right hand sides  $b_i, i = 1, \dots, n$  were used. The column *rcond* lists the ratio between smallest nonzero and largest eigenvalue of the tridiagonal matrix  $\mathbf{A}$ . All computations are performed in double precision.

n	$d(\mathbf{A})$	rcond	MaxP	BoosP	FraneP	DPTSV	IMTQL
100	0	.1140	.1570	.1570	.1720	.2500	.3750
100	10	.1048	.0310	.0310	.0310	-	.1250
100	20	.0941	.0310	.0310	.0320	-	.1090
300	0	.0940	2.2030	2.1100	2.0780	1.7810	19.609
300	30	.0952	.3440	.3280	.3430	-	2.9690
300	60	.0918	.2650	.2650	.2970	-	2.6400
500	0	.0956	9.2810	9.6870	9.2810	5.1870	84.172
500	50	.0936	.9370	.9220	.9380	-	13.172
500	100	.0920	.7350	.7190	.7970	-	12.250
800	0	.0913	36.922	38.235	37.000	6.3280	369.641
800	80	.0929	2.3910	2.3590	2.3900	-	68.203
800	160	.0910	1.9690	1.9370	2.0460	-	65.687
1000	0	.0913	76.125	73.062	71.109	6.9530	682.453
1000	100	.0913	3.7340	4.1100	3.7190	-	96.672
1000	200	.0917	3.6410	3.6100	3.2340	-	92.968

Since the large matrix  $\mathbf{A}$  is split into equally sized submatrices  $\mathbf{A}_\sigma$  (by setting the corresponding diagonal and subdiagonal elements to zero), the algorithms work much faster for the singular systems.

Table 2 shows the CPU time when only one right hand side is used. In a second column we also report the number of pivot swaps needed for the three pivoting strategies. The number of comparisons in the pivoting process are about  $n^2/2$  for MaxP and FraneP, whereas for BoosP many fewer comparisons are needed. Due to computer time restrictions we skipped executing the implicit triangular QL algorithm for  $n > 1000$ .

n	$d(\mathbf{A})$	MaxP		BoosP		FraneP		DPTSV	IMTQL
100	0	.0160	2,279	.0160	372	.0160	1,982	.0000	.6720
100	10	.0000	319	.0000	323	.0000	289	-	.0630
100	20	.0000	215	.0000	208	.0000	199	-	.0470
300	0	.0780	22,335	.0160	3224	.0310	17,136	.0000	7.6410
300	30	.0000	1,020	.0000	745	.0150	856	-	.3280
300	60	.0160	632	.0160	604	.0000	594	-	.2190
500	0	.0470	61,561	.0310	4,812	.1100	47,943	.0150	35.656
500	50	.0150	1,673	.0160	1,332	.0150	1,493	-	.9200
500	100	.0000	1,026	.0150	1,019	.0160	976	-	.5780
800	0	.1100	162,787	.0630	10,994	.2500	20,806	.0150	141.344
800	80	.0150	2,640	.0150	2,046	.0160	2,325	-	2.3910
800	160	.0150	1,702	.0150	1,675	.0160	1,586	-	1.4380
1000	0	.1560	250,210	.0940	10,189	.4060	18,4882	.0160	262.375
1000	100	.0150	2,477	.0320	2,631	.0160	2,951	-	3.6560
1000	200	.0160	2,100	.0160	2,025	.0150	1,952	-	2.2660
2000	0	.9220	1,004,585	.5000	50,968	2.442	750,596	.0320	-
2000	200	.0470	6,731	.0470	5,517	.0310	5,913	-	-
2000	400	.0470	4,182	.0310	4,044	.0320	3,934	-	-
3000	0	1.313	2,240,240	.7190	72,862	3.469	1,691,633	.0620	-
3000	300	.0630	10,164	.0630	8,216	.0620	8,898	-	-
3000	600	.0470	6,269	.0620	6,074	.0470	5,883	-	-

We found that in examples where the nullities are not covered by rounding error (the corresponding diagonal and subdiagonal entries are here zeroed) then the *BoosP* pivoting strategy works most efficiently. The applications described in the next section produce tridiagonal matrices with zero eigenvalues that are seriously hidden due to rounding. In those cases the *BoosP* and *MaxP* pivoting strategies were *not able* to discover the correct number of nullities especially for large  $n$ , and only the results of the *FraneP* strategy will be reported.

**5.3. Solving Systems with Dense *psd* Matrices.** More important for many applications in statistics (e.g. estimation of specific general linear models with singular matrices, computation of covariance matrices in nonlinear parameter estimation, estimating the parameters of severely overspecified problems like neural networks) is to show how our algorithm can be combined with other algorithms for the solution of linear systems  $\underline{x} = \mathbf{A}^\dagger \underline{b}$  with a dense  $n \times n$  positive semidefinite matrix  $\mathbf{A}$  with  $r(\mathbf{A}) < n$ . We want to compare the flop count and the real CPU times of our algorithm with three other algorithms currently used for MP inverses of singular systems.

**PSD3** : Tridiagonal L-D-L<sup>T</sup> Solver:

1. Tridiagonalize  $\mathbf{A}$  as  $\mathbf{A} = \mathbf{P}\mathbf{T}\mathbf{P}^T$  ([10] p. 419) using  $n - 1$  Householder transformations without explicitly computing  $\mathbf{P}$ . This uses  $2n^3/3$  flops.
2. Apply the sequence of Householder maps  $\mathbf{P}^T$  to  $\underline{b}$ , giving  $\underline{c} = \mathbf{P}^T \underline{b}$ . This requires  $n^2$  flops for each right-hand column.
3. Use our algorithm PSD3 to obtain the tri-sparse L-D-L<sup>T</sup> decomposition and hence find the least-squares solution to  $\mathbf{T}\underline{y} = \underline{c}$ . This uses  $OC = (7n - 7, 6n - 6, 3n - 2)$  for the

nonsingular case and  $OC = (11n - 20, 10n - 2, n + 7.5)$  for the case  $s = 1$  and  $t = n - 1$ , and for the general case see equation (4.6).

4. Apply the Householder transformations  $\mathbf{P}$  to  $\underline{y}$  to give the least-squares solution  $\underline{x} = \mathbf{P}\underline{y}$ . This uses  $n^2$  flops for each right-hand column.

The influence of rank deficiencies on the flop count depends on the size of the matrix split offs and is not easily predicted. When comparing later the cubic parts of the flop counts for large  $n$ , only the first step (tridiagonalizing  $\mathbf{A}$ ) counts.

**COS** : Complete Orthogonal Solver: The algorithm is based on [10], p.220 and 236, and implemented as in [11]:

1. Compute the *complete* orthogonal decomposition  $\mathbf{A} = \mathbf{Q} \begin{bmatrix} \mathbf{L}^T & \mathbf{0} \\ \mathbf{0} & \mathbf{0} \end{bmatrix} \mathbf{P}^T = \mathbf{Y}[\mathbf{L}^T \quad \mathbf{0}]\mathbf{P}^T$  with nonsingular upper triangular  $\mathbf{L}^T \in \mathcal{R}^{r \times r}$  and orthogonal  $\mathbf{Q} = [\mathbf{Y} \quad \mathbf{Z}]$  and  $\mathbf{P}$ . The orthogonal matrices  $\mathbf{Q}$  and  $\mathbf{P}$  are not explicitly computed. This takes  $2n^2r - nr^2 - r^3/3$  flops (see [11], appendix 1).
2. After or while performing the first step of the complete orthogonal decomposition we apply the  $r$  Householder transformations in  $\mathbf{Q}$  to the right-hand column  $\underline{b}$  forming  $\underline{c} = \mathbf{Q}^T \underline{b}$ . This takes  $rn$  flops.
3. The least-squares solution  $\underline{x} = \mathbf{P} \begin{bmatrix} \mathbf{L}^{-T} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} \end{bmatrix} \mathbf{Q}^T \underline{c}$  is completed by solving  $(\mathbf{L}^T)^{-1} \underline{c}$  via backward substitution and applying the  $r$  Householder transformations in  $\mathbf{P}$  which takes together  $r^2/2 + r(n - r)$  flops.

**SVS** : Singular Value Solver: Our implementation is a C version of the LINPACK subroutine DSVDC ([6] chapter 11). The svd for symmetric  $\mathbf{A}$  is  $\mathbf{A} = \mathbf{U}\mathbf{D}\mathbf{V}^T = \mathbf{V}\mathbf{D}\mathbf{V}^T$  and we only need to compute  $\mathbf{V}$  and  $\mathbf{D}$ :

1. Compute  $\mathbf{V}$  and  $\mathbf{D}$ , requiring  $6n^3$  flops ([10] p.248).
2. Compute  $\underline{c} = \mathbf{V}^T \underline{b}$ . This takes  $nr$  flops for each right-hand column.
3. Compute  $\underline{x} = \mathbf{V}\mathbf{D}^\dagger \underline{c}$ , using  $nr$  flops for each right-hand column.

**EVS** : Eigen Value Solver: Our implementation is a C version of the EISPACK subroutines TRED2 and IMTQL2 ([8] pp.308 and pp.291) which is basically unchanged in LAPACK except for blockwise computations which we not consider here.

1. Apply the symmetric QR algorithm to yield  $\mathbf{A} = \mathbf{Q}\mathbf{D}\mathbf{Q}^T$ , where  $\mathbf{Q}$  is orthogonal and  $\mathbf{D}$  is diagonal. Since  $\mathbf{Q}$  must be accumulated, this uses about  $5n^3$  flops ([10] p.424).
2. Compute  $\underline{c} = \mathbf{Q}^T \underline{b}$ , taking  $nr$  flops for each right-hand column using only the eigenvectors corresponding to nonzero eigenvalues.
3. Compute  $\underline{x} = \mathbf{Q}\mathbf{D}^\dagger \underline{c}$ , requiring  $nr$  flops for each right-hand column.

We construct randomly generated dense psd matrices  $\mathbf{A}$  with  $r(\mathbf{A}) = n, .9n, .8n$  using the following approach:

1. The  $n$  elements of an  $n \times n$  diagonal matrix  $\bar{\mathbf{D}}$  are randomly generated (uniformly distributed in  $[0,10]$ ). An orthogonal  $n \times n$  matrix  $\mathbf{V}$  is found by random generation (uniformly distributed in  $[0,1]$ ) and columnwise orthogonalization by Gram-Schmidt.
2. After sorting the diagonal entries of  $\bar{\mathbf{D}}$  in decreasing order,  $d = n - r$  entries in equally distant locations are set to zero defining the diagonal matrix  $\mathbf{D}$ .
3. Then matrix  $\mathbf{A}$  is computed by  $\mathbf{A} = \mathbf{V}^T \mathbf{D} \mathbf{V}$ .

Table 3 shows the CPU time in seconds on a Pentium Pro 200 PC when only one right hand side  $b$  was used. In the singular cases the hidden eigenvalues force our method to solve linear least-squares problems with submatrices  $\mathbf{A}_\sigma$ . Therefore, in general, no time savings are encountered when the tridiagonal matrix splits into smaller blocks. For some of the large examples we were able to detect the correct nullity only with the FraneP, but not with the MaxP or BoosP pivoting strategies.

n	$d(\mathbf{A})$	PSD3	COS	SVS	EVS
100	0	.0780	.0940	.5470	.6250
100	10	.0940	.0940	.4840	.5000
100	20	.0930	.1090	.4370	.4530
300	0	2.8750	3.4220	25.078	10.938
300	30	3.0000	3.5310	22.968	10.047
300	60	2.9060	3.6880	20.640	9.1570
500	0	13.812	17.859	120.859	54.312
500	50	8.9370	18.140	111.093	50.078
500	100	13.969	19.125	102.000	45.891
800	0	37.860	84.781	483.250	235.812
800	80	38.797	89.328	446.125	219.609
800	160	38.578	91.906	202.797	410.718
1000	0	74.953	143.797	921.797	441.141
1000	100	76.844	147.187	826.360	404.657
1000	200	76.094	158.234	764.031	372.984

Table 4 shows the CPU time when we solve for  $n$  right hand sides  $\mathbf{B} = \mathbf{I}_n$  which enables us to obtain  $\mathbf{X} = \mathbf{A}^\dagger$  and to check the size of the four MP residuals(2.2). In the singular cases our method is (due to the hidden eigenvalues) confronted with the solution of least-squares problems with submatrices  $\mathbf{T}_\sigma$  this time with a large number of right-hand sides. Therefore, for a modest nullity, the complete orthogonal decomposition and the eigenvalue decomposition have a slight advantage. For larger nullity the matrices  $\mathbf{T}_\sigma$  become smaller and our method again becomes faster. For a large number of right-hand columns and moderate nullity, the complete orthogonal decomposition becomes slightly faster than our method, since during the solution phase our method needs to apply the  $n - 1$  symmetric Householder transformations twice whereas the complete orthogonal decomposition only needs to apply  $n - d(\mathbf{A})$  Householder transformations from the first step and  $d(\mathbf{A})$  transformations from the second step.

n	$d(\mathbf{A})$	PSD3	COS	SVS	EVS
100	0	.3750	.2650	1.0470	.4840
100	10	.5000	.3600	.9220	.4530
100	20	.4530	.3440	.7970	.4220
300	0	9.9370	8.9370	46.797	17.954
300	30	17.000	15.281	41.687	15.281
300	60	14.219	13.953	38.485	14.219
500	0	45.907	45.500	228.328	85.437
500	50	82.781	77.468	201.063	77.156
500	100	70.516	72.313	179.687	68.313
800	0	204.687	242.922	1022.45	293.234
800	80	304.500	406.094	930.657	334.875
800	160	267.578	430.859	873.375	252.157
1000	0	399.688	408.078	1840.91	547.985
1000	100	593.360	591.250	1666.72	505.969
1000	200	529.453	605.047	1627.74	592.219

**5.4. Conclusions.** We have given a fast stable algorithm for computing least squares solutions for large singular p.s.d. systems. In the process we have demonstrated that (i) the conservative strategy of "relative pivoting" is more reliable than the "greedy" pivoting strategies, in that one keeps control of the rank of the matrix, and that (ii) using row operations is a viable alternative to the SVD approach to least squares. We have run exhaustive tests up to  $3000 \times 3000$  matrices and it is only the breakdown of the comparison algorithms that limited our tabular data.

**Acknowledgments.** The authors wish to thank the referees for their valuable suggestions which helped to improve the presentation of this algorithm.

#### REFERENCES

- [1] J.O. Aasen (1971), "On the reduction of a symmetric matrix to tridiagonal form", *BIT*, **11**, 233-242.
- [2] E. Anderson, Z. Bai, C. Bischof, J. Demmel, J. Dongarra, J. DuCroz, A. Greenbaum, S. Hammarling, A. McKenney, S. Ostrouchov, D. Sorensen (1991), *LAPACK User's Guide*, Philadelphia: SIAM Publications.
- [3] A. Ben-Israel and T.N. Greville (1974), *Generalized Inverses: Theory and Applications*, New York: John Wiley & Sons, Inc., p. 290-297.
- [4] J.R. Bunch and L. Kaufman (1977), "Some Stable Methods for Calculating Inertia and Solving Symmetric Linear Systems", *Mathematics of Computation*, **31**, 163-179.
- [5] S.L. Campbell and C.D. Meyer (1991), *Generalized Inverses of Linear Transformations*, New York: Dover.
- [6] J.J. Dongarra, J.R. Bunch, C.B. Moler, and G.W. Stewart (1979), *LINPACK User's Guide*, Philadelphia: SIAM Publications.
- [7] J.W. Frane (1977), "A Note on Checking Tolerance in Matrix Inversion", *Technometrics*, **19**, no. 4, pp. 513-514.
- [8] B.S. Garbow, J.M. Boyle, J.J. Dongarra, and C.B. Moler (1977), *Matrix Eigensystem Routines - EISPACK Guide Extension*, Berlin: Springer Verlag.
- [9] A. George, J.R. Gilbert, and J.W.H. Liu (1993), *Graph Theory and Sparse Computations*, New York: Springer.
- [10] G.H. Golub and C.F. Van Loan (1989), *Matrix Computations*, 2nd Ed., Baltimore: J. Hopkins University Press.
- [11] W.M. Hartmann and R.E. Hartwig (1996), "Computing the Moore-Penrose Inverse for the Covariance Matrix in Constrained Nonlinear Estimation", *SIAM Journal on Optimization*, **6**, p. 727-747.
- [12] R.E. Hartwig (1976), "Block Generalized Inverses", *Arch. Ratl. Mech. Anal.*, **61**, p. 197-251.
- [13] M.T. Jones and M.L. Patrick (1993), "Bunch-Kaufman Factorization for Real Symmetric Indefinite Banded Matrices", *SIAM Journal Matrix Anal. Appl.*, **14**, 553-559.
- [14] B.N. Parlett (1980), *The Symmetric Eigenvalue Problem*, Englewood Cliffs, NJ.: Prentice Hall.
- [15] E.L. Stiefel (1963), *An Introduction to Numerical Mathematics*, New York: Academic Press.
- [16] J.H. Wilkinson (1963), *Rounding Errors in Algebraic Processes*, Englewood Cliffs, NJ.: Prentice Hall, p.109.