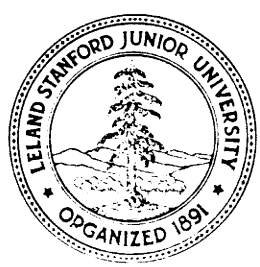
PRODUCT FORM OF THE CHOLESKY FACTORIZATION FOR LARGE-SCALE LINEAR PROGRAMMING

BY

MICHAEL A. SAUNDERS

STAN-CS-72-301
AUGUST 1972

COMPUTER SCIENCE DEPARTMENT
School of Humanities and Sciences
STANFORD UNIVERSITY



PRODUCT FORM OF THE CHOLESKY FACTORIZATION FOR LARGE-SCALE LINEAR PROGRAMMING

Michael A. Saunders
Computer Science Department
Stanford University
Stanford, California 94305

Abstract

A variation of Gill and Murray's version of the revised simplex algorithm is proposed, using the Cholesky factorization $BB^T = LDL^T$ where B is the usual basis, D is diagonal and L is unit lower triangular. It is shown that during change of basis L may be updated in product form. As with standard methods using the product form of inverse, this allows use of' sequential storage devices for accumulating updates to L . In addition the favorable numerical properties of Gill and Murray's algorithm are retained.

Close attention is given to efficient out-of-core implementation. In the case of large-scale block-angular problems, the updates to L will remain very sparse for all iterations.

This research was supported by the U. S. Atomic Energy Commission, Project SU326 P30-21. Reproduction-in whole or in part is permitted for any purpose of the United States Government.

Contents

1. Introduction

This paper is concerned with numerical solution of the standard linear programming problem

$$\begin{aligned} &\text{minimize} && c^T x \\ &\text{subject to} && Ax = b, \quad x \geq 0 \end{aligned}$$

where A is $m \times n$ and is usually very sparse. Following the work of Gill and Murray [6], an algorithm has been described in [14] which uses the orthogonal factorization $B = LQ$ to perform the steps of the revised simplex method [3]. Here B is the usual $m \times m$ basis, L is lower triangular and Q satisfies $QQ^T = Q^TQ = I$. Along with the methods of Bartels and Golub [1], [2] (which are based on the factorization $B = LU$ where U is upper triangular), the algorithms in [6], [14] constitute the only numerically stable versions of the simplex method that have yet been proposed.

An important feature of' Gill and Murray's approach is that the orthogonal matrix Q need not be stored. This follows from the identity

$$BB^T = LQQ^TL^T = LL^T$$

which chows that L is the Cholesky factor of BB^T, although we stress that the product BB^T is never computed. We shall often call L the Cholesky factor associated with B.

With Q discarded, our principal concern is with maintaining sparsity in L. In the Explicit Cholesky algorithm of [14], the emphasis was on the implications of retaining L in explicit form at all stages. By using a linked-list data structure to store only the non-zero elements of L,

it was shown that for in-core systems explicit updating of L can be practical in certain applications. When the matrix A is very sparse, and particularly if A has block-angular or staircase structure, the Cholesky factors remain sparse for all iterations. However, out-of-core implementation of the explicit algorithm is more difficult than with standard large-scale methods, since it is necessary to insert elements into the columns of L during change of basis and this cannot be done efficiently unless L is wholly contained in main memory.

In this paper, we show that L can be updated in product form. This means that L is not modified directly during basis changes; instead, certain transformations are accumulated in compact form in an update file, in the same way that eta-vectors are accumulated as updates to the standard product form of inverse (e.g. see Orchard-Hays [12]). During iterations, access to the update file is strictly sequential and therefore the file may extend conveniently onto disk or magnetic tape. It is this sequential mode of operation that enables product-form systems to deal with problems of very large size.

In sections 2, 3, and 4 we describe the Product-form Cholesky algorithm in mathematical terms. With regard to implementation it is similar to standard product-form algorithms, except that access is required to the current basis every iteration, which implies that B should be stored in a special sequential file of its own. Implementation aspects of this kind are discussed in sections 5,6, and 7. Methods for maintaining sparsity during reinversion (i.e. computation af an initial L) are considered in section 8. Finally, in section 9 we consider the question of sparsity within the transformations that modify L during changes of basis.

3. Modification of L during changes of basis

In order to minimize the number of square roots and divisions per iteration, we choose to work with the factorization

$$BB^T = LDL^T$$

where D is a diagonal matrix ($D=\mathrm{diag}(d_i), d_i>0$) and L is now unit triangular. An initial orthogonal factorization is computed explicitly during reinversion. We write this as

$$QB^T = R$$

where Q is a product of elementary orthogonal transformations and R is upper triangular. The matrices L and D are obtained by scaling the rows of R:

$$L^T = \mathrm{diag}(r_{ii}^{-1})R \ , \qquad D = \mathrm{diag}(r_{ii}^2) \ .$$

It can be readily shown that if column a_s replaces column a_r in B , then the new basis B^* satisfies

$$B^*B^{*T} = BB^T + a_s a_s^T - a_r a_r^T \ .$$

Such a change of basis will be accomplished in two steps, in each of which the current L and D are modified to produce $\bar{L}$ and $\bar{D}$ satisfying

$$\bar{L}\bar{D}\bar{L}^T = LDL^T + \alpha vv^T \ , \qquad (1)$$

where we take

1. $\alpha = +1$, $v = a_s$ to add column a_s ,
2. $\alpha = -1$, $v = a_r$ to delete column ar .

With these applications in mind we now consider the updating in (1) for a given vector v and any positive or negative α, assuming that the modified factorization exists.

Method C1

Let p, M and Δ be defined by .

$$Lp = v , \qquad M\Delta M^T = D + \alpha pp^T . \tag{2}$$

Thus p is obtained by forward substitution, and M and Δ are the Cholesky factors of a particularly simple matrix. From (1) we see that

$$\bar{L}\bar{D}\bar{L}^T = L(D + \alpha pp^T)L^T = L(M\Delta M^T)L^T = (LM)\Delta(LM)^T$$

and hence the modified factors are

$$\bar{L} = LM , \qquad \bar{D} = \Delta . \tag{3}$$

It can easily be verified that M in (2) is a special lower triangular matrix defined by two vectors p,β as follows:

$$M = \begin{bmatrix} 1 & & & & \\ p_2\beta_1 & 1 & & & \\ p_3\beta_1 & p_3\beta_2 & & & \\ \cdot & & \cdot & & \\ \cdot & & & \cdot & \\ \cdot & & & & 1 \\ p_m\beta_1 & p_m\beta_2 & & & p_m\beta_{m-1} \end{bmatrix} \tag{4}$$

where $P = (P, p_2 \ldots p_m)^T$,

$\beta = (\beta_1 \beta_2 \ldots \beta_m)^T$,

$D = \text{diag}(d_i)$,

$A = \text{diag}(\delta_i)$,

and the quantities β_i , δ_i are generated according to the following algorithm:

1. Set $\alpha_1 = \alpha$.
2. For i = 1, 2, ... m compute

$$\begin{aligned} &(a)\ \delta_i = d_i + \alpha_i p_i^2 \\ &(b)\ \beta_i = \alpha_i p_i / \delta_i \\ &(c)\ \alpha_{i+1} = \alpha_i d_i / \delta_i \ . \end{aligned} \tag{5}$$

This algorithm was derived independently by Gill and Murray [7]. Further details are given in [8].

Method C2

An alternative method for constructing M and A has been given by Gill and Murray [7], using elementary Hermitian matrices. This method has certain numerical advantages when $\alpha < 0$ and $LDL^T + \alpha vv^T$ is nearly singular. It may be summarized in slightly revised form as follows:

1. Set $\alpha_1 = \alpha$,

$s_1 = p^T D^{-1} p$,

$\sigma_1 = \alpha / [\ 1 + \sqrt{1 + \alpha s_1}\ ]$.

2. For i = 1, 2, . .., m compute

(a) $q_i = p_i^2/d_i$

(b) $\theta_i = 1 + \sigma_i q_i$

(c) $s_{i+1} = s_i - q_i$

(d) $\gamma_i^2 = \theta_{i-1}^2 + \sigma_i^2 q_i s_{i+1}$

(e) $\delta_i = \gamma_i^2 d_i$

(f) $\beta_i = \alpha_i p_i/\delta_i$

(g) $\alpha_{i+1} = \alpha_i/\gamma_i^2$

(h) $\sigma_{i+1} = \sigma_i(1 + \gamma_i)/[\gamma_i(\theta_i + \gamma_i)]$.

Again, further discussion is given in [8].

Both of these algorithms take α, p_i and d_i as input, and generate the appropriate β_i and δ_i which define M and A . When L is dense we normally use the special structure of M to compute the product LM explicitly in $m^2 + O(m)$ operations. In the present application we simply wish to record the vectors p,β in packed form and write them out to an update file for later re-generation of M . We will call the pair (p,β) an update, and a sequence of updates represents the product form of L . There are two updates to be stored each simplex iteration.

Although each update contains two distinct vectors (namely p and β), observe that $\beta_i = 0$ whenever $p_i = 0$, so the system overhead per update is essentially the same as for packing just one sparse vector. With regard to the rate of growth of elements in the update file, our principal claim for efficiency lies in knowing that with block-angular problems p is guaranteed to be very sparse for all iterations (see section 9). This will probably also be true for general sparse problems of sufficiently low density.

3. FTRAN and BTRAN

Let L_0 and D_0 be the Cholesky factors obtained from reinversion of a particular basis. The extension of equations (2), (3) to a sequence of updates should be clear. After k -iterations we will have

$$L_k = L_0 M_1 M_2 \cdots M_{2k-1} M_{2k} , \qquad (6)$$

$$D_k = A_{2k} ,$$

where each M_j is of the form shown in (4), and D_k is available explicitly.

Suppose at the next iteration that column a_s replaces column a_r in B. First we must find p satisfying

$$L_k p = a_s \qquad (7)$$

and then the corresponding β must be computed, for compact representation of M_{2k+1}. The arithmetic implied by equations (5) is best illustrated by the following pseudo-Algol program (Method C1 of section 2):

Algorithm 1. Computation of β from p, D

```
alpha:= 1;
for i:= 1 until m do
if p(i)≠0 then
begin
   Dsave:= D(i);
   temp  := alpha*p(i);
   D(i)  := Dsave + temp*p(i);
   beta(i):= temp/D(i);
```

```
        alpha:= alpha*Dsave/D(i);
    end;
```

A similar algorithm may be given for Method C2. In practice the test "if' p(i)≠0" would be replaced by "if abs(p(i))>eps", where eps is some suitable tolerance. Also the elements of β would not be stored explicitly in an m-dimensional array but would be packed along with the non-zero elements of p for immediate transfer to the update file.

Once it is determined that column a_r should be dropped from the basis, we must find a new p satisfying

$$(L_k M_{2k+1})p = a_r \ . \tag{8}$$

Given this p we compute β for M_{2k+2} by essentially the same method as in Algorithm 1. The first statement should be replaced by alpha:= -1 , and a test should be included to give an error exit if any of the new D(i) elements are negative, or smaller than some specified tolerance. (The new elements of D could never be negative if Method C2 were used.)

From equations (6),(7) and (8) it is clear that we must be able to solve systems of the form

$$M_j y = z \tag{9}$$

for as many M_j as are currently stored in the update file. Fortunately the structure of each M_j is so special (see equation (4)) that the forward substitution in (9) can be done very efficiently. This time (p,β) will already be in packed form, but for clarity we again assume they are stored in m-dimensional arrays:

Algorithm 2. Solution of My = z for FTRAN

```
S:= 0;
for i:= 1 until m do
if p(i)≠0 then
begin
   y(i):= y(i) - S*p(i);
   S:= y(i)*beta(i) + S;
end;
```

Here we assume that y and z occupy the same storage locations, as will be the case in any implementation. Observe that the elements of (p,β) are accessed sequentially in a "forward" direction (for i = 1, 2, ... m) and that computation of p from (7) requires M_j before M_{j+1}. Thus repeated use of Algorithm 2 for each M_j, $j = 1, 2, \ldots . 2k$, corresponds to the FTRAN operation of standard linear programming systems using the product form of inverse (e.g. see Orchard-Hays [12]).

Similarly, an operation corresponding to BTRAN is used for computation of the simplex multipliers π from a system of the form

$$L_k^T \pi = M_{2k}^T M_{2k-1}^T \cdots M_2^T M_1^T L_0^T \pi = \gamma_k \qquad (10)$$

for an appropriate right-hand-side vector γ_k. Here we need to solve systems $M_j^T y = z$ and again the special structure of each M_j leads to a very simple loop:

Algorithm 3. Solution of $M^T y = z$ for BTRAN

```
S:= 0;
for i:=m step -1 until 1 do
```

```
if p(i)≠0 then
begin
   y(i):= y(i) - &beta(i);
   S:= y(i)×p(i) + S;
end;
```

As before we assume y and z share the same storage. Comparison with Algorithm 2 shows that the roles of p and β are interchanged, while their elements are accessed sequentially in reverse order. This is completely convenient for buffered input/output, as we explain in section 5.

4. Computation of π

During reinversion the current basic cost vector $\hat{c}$ is regarded as the last row of the basis and is subjected to the same orthogonal transformation as B:

$$Q\,[B^T \mid \hat{c}] = [R \mid Q\hat{c}] .$$

Factoring out the diagonal of R gives L^T and a vector Y, say:

$$[R \mid Q\hat{c}] = \mathrm{diag}(r_{ii})\,[L^T \mid \gamma] ,$$

whereupon the system $B^T\pi = \hat{c}$ is equivalent to $L^T\pi = \gamma$, so that π can be computed by one back-substitution (i.e. one BTRAN operation). The general form of this system after k iterations was given in equation (10). We must store γ explicitly and transform it appropriately each change of basis. Suppose that column a_k is being added or dropped and the corresponding update (p,β) has been calculated. If the cost element c_k is stored in c(k) and if γ is contained in an array gamma(*), the following pseudo-Algol program illustrates what arithmetic is involved in updating γ :

Algorithm 4. Updating γ for solution of $L^T\pi = \gamma$

```
S:= c(k);
for i:=1 until m do
if p(i)≠0 then
begin
   s:= s - gamma(i)*p(i);
   gamma(i):= S*beta(i) + gamma(i);
end;
```

In practice this operation would not be performed separately but would be merged with computation of β . The two statements inside the above loop should be included as the last two statements of the loop in Algorithm 1.

Notice that all non-zero elements of p and β are required for modifying γ , whereas close inspection of Algorithms 2 and 3 shows that the first non-zero element of p and the last non-zero element of β (say p_f , β_ℓ respectively) are not required by FTRAN or BTRAN. Once γ has been modified, p_f and β_ℓ can be discarded. The corresponding elements β_f and p_ℓ must be written to the update file, but the unused space for p_f and β_ℓ could provide convenient storage for some of the flag and pointer information associated with packed vectors.

5. Buffered Input/Output for A, B and L

In an out-of-core linear programming system, part of' main memory must be allocated to a number of buffer regions to accommodate input/output (I/O) operations. Typically two regions are used for double-buffering the A-matrix into core during PRICE (when a column is selected for entry into the basis), while perhaps three-are devoted to the so-called eta-file, for use during FTRAN and BTRAN and for accumulation of updates to B^{-1}.

The particular algorithm proposed here differs from standard simplex algorithms in requiring access to the basis every iteration. Therefore certain differences arise in the organization of both main memory and auxiliary storage. The scheme we shall use is as follows:

1. Three sequential data sets reside on drum, disk or tape:
   (a) the A-file (fixed in size) containing A packed column-wise as usual.
   (b) the B-file (extendable) containing an initial basis and a sequence of columns that have recently entered the basis. (This is not required with standard methods.)
   (c) the L-file (extendable) containing an initial Cholesky factor L packed column-wise, followed by a sequence of updates to L.
2. (a) Three buffer regions are shared by the A- and B-files.
   . (b) Three further buffers are allocated to the L-file.

The A- and B-files may share the same I/O channel, but preferably should be on separate storage devices. The L-file should be accessed through a second I/O channel. To minimize the number of I/O operations each buffer region should be as large as possible, namely one sixth of whatever memory is available after allocation of various m-dimensional arrays to $\overset{A}{x}$, π, etc.

Use of three buffer regions for the L-file follows what a typical implementation of the eta-file might be in a system using the product form of inverse. We describe the mode of operation briefly. At any particular stage, two regions are used for double-buffering L into core during FTRAN and BTRAN, while the third is only partially filled and contains update vectors for the most recent iterations. (See Orchard-Hays [12, p. 113], Smith [15].) When this third buffer becomes filled it is written out to auxiliary storage as an extension of the L-file, and at this point the three L-buffers change roles in cyclic order.

With Algorithms 1, 2 and 3 of section 3 in mind we may ask what happens if an update (p,β) cannot fit into the unfilled portion of the third buffer above. It would be wasteful to write out the buffer half empty, and in any case even a whole buffer may not be large enough to contain all of a single update. Fortunately the sequential nature in which updates are used in FTRAN and BTRAN provides a simple answer. We can split $P = [p_1 \ / \ p_2]$, $\beta = [\beta_1 \ / \ \beta_2]$ at any convenient point and proceed to use (p_1,β_1), (p_2,β_2) as two distinct updates. It remains to associate with each update a flag which specifies the initial condition of variable S in Algorithms 2 and 3. (S is used to accumulate the inner-products $y^T\beta$ and $y^T p$ respectively.) Normally S will be initialized to zero, but if the flag is set then S retains its value from the previous update.

With regard to the basis file, observe that B is required for computation of a vector y (satisfying $By = a_s$) using the equation

$$y = B^{-T}u \ . \qquad (11)$$

Since this is just a matrix multiplication, rather than back-substitution say, the order in which columns of B are accessed is not important. In particular the order in which they occur in the B-file is quite acceptable. Hence it is clear that the three-buffer scheme previously described for the L-file is also an ideal design for the B-file. Two memory areas are used for double-buffering B into core for the computation in (11), while the third accumulates columns that have recently entered the basis. Accumulation will be very slow but the same rotation of buffers can accommodate overflow as before. Columns that are no longer in the basis need be purged from the B-file only occasionally (e.g. during reinversion), since only a small percentage of columns are changed between reinversions.

During PRICE the first two B-buffers provide access to the A-file in the standard way.

The single I/O-scheduling problem arises when a specific column must be retrieved from the B-file each iteration. (see step 8 in the next section.) Often the requisite column will already be in main memory after computation of $B^T u$, since basis changes frequently involve columns which entered during recent iterations. Of course it would be ideal if all of the basis could be contained in the three B-buffers, so in practice this situation may define a main memory size that is workable for a particular linear programming problem.

Alternatively, complete re-reading of the B-file for selection of a specific column would provide an excellent opportunity for performing one iteration of iterative refinement on the system $B\hat{x} = b$. This point is discussed in section 7 with reference to the main steps of the simplex algorithm.

6. Summary of algorithm

Suppose that k iterations have been performed following reinversion, that $B\hat{x} = b$ is the current basic solution, and that $L_k D_k L_k^T = BB^T$ is the current Cholesky factorization. The essential steps to be performed at iteration k+1 are:

1. BTRAN1 (Backward Transformation 1):

   Solve $L_k^T \pi = \gamma_k$ i.e. $M_{2k}^T \cdot \cdot \cdot \cdot M_2^T M_1^T L_0^T \pi = \gamma_k$.

2. PRICE: Read A-file to compute reduced costs for non-basic variables. Select column a_s for which $c_s - \pi^T a_s < 0$.

3. FTRAN1 (Forward Transformation 1):

   Solve $L_k p_1 = a_s$ i.e. $L_0 M_1 M_2 \cdot \cdot \cdot \cdot M_{2k} p_1 = a_s$.

4. UPDATE1: Compute $w = D_k^{-1} p_1$.

   Use p_1 to compute β_1 and modify D_k . Pack non-zero elements of (p_1, β_1) and add to end of L-file.

5. BTRAN2: Solve $L_k^T u = w$.

6. READB: Read B-file to compute $y = B^T u$. Add column a_s to B-file.

7. CHUZR: use $\hat{x}$, y and the usual ratio test to determine column a_r to be removed from B . Find $\theta = \hat{x}_r / y_r$.

8. SEEKR: Frequently, column a_r will already be in main memory. If not, position B-file at record containing a_r , while updating $\hat{x}$ according to $\hat{x} \leftarrow \hat{x} - \theta y$.

9. FTRAN2: Solve $L_k M_{2k+1} p_2 = a_r$.

10. UPDATE2: Compute β_2 , D_{k+1} from p_2 and add (p_2, β_2) to L-file.

It is interesting to note that FTRAN and BTRAN require no divisions at all, since L_0 and the M_j have unit diagonals.

7. Numerical considerations

Suppose that $B = LQ$ for some basis B. (For simplicity of notation we will temporarily use L in place of $LD^{\frac{1}{2}}$.) To find the current basic solution $\hat{x}$ satisfying $B\hat{x} = b$ we solve the equivalent system

$$BB^T u = b, \qquad \hat{x} = B^T u \tag{12}$$

by the following steps:

Method L: (a) $Lp = b$

(b) $L^T u = p$

(c) $\hat{x} = B^T u$.

An error analysis of this process has recently been given by Paige [1]. Let $\kappa(B)$ be the usual condition number of B and let $\tilde{x}$ be the computed approximation to $\hat{x}$. Paige's surprising result is that the relative error $\|\tilde{x} - \hat{x}\|_2 / \|\hat{x}\|_2$ depends essentially on $\kappa(B)$ and is not dominated by $\kappa^2(B)$ in spite of the occurrence of BB^T in (12). This is a very agreeable property indeed.

During some numerical tests to confirm this result we compared method L with the more obvious one which retains the orthogonal matrix Q:

Method Q: (a) $Lp = b$

(b) $\hat{x} = Q^T p$.

These tests involved Hilbert matrices of various order and showed that method Q is likely to give smaller relative error $\|\tilde{x} - \hat{x}\|_2 / \|\hat{x}\|_2$ than method L, and may even give small relative error $|\tilde{x}_i - \hat{x}_i| / |\hat{x}_i|$ in all components of 4 (which is more than could be asked of any method). Nevertheless the relative error achieved by method L was as small as could be expected from the magnitude of $u(B)$.

A further interesting effect was observed in connection with the process of iterative refinement (see Wilkinson [19, pp. 255-265]). This process involves correcting $\tilde{x}$ by the following steps:

(a) Compute the residual vector $r = b - B\tilde{x}$.

(b) Solve the system $B\delta x = r$, using the same factorization of B that was used for computing $\tilde{x}$.

(c) Take the new approximate solution to be $\tilde{x} + \delta x$.

These steps can be repeated. Wilkinson [19, p. 261] notes the possibility that single precision may be sufficient for computing the first residual vector r , if the method used for computing $\tilde{x}$ is somewhat less than ideal. (For further iterations double precision is essential.) In our tests r was not computed in double precision, and with method Q no improvement to the initial $\tilde{x}$ was obtainable. However when method L was used to compute $\tilde{x}$ and a single correction δx , very significant improvement was obtained. Similar results have been observed by Gill and Murray in refining the vector y of the system $By = a_s$.

This pseudo-refinement has been incorporated in the program discussed in [14] and tested on a rather ill-conditioned staircase problem of dimension: 357 x 385 . An IBM 360/91 computer was used (relative machine precision $16^{-13} = 2.2 \times 10^{-16}$) and $\tilde{x}$ was corrected after reinversion and also every 25 iterations between reinversions. Typically, $\max|r_i|$ was reduced from around 10^{-9} to 10^{-16} . In some cases when the basis was strongly ill-conditioned, $\max|r_i|$ was reduced from 10^{-5} to 10^{-14} by the single correction. (If $BB^T = LDL^T$, the condition number of B can be estimated using the lower bound $\kappa(B) > \{\max(d_i)/\min(d_i)\}^{\frac{1}{2}}$.)

In view of the above observations we suggest that a correction to $\hat{x}$ could be made at every iteration of the product-form Cholesky

algorithm. Here we are accepting the fact that an I/O hold-up may occur during the SEEKR operation (step 8 of section 6). Instead of waiting for the basis file to be positioned at a specific column, we could read the entire file and compute residuals for the current $\hat{x}$ at the same time.

Several steps of section 6 then need to be modified. Assuming that r will be computed separately following reinversion, the new steps are as follows:

3. FTRAN1: Solve $L_k p_1 = a_s$ and $L_k p_3 = r$ in parallel.
(Since the overhead of unpacking L_k consumes most of the time, this does not involve much more work than before.)

4. UPDATE1: Compute $w_1 = D_k^{-1} p_1$ and $w_2 = D_k^{-1} p_3$.
Use p_1 to compute β_1 and modify D_k. Pack non-zero elements of (p_1, β_1) and add to end of L-file.

5. BTRAN2: Solve $L_k^T u_1 = w_1$ and $L_k^T u_2 = w_2$ in parallel.

6. READB: Read B-file to compute $y = B^T u_1$ and correct $\hat{x}$ according to $\hat{x} \leftarrow \hat{x} + B^T u_2$.

7. CHUZR: Use $\hat{x}$, y and the usual ratio test to determine column a_r to be removed from B. Find $\theta = \hat{x}_r / y_r$. Update $\hat{x}$ according to $\hat{x} \leftarrow \hat{x} - \theta y$.

8. SEEKR: Read B-file to select column a_r while computing residuals $r = b - B\hat{x}$ for the new B.

Vectors r, p_3, w_2, u_2 may all share the same storage.

8. Sparsity considerations during reinversion

Let P_1, P_2 be row and column permutations to be applied to some basis B. The orthogonal factorization $P_1BP_2 = (LD^{\frac{1}{2}})Q$ is well known to be numerically stable for all choices of P_1 and P_2, so we are free to choose whatever permutations might lead to the most sparse L. However, on eliminating Q to obtain the associated Cholesky factorization we see immediately that L and D are independent of P_2:

$$(P_1BP_2)(P_2^TB^TP_1^T) = (LD^{\frac{1}{2}}Q)(Q^TD^{\frac{1}{2}}L^T)$$

i.e.
$$P_1BB^TP_1^T = LDL^T .$$

Hence our search for sparsity in L is reduced to finding an optimal ordering for the rows of B.

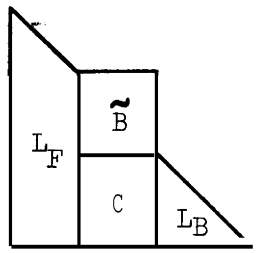
The fact that column-ordering is irrelevant was put to good use in [14] where L was being updated explicitly. We were led to the seemingly naïve strategy of selecting P_1 from an initial inspection of the full A-matrix, on the grounds that this might provide a permutation that would be reasonably close to optimal for all subsequent bases. By such means we hoped to avoid large fluctuations in the density of L during the simplex iterations. Although it is not clear which single row-ordering is best, the strategy is most likely to be successful if P_1 transforms A into block-angular form. (See [14].)

When L is updated in product form we favor the conventional approach of choosing a new P_1 each time L is re-computed directly from B (i.e. each reinversion). This is because the sparsity of such an initial L strongly affects the efficiency of subsequent iterations up till the next

reinversion. (The other important factor is the sparsity of updates, which we discuss in section 9.)

A first step towards obtaining a good row permutation is to partition B into the following familiar form:

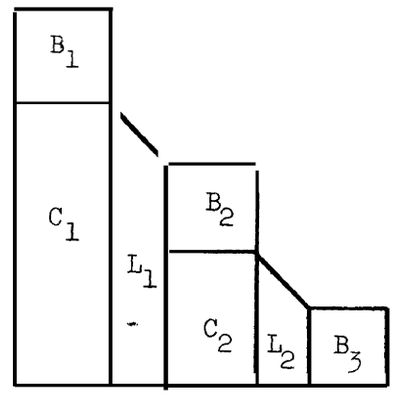
B =



where L_F , L_B are triangular columns with non-zeros on the diagonal. (These are the forward and backward triangles respectively, and finding them is a straightforward process; e.g. see Hellerman and Rarick [9].) Partition $\tilde{B}$ is called the "bump". It is square and in general sparse, and we have yet to compute its orthogonal factorization. This amounts to taking linear combinations of the columns of $\tilde{B}$ such that $\tilde{B}$ is reduced to lower triangular form. We see that fill-in will occur in partition C , but not in partitions L_F or L_B .

To minimize fill-in we need to permute the rows of $\tilde{B}$ in some optimal fashion. One promising possibility is to make use of the pre-assigned pivot procedures due to Hellerman and Rarick (called P^3 and P^4 in [9], [10]). These are algorithms for isolating further bumps within $\tilde{B}$. Thus after the main bump is located, the next stage of P^4 is to find a row and column ordering which arranges $\tilde{B}$ into the form

$\tilde{B}$ =



where B_1 , B_2 and B_3 are new square sub-bumps (there may be any number), and L_1 and L_2 are triangular columns with non-zeros on the diagonal. This strategy ensures that there will be no fill-in for at least some of the columns of $\tilde{B}$ and C , namely those columns corresponding to L_1 and L_2 . Each sub-bump remains to be triangularized, but whatever we do to bump B_1 , for example, will have no effect on B_2 or B_3 . Thus our final problem is to find optimal row orderings for all bumps, treating each independently.

At this point it is interesting to note the statistics given by Hellerman and Rarick in [9] for their algorithm P^3 . When applied to basis matrices of dimension ranging from 589 up to 977, the number of bumps that were isolated by P^3 ranged from 3 up to 22. This is encouraging for the following reasons. Suppose a basis B is reduced to lower triangular form by a sequence of elementary orthogonal transformations Q_{ij} $(i<j)$:

$$BQ_{i_1j_1}Q_{i_2j_2}\cdots Q_{i_kj_k} = L .$$

Each Q_{ij} represents a linear combination of columns i and j of the current B (transformed by all previous Q_{ij}), and the sparsity structure of both columns after the transformation is the logical OR of their sparsity structure before Q_{ij} is applied. (In contrast, if Q_{ij} were just an elementary elimination operation, column j would be affected in the same way but column i would not change at all.) This property of orthogonal transformations implies that fill-in is at least as much as with simple elimination, and it appears that a single rather dense column in B is potentially capable of propagating non-zeros throughout the whole of L . Fortunately Hellerman and Rarick's results indicate that a typical LP basis can be permuted in a way which reveals a number of "fire-breaks." Thus propagation of non-zeros may occur below each bump but certainly can not spread across the triangular columns between bumps.

Of course the same is true with the product form of inverse (hereafter called PFI). Our point is that with orthogonal factorization the effect of propagation can be very serious if allowed to continue over a large number of columns. We may gain some relief in the knowledge that propagation must stop at the end of each bump.

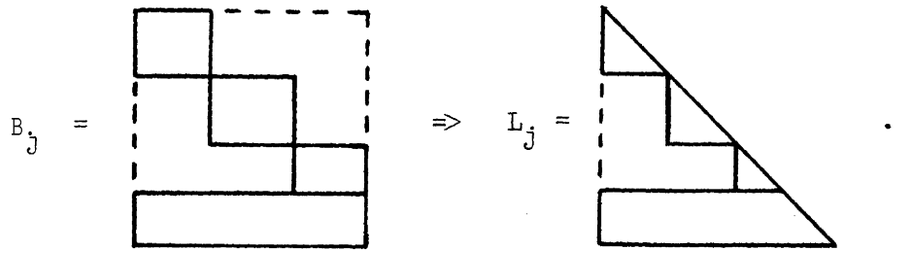
We return now to the problem of permuting the rows of each bump. The strategy of P^4 is to find a permutation of both rows and columns which reveals a spike structure of this form:

B_j =

.

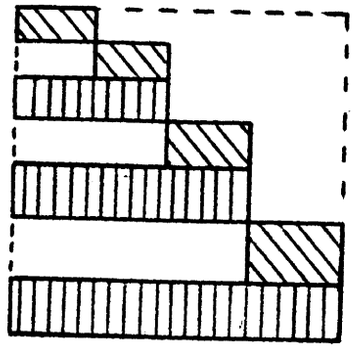
A similar spike-finding algorithm has been given by Kalan [11]. When the PFI of B_j is computed the only fill-in that occurs is below the spikes. The columns between the spikes of any bump are like the lower-triangular columns between the bumps of $\tilde{B}$, and since their sparsity is not altered, it appears that the spike-finding strategy is ideal for PFI.

On the other hand, orthogonal factorization of such a structure will be less successful in terms of fill-in, since if $B_j = L_j Q_j$ say, we know that a particularly dense spike is likely to propagate non-zeros throughout both L_j and the columns beneath B_j. As an alternative, consider the Cholesky factor associated with a bump that has block-angular form:



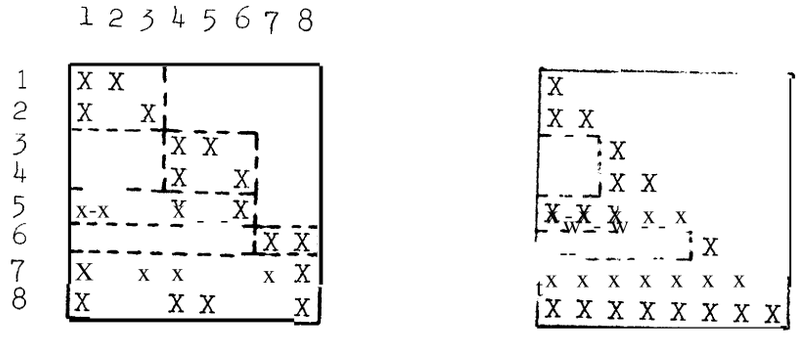
The preservation of zeros in L_j below the angular blocks leads us to propose looking for a more general structure which we shall call nested block-angularity. In its simplest form this amounts to isolating two angular blocks (plus a set of coupling rows) and then applying the same operation to each block recursively, until at all levels of recursion neither of the blocks is further decomposable. The structure thus obtained depends on whether or not both blocks are decomposable at each stage. If just the first block has further structure each time we will get the following nested pattern:

B_j =

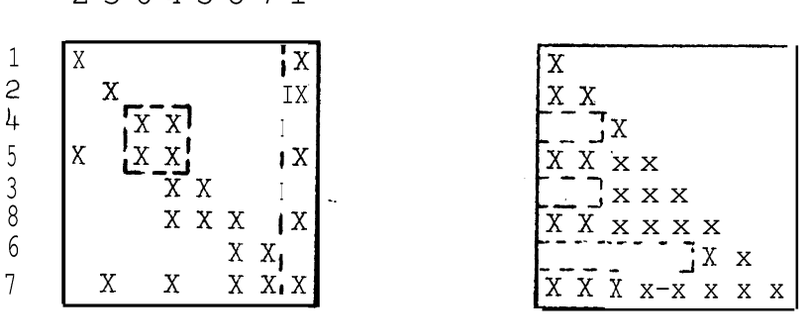


This is an extreme case and there are many variations. An algorithm for detecting block-angularity within a general rectangular matrix has been given by Weil and Kettler [18], and it can be applied directly to the problem of finding nested block-angularity. Cur motivation is that by so doing we can guarantee preservation of zeros inside the angular blocks as well as below them.

To illustrate that this strategy may sometimes be as good as looking for spikes, here is an 8x8 bump with nested block-angular structure, along with its associated Cholesky factor:



If the last stage of P^4 is applied to the same matrix we get the following spike structure and a slightly different Cholesky factor:



By coincidence both orderings give exactly the same number of zeros in the lower triangle. Without further experimentation we cannot draw any conclusions about the relative efficiency of each approach.

8.1 Numerical aspects of preassignment

Finally we must look at the numerical implications of preassigning pivots, Suppose that a basis B is to be "inverted" either by PFI or by orthogonal factorization (LQ for short). The strategy of isolating square bumps in B is certainly justified in both cases, since neither PFI nor LQ alters the triangular columns L_F, L_1, L_2,... L_B between and around the bumps. Any near-singularity in these columns implies near-singularity of B itself',

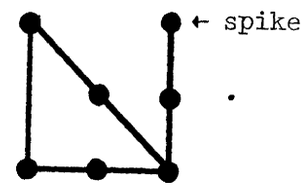
Similarly (since the bumps are square), near-singularity of any bump implies that the original B is almost singular, and no amount of reordering or merging of bumps can improve the condition of B . This further justifies our earlier statement that the bumps can be treated independently.

The only numerical difference between PFI and LQ (during reinversion) arises in the factorization of each bump. With PFI it is unlikely that any pivot order assigned to a bump will be completely

satisfactory if that order has been chosen without regard to the magnitude of non-zeros in the bump. It may be thought (e.g. [9, p. 214]) that the only possible complication would be with the spike columns, whenever the updated pivot element of a spike becomes too small. A strategy currently being used is to interchange an unsuitable spike column with some other spike column, on the grounds that at least one of the spikes will have an acceptable pivot element. This will often mean moving a spike and its pivot row out of one bump into the next. The aim is to retain as much of the preassigned pivot order as possible.

The following example, however, shows that it is unreasonable to assume that the non-spike columns will have acceptable pivots. Suppose a bump B_j is of the form

$$B_j = \begin{bmatrix} 10^{-3} & & 1 \\ & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix} =$$



If the preassigned pivot order is retained here, the first eta-transformation will be

$$E_1 B_j = \begin{bmatrix} 10^{3} & & \\ & 1 & \\ -10^{3} & & 1 \end{bmatrix} \begin{bmatrix} 10^{-3} & & 1 \\ & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix} = \begin{bmatrix} 1 & & 10^{3} \\ & 1 & 1 \\ & 1 & -999 \end{bmatrix}$$

and hence computation of PFI will introduce unnecessary numerical error into an otherwise well-conditioned matrix. Clearly an interchange should be made between the first and third rows, or between the first and third columns.

This problem would be overcome by treating small non-spike pivots in the same way as small spike pivots. In the extreme case of choosing maximal pivot:; (i.e. taking the relative pivot tolerance to be 1), we assert that PFI would be numerically stable during reinversion ii' provision were made to interchange either all of the rows or all of the columns within each bump, and under these conditions it would never be necessary to move columns from one bump into another. In practice it may be feasible to localize interchanges in this way even if the relative pivot tolerance is somewhat less than 1 .

In any event, numerical precautions must be taken when PFI is used, and some-revision of a preassigned pivot order will often be necessary. The anticipated reduction in basis matrix I/O may therefore not always be achieved.

With orthogonal factorization, as we have said before, all pivot orders are numerically acceptable, and in such a context the philosophy of preassigning pivots becomes fully justifiable. We are paying the price of higher density in the basis factorization, but by this means alone can the advantages of preassignment be fully realized.

9. Sparsity of updates

In any algorithm based upon product-form updating, a principal factor governing reinversion frequency is the rate of growth of elements in the update file. It would be pleasing if the energy expended during reinversion had some optimizing effect on the sparsity of subsequent updates. To some extent this proves to be the case with the Cholesky algorithm.

Suppose that column a_s replaces column a_r in B_k at some iteration k. In the case of standard PFI updating, the number of non-zeros in the updated column vector $\alpha = B_k^{-1} a_s$ determines how many elements are added to the eta-file. Clearly this number is uniquely determined by B_k and a_s, and would not be altered by any permutations to the rows and columns of B_k. Neglecting numerical error we conclude that with PFI the rate of growth of the eta-file is independent of whatever sparsity was achieved last reinversion, or how recently that reinversion was performed.

For the Cholesky algorithm the relevant update vectors are p_1 and p_2, as given in FTRAN1 and FTRAN2 of section 6:

$$L_k p_1 = a_s , \qquad L_k M_{2k+1} p_2 = a_r . \tag{13}$$

Now if P is the row permutation chosen during reinversion of an initial basis B_0, then L_k is the Cholesky factor associated with PB_k ($k \geq 0$). A change in P would affect all L_k and therefore would alter p_1 and p_2 above. In other words, the choice of P during reinversion affects the sparsity of updates for all subsequent iterations. We will discuss this situation in general terms first, and then specialize to block-angular problems.

9.1 General sparse problems

Given a large sparse linear programming problem, we should bear in mind the following points:

(a) In a triangular system of the form $Lp = v$ such as in (13) above, the first non-zero element of p coincides with the first non-zero in the right-hand-side vector v (counting from the top down).

(b) A reinversion algorithm such as Hellerman and Rarick's (section 8) is usually capable of permuting a basis B_0 into almost lower triangular form. By this we mean that only a small fraction of columns (viz. the spikes) have non-zeros above the diagonal.

(c) The number of iterations performed between reinversions is usually small relative to m (e.g. reinversion every 50 iterations when m=1000 gives a ratio of 5%). Hence after k iterations, B_k differs from B_0 in only a small percentage of its columns.

(d) Since the Cholesky factor associated with B_k does not depend upon column order, the row permutation P chosen as optimal for B_0 should remain close to optimal for all B_k.

To formalize point (a), f'or any m-vector v, define an integer function $\theta(v)$ as follows:

$$\theta(v) = k \quad \text{iff} \quad \begin{cases} v_1 = 0 & \text{for all } i<k, \\ v_k \neq 0. \end{cases}$$

Then $Lp = v$ implies that $\theta(p) = \theta(v)$, so the maximum possible number of non-zeros in p is $m - \theta(v)$. Now points (b) and (c) together show that all bases B_k are essentially triangular (for the purposes of this argument), so on the average it is likely that $\theta(a_r) \geq m/2$, where a_r

is the column being deleted from B_k . Hence in (13), $\theta(p_2) > m/2$ on average. In words this means that the vector p_2 is likely to be less than 5% dense, even if there is complete fill-in below the first non-zero. The same may be said about p_1 in (13), since it seems reasonable to assume that the incoming columns a_s will in the long run have non-zeros distributed much like the columns they are replacing.

A similar argument may be applied to the method of Forrest and Tomlin [5], [16], [17] for updating LU factors of the basis. If the "partially updated" form of the incoming column is $\gamma = L^{-1}a_s$, it is likely that $\theta(\gamma) \geq m/2$ on the average.

Even with PFI the same could be said about the vector $\alpha = B_k^{-1}a_s$ in the case of transportation problems, since then B_k is always a permuted triangle. With more general problems the size of the forward triangle of each basis would be the critical factor.

To summarize, at each iteration a strict upper bound on the number of non-zeros in the updates $(p_1,\beta_1), (p_2,\beta_2)$ is $4m$, and the above discussion has reduced the bound to a "likely average" of $2m$. This is not entirely satisfactory yet, since a strict upper bound for the α-vector in PFI is just m elements. However, just as we expect α to be sparse in most cases, we also expect that vectors p_1, p_2 will not be completely dense below positions $\theta(p_1)$, $\theta(p_2)$ respectively.

Except in the case of block-angular problems (see below), we must resort to further heuristics by comparing

$$p_1 = L^{-1}a_s$$

with

$$\alpha = B^{-1}a_s = B^T L^{-T} D^{-1} p_1 .$$

(Equivalently, $\alpha = Q^T D^{-\frac{1}{2}} p_1$ if Q is retained in the orthogonal factorization $B = LD^{\frac{1}{2}}Q$.) Since pl is only "partially updated" by comparison with α it seems that if α is at all sparse in PFI then p_1 is likely to be even more sparse. Point (d) above indicates that if pl is sparse during iterations immediately after reinversion, its density should increase only slowly during later iterations.

Under the final assumption that p_2 will be as sparse as Pl , we conclude with the following conjecture:

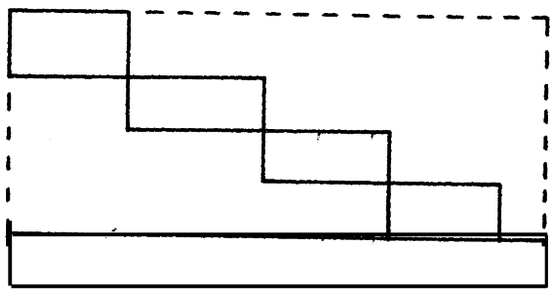
> For a given problem, if the a/-vectors occurring in PFI are of low density, the growth of elements in the L-file of the Cholesky algorithm will be comparable to the growth of elements in the eta-file of PFI .

As always, such a heuristically-derived conjecture must be verified by practical experimentation. In addition, comparison must at some stage be made with the method of Forrest and Tomlin [5], [16], [17], since relative to PFI this method has demonstrated considerably lower growth rate of the eta-file.

9.2 Block-angular problems

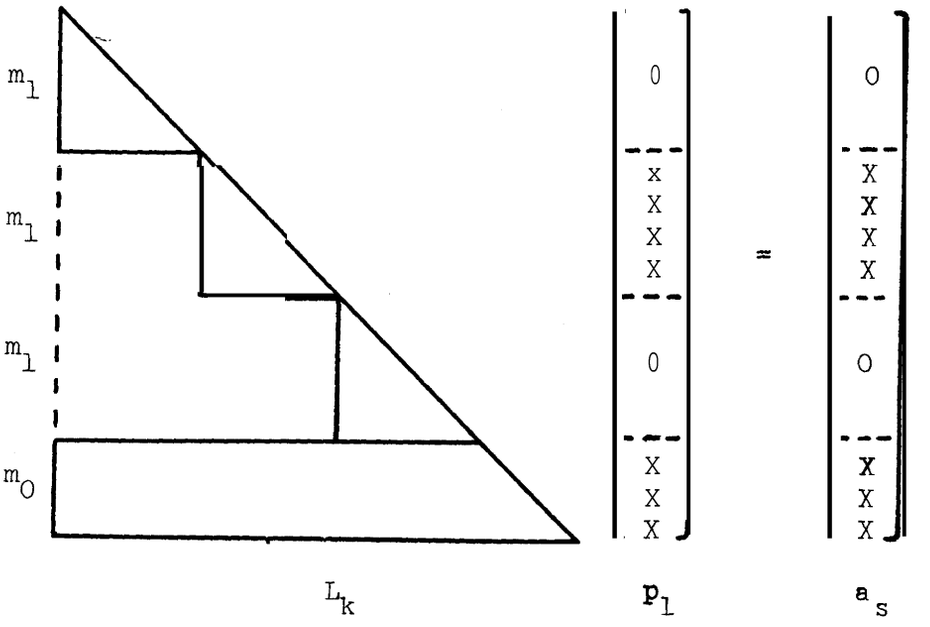
The one case where we can guarantee in advance that the update vectors will be sparse, is when the matrix A has block-angular structure (e.g. [4]):

A =



Just a trivial modification to the general algorithm is required to take advantage of this special structure. Specifically, the reinversion routine must choose row permutations for each block individually; i.e. it must not move rows from one block to another. Under such circumstances we know that the Cholesky factors L_k will be block-triangular for all iterations (Saunders [14]).

For simplicity, suppose there are m_0 coupling constraints and three blocks each of dimension m_1. Suppose also that a column a_s is entering from block 2. Then the system $L_k p_1 = a_s$ looks like this:



The zeros that are shown in p_1 follow directly from the zeros in column a_s. The same is true for the second update vector when a column is deleted from some block. The total number of elements added to the L-file each iteration can be at most $4(m_0 + m_1)$, and should be considerably fewer if the problem is sparse within the blocks. This upper bound is independent of the number of angular blocks.

10. Conclusion

Among the many implementations of Dantzig's original simplex method, those which have become established because of their sparseness properties are unfortunately numerically unstable. The foremost examples are algorithms based on the product form of inverse. A more recent example is the LU implementation of Forrest and Tomlin [5], [16], [17]; this has demonstrated extremely good sparseness characteristics on large practical problems, but the method used for updating the LU factors cannot be classed as numerically stable. Conversely, the LU algorithm of Bartels and Golub [1], [2] has excellent numerical properties, but because of the difficulty of inserting new non-zero elements into U during changes of basis (see Tomlin [17]), this method cannot yet be applied to large-scale problems.

At present, the algorithm described in this paper represents the only general linear programming method which is both numerically stable and capable of efficient out-of-core implementation. We have shown that product-form updating of the Cholesky factorization is feasible, and we have retained the numerical advantages inherent in Gill and Murray's version of the simplex method [6].

The aim in writing has essentially been two-fold:

1: To present a mathematical description of the product-form Cholesky algorithm, along with sufficient practical detail to indicate how implementation might proceed.

2. To discuss qualitatively some aspects of orthogonal factorization in the context of general sparse matrices, in order to gain some assurance that the expense of a non-trivial implementation might

be justified.

We anticipate that the class of problems for which the method will prove to be efficient will include those which are extremely sparse and those which have block-angular structure.

Acknowledgements

I wish to thank John Tomlin for many discussions of the implementation aspects of large-scale systems, including the buffering scheme described in section 5. I am also very grateful to Walter Murray and Chris Paige for discussions of the numerical aspects; to Phil Gill, Gene Golub and Richard Underwood for their valuable criticisms of the first draft; and to Mary Bodley for her speed and care in typing the manuscript.

References

[1] R. H. Bartels, "A stabilization of the simplex method," Numerische Mathematik 16 (1971), pp. 414-434.

[2] R. H. Bartels and G. H. Golub, 'The simplex method of linear programming using LU decomposition," Comm. ACM 12(1969) pp. 266-268, 275-278.

[3] G. B. Dantzig, Linear programming and extensions, Princeton University Press, Princeton, New Jersey (1963).

[4] G. B. Dantzig, "Large-scale linear programming," Operations Research Department Report No. 67-8, Stanford University, Stanford, California (1967).

[5] J. J. H. Forrest and J. A. Tomlin, "Updated triangular factors of the basis to maintain sparsity in the product form simplex method,' Mathematical Programming 2 (1972) pp. 263-278.

[6] P. E. Gill and W. Murray, "A numerically stable form of the simplex algorithm," J. of Linear Algebra and its Applications 6(1973).

[7] P. E. Gill and W. Murray, "Quasi-Newton methods for unconstrained optimization," J. Inst. Maths Applications 9(1972) pp.91-108.

[8] P. E. Gill, G. H. Golub, W. Murray and M. A. Saunders, "Methods for modifying matrix factorizations," Computer Science Department Report (to appear) , Stanford University, Stanford, California (1972).

[9] E. Hellerman and D. Rarick, "Reinversion with the preassigned pivot procedure," Mathematical Programming 1 (1971) pp.195-216.

[10] E. Hellerman and D. Rarick, "The partitioned preassigned pivot procedure (P^4)," pp. 67-76 in Sparse matrices and their application:: D. J. Rose and R. A. Willoughby (Editors), Plenum Press, New York' (1972).

[11] J. E. Kalan, "Aspects of large-scale in-core linear programming," proceedings of ACM Annual Conference, Chicago, Illinois (August 3-5, 1971).

[12] W. Orchard-Hays, Advanced linear-programming computing techniques, McGraw-Hill. New York (1968).

[13] C. C. Paige, "An error analysis of a method for solving matrix equations," Computer Science Department Report No. CS 297, Stanford University, Stanford, California (1772).

[14] M. A. Saunders, "Large-scale linear programming using the Cholesky factorization," Computer Science Department Report No. CS 252, Stanford University, Stanford, California (1972).

[15] D. M. Smith, "Data logistics for matrix inversion," in Sparse matrix proceedings, Ed. R. A. Willoughby, IBM Research Center, Yorktown Heights, New York (1968) pp. 127-132.

[16] J. A. Tomlin, "Maintaining a sparse inverse in the simplex method," IBM Journal of Research and Development 16 No. 4 (1972) pp. 415-423.

[17] J. A. Tomlin, "Modifying triangular factors of the basis in the simplex method," pp. 77-85 in Sparse matrices and their applications, D. J. Rose and R. A. Willoughby (Editors), Plenum Press, New York (1972).

[18] R. L. Weil and P. C. Kettler, "Rearranging matrices to block-angular form for decomposition (and other) algorithms," Management Science 18, No. 1 (1971) pp. 98408.

[19] J. H. Wilkinson, The algebraic eigenvalue problem, Oxford University Press, London (1965).