SU326 P30-14

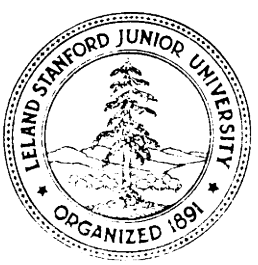
LARGE-SCALE LINEAR PROGRAMMING USING THE CHOLESKY FACTORIZAT ION

BY

M. A. SAUNDERS

STAN-CS-72-252
JANUARY 1972

COMPUTER SCIENCE DEPARTMENT
School of Humanities and Sciences
STANFORD UNIVERSITY



SU326-P30-14

Large-Scale Linear Programming using the Cholesky Factorization

by

M. A. Saunders
Stanford University

This work was supported by the U. S. Atomic Energy Commission, the New Zealand University Grants Committee, and the New Zealand Department of Scientific and Industrial Research.

Abstract

A variation of the revised simplex method is proposed for solving the standard linear programming problem. The method is derived from an algorithm recently proposed by Gill and Murray, and is based upon the orthogonal factorization

$$B = LQ$$

or, equivalently, upon the Cholesky factorization

$$BB^T = LL^T$$

where B is the usual square basis, L is lower triangular and Q is orthogonal.

We wish to retain the favorable numerical properties of the orthogonal factorization, while extending the work of Gill and Murray to the case of linear programs which are both large and sparse. The principal property exploited is that the Cholesky factor L depends only on which variables are in the basis, and not upon the order in which they happen to enter. A preliminary ordering of the rows of the full data matrix therefore promises to ensure that L will remain sparse throughout the iterations of the simplex method.

An initial (in-core) version of the algorithm has been implemented in Algol W on the IBM 360/91 and tested on several medium-scale problems from industry (up to 930 constraints). While performance has not been especially good on problems of high density, the method does appear to be efficient on problems which are very sparse, and on structured problems which have either generalized upper bounding, block-angular, or staircase form.

Contents

Large-scale Linear Programming using the Cholesky Factorization.

1. Introduction

The standard linear programming problem is

$$\begin{aligned} &\text{minimize} && c^T x \\ &\text{subject to} && Ax=b, \quad x \geq 0 \end{aligned} \tag{1}$$

where A is mx n and is usually very sparse. Virtually all methods currently in use for solving (1) are variations of the Revised Simplex Method (Dantzig [4]). If B is the usual mx m basis, the principal source of variation lies in the method chosen for solving two systems of equations* of the form

$$B^T\pi = \hat{c}\,, \quad By = a \tag{2}$$

at each iteration of the algorithm. This effectively means there are two areas in which methods can differ:

(a) the representation used for B^{-1} or its equivalent, for any particular initial B ;

(b) the technique used for updating B^{-1} when columns of B are changed one by one.

In both areas there are two problems to be faced:

(1) maintaining sparsity

(2) maintaining numerical stability,

and the aim here is to present a method which reaches a compromise between these requirements. 'The method is derived from an algorithm

---

*Or three systems, if the current basic solution $\hat{x}$ is obtained by solving $B\hat{x}=b$ directly (see Section 5).

recently proposed by Gill and Murray [8], and is based up the orthogonal factorization

$$B = LQ \tag{3}$$

or, equivalently, upon the Cholesky factorization

$$BB^T = LL^T \tag{4}$$

where L is lower triangular and Q is orthogonal ($QQ^T = I$). While the favorable numerical properties of the factorization (3) are widely recognized, the unknown quantity has been how to keep L sparse. We hope to make some progress in this direction.

In standard methods the conflict between sparsity and stability arises in the choice of pivot sequence, as is well known. Stage (a) above is called the reinversion phase, and most reinversion routines use either the product form of inverse (PFI) or the more recent elimination form of inverse (EFI). For example in EFI we have

$$P_1 B P_2 = LU \tag{5}$$

where P_1, P_2 are permutation matrices defining the pivot sequence, and L, U are respectively lower and upper triangular. Now for some choices of P_1, P_2 the LU factorization does not even exist, while for other choices it can be poorly determined. Therefore the search for permutations which lead to sparse factors must always be tempered by the fact that the resulting numerical error could sometimes be unacceptably high. Without judging the merit of different methods, we note that both extremes have been proposed in the literature: on one hand the method of Bartels and Golub [1], [2] gives top priority to numerical stability in choice of pivot elements, while in contrast the new *'preassigned pivot procedure" of Hellerman and Rarick [13], [14]

endeavors to choose an optimal pivot sequence by consideration solely of the zero/nonzero structure of B .

Again in the updating phase, once a change of basis has been determined by the rules of the simplex algorithm, the standard methods of updating PFI or EFI allow no freedom whatever in choice of pivot element. The method of Bartels and Golub (for updating the Hessenberg form encountered) is the only method which retains the possibility of pivoting for numerical stability.

Turning now to the orthogonal factorization, corresponding to (5) we have

$$P_1 B P_2 = LQ \qquad (6)$$

and in contrast to the above, this factorization exists for <u>all</u> permutations P_1, P_2 . This means that we are free to choose permutations from sparsity considerations alone, without fear that in so doing we might be compromising numerical stability. Furthermore, following Gill and Murray we do not store Q , and therefore we are concerned only with maintaining sparsity within L .

Unfortunately it happens that the degrees of freedom in (6) are much fewer than in (5) , because P_2 (being orthogonal) should really be incorporated into Q :

$$P_1 B = LQP_2^T = L\tilde{Q}$$

Thus for a given P_1, a change of P_2 will affect only Q , and the sparsity of L is therefore affected only by the choice of P_1 . Nevertheless, we are able to turn this fact to advantage, as described in the remainder of this paper. We choose P_1 not by examining any particular B but rather by taking a broader view and considering the

full matrix A itself. Any a priori knowledge of special. structure within A can often be put to good use at this stage.

An in-core version of the algorithm has been implemented, and the presentation here remains primarily within that context. Nevertheless, the algorithm is intended to be a practical method for solving a wide range of large, sparse linear" programs, and methods for implementing it out-of-core will be the subject of future research.

2. The Cholesky Factorization

If M is a symmetric, positive definite matrix, there always exists a lower-triangular L such that $M = LL^T$. L is called the Cholesky factor of M, and its elements are uniquely determined, apart from the sign of each column. In our particular application, $M = BB^T$, which is clearly symmetric and is also positive definite if B is non-singular. Hence the LL^T factorization exists for all bases B which arise in the simplex method.

It is emphasized now that the product BB^T is never actually computed, but rather L is obtained from a factorization of B itself. As is well known there always exists an orthogonal matrix Q ($Q^TQ = QQ^T = I$) such that

$$QB^T = R \tag{7}$$

where R is upper-triangular and has the same rank as B. It follows that

$$R^TR = (BQ^T)(QB^T) = BB^T$$

and hence the lower-triangular matrix we require is simply

$$L = R^T \tag{8}$$

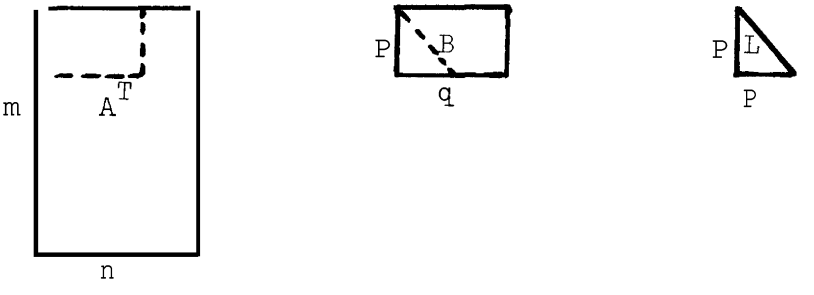
Note that (7) may now be written as in equation (3), $B = IQ$. In discussing the modification of L during change of basis, we will find it convenient to make use of equation (7), but at the same time equations (4) and (8) ($BB^T = LL^T$, $L = R^T$) will serve as reminders that Q is neither stored nor updated at any stage of the algorithm.

In the context of both linear and nonlinear programming, the use of the Cholesky factorization has recently been advocated by Gill and Murray [8], [9], [19]. As it happens, the good numerical properties of

the factorization constitute only one of several. attractive features. Thus in the linear programming application [8], Gill and Murray choose to consider the non-standard problem

$$\begin{aligned} &\text{minimize} && c^T x \\ &\text{subject to} && A^T x \geq b \end{aligned}$$

where A^T is now $m \times n$, $m > n$. They are then able to take advantage of the fact that the $BB^T = LL^T$ factorization exists even when B is not square. Thus B is allowed to have dimensions $p \times q$ where $p \leq q \leq n \leq m$, so that L will be $p \times p$ and the work and storage per iteration will usually be much reduced. Here p is the number of active inequality constraints, and since it will usually be true that $p \ll n$, the reduction in size can be quite significant:



Note in particular that the reduction in column-dimension to $q < n$ is obtained by giving special attention to constraints of the simple form $\pm x_j \geq b_j$, which is one very special form of sparsity within A.

Since linear programming problems arise in many different areas and can be widely varying in dimension and sparsity, it is unreasonable to expect that any particular algorithm would be ideal for all problems.

Thus, in cases where A is very dense except for simple upper and lower bounds, the algorithm of Gill and Murray will be considerably more efficient than standard methods, with regard to storage and computational requirements. On the other hand, in the area of large-scale linear programming the constraint matrix can be extremely large and in general will exhibit rather arbitrary sparseness. In such cases, even the $p \times p$ L above would be much too large for efficiency, if regarded as a dense matrix.

Our aim, then, is to extend the application of LL^T to large-scale problems by attempting to maintain sparsity within L . To this end we are forced to restrict ourselves to bases B which are square (thus treating the standard problem (1) and allowing exchange of columns as usual, but not allowing exchange of rows). We are then able to exploit yet another property of the Cholesky factorization, as stated in the following (trivial) theorem:

Theorem 1

The Cholesky factor of BB^T is independent of the ordering of the columns of B .

Proof

Suppose $BB^T = LL^T$, and let $\tilde{B}$ be the same as B except that its columns may be in a different order. Thus $\tilde{B} = BP$ for some permutation matrix P . Since $PP^T = I$ it follows that

$$\tilde{B}\tilde{B}^T = BPP^TB^T = BB^T = LL^T$$

and hence $\tilde{B}$ is associated with the same factor as B .

During both "reinversion" and subsequent updating, the storage of L will remain explicit (as opposed to product form), with linked lists

being used to represent the non-zero elements of each column. Further, a pre-processing of the full matrix A will select a particular row permutation, to be applied to A at the beginning and not changed thereafter. Theorem 1 then shows that the density of L for any particular basis depends only on which columns are in the basis, not on the order by which these columns happened to enter the basis during the iterations of the simplex method.

3. Motivation for pre-processing sparse A

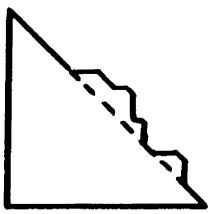
We suppose that A can be stored (compactly) in core and can therefore be subjected to an initial inspection of its rows and columns, as follows. We wish to find some row permutation P_1 and some column permutation P_2 such that the matrix P_1AP_2 is "as close to lower-triangular form as possible." Pictorially, this is intended to mean that the constraint matrix should look something like this:

P_1AP_2 =

(9)

where the lower-triangular part will still be very sparse. In general, any basis B will be made up of a fairly random selection of columns from A , but if P_3 is the permutation which sorts the columns of B according to their order of appearance in (9), we can expect the permuted basis to look something like this:

P_1BP_3 =



(10)

Thus if P_1 is chosen carefully-, there will always exist a permutation of the columns of P_1B (namely, P_3) such that P_1BP_3 has relatively few of its non-zero elements occurring above the diagonal.

Now it is well known that once a column has been selected to enter the basis, the simplex method allows no choice whatever about which column must leave (neglecting degeneracy), so that P_1B will generally

show no sign of being anything better than an arbitrarily sparse matrix. Thus it is here that we make use of Theorem 1, which tells us that the Cholesky factor L associated with P_1B is independent of the ordering of columns within P_1B. The mere existence of P_3 in (10) is all that we need.

In summary, the important points about pre-processing A are as follows:

1. Given a sparse matrix A there must exist permutations P_1, P_2 which arrange A in the form shown in (9). For if not, A would necessarily be quite dense.
2. With P_1 chosen and fixed, the existence of P_2 in (9) guarantees the existence of P_3 in (10).
3. The near-triangularity of P_1BP_3 gives reasonable justification for expecting that the associated L might have a density not much greater than that of B.
4. In deriving an initial row-ordering from the full matrix A, we clearly do not have an optimal ordering for any particular B. Instead we hope to obtain an ordering which is reasonably close to optimal for all B's encountered during the iterations, and we thereby justify storing the non-zero elements of each L explicitly. The density of the L's will fluctuate from one iteration to the next, but it is hoped that the average number of elements will remain within a range of say 2 to 5 times as many elements as in any B. "Reinversion" will never be necessary except for numerical reasons, because we do not wish to alter the row-ordering of B, and L is otherwise unique.

5. The above is in marked contrast to most existing LP systems, where the reinversion routine produces an extremely compact representation of B^{-1} for any particular B, but the updates during subsequent iterations are kept in product form so that the number of elements involved between reinversions is strictly increasing. It is this very property which enables conventional systems to operate out-of-core, but it is argued that in many cases (particularly with problems whose special structure is reflected in L) the average amount of data to be manipulated using LL^T might be significantly less than that involved in standard methods.

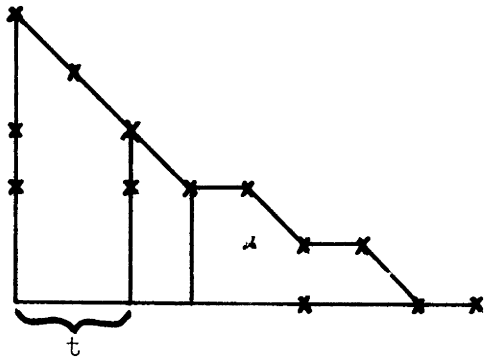
4. Finding the initial row permutation of A

We assume that the elements of A are already available via column lists. The row permutation P_1 is then easily found with the help of a (temporary) row-list and a set of row-counts giving the number of non-zero elements in each row of A . The procedure to be used follows one of the steps that is performed by existing reinversion routines for locating the so-called "forward triangle" of a basis (see Orchard-Hays [20]). The procedure is readily extended to the full matrix A . We will call the process triangularization.

Initially all rows and columns are considered to be eligible. Rows become ineligible as they are moved one by one to the top, and a column j becomes ineligible as soon as a row is chosen which contains an element in column j . The steps are:

1. Find the smallest row-count among any remaining eligible rows. Ties can be broken by keeping an unmodified copy of the counts for the full matrix.
2. Let the above row be number i in the original A . Take this row to be next-nearest-the-top, and make it ineligible.
3. Using the row-list, search row i and suppose there is an element in column j . Then use the column-list to reduce by 1 the count for any row which contains -an element in column j .
4. Make column j ineligible and repeat step 3 for any further elements in row i .
5. Repeat from step 1 if there are still. any eligible rows.

To illustrate the process, it can be verified that the following example is already ordered according to the algorithm.



Rows will be marked off from the top down, and columns become ineligible from left to right.

There is one obvious advantage in performing this operation on the full matrix A . It will often happen through redundancies in formulation of the linear program that the first t columns, say, will be strictly lower-triangular after the permutations (t=3 in the above example). This means that the first t variables are effectively fixed and can be immediately eliminated from the problem by a partial forward-substitution. Thus, the above 6 x 9 problem can be deflated from the beginning to dimensions 3 x 6 .

It is not at all clear that the triangularization method produces the best ordering of rows of A , and it has been suggested by J. A. George that a simple sorting of rows according to row-count might do just as well.* This is certainly easy to do, although it would not allow detection of any strict forward triangle. Also when the "cold start" technique described in section 11 is used, triangularization is likely to lead to an initial basis containing fewer artificial variables. As a compromise, the procedure currently being used is to

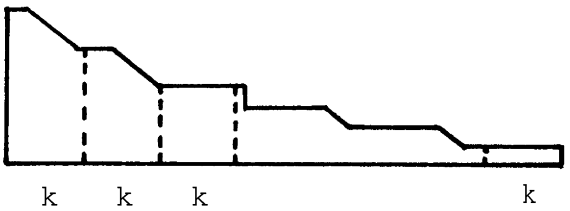
---

*See also the discussion of structured problems in section 12 .

move any markedly-dense rows to the bottom (maybe 5 or 10%) and then to proceed with triangularization of the remainder.

Once the permutation P_1 has been found, it is convenient for later purposes to permute the elements in each column accordingly. This is easily done within the column-list, one column at a time.

The column permutation P_2 can be discarded (it is never necessary to re-order the columns physically) or else the following refinement in "pricing" strategy could be used. As will be seen, the simplex multipliers are given by $L^T\pi = d$, so they are computed one by one from the bottom up. Clearly the back-substitution process can be stopped short at any time. Suppose now that columns are priced-out in groups of k, where $k \ll n$, and that the grouping is the one defined by P_2:



(The best column to enter the basis will be found in one group, an iteration performed, and then the next group examined.) Then for all but the first group the computation of the π_i can be stopped short, and for the last groups only a fraction of the multipliers need be computed. This strategy should lead to a significant saving on large problems. It is one which is available to any implementation whose representation of B^{-1} is explicit (as L is here) rather than in product form.

5. Solution of the linear eauations

Let $\hat{x} \geq 0$ be the current basic feasible solution, satisfying $B\hat{x} = b$. Then each iteration o the revised simplex method involves the following steps:

1. Solve the system

$$B^T\pi = \hat{c} \tag{11}$$

for the current simplex multipliers π.

2. Select a column a_s from A satisfying $c_s - \pi^T a_s < 0$.

3. Solve the system

$$By = a_s \tag{12}$$

4. Find r such that

$$\theta = \frac{\hat{x}_r}{y_r} = \min_{y_i > 0} \frac{\hat{x}_i}{y_i}$$

5. Update $\hat{x}$ according to

$$\hat{x}_i \leftarrow \hat{x}_i - \theta y_i \qquad (i \neq r)$$

$$\hat{x}_r \leftarrow \theta$$

6. Exchange columns a_r, a_s in the basis, and update the factorization of the new B.

Apart from step 6, the main work is in the solution of the linear systems (11), (12).

Observe that the updating of $\hat{x}$ in step 5 could involve numerical cancellation, and ideally should be replaced by a direct computation of $\hat{x}$ from $B\hat{x} = b$, after step 6. However this would imply a great'deal more work, and in practice no significant problems have been encountered

with updating, given that $\hat{x}$ is reset by direct solution of $B\hat{x} = b$ following reinversions.

Consider first the solution of (12). Following Gill and Murray [8] we see that y is given by

$$LL^T u = as \ , \ y = B^T u \tag{13}$$

The first of these equations is equivalent to $BB^T u = a_s$, so that $B^T u$ is one solution of $By = a_s$, and the non-singularity of B guarantees it is the only solution.

Similarly, when the system $B\hat{x} = b$ is solved after recomputation of L, the solution is given by $LL^T v = b$, $\hat{x} = B^T v$. Note that I-3 itself is required here, which is why it is convenient to have A (or at least, B) in-core.

The simplex multipliers are given by (11), $B^T \pi = \hat{c}$, and the non-singularity of B ensures that this is equivalent to $BB^T \pi = B\hat{c}$. Thus π could be found from

$$LL^T \pi = B\hat{c} \tag{14}$$

which is the method originally proposed by Gill and Murray in [8]. However it is considerably more efficient to transform $\hat{c}$ as though it were the last row of B. Suppose that the orthogonal factorization (7) gives

$$Q\,[B^T \mid \hat{c}] \ = [R \mid d] = [L^T \mid d] \tag{15}$$

Then (11) is equivalent to

$$L^T \pi = d \tag{16}$$

so that just one back-substitution is required to find π if d is updated along with L from one iteration to the next.*

*Since publication of [8], Gill and Murray have independently adopted this method also.

Another important advantage in using (16) arises from error considerations. Here we need a quantity called the condition number, $\kappa(B)$, defined as

$$\kappa(B) = \kappa(B^T) = ||B||\ ||B^{-1}||$$

where $||\cdot||$ denotes the euclidean norm. Suppose the system $B^T\pi = \hat{c}$ is perturbed slightly, by small changes to either B or the right-hand side (such as will be incurred by storing the data in a computer's finite-length word). The exact solution π will be perturbed by some amount proportional to the perturbations in the data, and it can be shown that the constant of proportionality is $\kappa(B)$. Thus $\kappa(B)$ provides an estimate of the intrinsic uncertainty in π . Hence it is reasonable to discuss in terms of $\kappa(B)$ also, the error that can result from round-off when a particular numerical method is used to compute π .

Returning to (16), it has been shown (e.g. Golub and Wilkinson [12]) that if $L^T\pi = d$ is used to solve (11) the relative error in π can be bounded by a term involving $\kappa(B)$, whereas if (14) is used the bound involves $\kappa(B^TB) = \kappa^2(B)$. This is the above mentioned advantage in using (16), from a standpoint of round-off error.

Unfortunately the situation is not so favorable when we use equations (13) to solve $By = a_s$. The relative error bound for y again involves $\kappa^2(B)$, and this could be a problem with severely ill-Conditioned data. (In some cases the algorithm of this paper could be applied to the dual linear program, since errors in π are often less important than errors in x .) We point out that if B = LQ the solution y is given by

$$Lw = a_s , \qquad QY = w$$

and the error in y would be bounded by $\kappa(B)$. Thus the above problem arises only because we are choosing to represent Q^{-1} by

$$Q^{-1} = Q^T = (L^{-1}B)^T = B^T L^{-T}$$

rather than storing and updating Q itself. Since B , and therefore A , would no longer be required in-core, an alternative implementation which maintained Q in product form might in some cases be preferable.

A full error analysis of the QR factorization has been given by Wilkinson [24]. The use of plane rotations to solve linear systems as in equations (15), (16) has also been analyzed by Van der Sluis [22].

6. Updating L upon change of basis

A change of basis will be accomplished in two stages:

1. Column a_r is deleted, giving an intermediate L which is singular.
2. Column a_s is added and the intermediate L is modified to produce a new L corresponding to the basis of the next iteration.

The reason for deleting before adding, rather than vice versa, is given by the following results. Suppose

$$LL^T = BB^T \qquad \text{and} \qquad \bar{L}\bar{L}^T = BB^T + aa^T$$

so that $\bar{L}$ is the Cholesky factor obtained by adding column a to B.

Theorem 2

The density of $\bar{L}$ can not be less than the density of L (neglecting numerical cancellation).

Corollary

When a column is removed from B, the density of the new factor can not (significantly) increase.

The theorem is obtained by considering the effect of the elementary orthogonal transformations used to update the QR factorization of a matrix when a row is added to the matrix (see section 8 and recall $L = R^T$). Briefly it is due to the fact that if

$$\begin{bmatrix} \bar{\alpha} \\ \bar{\beta} \end{bmatrix} = Z \begin{bmatrix} \alpha \\ \beta \end{bmatrix}$$

where Z is a 2 x 2 orthogonal matrix, then usually

$$\bar{\alpha} \neq 0, \qquad \bar{\beta} \neq 0$$

unless $\alpha=0$ and $\beta=0$.

The corollary follows because the uniqueness of L with respect to column permutations on B implies that removal of a column is the exact reverse of the process of adding it back again.

The main point is that we wish to ensure that the intermediate L above will in general be less dense than its predecessor. We note here that the factorization

$$BB^T = LDL^T$$

has also been (successfully) used, where D is diagonal and $\ell_{ii} = 1$, to take advantage of the fact that no square roots are required during updates, and less divisions are required during back-substitutions. However for numerical stability it is essential with this factorization to avoid singularity by adding before deleting, and consequently the intermediate L will generally be more dense than its predecessor, sometimes markedly so. It is now felt that the possible severity of this fluctuation outweighs the other advantages that LDL^T might have over LL^T.

The corollary has a useful practical implication. If the density of L does increase significantly when a column is deleted, then most of the new non-zeros must be due to propagation of noise, in the form of very small numbers which should be treated as zero. This can happen after a large number of modifications and provides one of the several indicators needed in practice to trigger reinversion.

7. Removal of a row from the QR factorization

In preparation for modifying L when column a_r is deleted from B, let us adopt the QR notation commonly used in numerical linear algebra. As noted in equation (7), the QR factorization is

$$QA = R$$

where Q is orthogonal, $R = L^T$, and we are temporarily defining $A = B^T$. We wish to delete row a_r^T from A, and we now give a new method for accomplishing this using elementary orthogonal matrices in a manner which is becoming increasingly well known (cf. Golub [11], Gill and Murray [8],[9]). Suppose for notational purposes that A is $m \times n$, $m \geq n$, although we are mainly interested here in the special case when A is square. Application of the method to rectangular A will be discussed more fully in [10].

Let $R^T p = a_r$, and consider the sequence of elementary orthogonal matrices Z_i such that

$$Z_i \begin{bmatrix} p_i \\ \delta_i \end{bmatrix} = \begin{bmatrix} 0 \\ \delta_{i-1} \end{bmatrix} \qquad i=n, n-1, \ldots, 1$$

Here, the Z_i could be plane rotations of the form

$$\begin{vmatrix} \cos\theta_i & -\sin\theta_i \\ \sin\theta_i & \cos\theta_i \end{vmatrix}$$
$$1$$

or else 2×2 Householder transformations, and the equation serves to define θ_i, δ_{i-1} in terms of p_i, δ_i. The starting value δ_n remains to be chosen.

The Z_i are now inflated with the appropriate parts of the identity matrix and applied in turn to the matrix

$$\left[\begin{array}{c|c} p & R \\ \hline \delta_n & 0 \end{array}\right]$$

As the elements of p are reduced to zero from the bottom up, the row below R gets filled up from right to left. R is modified row by row, but retains its triangular structure. Thus we have

$$Z_1 Z_2 \cdots Z_n \left[\begin{array}{c|c} p & R \\ \hline \delta_n & 0 \end{array}\right] = \left[\begin{array}{c|c} 0 & \bar{R} \\ \hline \delta_0 & s^T \end{array}\right] \quad 1$$

and the orthogonality of the Z_i gives

$$\left[\begin{array}{c|c} p^T & \delta_n \\ \hline R^T & 0 \end{array}\right] \left[\begin{array}{c|c} p & R \\ \hline \delta_n & 0 \end{array}\right] = \left[\begin{array}{c|c} 0 & \delta_0 \\ \hline \bar{R}^T & s \end{array}\right] \left[\begin{array}{c|c} 0 & \bar{R} \\ \hline \delta_0 & s^T \end{array}\right]$$

so that

$$p^T p + \delta_n^2 = \delta_0^2$$

$$R^T p = \delta_0 s$$

$$R^T R = \bar{R}^T \bar{R} + s s^T$$

Now the natural choice for δ_n is clearly the value which gives $\delta_0 = 1$ since this implies $R^T p = s$ and therefore

$$S = a_r$$

$$\bar{R}^T \bar{R} = R^T R - a_r a_r^T$$

as required.

In general this means setting $\delta_n = \sqrt{1 - p^T p}$. For the special case $m = n$, comparing the equations

$$R^T Q = B, \qquad R^T p = a_r$$

shows that p is just the r-th column of Q . Hence $p^T p = 1$ and we set $\delta_n = 0$.

The discussion above brings to light two properties of the updating method which might be used as a measure of numerical error. First, the vector being eliminated from A is re-generated as the vector s , and thus a non-trivial discrepancy between s and a_r could imply significant numerical error in R .

The second check is available only in the special case $m = n$, where $\delta_n = 0$, $\delta_0^2 = p^T p$. Since δ_0^2 is actually computed from p as a by-product of the updating, it is available at no extra cost, and any significant deviation of δ_0^2 from 1 implies numerical error in p and therefore either similar error in R or ill-conditioning of the current B (or both). In practice, the size of $|\delta_0^2 - 1|$ can be monitored and it provides us with some sort of numerical check every iteration. A continuous numerical check of this kind is something which is not common in standard linear programming systems. (Forrest and Tomlin report a similar check in their new LU implementation [6]).

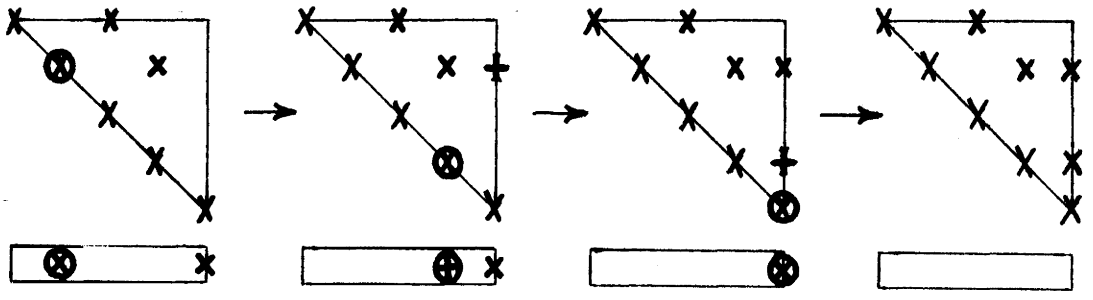
8. Adding a column to the basis

We will continue to use QR notation here, as in the previous section. The modification of I when column a_s is added to B is the process of modifying R when a row a_s^T is added to B^T. It is well known how to do this using plane rotations, when R is dense (e.g. Golub [11]). In general, with R $m \times m$, we would have

$$Z_m \, Z_{m-1} \cdots Z_1 \begin{bmatrix} R \\ v^T \end{bmatrix} = \begin{bmatrix} \bar{R} \\ 0 \end{bmatrix}$$

where each Z_i is an elementary orthogonal transformation defined at each stage by two elements, namely r_{ii} and the i-th element of v after modification by the previous transformations $Z_1, \ldots, Z_{i-1}$.

When R and v are sparse, the algebra is the same but many of the Z_i will simply be I. To minimize computation time we need a data structure which indicates directly which transformations are non-trivial. To illustrate, let us consider an example with $m = 5$. Suppose R has two off-diagonal elements as shown below, and suppose that the new row has only two non-zero elements. The steps by which V is reduced to zero are as follows:

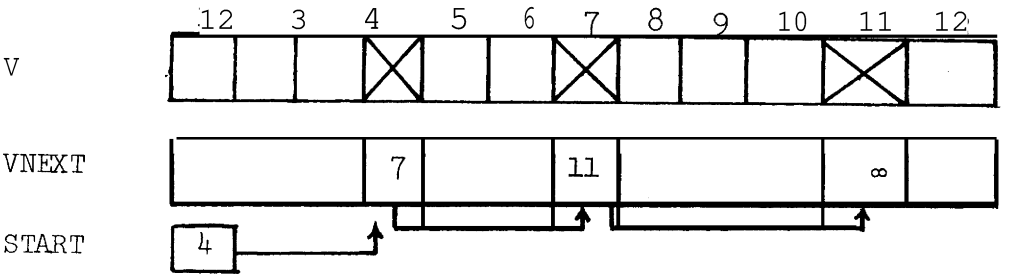


Circled elements define the transformation each stage. Non-zero elements are marked by x , and new non-zeros (which were zero in the previous stage) are marked by + . We see that

1. Rows 1 and 3 of R are unchanged.
2. A new element has appeared in each of rows 2 and 4 .
3. A non-zero element was produced'in v at the second stage, and this element had to be reduced to zero by a non-trivial transformation.

Some of the computed elements could prove to be below some pre-specified tolerance, and should be eliminated from the data structure. This will happen occasionally during the addition process, and will occur quite frequently during the reverse process of removing a column from the basis.

Because of this need to insert and delete elements, we have chosen for in-core implementation to use a linked-list to represent the non-zero elements in each row of R , as described in the next section. A simpler kind of list can be used to keep track of the elements in v , at a slight cost in storage. Suppose the non-zero elements of the initial v are placed in the appropriate positions of an array V(* of dimension m . Then an integer array VNEXT(*) is used to point to these elements in an obvious way:



Clearly there is space available for any new non-zero elements that arise during the addition process. It would be possible to set $V(j) = 0$ wherever $v_j = 0$ initially and eliminate VNEXT(*) altogether, but note that each non-trivial transformation Z_i involves a significant amount of computation. If the essential part of Z_i is the 2x 2 orthogonal matrix Q_i satisfying

$$Q_i \begin{bmatrix} r_{ii} \\ v_i^{(i)} \end{bmatrix} = \begin{bmatrix} \bar{r}_{ii} \\ 0 \end{bmatrix}$$

then we must compute

$$\begin{bmatrix} \bar{r}_{ij} \\ v_j^{(i+1)} \end{bmatrix} = Q_i \begin{bmatrix} r_{ij} \\ v_j^{(i)} \end{bmatrix}$$

for $j = i+1, \ldots, m$. (Here, $v_j^{(i)}$ is the j-th element of v before the i-th transformation.) The list structures for R and v enable us to economize on arithmetic when either r_{ij} or $v_j^{(i)}$ is zero, and more importantly allow us to skip directly past any computation for which r_{ij} and $v_j^{(i)}$ are <u>both</u> zero (which is the most frequent case).

The "ADD" routine for performing the above process is employed in two situations:

1. To add a column to the basis each iteration, following the removal of a column.
2. To recompute $L = R^T$ from scratch whenever a "reinversion" is required see section 10).

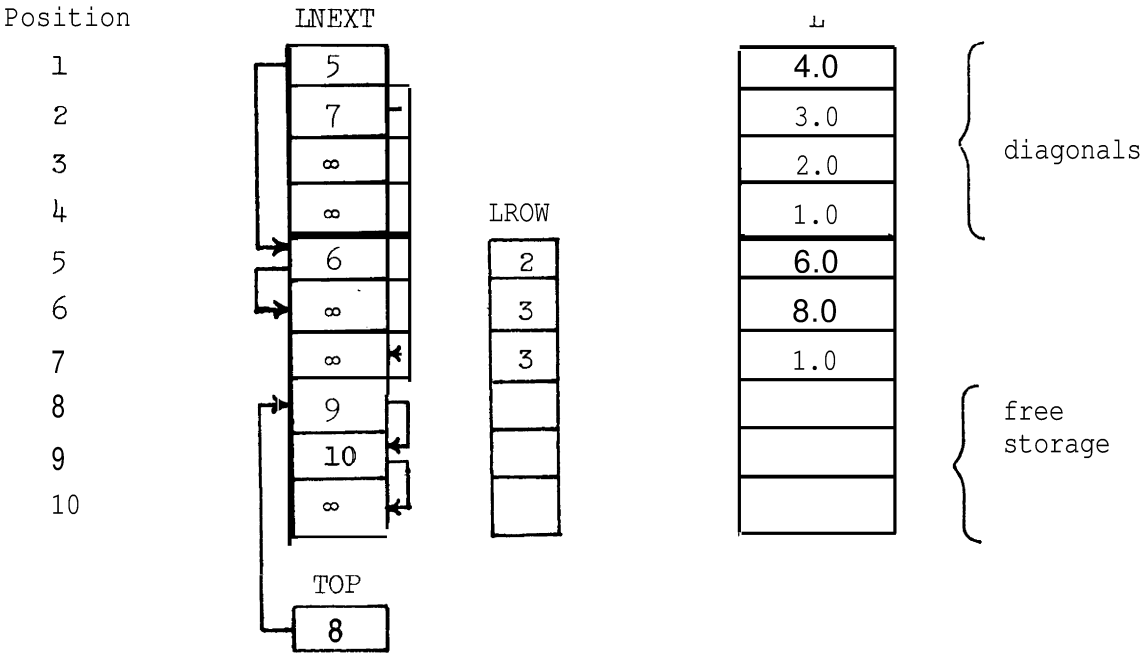
In both cases it happens that R is singular upon entry to the routine. We wish to emphasize that the process is nevertheless well-defined and numerically stable.

9. Storage of sparse L

As indicated in the previous section, we use a linked-list to store the elements of each column of L. In a high-level language, this can be done with three parallel arrays as follows. An integer array LNEXT(*) serves as a set of pointers into another integer array LROW(*) which contains the row index of the next non-zero element in a particular column. The element itself is stored in a floating-point array L(*). For example, to store the matrix

$$L = \begin{bmatrix} 4 & & & \\ & 3 & & \\ 6 & & 2 & \\ 8 & 1 & & 1 \end{bmatrix}$$

the arrays might be set up as follows:

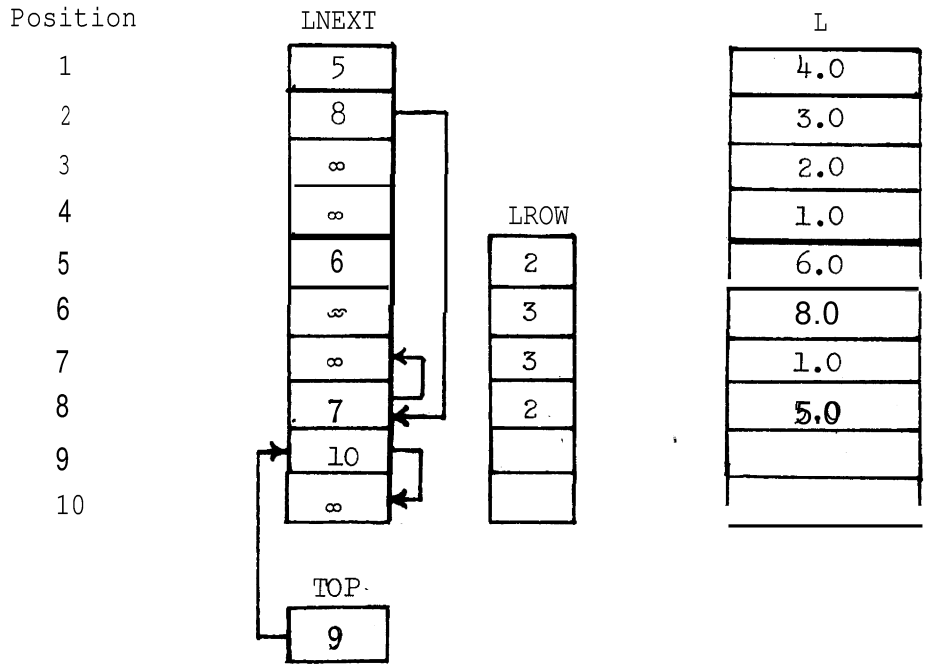


The ∞ sign marks the end of each column, and TOP points to the beginning of the linked-list of free storage, also contained in LNEXT(*) .

If an element 5.0 is inserted into column 2 to give

$$L = \begin{bmatrix} 4 & & & \\ & 3 & & \\ 6 & 5 & 2 & \\ 8 & 1 & & 1 \end{bmatrix}$$

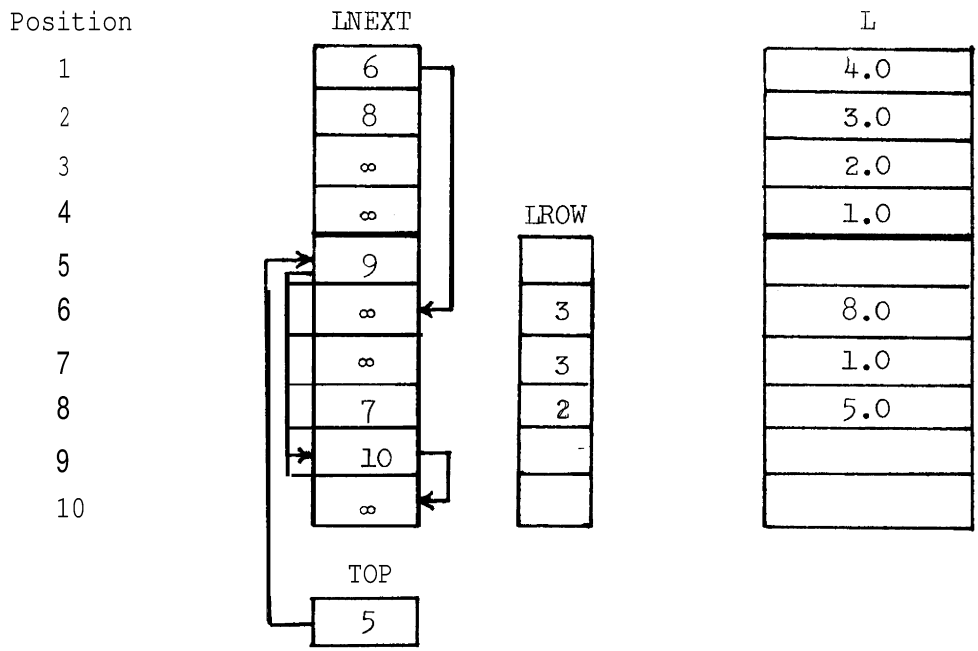
the arrays would be modified to look like this:



and similarly if the element 6.0 were deleted from column 1, we would have

$$L = \begin{bmatrix} 6 & & & \\ & 3 & & \\ & 5 & 2 & \\ 8 & 1 & & 1 \end{bmatrix}$$

and the arrays would change to the following:



The purpose of lists is, of course, to enable elements to be inserted and deleted without having to push existing elements around to make room. We see that "holes" can appear, which are linked into the free-storage list, and that the elements of any particular column will diffuse through the list as modifications proceed. Since L is accessed column by column during each simplex iteration, if the algorithm is implemented in a virtual-memory environment it would probably be profitable to group together occasionally the elements in each column, to alleviate the problem of memory fragmentation.

10. Tieinversion

Following convention, we use the term reinversion to mean computation of L from a particular basis B , despite the fact that no matrix inversion is involved. Orthogonal triangularization would be a more accurate description. There are two main parts to the process, as follows.

First, as many columns as possible are taken from the current B and placed directly in L . Usually the majority of columns in B can be so placed, because the pre-processing of A should ensure that all bases are nearly triangular. No arithmetic is required at this stage and the order of placement is irrelevant. Thus in a single pass through A each basic column is copied into the position defined by its first non-zero element, as long as there is no column already in that position.

Next, unfilled columns of L are set to zero (via lists of zero length) and unplaced columns of B are added one by one, by repeated calls to the ADD routine of' section 8 . Again any order will do, so a second pass through A is all that is required.

Reinversion time depends heavily on the growth of non-zero elements within L , and also on the number of columns of L that are modified by each call to the ADD routine in the second stage. As might be expected, triangularization of a relatively dense B can involve a great deal of computation, while for very sparse problems the process can be quite rapid, since there is no permutation-finding logic involved, and only a small percentage of the columns are affected during the second stage.

The principal reasons for invoking reinversion are listed below. An indication is given of tolerances that have been used on the IBM 360/91 when computation is performed in long precision (approximately 15D), aiming for about 7D precision in the solution (or better). We assume that the data has been scaled as described in section 13.

1. The row and column residuals

$$\rho = b - B\hat{x}, \quad \gamma = \hat{c} - B^T\pi$$

are computed at regular intervals (e.g. every 25 or 50 iterations). Reinversion is called if

$$|\rho| > 10^{-7} \quad \text{or} \quad |\gamma| > 10^{-6}$$

where $|\rho|$ is the average of $|\rho_i|$.

2. The pivot element y_r in equation (12) should not be too small. Reinversion is performed if $y_r < 10^{-3}$. (This may be too large for some problems.)

3. During deletion of a column from L, reinversion is called if the quantity $|\delta_0^2 - 1|$ (section 7) exceeds 10^{-7}.

4. Also during deletion, if the number of new non-zero elements in 1, (which should be negative) exceeds about 5% of the total non-zeros, reinversion is called to eliminate what must be noise (see section 6).

11. Finding an initial basis

The following simple "cold start" procedure can be used to find an initial non-singular basis. Its implementation is made easy by the preliminary triangularization of A . A full identity matrix of either slack or artificial variables could be used, but the aim is to do better and instead we look for a basis which is strictly lower-triangular (so that B = L) . This guarantees non-singularity equally well, and on a typical sparse problem can usually be done with the help of only a few artificial columns.

In a single pass through the column-list for A we look at the position and sign of the first element in each column, and record for each i (i = 1,2, . . . , m) the "best positive column" and the "best negative column." By this we mean the following. Suppose the first non-zero element in column j is a_{ij} and suppose $a_{ij} > 0$. Then column j is a possible candidate for "best positive column for row i", depending say on the size of c_j relative to the previous best. Similarly if $a_{ij} < 0$, column j might become the best negative column for row i .

An initial B = L is now selected from the above candidates, and a forward substitution $L\hat{x} = b$ is performed in parallel in order to ensure that the resulting $\hat{x}$ is feasible $(\hat{x} > 0)$. At the i-th stage, the sign of

$$b_i - \sum_{j=1}^{i-1} l_{ij} \hat{x}_j$$

determines whether a positive or negative column should be used as the

i-th column of L . If there is no acceptable candidate, we must introduce an artificial column of appropriate sign $(\pm e_i)$.

12. Structured problems

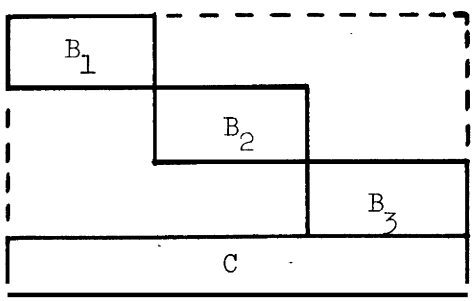
The techniques discussed so far are intended for use on general sparse linear programs. Now it often happens that the matrix A in (1) has special structure, and if we have knowledge of this it is natural to want to exploit such information wherever possible. As we shall see, the LL^T factorization does allow us to utilize structural information, and it is only during the preliminary triangularization of A (see sections 3, 4) that special care need be taken. No modification is required to the simplex algorithm itself.

The discussion here bears certain similarities to the compact basis triangularization proposed by Dantzig [3] for staircase problems, in that we are talking about preserving structure from one iteration to the next.

(a) Block-angular problems

In this case, the constraint matrix has the form

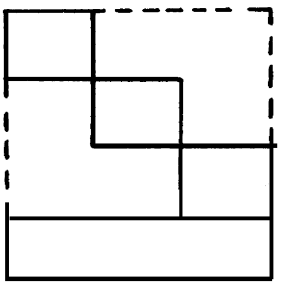
A =



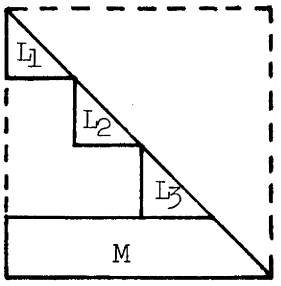
(a 3-block example) where each block B_i is usually sparse, and B_3 may have zero row-dimension. Recall that the preliminary triangularization of A is effectively just a row permutation. We wish to restrict the permutation now to be one which triangularizes each B_i individually, and at the same time moves the coupling constraints C to the bottom,

as shown. For some suitable column-ordering P each basis B will then possess the same block-angular form, and since the corresponding Cholesky factor is independent of column order, it follows that L has the form shown below:

BP =



L =



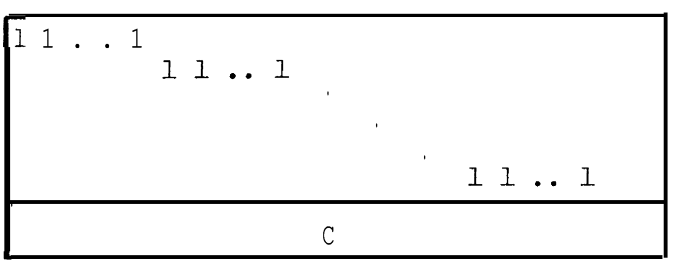
Even if no row permutation were applied to A , L would maintain the block-triangular structure, but the triangularization of each B_i should ensure that each triangle L_i will remain sparse. Similarly, triangularization of sparse C should lead to reasonably sparse M .

This example emphasizes the point already made, that the Cholesky factorization automatically takes advantage of useful structure during the simplex iterations, even though it is "unaware" that such structure is present. Once an appropriate row permutation has been fixed it is unnecessary to retain information on row or column partitions within A .

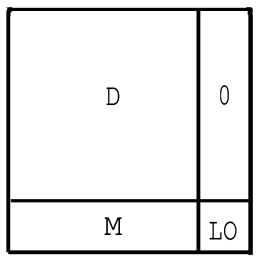
(b) Generalized Upper Bounding (GUB)

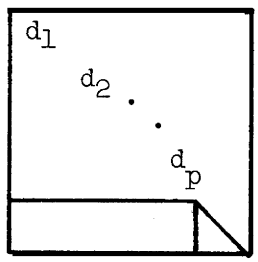
This is a special case of the block-angular structure, in which each block has only one row, usually with all elements $\pm$ 1 (see Dantzig and Van Slyke [5]):

A =



'kc number of GUB sets is usually very large compared to the number of coupling constraints in C , so each basis is almost completely triangular. The Cholesky factors are of the form

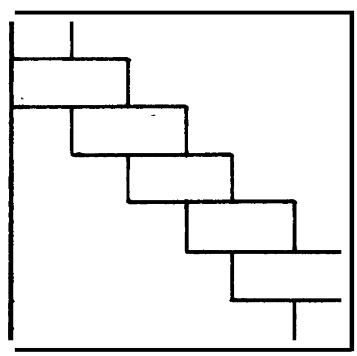
L = 

= 

with D diagonal and L_0 triangular. If the i-th GUB set has n_1 members in a particular basis, the corresponding L will have $d_i = \sqrt{n_i}$. Since it is true that most $n_i=1$, efficiency could be improved by taking this and certain other simplifications into account. Nevertheless a general implementation of the LL^T method can derive high efficiency from the GUB structure automatically, and in contrast to standard GUB codes does not require any specialized "housekeeping" for monitoring the status of the variables in each set.

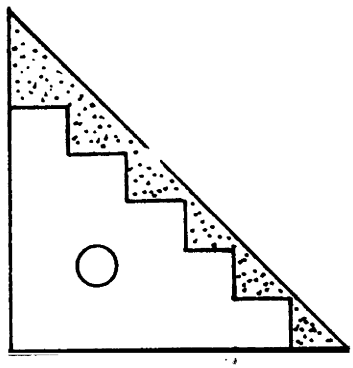
(c) Staircase structure

Multi-stage systems give rise to problems in which each basis has the following staircase form:

B = 

In this case also, the Cholesky factors preserve the profile of each B below the diagonal:

L =

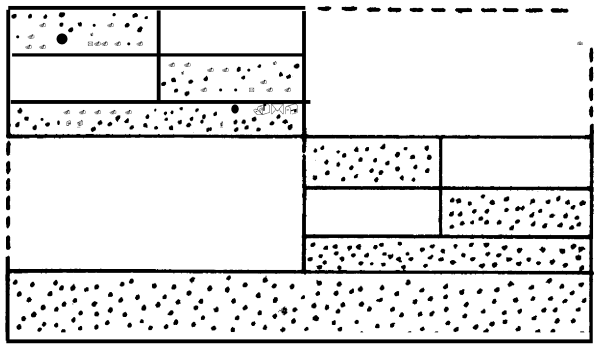


The full matrix A has a form similar to the B shown, and an individual triangularization of the rows within each horizontal stair should minimize the density of each L within the profile.

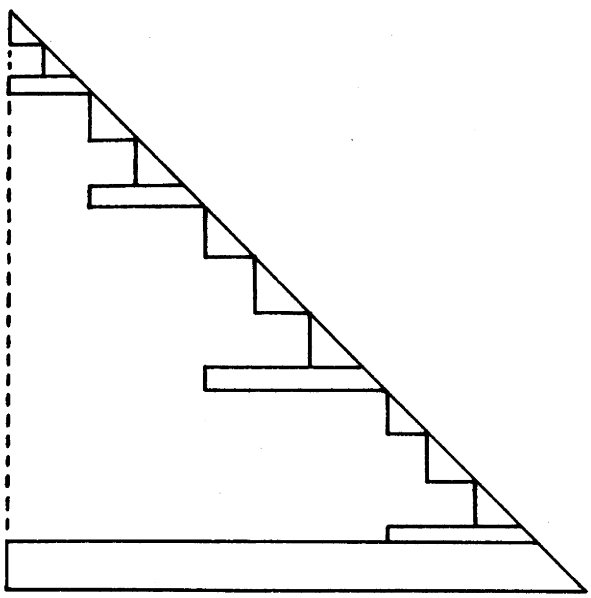
(d) Unstructured problems

Since it is often true that structure within a problem reflects important characteristics of the physical system which the problem is modeling, it would not be unrealistic to recommend that grouping of constraints into one of the above forms be done during formulation of the model (i.e. by human hand). The grouping can then be input to a preliminary computer routine for further (more localized) triangularization.

If a given problem has no apparent structure at all, it may be profitable to adopt as the pre-processing phase, the method of Weil and Kettler [23] for rearranging A into block-angular form. Since the Cholesky factorization takes best advantage of this particular structure, it is conceivable that even the diagonal blocks thus obtained should in turn be processed by the Weil and Kettler algorithm, to produce structures of the following form:



The sub-structure within each block will be preserved in L just as it is for the broader partitions. If there are many blocks of this form, each L will have the following interesting profile:



This is strongly reminiscent of the profile arising in the work of J. A. George [7,Ch.4], wherein the Cholesky factorization $A = LL^T$ is considered, where A is a given symmetric, positive definite sparse matrix. Symmetric row/column permutations applied to A lead to L's of varying density, and since the profile of A is preserved in L the least dense factors are obtained by minimizing the number of elements within the profile (rather than minimizing bandwidth, say). Thus an

ordering was found which had a "spike" structure similar to the L above. Possibly this observation will throw some light on the problem of finding an optimal permutation when A is of the form BB^T .

A further possibility for unstructured problems arises from the work of H. Konno [16], [17]. An algorithm is given in [16] for solving the so-called bilinear programming problem (BLP) using a sequence of linear programs. One of the applications of BLP given in [17] relates to the triangularization philosophy. The measure of lower-triangularity used for a square sparse-matrix A is the number of non-zero elements occurring above the diagonal of A . It is shown that the problem of finding row and column permutations which maximize triangularity can be cast as a BLP , and hence in principle can be solved. An extension to rectangular A should be possible. [Unfortunately this approach would not be practical for large problems unless the same structure were to be used many times.]

13. Computational results

Most of the ideas described here have been implemented in an Algol W program* on the IBM 360/91 at the Stanford Linear Accelerator Center. Data is input in MPS/360 format (see [18]), including simple upper and lower bounds on any number of variables. Algol W was chosen as a flexible and convenient programming language for development purposes, without which evolution of the algorithm would have been very much slower. However no direct-access I/O facilities are available and the implementation is therefore strictly in-core. Comparison with other systems is difficult in view of the different machines, programming techniques and use of core, but we give performance figures where they are available. The run times recorded below would be reduced by a factor of 3 or 4 if assembly language were used in place of Algol W . Alternatively the times should be multiplied by a factor somewhere between 1.2 and 1.5 to give equivalent run times of an assembly language implementation on an IBM 360/67 (very approximately).

Four medium-scale problems have been used as test cases. They are listed in Table 1, in order of increasing difficulty. Some relevant run-time statistics are given in Table 2, where time is measured in seconds of 360/91 CPU utilization, and the number of elements in L refers to non-zeros below the diagonal. optimal

*Algol W was developed for the IBM 360 by faculty and students of the Computer Science Department at Stanford University, as a refinement and extension of Algol 60. No facilities were used here that are not available in Algol 60, except for character strings used for reading the MPS/360 data.

solutions were found for problems A, B and C , but problem D could not be run to completion because there was insufficient storage available for L.

The main features of the solution strategy are as follows:

1. The input data was stored in short precision (6 hexadecimal digits in the mantissa) but all computation and working storage used long precision (14 hexadecimal digits).
2. A row scaling was applied to A to make the largest element in each row approximately equal to 1 (to the closest power of 2). Then a column scaling was applied to reduce the euclidean length of each column to approximately 1 . This is an attempt to improve the condition number of each basis.
3. Except on problem C, the procedure of section 4 was used to permute the rows of A into approximately lower-triangular form.
4. The cold start procedure of section 11 was used to find an initial triangular basis, with the help of a number of artificial variables.
5. For ease of implementation, the usual two phases of the simplex method were replaced by a single minimization (the "big M" method) in which artificial variables are given a value $c_j = M$ in the cost vector c , where M is-sufficiently large that their value in an optimal solution is zero. Usually $M = 1000 \times \max|c_j|$ is large enough.
6. The pricing strategy used was also non-ideal but easy to implement. The first k columns are considered for entry into the basis, where k is pre-set to something like 300 or 400 , depending on the number of variables and the expense of computing

reduced costs relative to the expense of changing basis. An iteration is performed using the column with most-negative reduced cost, and then the next k columns are considered.

7. Reinversions were performed only when the error conditions of section 10 were encountered.

8. The maximum amount of memory available to the Algol W program for work space is approximately 500K bytes (62,500 long words). This was more than enough for all cases except problem D .

As Table 2 shows, problems A and B were solved quite easily, but the high density of problems C and D caused considerably more difficulty. We will discuss each problem briefly.

Problem A Being a network problem, this example is numerically well-conditioned (all elements ± 1) and highly triangularizeable. With only 3 elements per column its density is also very low and it is not surprising that L remained very sparse throughout the iterations. Figure 1 shows the growth of off-diagonal elements in L (N_L) along with the number of artificial variables (N_A) as functions of iteration number. It is to be expected that L should become more dense as unit vectors are replaced by somewhat denser columns. If iterations had continued, a levelling off would have occurred, as exhibited by problem C .

This problem was obtained via R. R. Meyer from the Shell Development Company, California who also obtained the following comparative performances:

Code	Machine	Iterations	Time (seconds)
FMPS	Univac 1108	477	81
ILONA	Univac 1108	?	50

The iteration time of 21.3 seconds and total solution time of 30.2 seconds shown in Table 2 (Algol W, 360/91) compare reasonably well with these figures.

Problem B This problem is of generalized upper bounding type, with 890 GUB sets and 39 coupling constraints. The triangularization procedure of section 4 was successful in moving most of the coupling constraints to the bottom of A (no special effort was made to do this exactly). As explained in section 12, the Cholesky factors are almost triangular, and the number of off-diagonal elements in L was virtually constant at around 1500 throughout the 958 iterations. Starting at 1428 elements, this number never exceeded 1534. Again the problem was well-conditioned and only 1 reinversion was called, at iteration 555 using $|\delta_o^2 - 1|$ as control (see section 7), which served to avert a slight onset of noise within L .

This problem was obtained from the Crown Zellerbach Corporation, via M. G. Kazatkin, and provides a good example of how the Cholesky factorization, together with explicit storage and updating of L , can take advantage of structure. R. B. Johnston of Crown Zellerbach obtained the following benchmark results on several commercial systems (last quarter, 1970):

Code	Machine	Iterations	Time (minutes)
FMPS	Univac 1108	1700	13.5
UMPIRE	Univac 1108	1852	13.5
ILONA	Univac 1108	1491	9.5
MPS/360	IBM 360/65	1885	36.3
OPTIMA	CDC 6600	420	2.4

The first two systems also solved the problem with special GUB codes, and returned times of 13.0 and 4.5 minutes respectively. It can be seen that the results in Table 2 (95.5 seconds iteration time, less than 2 minutes total time) compare quite favorably.

Problem C This is a dynamic multi-sector model with staircase structure, obtained from Professor A. S. Manne and K. W. Kohlhagen at Stanford University. There are six main "stairs" or blocks, each approximately 50×100 . Ideally each block should be triangularized individually, but this was done only crudely by hand, and further triangularization by program was suppressed. (Triangularization of A as a whole, destroyed the staircase structure and led to very dense L's .) Although small in absolute dimensions, this problem was rather difficult to solve for two reasons:

1. The density of 2.3% is moderately high, but since all elements are concentrated within the staircase structure, the density of each block is more like 10% , which is very high.
2. Numerically speaking the problem is ill-conditioned, with the size of matrix elements ranging from order 10^{1} down to order 10^{-5} . (This range was not altered significantly by the row and column scaling.)

All reinversions were called following selection of a column for which the pivot element was unusually small. (Such a column is then temporarily rejected and a different one chosen for entry into the basis.) The relatively high reinversion time of 6 seconds reflects the strong linkage between variables and indicates that many columns of L are affected by each basis change. This in turn emphasizes that with dense problems it can be expensive to update L explicitly. (Correspondingly, standard methods of updating in product form would lead, in PFI for example, to a rapid growth of eta elements and consequently to relatively frequent reinversion.)

As figure 2 shows, the number of elements in L increased steadily while artificial variables were being removed from the basis, and then levelled off at a little over 12000 . This steady state is due to the fact that the staircase structure is being preserved by the sequence of Cholesky factors. Though the figure of 12000 is large considering the size of the problem, it simply reflects the high density of the data and would have been much larger if structure were not preserved.

Similar difficulties are reflected in the performance of MPS/360* on a smaller (316 x 463) unstructured formulation of the same problem. An initial reinversion, starting from an advanced basis, failed with a row error of 105 . All subsequent reinversions were successful, but illustrate well the possible disadvantages of the product form of

*This is MPS/360 V2-M9 , running on an IBM 360/67 at the Stanford University Computation Center, Campus Facility.

inverse in certain cases. The number of eta elements ranged from around 24,000 after reinversion, up to nearly 40,000 about 50 iterations later. Reinversion time was between 0.6 and 0.8 minutes, and total run time from cold start was approximately 21 minutes.

Problem C (modified)

A more direct comparison with MPS/360 was obtained using the staircase model with many of the variables fixed in value. Resultant problem size was 357 x 385 , plus 148 slack variables. Only 6 of the slacks appeared in the optimal basis.

The performance of each method is summarized in Table 3, and it appears that on this test case the Cholesky method has performed significantly better than the standard method using product form of inverse. The growth of elements in L and PFI are plotted in figure 3. We must point out that L is used four times each iteration, whereas PFI is used only twice. Nevertheless the results are interesting from a storage point of view. The jump in density of PFI about 250 iterations before optimum was due to a row check failure, followed by a repeated reinversion with a tighter pivot tolerance.

Problem D

This is the first of three problems used experimentally by Forrest and Tomlin, called problem A in [6], [21] . It was treated as a general sparse linear program. During the run shown in Table 2, the number of elements in L increased steadily to 21000 , which represents the maximum storage that could be allocated for this

particular problem in the Algol W implementation. The run was terminated before an optimum solution was found.

Forrest and Tomlin give comparative figures for two methods, both starting from a full basis and an LU factorization stored in product form (EFI). With the standard product form of update, the number of eta elements increased from 4861 to 35885 after 70 iterations, whereas with their own method for updating the LU factors the number grew from 4861 to only 8958 , which is a significant improvement.

The poor performance of the Cholesky factorization on this example was partly explained by an inspection of the constraint matrix, which proved to be approximately dual-angular in structure (containing coupling variables rather than coupling constraints), with 6 main diagonal blocks of relatively high density, and about 400 coupling variables. This structure is not one which is preserved by the LL^T factors. It is possible that the Cholesky method would perform better on the dual problem, since this would have standard block-angular form (but would be considerably larger in overall dimension).

Problem	A	B	C	D
Structure	Network	GUB	Staircase	General
Rows	537	930	357	822
Structural columns	1775	3562	467	1571
Elements	3556	15103	3856	11127
Density	0.37%	0.36%	2.31%	0.86%
Bounds	Yes	Yes	Yes	No

Table 1. Test problem characteristics.

	A	B	C	D
Time to input data	5.5	11.0	3.5	10.5
Time to triangularize A	3.4	10.4	Rows not permuted	8.7
No. of artificial vars.	82	6	137	168
Initial no. of elements in L; density	407 0.28%	1428 0.33%	2248 3.5%	2262 0.67%
Final no. of elements in L; density	1046 0.73%	1510 0.35%	12228 19.4%	21000 6.2%
No. of reinversions	0	1	8	14
Typical reinversion time	3.18	0.23	6.0	5.0
No. of iterations	343	958	488	284*
Time for iterations	21.3	95.5	311	199*
Row and column residuals, before final reinversion	$10^{-17}, 10^{-13}$	$10^{-14}, 10^{-14}$	$10^{-10}, 10^{-13}$	$10^{-11}, 10^{-9}$
Row and column residuals, after final reinversion	$10^{-16}, 10^{-16}$	$10^{-17}, 10^{-18}$	$10^{-12}, 10^{-14}$	$10^{-12}, 10^{-11}$

Table 2. Solution statistics using the Cholesky factorization.
All times in seconds of 360/91 CPU utilization.
(*Optimum not reached for problem D.)

LL^T, Algol W, 360/91		PFI, MPS/360, 360/67	
Cold start	0 iterations 0.02 minutes 151 artificial variables	Crash	204 iterations 0.98 minutes 66 infeasibilities
Phase 1	212 iterations 1.65 minutes	Phase 1	3 3 4 iterations 7.42 minutes
Reinversion	0.09 minutes 11923 elements in L	Reinversion	0.22 minutes 19186 eta elements before invert 11710 eta elements after invert 3264 elements in B
Phase 2	1 8 1 iterations 2.45 minutes	Phase 2	227 iterations 7.33 minutes
Reinversion at optimum	0.12 minutes 12401 elements in L	Reinversion at itn 731	0.56 minutes 39765 eta elements before invert 23013 eta elements after invert 3533 elements in B
Total	3 9 3 iterations 4.12 minutes	Total	7 6 5 iterations 15.73 minutes

Table 3. Staircase model (problem C) with reduced no. of variables. Comparison of Cholesky factors with Product Form of Inverse.

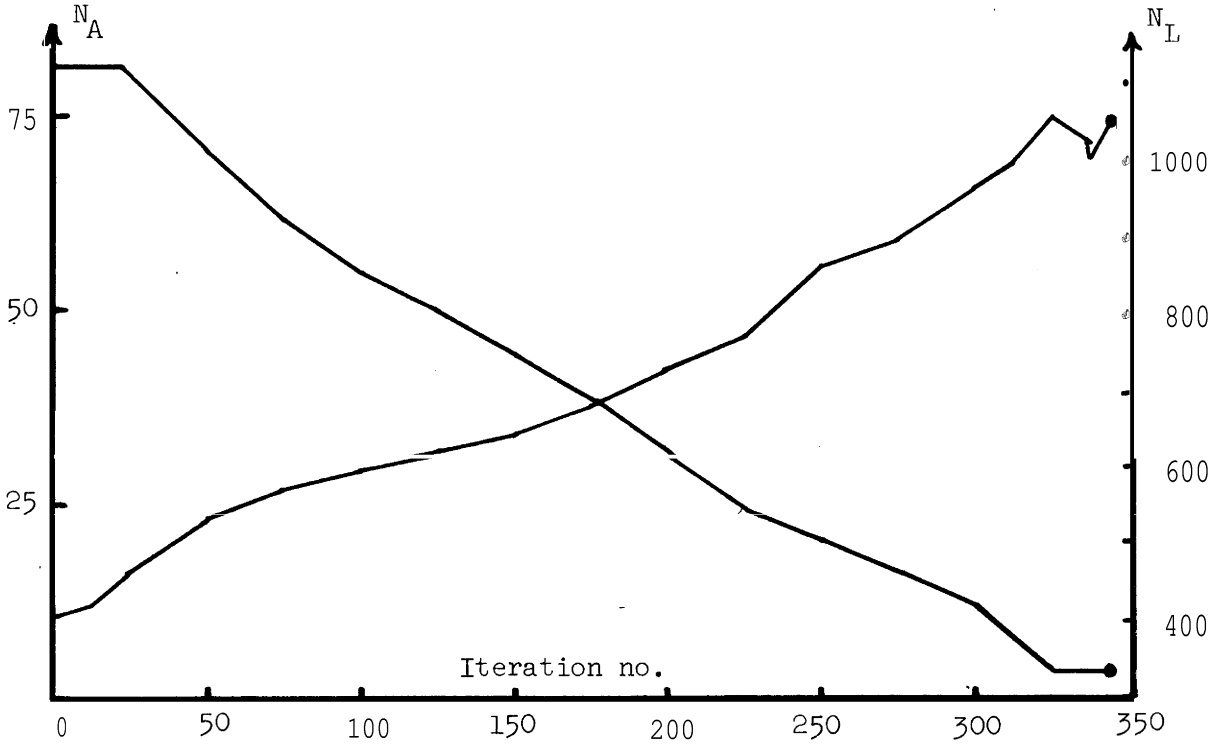


Figure 1. Problem A: growth of non-zeros in L with elimination of unit vectors from basis.

N_A = No. of artificial variables
N_L = No. of off-diagonal elements in L

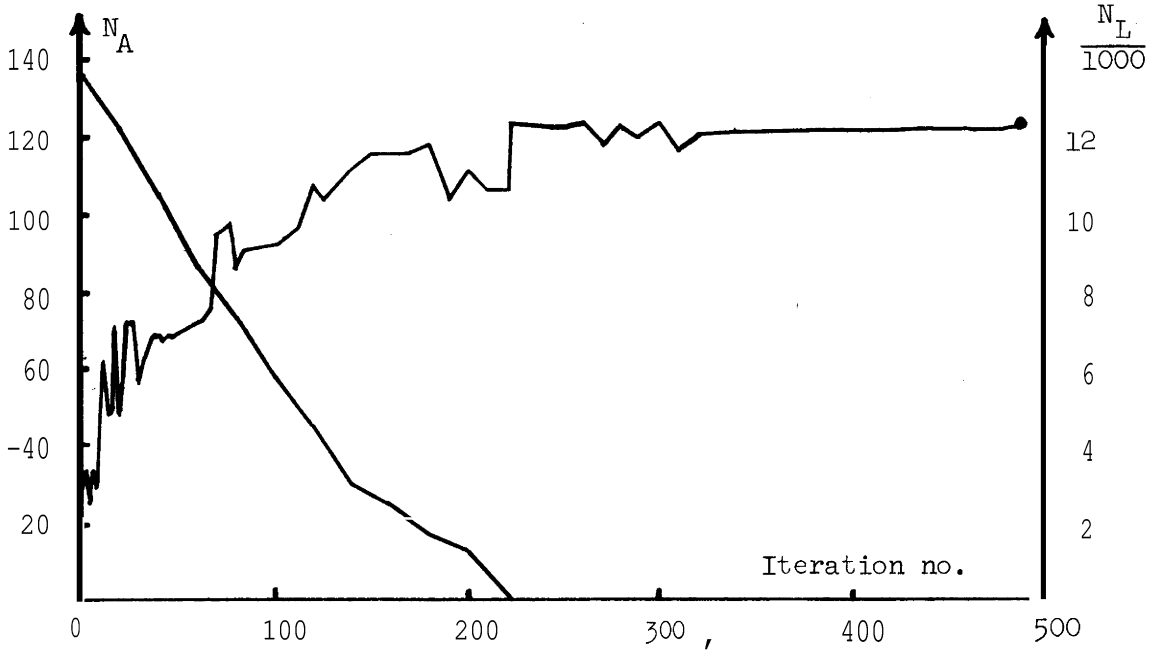


Figure 2. Problem C: steady-state density of L.

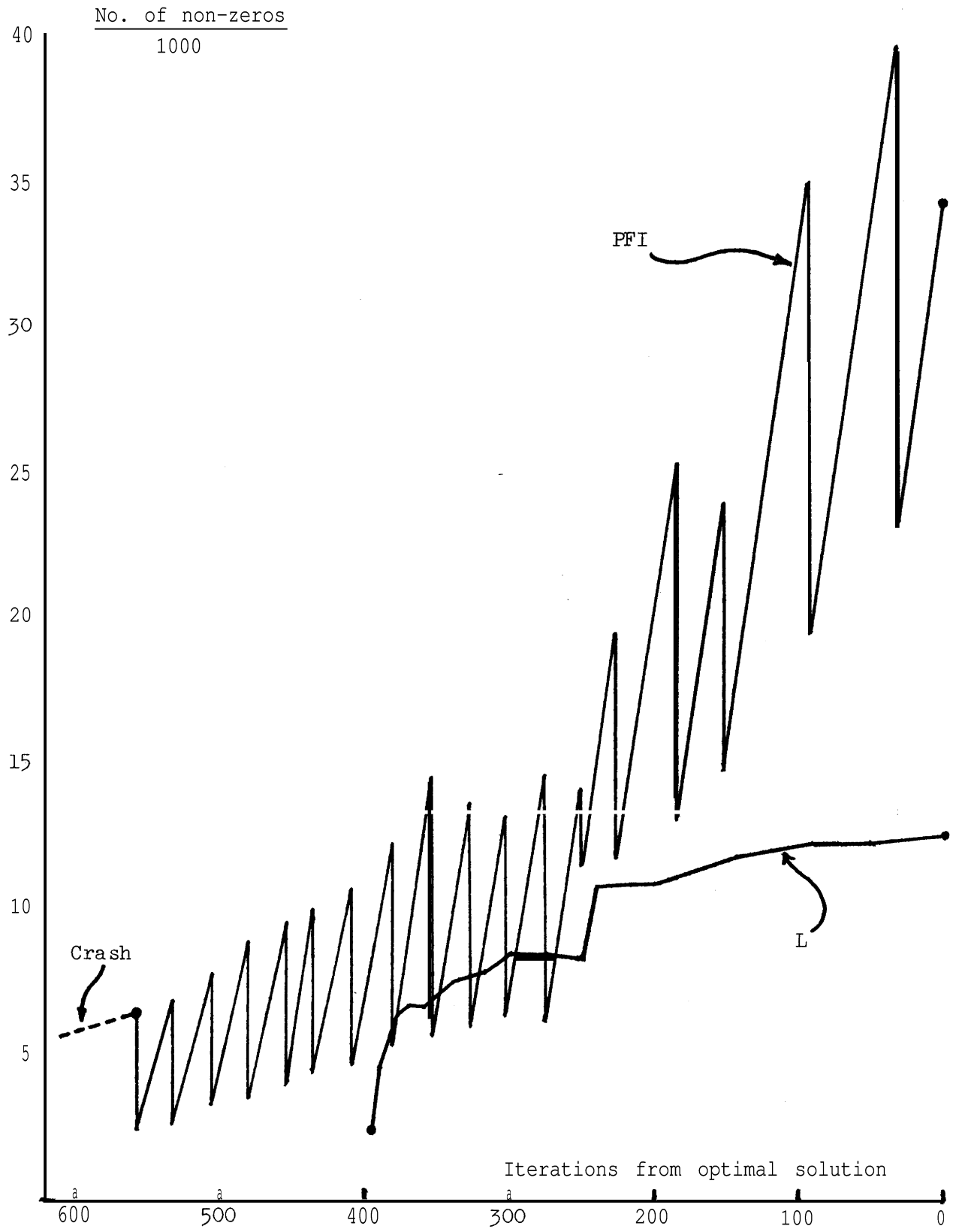


Figure 3. Staircase model (problem C, modified).
Growth of non-zeros in Product Form of Inverse
and in Cholesky factor.

14. Summary and suggestions for out-of-core implementation

Given a basis B with corresponding Cholesky factor L and basic solution $\hat{x}$, the main steps to be performed each iteration of the algorithm are as follows (using notation defined in previous sections).

1. Compute the Lagrange multipliers from

$$L^T\pi = d$$

2. Use π to select a column a_s for entry into the basis, and compute the rate of change of $\hat{x}$ from

$$LL^Tu = a_s \quad , \quad y = B^Tu .$$

3. Use y to select an out-going column a_r and update $\hat{x}$.

4. Solve

$$Lp = a_r$$

in preparation for modifying L.

5. Modify L in two stages, such that

(a) $\hat{L}\hat{L}^T \leftarrow LL^T - a_r a_r^T$ (using P)

(b) $\bar{L}\bar{L}^T \leftarrow \hat{L}\hat{L}^T + a_s a_s^T$.

It can be seen that in steps 1 through 4 above, access to L is sequential (column by column) and is alternately backwards and forwards. Thus for these calculations a disk file (for column-wise storage of L) would be as convenient as in standard product-form systems.

Modification of L (step 5) again requires a backward pass and a forward pass, but the main difficulty is that elements must be inserted into L. A possible solution to this problem is as follows. Storage on disk will be in the form of a sequence of fixed length records, each large enough to hold about 20 elements of L. Now

consider the modification of a particular column which has been allocated r records on disk (containing d = 20r "disk-elements," say). In general this column will have a further c "core-elements", which are held in main memory in linked-list form, as described earlier in this paper. During modification, the d disk-elements will be read into core and linked into the appropriate part of the list, giving d+c elements in-core for the column in question. The modification can be performed conveniently within the list structure, and the first d (modified) elements will then be written out to disk in place of their predecessors and deleted from the list, leaving some small number of core-elements behind.

In this way the total core required by the lists for all m columns of L should change relatively little at each iteration. During early iterations while L is filling in, periodical re-writes can be performed (e.g. during reinversion) in order to allocate additional disk records to the densest columns. Storage requirements should stabilize after 100 or 200 iterations.

Note that for small problems we would initially set r = 0 for all columns and operation would be completely in-core. Transition to disk would be smoothly accomplished, if necessary, by increasing r for the densest columns.

Note also that unless a problem is very dense, only a small percentage of the columns of L are affected by a basis change. This is why fixed length records are specified, so that "seek" operations can be requested in order to skip past columns' on disk which are not to be modified. Drums or fixed-head disks would alleviate this problem to some extent.

Storage of the constraint matrix A remains to be discussed. The recent work of J. E. Kalan on the concept of super-sparseness (see [15]) indicates that even for extremely large problems, in-core storage of A is within the realms of practicality. However we cannot imbed any part of L within A , in the way that Kalan advocates imbedding the product form of B^{-1} , and as Tomlin points out in [21], relying on the extended-core storage of current large machines "can only be a postponement at best."

Fortunately the primal simplex algorithm does not require a scan of all columns of A each iteration, so if A has many more columns than rows the simplest solution is to perform a sequence of suboptimizations. At each stage the current basis B and as many non-basic columns as possible are retained in core. (B is required in step 2 above, and a random column from B is needed in step 4.) After a number of iterations, a pass through A can select the current basis and a new set of columns for a further suboptimization.

Although standard systems do not retain B in-core, there are some advantages in doing so, in particular for checking of row and column residuals and for reinversion whenever necessary. We assume that Kalan's super-sparseness techniques for compacting B should make this practical.

15. Conclusion

In presenting a new linear programming algorithm we do not claim to be able to solve all problems efficiently. Instead we hope to have demonstrated that for certain well-defined classes of problem the method does have some useful advantages, in terms of both numerical stability and preservation of sparseness.

The problems to which the method is immediately applicable are those for which a preliminary ordering of the rows of A can be guaranteed to give a sparse factorization for every basis B arising in the simplex method. The uniqueness of the Cholesky factor L with respect to column permutations on B then makes it profitable to store and update the non-zero elements of L explicitly rather than in product form.

In the case of GUB , block-angular and staircase problems it is clear what the row-ordering of A should be, and the method then takes advantage of the structure without further overhead (e.g. problems B,C). For unstructured problems, triangularization of A appears to be sufficient if the density is low enough (e.g. problem A). However, unless there is structure to be preserved there seems to be a threshold density (at about 0.5%) above which the Cholesky factors fill in significantly, even when triangularization of A is carried out (e.g. problem D). In such cases, a structure-finding algorithm such as-that of Weil and Kettler appears to be necessary.

An interesting unsolved problem has arisen:

For which permutation P does the factorization

$$PBB^TP^T = LL^T$$

give an L which is most sparse?

If this question can be answered for square B (and possibly then for rectangular B) the algorithm in this paper may find broader application. In the meantime, the method is already applicable to many problems and it is felt that the unusual properties of the Cholesky factorization deserve further investigation.

Acknowledgements

There are many people to whom I am deeply grateful for their help during this work. In the first place I wish to thank Professor Gene Golub for his generous devotion of time to discussions, for his advice on the numerical problems involved and for many suggestions toward improving the presentation. I also wish to thank Professor George Dantzig especially for his valuable time and advice and for his much-needed encouragement throughout. Professor George Forsythe, too, was a patient listener during the early stages.

I welcome this chance to thank Philip Gill and Dr. Walter Murray for their advice and hospitality during a visit to the National Physical Laboratory where I first learned of their new approach to linear programming. I wish to thank them also for much subsequent discussion of updating methods and related points in question.

Among my colleagues at Stanford University I wish to express my thanks particularly to Richard Underwood for his constantly willing ear and sound advice on every aspect of the subject, and to Alan George for his help in many ways. Thanks are also due to John Tomlin, Philip Wolfe, Professor Evan Porteus, Professor Michael Osborne, Michael McGrath, Michael Malcolm, Richard Brent and Stanley Eisentat for many useful comments.

The people who have afforded their time in providing the test problems are almost too numerous to list. I wish to thank Robert Meyer of the Shell Development Company for supplying problem A; Michael Kasatkin and Craig Chioino of the Crown Zellerbach Corporation for providing problem B; Professor Alan Manne of the Department of Operations Research and Steve Kohlhagen of the Department of Economics, both at Stanford University, for providing problem C and making the comparative test runs; and John Tomlin of Scientific Control Systems Ltd. and his associates at British Petroleum (London) for providing problem D. With regard to problems not used here I am grateful to Bruce Murtagh of the Fluor Corporation for obtaining a smaller but all-importantly authentic model, and conversely to the people who have provided extremely large models which serve to emphasize the astonishing proportions of linear programming problems that are apparently being solved in industry today on a relatively routine basis.

Thanks finally to Trina Enderlein for her willingness and expertise in typing the manuscript.

References

[1] Bartels, R. H., "A Stabilization of the Simplex Method", Numerische Mathematilc 16, pp. 414-434, 1971.

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

[3] Dantzig, G. B., "Compact basis triangularization for the simplex method," from Recent advances in mathematical programming, edited by R. Graves and P. Wolfe, McGraw-Hill, 1963.

[4] Dantzig, G. B., Linear programming and extensions, Princeton University Press, 1963.

[5] Dantzig, G. B. and R. M. Van Slyke, "Generalized upper bounded techniques for linear programming," Journal of Computer and System Sciences, Vol. 1, No. 3, pp. 213-216, October 1967.

[6] Forrest, J. J. H. and J. A. Tomlin, "Updating triangular factors of the basis to maintain sparsity in the product form simplex method," presented at NATO conference, "Applications of optimization methods for large-scale resource-allocation problems," Elsinore, Denmark, July 5 -9, 1971.

[7] George, J. A., "Computer implementation of the finite element method," Technical report No. cs 208, Computer Science Department, Stanford University, 1971.

[8] Gill, P. E. and W. Murray, "A numerically stable form of the simplex algorithm," Technical report No. Maths 87, National Physical. Laboratory, Teddington, 1970.

[9] Gill, P. E. and W. Murray, "Quasi-Newton methods for unconstrained optimization,!' Technical report No. Maths 97, National Physical Laboratory, Teddington, 1971.

[10] Gill, P. E., G. H. Golub, W. Murray and M. A. Saunders, "Methods for modifying matrix factorizations," to be published.

[11] Golub, G. H., "Numerical methods for solving linear least squares problems," Numerische Mathematik 7, pp. 206-216, 1965.

[12] Golub, G. H. and J. H. Wilkinson, "Note on the iterative refinement of least squares solution," Numerische Mathematik 9, pp. 139-148, 1966.

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

[14] Hellerman, E. and D. Rarick, "The partitioned preassigned pivot procedure (P^4)," presented at the Symposium on Sparse Matrices and their Applications, IBM Thomas J. Watson Research Center, Yorktown, New York, September 9-10, 1971.

[15] Kalan, J. E., "Aspects of large-scale in-core linear programming," presented at ACM Annual Conference, Chicago, Illinois, August 3-5, 1971.

[16] Konno, H., "Bilinear programming: Part I. Algorithm for solving bilinear programs," Technical report No. 71-9, Dept. of Operations Research, Stanford University, August 1971.

[17] Konno, H., "Bilinear programming: Part II. Application of bilinear programming," Technical report No. 71-10, Dept. of Operations Research, Stanford University, August 1971.

[18] MPS/360, Linear and separable programming user's manual, IBM publication HZO-0476.

[19] Murray, W., "An algorithm for indefinite quadratic programming," Technical report No. DNAC 1, National Physical Laboratory, Teddington, June 1971.

[20] Orchard-Hays, W., Advanced linear-programming computing techniques, McGraw-Hill, 1968.

[21] Tomlin, J. A., "Modifying triangular factors of the basis in the simplex method," presented at the Symposium on Sparse Matrices and their Applications, IBM Thomas J. Watson Research Center, Yorktown, New York, September 9-10, 1971.

[22] Van der Sluis, A., "Condition, equilibration and pivoting in linear algebraic systems," Numerische Mathematik 15, pp. 74-86, 1970.

[23] Weil, R. L. and P. C. Kettler, "Rearranging matrices to block-angular form for decomposition (and other) algorithms," Management Science 18, no. 1, pp. 98-108, 1971.

[24] Wilkinson, J. H., The algebraic eigenvalue problem, Oxford University Press, 1965.