# Application of Dense Householder Transformations to a Sparse Matrix

LINDA KAUFMAN
Bell Laboratories

Householder transformations are a type of orthogonal transformation used to zero elements of a vector in many least squares and eigenvalue computations. This paper is concerned with the application of a given set of Householder transformations to a sparse matrix $X$. Normally, the application of the first transformation ruins the zero structure of $X$. Despite this fact it is shown that the transformed matrix has some algebraic structure which might be exploited. Algorithms are given which exploit this structure during computation of the orthogonal decomposition of a matrix, the null space of a matrix, and the singular value decomposition of a matrix.

Key Words and Phrases: Householder transformations, sparse matrices, singular value decomposition
CR Categories: 5.14

## 1. INTRODUCTION

Many algorithms in numerical linear algebra use orthogonal transformations to zero elements of a vector. One such transformation is the Householder transformation

$$P = I + \beta \mathbf{u}\mathbf{u}^{\mathrm{T}}, \tag{1.1}$$

where $\beta = -2/\mathbf{u}^{\mathrm{T}}\mathbf{u}$ and $\mathbf{u}$ is chosen so that for a given vector $\mathbf{a}$, certain elements of $P\mathbf{a}$ are zero. Householder transformations are used to solve least squares problems [3] and in some eigenvalue calculations. They are used because they are easy to store (only $\mathbf{u}$ need be saved) and easy to apply: For a given vector $\mathbf{x}$, $P\mathbf{x} = \mathbf{x} + \beta(\mathbf{u}^{\mathrm{T}}\mathbf{x})\mathbf{u}$, so that if $\mathbf{u}$ has $m$ nonzero elements, forming $P\mathbf{x}$ requires $2m + 1$ multiplications and $2m - 1$ additions.

In some problems it is necessary to apply a sequence of $n$ given Householder transformations with dense $u$'s to a sparse $m \times k$ matrix $X$. For example, in Section 3 we describe one case in which the columns of $X$ are columns of the identity matrix and another case in which most of the rows of $X$ are zero. Unfortunately, the application of the first transformation ruins the sparsity of $X$, and therefore its zero structure is usually ignored. However, the transformed matrix still has some structure which may be exploited whenever $k \geq n/2$. We


Author's address Bell Laboratories, Murray Hill, NJ 07974.

give two applications in which the operation counts and storage requirements are decreased if this structure is used.

## 2. USING HOUSEHOLDER TRANSFORMATIONS

Let

$$P^{(i)} = I + \beta_i \mathbf{u}^{(i)} \mathbf{u}^{(i)^T}, \qquad i = 1, 2, \ldots, n$$

be $n$ given Householder transformations. Let $X$ be a given sparse $m \times k$ matrix with columns $\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_k$. Assume that the matrix $Y = P^{(n)} P^{(n-1)} \cdots P^{(1)} X$ is required. In this section we describe two algorithms for computing $Y$. The first is the traditional method; the second is an algebraically equivalent method which takes advantage of the sparseness of $X$.

Normally $Y$ is determined in $n$ stages: Let

$$X^{(1)} = X,$$
$$X^{(i+1)} = P^{(i)} X^{(i)} \quad \text{for} \quad i = 1, 2, \ldots, n.$$

Then $Y = X^{(n+1)}$. This is equivalent to

### ALGORITHM A

For $i = 1, 2, \ldots, n$
  For $j = 1, 2, \ldots, k$
    set $c_{ij} = \beta_i \mathbf{u}^{(i)^T} \mathbf{x}_j^{(i)}$ (2.1)
    and $\mathbf{x}_j^{(i+1)} = \mathbf{x}_j^{(i)} + c_{ij} \mathbf{u}^{(i)}$. (2.2)

Since dense $u^{(1)}$ causes $X^{(2)}$ to be dense even when $X$ is sparse, it is difficult to exploit with Algorithm A any zero structure that $X$ possesses. However, from (2.2) we see that

$$\mathbf{x}_j^{(i+1)} = \mathbf{x}_j + \sum_{q=1}^{i} c_{qj} \mathbf{u}^{(q)}. \tag{2.3}$$

This means that the $c_{ij}$ in (2.1) can be computed using the formula

$$c_{ij} = \beta_i \left( \mathbf{u}^{(i)^T} \mathbf{x}_j + \sum_{q=1}^{i-1} c_{qj} (\mathbf{u}^{(i)^T} \mathbf{u}^{(q)}) \right). \tag{2.4}$$

Equation (2.4) of course requires much more work than (2.1) if $X$ is dense, but if $X$ is so sparse that the cost of the computation of $\mathbf{u}^{(i)^T} \mathbf{x}_j$ can be ignored, eq. (2.4) can lead to a more efficient method. It is the basis of Algorithm B below for computing the columns $\mathbf{y}_j$ of $Y$.

### ALGORITHM B

For $i = 1, 2, \ldots, n$
  For $q = 1, 2, \ldots, i - 1$
    $d_q = \mathbf{u}^{(i)^T} \mathbf{u}^{(q)}$. (2.5)

For $j = 1, 2, \ldots, k$

$$\text{set } c_{ij} = \beta_i\left(\mathbf{u}^{(i)^1}\mathbf{x}_j + \sum_{q=1}^{i-1} c_{qj}d_q\right). \tag{2.6}$$

For $j = 1, 2, \ldots, k$

$$\mathbf{y}_j = \mathbf{x}_j + \sum_{i=1}^{n} c_{ij}\mathbf{u}^{(i)}. \tag{2.7}$$

If $X$ is sparse enough that the cost of computing $\mathbf{u}^{(i)^T}\mathbf{x}_j$ is negligible, the cost of computing the $c$'s in (2.6) is essentially the $n^2/2 - n/2$ inner products required to form the $d$'s in (2.5). In comparison $kn$ inner products are needed to compute the $c$'s in Algorithm A. Thus, if $k > n/2$, Algorithm B should be more efficient. Of course, Algorithm B requires the storage of the $c$'s, while Algorithm A does not, but in many applications this space is available. Since all the $\mathbf{u}$'s are known beforehand and since there is no need to compute the $\mathbf{x}_j^{(i)}$'s of (2.3), step (2.7) of Algorithm B can be done in one shot.

## 3. APPLICATIONS

In the applications given below we count the number of operations. Since each multiplication is usually coupled with an addition, an operation is defined as one addition and one multiplication.

*Application* 1: *Computation of Q from the QR Decomposition.* Any $m \times s$ $(m \geq s)$ matrix $A$ of rank $n$ can be written as

$$A = Q\begin{bmatrix} R & S \\ 0 & 0 \end{bmatrix}T,$$

where $Q$ is an $m \times m$ orthogonal matrix, $R$ is an $n \times n$ upper triangular matrix, and $T$ is a permutation matrix. This decomposition is particularly useful for solving least squares problems [3]. It is often computed by applying a sequence of Householder transformations $P^{(n)}, \ldots, P^{(1)}$ to $A$. Thus

$$Q = P^{(n)} \cdots P^{(2)}P^{(1)}. \tag{3.1}$$

Rarely is $Q$ formed explicitly, but in some linearly constrained minimization algorithms [2] it is useful to construct an explicit representation by applying the Householder transformations to the identity matrix.

The matrix $Q$ can be partitioned as

$$Q = [\underset{n}{Q_1} \mid \underset{m-n}{Q_2}],$$

where $Q_1$ is an orthonormal basis for $A$ and $Q_2$ forms a basis for the null space of $A$. In least squares only $Q_1$ may be required while in constrained minimization only $Q_2$ may be required. In the first case $X$ is the first $n$ columns of the $m \times m$ identity matrix while in the second case $X$ is the last $m - n$ columns of the $m \times m$ identity matrix.

The operation counts for computing $Q_1$ only, $Q_2$ only, and all of $Q$ are given in Table I. The operation counts consider the fact that the first $(n - i)$ elements of $\mathbf{u}^{(i)}$ are zero, which has the following influence on the computation:

(1) From (2.4), $c_{ij} = 0$ for $i \leq n - j$ for $Q_1$ and $Q$.

Table I. Operation Counts for Application 1: $r = (m - n)/n$

| | Algorithm A | Algorithm B |
|---|---|---|
| Calculation of $Q_1$ only, i.e $X = \begin{bmatrix} I_n \\ 0 \end{bmatrix}$ | | |
| Calculating $c$'s | $\sum_{i=1}^{n} (m - n + i)(i - 1) + \sum_{i=1}^{n} i$ | $\sum_{i=1}^{n} (m - n + i)(n - i) + \sum_{i=1}^{n} \sum_{j=1}^{i} j$ |
| Rest of calculation | $\sum_{i=1}^{n} (m - n + i)i$ | $\sum_{i=1}^{n} (m - n + i)i$ |
| Total | $(m - n)n^2 + \frac{2}{3}n^3 + n^2 + n/3$ $= n^3(r + \frac{2}{3}) + n^2 + n/3$ | $(m - n)n^2 + \frac{2}{3}n^3 + n^2 + n/3$ $= n^3(r + \frac{2}{3}) + n^2 + n/3$ |
| Calculation of $Q_2$ only, i.e $X = \begin{bmatrix} 0 \\ \cdots \\ I_{m-n} \end{bmatrix}$ | | |
| Calculating $c$'s | $\sum_{i=1}^{n} (m - n + i)(m - n) + (m - n)n$ | $\sum_{i=1}^{n} (m - n + i)(n - i) + (m - n)(n^2 + n)/2$ |
| Rest of calculation | $\sum_{i=1}^{n} (m - n + i)(m - n)$ | $\sum_{i=1}^{n} (m - n + i)(m - n)$ |
| Total | $2(m - n)^2n + (m - n)(n^2 + 2n)$ $= n^3(2r^2 + r) + 2rn^2$ | $(m - n)^2n + \frac{3}{2}(m - n)(n^2 + n) + n^3/6 - n/6$ $= n^3(r^2 + \frac{1}{2}r + \frac{1}{6}) + \frac{3}{2}rn^2 - n/6$ |
| Calculation of all of $Q$, i.e. $X = I_m$ | | |
| Calculating $c$'s | $\sum_{i=1}^{n} (m - n + i)^2$ | $\sum_{i=1}^{n} (m - n + i)(n - i) + (m - n)(n^2 + n)/2 + \sum_{i=1}^{n} \sum_{j=1}^{i} j$ |
| Rest of calculation | $\sum_{i=1}^{n} (m - n + i)^2$ | $\sum_{i=1}^{n} (m - n + i)^2$ |
| Total | $2(m - n)^2n + (m - n)(2n^2 + 2n) + \frac{2}{3}n^3 + n^2 + n/3$ $= n^3(2r^2 + 2r + \frac{2}{3}) + n^2(2r + 1) + n/3$ | $(m - n)^2n + (m - n)(2n^2 + n) + \frac{2}{3}n^3 + n^2 + n/3$ $= n^3(r^2 + 2r + \frac{2}{3}) + (r + 1)n^2 + n/3$ |

(2) The cost of computing the nonzero inner products in (2.1) for the $i$th iteration is $(m - n + i)(i - 1)$ for $Q_1$, $(m - n + i)(m - n)$ for $Q_2$, and $(m - n + i)(m - n + i - 1)$ for all of $Q$. Note that for $Q_1$ and $Q$ we count no cost for computing $\mathbf{u}^{(i)^T}\mathbf{x}_j^{(i)}$ for $j \leq n + 1 - i$ because by (2.3), $\mathbf{x}_j^{(i)} = \mathbf{e}_j$ for $j \leq n + 1 - i$, where $\mathbf{e}_j$ is the $j$th column of the identity matrix.

(3) The cost of (2.2) at the $i$th iteration is $(m - n + i)i$ for $Q_1$, $(m - n + i)(m - n)$ for $Q_2$, and $(m - n + i)(m - n + i)$ for all of $Q$.

(4) Algorithm B for computing $Q$ becomes simply

For $i = 1, 2, \ldots, n$
  For $q = 1, 2, \ldots, i - 1$
    $d_q = \mathbf{u}^{(i)^T}\mathbf{u}^{(q)}$. (3.2)

For $j = n - i + 1, \ldots, n$

$$c_{ij} = \beta_i \left( u_j^{(i)} + \sum_{q=n-j+1}^{i-1} c_{qj} d_q \right). \tag{3.3}$$

For $j = n + 1, \ldots, m$

$$c_{ij} = \beta_i \left( u_j^{(i)} + \sum_{q=1}^{i-1} c_{qj} d_q \right). \tag{3.4}$$

For $j = 1, 2, \ldots, n$

$$\mathbf{q}_j = \mathbf{e}_j + \sum_{i=n-j+1}^{n} c_{ij} \mathbf{u}^{(i)}. \tag{3.5}$$

For $j = n + 1, \ldots, m$

$$\mathbf{q}_j = \mathbf{e}_j + \sum_{i=1}^{n} c_{ij} \mathbf{u}^{(i)} \tag{3.6}$$

When computing $Q_1$ only or $Q_2$ only, all the $d$'s in (3.2) still have to be computed at a cost of $\sum_{i=1}^{n} (m - n + i)(n - i)$ operations. If only $Q_1$ is required, steps (3.4) and (3.6) should be eliminated and the steps at (3.3) and (3.5) should be executed at a cost of $\sum_{i=1}^{n} \sum_{j=1}^{i} j$ and $\sum_{i=1}^{n} (m - n + i)i$ operations, respectively. If only $Q_2$ is required, steps (3.3) and (3.5) should be eliminated and steps (3.4) and (3.6) should be executed at a cost of $(m - n) \sum_{i=1}^{n} i$ and $(m - n) \sum_{i=1}^{n} (m - n + i)$ operations, respectively.

In Table I, $r$ is the ratio $(m - n)/n$. Certainly for sufficiently large $n$, no case based on operation counts can be made to prefer either algorithm for computing $Q_1$. When $m = 2n$, i.e. $r = 1$, and $n$ is sufficiently large, Table I indicates that for the calculation of $Q_2$ the time for Algorithm B is about 89 percent that for Algorithm A, while for $Q$ the time for Algorithm B is about 79 percent that for Algorithm A. As the ratio of $m$ to $n$ increases, the term with $r^2$ in each of the total operation counts in the table begins to dominate and the ratio of the operation counts of Algorithm B to Algorithm A approaches $\frac{1}{2}$ for both matrices.

If in Algorithm B in (3.5) and (3.6) rows $n + 1$ through $m$ of $Q$ are computed first followed by the computation of row $n$, row $n - 1$, etc., then the $c$'s may be safely stored temporarily in the first $n$ rows of the $Q$ matrix. Thus no extra storage is needed for the $C$ matrix.

Figure 1 contains a graph of the theoretical ratio of the computation times of the two algorithms for computing $Q$ and the actual ratio as obtained using an optimizing Fortran compiler on the Honeywell 6000 computer at Bell Laboratories. The timer on that machine has about a 1 percent accuracy. Since the theoretical curves for $n = 5$, $n = 10$, and $n = 15$ were nearly indistinguishable, only the theoretical curve for $n = 15$ is printed. As the graph indicates, one can expect the actual curves to approach the theoretical curves as $n$ increases. Of course timing results are compiler dependent, and indeed with one nonoptimizing compiler on the Honeywell the computed curve for $n = 15$ lay below the theoretical curve.

To study the numerical properties of algorithm B, $Q$ matrices generated by methods A and B were compared. Random $\mathbf{u}$ vectors, whose elements were in the range (0, 1), were generated in single precision for various values of $m$ and $n$. The $Q$ matrix was then computed in single precision by methods A and B and is denoted by $Q_A$ and $Q_B$, respectively. The matrix $Q_D$, the $Q$ matrix calculated by

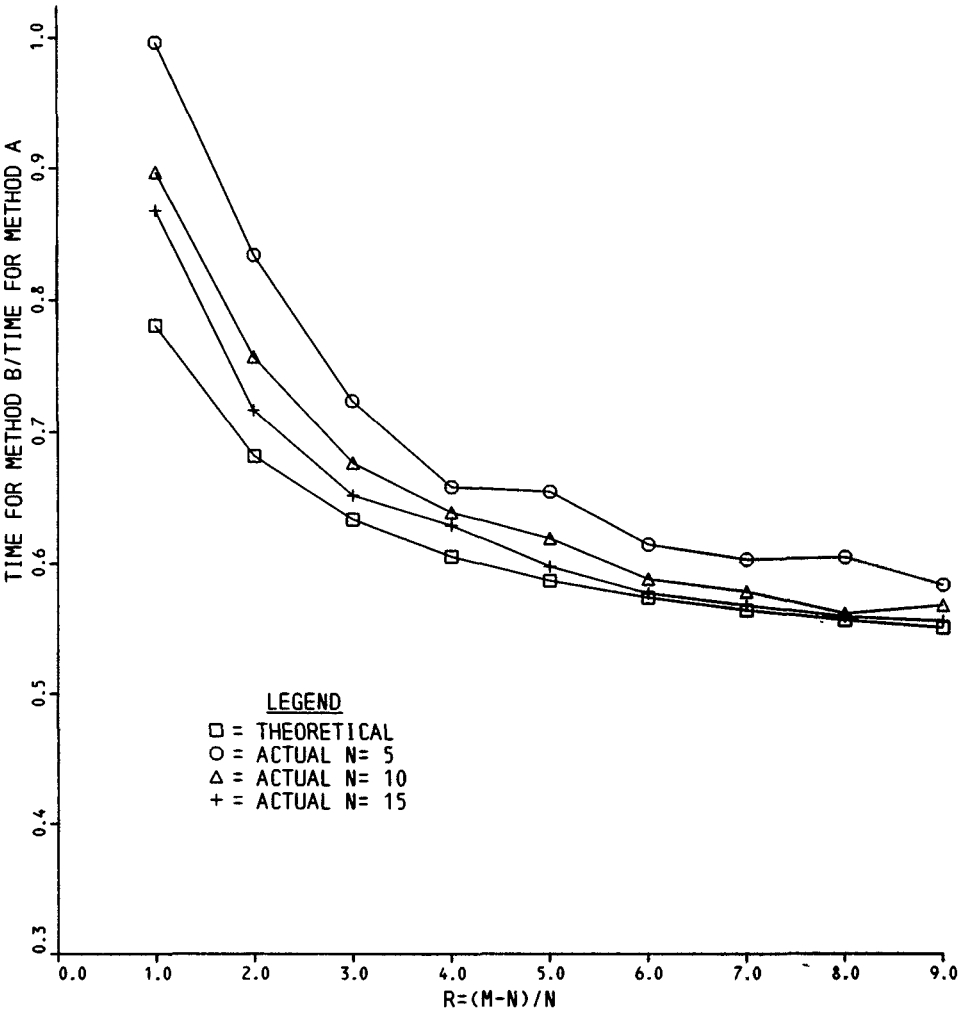


Fig. 1. Computation times for $Q$

method B in double precision, was also generated. Table II, which gives $\|Q_A - Q_D\|_\infty$ and $\|Q_B - Q_D\|_\infty$ for various values of $m$ and $n$, indicates that neither algorithm can be said to be consistently more stable. All calculations were performed using a Fortran program on the Interdata 8/32 running with a UNIX operating system. The machine precision of the Interdata is $1.0 \times 10^{-6}$.

*Application* 2: *The Singular Value Decomposition.* The singular value decomposition of a matrix can be used to solve least squares problems [3] and to determine the condition of a matrix.

To determine the singular value decomposition (SVD) of an $m \times n$ matrix $A$ when $m \gg n$, Chan [1] suggests first applying Householder transformations

Table II. Comparison Between Methods A and B

| $m$ | Error in $Q_A$ | Error in $Q_B$ | $m$ | Error in $Q_A$ | Error in $Q_B$ |
|---|---|---|---|---|---|
| | | $n = 5$ | | | |
| 10 | 2.9807e−6 | 7.9261e−6 | 15 | 7.4025e−6 | 8.7826e−6 |
| 20 | 1.2268e−5 | 1.1215e−5 | 25 | 1.4802e−5 | 9.8649e−6 |
| 30 | 1.6930e−5 | 1.2713e−5 | 35 | 1.8286e−5 | 1.2596e−5 |
| 40 | 2.0306e−5 | 1.2320e−5 | 45 | 2.6124e−5 | 1.2040e−5 |
| 50 | 3.2037e−5 | 3.9735e−5 | 55 | 2.9788e−5 | 3.0797e−5 |
| | | $n = 10$ | | | |
| 20 | 1.2657e−5 | 1.3698e−5 | 30 | 1.9773e−5 | 1.2366e−5 |
| 40 | 2.4471e−5 | 1.3471e−5 | 50 | 2.4471e−5 | 1.4633e−5 |
| 60 | 4.5954e−5 | 5.0927e−5 | 60 | 2.9015e−5 | 6.9455e−5 |
| 80 | 5.3387e−5 | 9.7508e−5 | 90 | 5.8668e−5 | 9.4817e−5 |
| 100 | 6.3092e−5 | 1.0715e−4 | | | |
| | | $n = 15$ | | | |
| 15 | 1.7971e−5 | 2.1452e−5 | 45 | 2.8490e−5 | 2.5765e−5 |
| 60 | 4.0572e−5 | 3.3337e−5 | 75 | 5.2284e−5 | 1.0718e−4 |
| 90 | 6.9425e−5 | 1.0410e−4 | | | |

$P^{(n)}, \ldots, P^{(1)}$ to $A$ to reduce it to an upper triangular matrix $R$ so that

$$A = P^{(n)} \cdots P^{(1)} \begin{bmatrix} R \\ 0 \end{bmatrix}$$

and then determining the singular value decomposition of $R$. If the SVD of $R$ is given by

$$R = W\Sigma V^{\mathrm{T}}$$

where $W$ and $V$ are $n \times n$ unitary matrices and $\Sigma$ is an $n \times n$ diagonal matrix, then the SVD of $A$ is given by

$$A = U\Sigma V^{\mathrm{T}}$$

with

$$U = P^{(n)} \cdots P^{(1)} \begin{bmatrix} W \\ 0 \end{bmatrix}_{m-n}^{n} \tag{3.7}$$

Once the $P$'s and $W$ have been computed, Chan mentions two methods for computing $U$. The first method applies the Householder transformations to $\left[\begin{smallmatrix} W \\ 0 \end{smallmatrix}\right]$, fills it up with nonzeros, and requires $2mn^2 - n^3$ operations and $2mn$ storage locations. The second method applies the Householder transformations to $\left[\begin{smallmatrix} I_n \\ 0 \end{smallmatrix}\right]$, postmultiplies the result by $W$, and requires $2mn^2 - n^3/3 - n^2/2 + (5/6)n$ operations and $mn + n^2$ storage locations. The original motive of the present paper was to find an algorithm with the favorable storage requirements of the second scheme and no more operations than the first. An algorithm was found which meets the storage requirement and decreases the operation count to $\frac{3}{2}mn^2$ operations. It partitions $U$ of (3.7) into

$$U = \begin{bmatrix} U_U \\ \cdots \\ U_L \end{bmatrix}_{m-n}^{n}$$

and uses a combination of Algorithm A and Algorithm B to find $U_U$ and eq. (2.3) to find $U_L$.

Let $X^{(1)} = \begin{bmatrix} W \\ 0 \end{bmatrix}_{m-n}^{n}$ and $X^{(\iota+1)} = P^{(\iota)}X^{(\iota)}$ where $P^{(\iota)} = I + \beta_\iota \mathbf{u}^{(\iota)}\mathbf{u}^{(\iota)^{\mathrm{T}}}$. Partition $\mathbf{u}^{(\iota)}$ as

$$\mathbf{u}^{(\iota)} = \begin{bmatrix} \mathbf{f}^{(\iota)} \\ \mathbf{g}^{(\iota)} \end{bmatrix}_{m-n}^{n} \tag{3.8}$$

and $X^{(\iota)}$ as

$$X^{(\iota)} = \begin{bmatrix} W^{(\iota)} \\ Z^{(\iota)} \end{bmatrix}_{m-n}^{n}$$

Hence

$$U_U = W^{(n+1)} \quad \text{and} \quad U_L = Z^{(n+1)}.$$

From (2.1) we find that

$$c_{\iota j} = \beta_\iota(\mathbf{f}^{(\iota)^{\mathrm{T}}}\mathbf{w}_j^{(\iota)} + \mathbf{g}^{(\iota)^{\mathrm{T}}}\mathbf{z}_j^{(\iota)}).$$

But from (2.2) and (2.3), using the fact that $Z^{(1)} = 0$, we have

$$\mathbf{z}_j^{(\iota+1)} = \mathbf{z}_j^{(\iota)} + c_{\iota j}\mathbf{g}^{(\iota)} = \sum_{q=1}^{\iota} c_{qj}\mathbf{g}^{(q)}. \tag{3.9}$$

Hence

$$c_{\iota j} = \beta_\iota\left(\mathbf{f}^{(\iota)^{\mathrm{T}}}\mathbf{w}_j^{(\iota)} + \sum_{q=1}^{\iota-1} c_{\iota j}\mathbf{g}^{(\iota)^{\mathrm{T}}}\mathbf{g}^{(q)}\right),$$

and there is no need to store the $Z$ matrices while constructing the $c$'s.

We are thus led to the following algorithm for computing $U_U$:

ALGORITHM C

For $\iota = 1, 2, \ldots, n$
  For $q = 1, 2, \ldots, \iota - 1$
    set $s_q = \mathbf{g}^{(\iota)^{\mathrm{T}}}\mathbf{g}^{(q)}$.
  For $j = 1, 2, \ldots, n$
    set $c_{\iota j} = \beta_\iota\left(\mathbf{f}^{(\iota)^{\mathrm{T}}}\mathbf{w}_j^{(\iota)} + \sum_{q=1}^{\iota-1} s_q c_{qj}\right)$
    set $\mathbf{w}_j^{(\iota+1)} = \mathbf{w}_j^{(\iota)} + c_{\iota j}\mathbf{f}^{(\iota)}$.

The computation of $U_U$ requires $mn^2/2 + n^3 + m/2$ operations.

From (3.9) we see that

$$\mathbf{z}_j^{(n+1)} = \sum_{q=1}^{n} c_{qj}\mathbf{g}^{(q)}$$

so that

$$U_L = Z^{(n+1)} = GC,$$

where the columns of $G$ are the $\mathbf{g}^{(\iota)}$'s of (3.8). The computation of $U_L$ requires $mn^2 - n^3$ operations. Hence there is about a 25 percent decrease in the operation count when $m \gg n$ from Chan's methods.

Algorithm C can be implemented in several ways which require only $mn + n^2$ storage locations and no additional locations to store the $C$ matrix. In one such implementation the $W$ matrix would initially occupy the first $n$ rows of an $m \times n$ array and $G$ would occupy the last $m - n$ rows of that array. The lower triangular matrix $F$ whose columns are the $\mathbf{f}^{(i)}$'s of (3.8) would occupy a separate $n \times n$ array into which the rows of $C$ would be inserted as they are computed.

## 4. CONCLUSION

We have shown two applications in which Algorithm B gives a smaller operation count than Algorithm A. The idea behind the algorithm can also be used during least squares computations when there are many sparse right-hand sides and during the computation of the orthogonal decomposition of a sparse but structured matrix. Our examples indicate that sparse matrices which have suffered Householder transformations still have some useful algebraic structure.

## REFERENCES

1. CHAN, T.C. On computing the singular value decomposition. Stanford Computr. Sci. Rep. 588, Stanford U., Stanford, Calif., Feb. 1977
2. GILL, P E., AND MURRAY, W. *Numerical Methods for Constrained Optimization.* Academic Press, London, 1974.
3. LAWSON, C.L., AND HANSON, R.J. *Solving Least Squares Problems.* Prentice-Hall, Englewood Cliffs, N.J., 1974.

Received December 1977, revised January 1979