The WY Representation for Products of Householder Matrices

Christian Bischof
Charles Van Loan

TR 85-681
December 1985

Department of Computer Science
Cornell University
Ithaca, NY 14853

The WY Representation for Products of Householder Matrices

Christian Bischof
Department of Computer Science
Cornell University
Ithaca, New York 14853

Charles Van Loan
Department of Computer Science
Cornell University
Ithaca, New York 14853

Abstract

A new way to represent products of Householder matrices is given that makes a typical Householder matrix algorithm rich in matrix-matrix multiplication. This is very desireable in that matrix-matrix multiplication is the operation of choice for an increasing number of important high performance computers. We tested the new representation by using it to compute the QR factorization on the FPS-164/MAX. Preliminary results indicate that it is a very efficient way to organize Householder computations.

Computations associated with this research were performed on the Production Supercomputer Facility at Cornell University which is supported in part by the National Science Foundation and IBM Corporation. Additional funding was provided by ONR contract N00014-83-K-0640.

1. Introduction

During the past five years a great deal has been written about matrix computations in high performance "supercomputing" environments. For example, techniques have emerged for designing linear algebra codes that "squeeze" the most out of various vector pipeline architectures. The reader is referred to the excellent survey article by Dongarra, Gustavson, and Karp (1983). One theme in their work is how to take an algorithm such as Gaussian elimination and organize it in such a way that the resulting code is rich in matrix-vector multiplication. The regularity of that operation allows for maximum pipelining and minimum traffic to and from memory. The papers by Dongarra and Eisenstat (1984) and Dongarra, Kaufman, and Hammarling (1985) further dramatize this point.

It is now the case with some new architectures that high performance can best be achieved with algorithms that are rich in *matrix-matrix* multiplication. Such a requirement leads naturally to block algorithms. For methods such as Gaussian elimination and the Cholesky factorization the course of action is relatively straight-forward. Consider the LU factorization:

$$\begin{bmatrix} A_{11} & A_{12} & A_{13} \\ A_{21} & A_{22} & A_{23} \\ A_{31} & A_{32} & A_{33} \end{bmatrix} = \begin{bmatrix} L_{11} & 0 & 0 \\ L_{21} & L_{22} & 0 \\ L_{31} & L_{32} & L_{33} \end{bmatrix} \begin{bmatrix} U_{11} & U_{12} & U_{13} \\ 0 & U_{22} & U_{23} \\ 0 & 0 & U_{33} \end{bmatrix}$$

If we assume that the first block row of U and the first block column of L are known, then the blocks L_{22} , U_{22} , L_{32} , and U_{23} can be resolved from the equations

$$\begin{aligned} L_{22}U_{22} &= A_{22} - L_{21}U_{12} \\ L_{22}U_{23} &= A_{23} - L_{21}U_{13} \\ L_{32}U_{22} &= A_{32} - L_{31}U_{12} \end{aligned}$$

Note that the righthand sides are rich in matrix-matrix multiplication. If block dimensions are chosen properly then the overall method will be sufficiently rich in that operation to ensure high performance. This has been demonstrated by Steve Oslon (1985) of Floating Point Systems who has implemented the above scheme (with pivoting) on the FPS-164/MAX.

The FPS-164/MAX has an architecture that "likes" matrix-matrix multiplication and it is for this machine that we have developed a new algorithm for the QR factorization that performs extremely well. Although our technique has been tailored to the FPS-164/MAX, we suspect that our work will be of interest to users of other high performance computers such as the Cray-2 and the IBM 3090.

It should be noted that block procedures for matrix decompositions involving orthogonal transformations are not so straight-forward. Consider the QR factorization:

$$A = [A_1 , A_2 , A_3] = QR = \left[Q_1 , Q_2 , Q_3\right] \begin{bmatrix} R_{11} & R_{12} & R_{13} \\ 0 & R_{22} & R_{23} \\ 0 & 0 & R_{33} \end{bmatrix} .$$

If the first block column of Q and the first block row of R are known, then Q_2 and R_{22} can be obtained by computing the QR factorization

$$Q_2R_{22} = A_2 - Q_1R_{12}$$

whereupon $R_{23} = Q_2^TA_3$. These are essentially the key steps of a block version of modified Gram-Schmidt (MGS). Block MGS is certainly rich in matrix multiplication but the "Q" matrix may not be sufficiently orthonormal for certain applications. (See Golub and Van Loan (1983,pp.151-152) for a description of MGS and its properties.) It is an open question how to organize block MGS and what its numerical properties are. We shall not treat the matter further in this paper.

Our intention is to examine how the perfectly stable Householder QR factorization procedure can be written in a form that is rich in matrix multiplications. Householder transformations underpin the eigenvalue and least squares solvers in EISPACK and LINPACK. A discussion of the algorithms in these packages may be found in Golub and Van Loan (1983).

The key to obtaining block Householder algorithms that are rich in matrix multiplication turns out to be what we call the "WY representation". It amounts to writing products of Householder matrices like

$$Q_k = P_1 \cdots P_k \qquad P_i \in R^{mxm}$$

in the form

$$Q_k = I + WY^T \qquad W , Y \in R^{mxk} . \tag{1.1}$$

Several authors have looked at similar ways to group and generalize Householder matrices and we review this work in §2. The basic properties of the WY representation are detailed in §3. In §4 we show to compute the QR factorization using the WY representation. An implementation of our algorithm on the FPS-164/MAX is then described in §5. The last section suggests how the WY representation can be used to compute other decompositions that involve orthogonal transformations.

2. Aggregating Householder Transformations

The idea of "aggregating" Householder transformations for performance is not new. Dongarra, Kaufman, and Hammarling (1985) suggest pairing Householder matrices in order to reduce the number of vector memory references. For example, they recommend that the update

$$A := (I - \alpha ww^T)(I - \beta uu^T)A$$

be computed as follows:

$$\begin{aligned} v^T &= w^T A \\ x^T &= u^T A \\ y^T &= v^T - (\beta w^T u)x^T \\ A &:= A - \beta u x^T - \alpha w y^T \ . \end{aligned}$$

As the authors point out, this manuever actually increases the number of required flops. But the resulting algorithm is faster in pipelined environments because there is less memory traffic. The WY representation (1.1) is a generalization of this pairwise grouping strategy.

Note that an orthogonal matrix of the form $I + WY^T$ can be thought of as a generalized Householder matrix, an idea that was examined by Dietrich(1976). Given an m-by-p matrix partitioned as follows

$$X = \begin{array}{[c]c} X_1 & p \\ X_2 & m-p \end{array}$$

Dietrich shows how to compute a full rank m-by-p matrix

$$W = \begin{array}{[c]c} W_1 & p \\ W_2 & m-p \end{array}$$

so that if

$$P = I - 2W(W^TW)^{-1}W^T$$

then

$$P^TX = \begin{array}{[c]c} T & p \\ 0 & m-p \end{array} \qquad (2.1)$$

The matrix P above is clearly a block generalization of a Householder matrix. A stable method for computing W using some recent algorithmic developments is as follows:

Step 1. Compute the QR factorization:

$$X = \begin{bmatrix} Q_1 \\ Q_2 \end{bmatrix} R \ .$$

Step 2. Compute the CS decomposition:

$$\begin{aligned} Q_1 &= U_1 C V^T \ , \quad C = diag(c_1, \ldots, c_p) \ , \quad 1 \ge c_1 \ge \cdots \ge c_p \ge 0 \\ Q_2 &= U_2 S V^T \ , \quad S = diag(s_1, \ldots, s_p) \ , \quad 0 \le s_1 \le \cdots \le s_p \le 1 \ . \end{aligned}$$

(Here, U_1 , U_2 and V are orthogonal.)

Step 3 . Set

$$W = \begin{bmatrix} U_1(C + I) \\ U_2 S \end{bmatrix} .$$

The trouble with this method of aggregating Householder transformations is that some delicate SVD computations are involved in Step 2. See Stewart (1983) or Van Loan (1985) for details concerning the computation of the CS decomposition. In addition to these difficulties, the matrix T in (2.1) will not generally be upper triangular. ($T = -U_1 V^T$.) Thus, if we incorporated Dietrich's idea in a QR factorization algorithm the resulting R would be block upper triangular. True upper triangular form could easily be achieved but it's an inconvenient post-computation.

3. The WY Representation

Recall that an m-by-m Householder matrix is an orthogonal matrix of the form

$$P = I + uv^T$$

where $v \in R^m$ has unit 2-norm length and $u = -2v$. Householder matrices can be used to compute all kinds of important matrix decompositions and the large number of subroutines in LINPACK and EISPACK that rely on these transformations underscores their importance.

In a typical application a sequence of Householder transformations $P_i = I + u_i v_i^T$ is applied to a given matrix. Consider the following algorithm for computing the QR factorization of

$$A = [\, a_1 , \cdots , a_n \,] \qquad a_j \in R^m .$$

Algorithm 3.1

```
For $k = 1 : n$
    For $j = 1 : k-1$
        $a_k := a_k + (u_j^T a_k) v_j$        $(a_k := P_{k-1} \cdots P_1 a_k)$
    end $j$
    Compute unit 2-norm $v_k$ so that if $u_k = -2v_k$ and
        $P_k = I + u_k v_k^T$ then $P_k a_k$ is zero in components $k+1$ to $m$ .
end $k$
```

This computation is rich in inner products and SAXPY's but not in matrix multiplication. (A "SAXPY" is the operation $y := \alpha x + y$ where x and y are vectors and α is a scalar.) The SAXPY's in the j loop cannot be performed in parallel because the a_k are different for each j.

Reversing the order of the loops in Algorithm 3.1 essentially gives the LINPACK QR factorization procedure. In this version the key computations are updates of the form $A := (I + u_j v_j^T)^T A$. The inner products associated with $u_j^T A$ can be computed in parallel as can the SAXPY's associated with $A := A + v_j (u_j^T A)$. What prevents "supervector performance" is an unfavorable ratio of vector loads and unloads to actual computation. See Dongarra, Kaufman, and Hammarling (1985).

We propose a new way to represent products of Householder matrices that rectifies this problem. In particular, we exploit the fact that a product of Householders

$$Q_k = P_1 \cdots P_k$$

can be written in the form

$$Q_k = I + W_k Y_k^T$$

where W_k and Y_k are m-by-k matrices. Note that the mission of the j loop in Algorithm 3.1 is to compute $Q_{k-1}^T a_k$.

To anticipate the value of the WY representation observe that if B is a matrix then the computation $B := Q_k^T B$ is rich in matrix multiplication:

$$B := B + Y_k(W_k^T B)$$

However, before we can pursue the algorithmic benefits of the WY representation be must establish its existance and show how it can be updated.

The WY form certainly exists for the $k = 1$ case. Indeed, since $Q_1 = P_1 = I + u_1 v_1^T$ we just set $W = u_1$ and $Y = v_1$. To establish the result for general k we show how to obtain (W_k , Y_k) from (W_{k-1} , Y_{k-1}) upon receipt of the k-th Householder matrix $P_k = I + u_k v_k^T$. From the manipulations

$$\begin{aligned} Q_k = Q_{k-1}P_k &= (I + W_{k-1}Y_{k-1}^T)(I + u_k v_k^T) \\ &= I + [W_{k-1} , Q_{k-1}u_k][Y_{k-1} , v_k]^T \\ &= I + [W_{k-1} , u_k][P_k Y_{k-1} , v_k]^T \end{aligned}$$

we see that there are actually two ways to accomplish this:

Method 1.

$$\begin{aligned} W_k &= [W_{k-1} , Q_{k-1}u_k] \\ Y_k &= [Y_{k-1} , v_k] \end{aligned}$$

Method 2.

$$\begin{aligned} W_k &= [W_{k-1} , u_k] \\ Y_k &= [Y_{k-1} , P_k Y_{k-1}] \end{aligned}$$

We have used Method 1 in our implementations feeling that the matrix-vector multiplication $Q_{k-1}u_k$ is easier to optimize than the rank one update $P_k Y_{k-1}$. However, this is not a clear choice. Indeed, it may be wiser to update WY factors using Method 2 for the following reason. In many applications such as in Algorithm 3.1 the first $k - 1$ components of the Householder vector v_k are zero. Thus, if Method 2 is used then a simple check reveals that *both* Y_k and W_k are lower trapezoidal. However, if Method 1 is used then *only* Y_k is lower trapezoidal.

We point out that manipulation of Householder matrices through the WY form is stable. The same favorable roundoff properties that attend "conventional" Householder algorithms apply if the computations are arranged in WY form. This claim is justified in a brief appendix to be found at the end of the paper.

Finally, we mention that the above updating formulae generalize. In particular, if $U = I + W_U Y_U^T$ and $V = I + W_V Y_V^T$ are orthogonal matrices in WY form then $Q = UV$ has its WY factors given by $W_Q = [W_U , U W_V]$ and $Y_Q = [Y_U , Y_V]$.

4. Computing the QR Factorization Using the WY Representation

To illustrate the attractiveness of the WY representation we show how it can be used to compute the QR factorization of a rectangular matrix $A = [a_1, \ldots, a_n] \in R^{mxn}$. A naive (but instructive) WY implementation of Algorithm 3.1 is as follows:

Algorithm 4.1

```
For $k = 1 : n$
    $a_k := (I + WY^T)^T a_k$        ($a_k := Q_{k-1}^T a_k$ , skip if $k = 1$)
    Compute unit $v$ such that if $u = -2v$ then $(I + uv^T) a_k$
        is zero in components $k+1$ to $m$ .
    $W := [W , (I + WY^T)u]$        ( $W = [u]$ if $k = 1$ )
    $Y := [Y , v]$                  ( $Y = [v]$ if $k = 1$ )
end $k$
```

There are three problems with this algorithm. (1) It requires $2mn^2 - n^3/2$ flops, approximately twice what is required by the usual Householder QR factorization scheme, e.g., Algorithm 3.1. (2) It requires an m-by-n workspace for W. (3) It is not rich in matrix-matrix multiplication.

The same criticisms would essentially apply if Method 2 updating was used although one could get by with $n^2/2$ fewer storage locations and $p^3/2$ fewer flops.

To obtain a successful QR factorization scheme using the WY representation it is necessary to partition A into block columns:

$$A = [A_1 , \cdots , A_N]$$

where each A_i has p columns. (If p does not divide n then A_N may have fewer than p columns, an unimportant detail that we hereafter suppress.) The appropriate column width p depends on the underlying architecture. (For the FPS-164/MAX it will turn out to be equal to the maximum number of parallel dot products that can be performed.)

The basic idea of our procedure is as follows. At the beginning of Step k $(1 \le k \le N)$ the matrix A has been overwritten with

$$P_{(k-1)p} \cdots P_1 A = \begin{array}{cc} \left[\begin{array}{ccc} R_{11} & R_{12} & R_{13} \\ 0 & \bar{A}_k & B \end{array}\right] & \begin{array}{c} (k-1)p \\ r \end{array} \\ \begin{array}{ccc} (k-1)p & p & q \end{array} & \end{array}$$

where $r = m-(k-1)p$ and $q = n-kp$. The QR factorization of $\bar{A}_k$ is then computed and the orthogonal matrix (in WY form) is applied to B.

In our formal description of this procedure we use the notation $A(i_1\ ,\ i_2\ ,\ j_1 : j_2\ )$ to designate the submatrix of A defined by rows i_1 through i_2 and columns j_1 through j_2.

Algorithm 4.2

```
For $k = 1: N$
    $s = (k-1)p + 1$
    Compute $W$ and $Y$ such that $I + WY^T$ is orthogonal
        and $(I+WY^T)^T A(s{:}m, s{:}s+p-1)$ is upper triangular.
    $A(s{:}m, s{:}n) \ := \ (I + WY^T)^T A(s{:}m, s{:}n)$
end $k$
```

Upon emergence from this algorithm the matrix A is overwritten by R. The Householder vectors v_i that are generated can be stored below the diagonal in the usual manner. (See Golub and Van Loan (1983,p.148).) It is not necessary to save the W matrices from step to step and so we just need an m-by-p workspace for W (This is in contrast to the m-by-n workspace required by Algorithm 4.1). Algorithm 4.2 is clearly rich in matrix-matrix multiplication. Indeed much more work is spent applying the WY factors than generating them which is good in that it implies richness in matrix multiplication. To be specific, $(2mn^2 - n^3)/N$ flops are required to generate the W's and Y's while $(mn^2 - n^3/3)$ flops are needed to apply them. Since the LINPACK QR algorithm requires $mn^2 - n^3/3$ flops we see that Algorithm 4.2 is more expensive by a factor of $(1 + 2/N)$ from the standpoint of flops. For modest (and realistic) values of N this factor is for all practical purposes equal to one. Moreover, in high performance environments just counting flops is an inadequate measure of efficiency. The number of references to memory usually has a much greater bearing on the efficiency of an algorithm. This is borne out by our experience implementing Algorithm 4.2 on the FPS-164/MAX.

5. Performance on the FPS-164/MAX System

Up to now a "flop" has meant the amount of work that is roughly associated with an operation of the form $a_{ij} := a_{ij} + a_{ik}\, a_{kj}$. However, manufacturers of "high performance" computers quantify speed using "high performance flops". A "high performance flop" equals two "regular" flops. Thus, n-by-n matrix multiplication requires $2n^3$ flops. In this section we'll quantify speed in terms of millions of high performance flops per second as we will be reporting on the FPS-164-MAX.

The FPS-164 is an 11 Megaflop (Mflop), 64-bit general purpose scientific processor built by Floating Point Systems Inc. It is attached to the I/O system of a general purpose host computer. In a typical situation the program executing on the 164 is a subroutine called from a host FORTRAN program.

The FPS-164 can be enhanced with additional "MAX boards" that can perform selected linear algebra calculations very fast. Each board buys 22 Mflops and up to fifteen can be installed in a single 164. A fully configured FPS-164/MAX thus has a theoretical peak performance level of 341 Mflops. A nice overview of the architecture is given in Madura, Broussard, and Strickland (1985). We cover just enough of its features to build an appreciation for the type of algorithm that performs efficiently in the MAX environment.

Each MAX module operates synchronously and in parallel with the 164's CPU. The modules appear as memory to the 164's CPU in that they are accessed via reads and writes to reserved addresses. The essential components of a MAX module are two fully pipelined adder/multiplier pairs and eight vector registers (length = 2047). The MAX modules have a limited repertoire of operations but by design they figure heavily in matrix computations. If $n_{\max}$ denotes the number of installed MAX boards then the system is able to perform

$$n_d = 4 + 8 \cdot n_{\max}$$

parallel dot products or

$$n_v = 1 + 4 \cdot n_{\max}$$

parallel SAXPY's as follows:

Parallel Dot Product

If $V = [v_1, \ldots, v_d] \in R^{mxd}$ $(d \le n_d)$ resides in the MAX vector registers and $w \in R^m$ is stored in main memory, then it is possible to compute the d inner products $v_i^T w$ concurrently. If $d = n_d$ then this means that $V^T w$ can be formed at an approximate rate of $11 + 22 \cdot n_{\max}$ Mflops.

Parallel SAXPY

If $V = [v_1, \ldots, v_d] \in R^{mxd}$ $(d \le n_v)$ resides in the MAX vector registers and the vectors $z \in R^d$ and $u \in R^m$ are in main memory, then it is possible to compute the s SAXPY's $v_i := v_i + z_i u$ concurrently. If $d = n_v$ then the computation $V := V + u\, z^T$ can be formed at the approximate rate of $5 + 11 \cdot n_{\max}$ Mflops

FPS has extended its ANSI FORTRAN 77 to accomodate a set of subroutines that perform the above computations. This includes a set of routines for executing the required loads and unloads that are responsible for moving data between the MAX modules and main memory. (To those familiar with the FPS-164, table memory essentially acts like a half MAX board. So when we refer to "MAX boards" we really mean "MAX boards plus table memory.")

Because the overhead associated with the loading and unloading of the MAX modules can easily dominate a floating point computation, it is desirable that the vectors loaded into a MAX module be reasonably long and (more critically) that they be "re-used" as much as possible before they are returned to main memory. That way, the cost of a load (or unload) is spread over a significant amount of high speed computation.

With this review of the FPS-164/MAX we can present the MAX version of Algorithm 4.2. In the k-th step of the algorithm attention is focussed on the submatrix

$$A(\,s{:}m\ ,\ s{:}n\,) = \begin{array}{cc|c} \bar{A}_k , & B & q \\ p & r & \end{array}$$

where $q = m-(k-1)p$, $r = n-kp$, and $s = 1 + (k-1)p$. The block width p is taken to be the maximum number of parallel dot products that can be performed, i.e., $p = n_d$. There are three steps to consider.

Step 1. Compute the QR Factorization of $\bar{A}_k$.

Algorithm 4.1 is used for this purpose. The MAX boards are *not* used here as the algorithm is not rich in matrix multiplication.

Step 2. Compute $Z^T = W^T B$.

Let $B = [b_1, \ldots, b_r]$ be a column partitioning of B . Noting that W entirely fits into the MAX vector registers, $Z \in R^{r \times p}$ is computed as follows:

```
Load $W$ into the MAX boards.
**For** $i = 1{:}r$
    Compute $z(\,i{:}i\ ,\ 1{:}p\,) := b_i^T W$     (Parallel Dot Product)
**end** $i$
```

Note that W is reused $r = n-kp$ times. Except at the very end of the algorithm this portion of the computation can proceed at close to peak parallel dot product speed.

Step 3. Overwrite B with $B + YZ^T$

This is a parallel SAXPY operation. For clarity, assume that s divides r and that $t = r/s$. Partition B, Y, and Z as follows:

$$B = [B_1, \ldots, B_t] \qquad B_i \in R^{qxs}$$

$$Y = [y_1, \cdots, y_p] \qquad y_i \in R^q$$

$$Z = \begin{bmatrix} z_{11} & \cdots & z_{1p} \\ \vdots & \vdots & \vdots \\ \vdots & \vdots & \vdots \\ z_{t1} & \cdots & z_{tp} \end{bmatrix} \qquad z_{ij} \in R^s$$

With this partitioning the update $B := B + YZ^T$ can be computed as follows:

```
For $i = 1: t$
    Load $B_i$ into the MAX vector registers.
    For $j = 1: p$
        $B_i := B_i + y_j z_{ji}^T$    (Parallel SAXPY)
    end $j$
    Unload $B_i$ into main memory.
end $i$
```

The re-use factor for this portion of the algorithm is p.

The MAX algorithm just described has been tested on Cornell's FPS-164/MAX system. Currently, this is a one MAX Board installation implying 33 Mflop peak dot product performance and 15 Mflop peak SAXPY performance. Table 1 indicates the megaflop rates that our code has achieved for problems of various dimensions.

m	n=.25m	n=.50m	n=.75m	n=m
250	7	9	11	11
500	10	13	14	15
750	13	14	16	17
1000	14	15	17	18

Table 1. Performance in Megaflops (Mflops)

We point out that the fraction of the overall computation that is performed on the MAX boards is $N/(N + 2)$ where $N = n/n_d = n/12$. For the various values of n represented in the table, this fraction ranges from .71 to .97. Of the all the MAX computations in our implementation half are parallel dot products and half are parallel SAXPY's. Table 5.1 indicates the megaflop rates that our code has achieved for problems of varying dimension. An optimized "LINPACK QR" code running on the 164 without MAX boards would perfrom in the vicinity of 5 Mflops. We feel that these benchmarks confirm that the WY representation is a viable way of organizing Householder computations on the FPS-164/MAX.

6. Other Factorizations

Householder matrices are traditionally used in the computation of the following decompositions:

$$\begin{aligned} Q^H A Q &= T && (A \text{ symmetric} , T \text{ tridiagonal}) \\ Q^H A Q &= H && (H \text{ Hessenberg}) \\ U^T A V &= B && (B \text{ bidiagonal}) \end{aligned}$$

When one contemplates WY versions of these algorithms some new difficulties arise that are not present in the QR factorization. These problems stem from the fact that the above decompositions involve both left and right transformations. This makes "delayed" application of the Householder matrices problematical. In the QR scheme we can reduce a subset of columns without touching the "rest" of the matrix. This is because the Householders are applied from just one side. However, in Householder tridiagonalization (for example) this is not possible. The second Householder P_2 is a function of *all* the matrix elements because it is based on the second column of

$$P_1^T A P_1 = \begin{bmatrix} x & x & 0 & 0 & 0 \\ x & x & x & x & x \\ 0 & x & x & x & x \\ 0 & x & x & x & x \\ 0 & x & x & x & x \end{bmatrix} .$$

In particular, n^2 flops are required to compute this column. This is *not* an order of magnitude less work than performing the entire update $A := P_1^T A P_1$ which costs $2n^2$ flops. Thus, unlike in the QR factorization method, nothing is gained by aggregating Householders and *then* applying them in WY form.

One way out of this jam is to strive for block tridiagonalization. Partition A as follows:

$$A = \begin{array}{cc} \left[\begin{array}{cc} A_{11} & A_{21}^T \\ A_{21} & A_{22} \end{array} \right] & \begin{array}{c} p \\ n-p \end{array} \\ \begin{array}{cc} p & n-p \end{array} & \end{array} .$$

If we compute the QR factorization

$$A_{21} = QR \qquad Q = I + WY^T$$

and then update

$$B = Q^T A_{22} Q = (I + WY^T)^T A_{22} (I + WY^T)$$

we obtain

$$A := diag(I_p , Q)^T A \, diag(I_p , Q) = \left[\begin{array}{cc} A_{11} & R^T \\ R & B \end{array} \right] .$$

This illustrates the basic step of the algorithm. Although it is rich in matrix multiplies it leaves us with a bandwidth p eigenproblem. One posibility would be to reduce the resulting matrix to tridiagonal form using the method of Schwartz(1968). The overall success of this procedure in the MAX environment is the next item on our agenda and will be reported elsewhere along with some related procedures for bidiagonalization and Hessenberg reduction.

Acknowledgements

We are indebted to Steve Oslon of FPS who kindly shared his MAX expertise with us as we developed our code. We're also obliged to Gene Golub of Stanford University for calling our attention to the Dietrich (1976) reference.

Appendix. Roundoff Properties of the WY Representation

In this appendix we have opted for an "O(**u**)" analysis feeling that complete rigor would add only tedium to an otherwise simple argument. For further comments on the "philosophy" of roundoff analysis, see Golub and Van Loan (1983,p.32ff).

Suppose **u** is the unit roundoff and A and B are floating point matrices. If $fl(AB)$ denotes the computed product of A and B then

$$fl(AB) = AB + E$$

where

$$\| E \|_2 = O(\mathbf{u}) \| A \|_2 \| B \|_2 .$$

Likewise, if A and B are compatible for addition then

$$fl(A + B) = A + B + E$$

with

$$\| E \|_2 = O(\mathbf{u}) (\| A \|_2 + \| B \|_2) .$$

Repeated use will be made of these well known facts.

We begin with a definition. Suppose W and Y are m-by-k matrices with floating point entries. We say that

$$Q = I + WY^T$$

is **u**-orthogonal if the following three properties hold:

$$\| W \|_2 = O(1) \tag{A1}$$

$$\| Y \|_2 = O(1) \tag{A2}$$

$$\| Q^TQ - I \|_2 = O(\mathbf{u}) . \tag{A3}$$

If we apply Q to a floating point matrix A then

$$fl(QA) = QA + F$$

where

$$\| F \|_2 = O(\mathbf{u}) \| A \|_2 .$$

Since (A3) implies that the singular values σ_i of Q are all of the form $\sigma_i = 1 + O(\mathbf{u})$, it follows that

$$fl(QA) = Q(A + Q^{-1}F) = Q(A + E) \tag{A4}$$

where

$$\| E \|_2 = \| Q^{-1}F \|_2 = O(\mathbf{u}) \| A \|_2 . \tag{A5}$$

Thus, if (A1), (A2) , and (A3) hold then manipulations with Q in WY form are stable because it allows for a favorable inverse error analysis.

A corollary of this result is that

$$P = I + uv^T$$

is **u**-orthogonal provided the floating point vectors v and u satisfy

$$\| v \|_2 = 1 + O(\mathbf{u}) \tag{A6}$$

and

$$u = -2v \ . \tag{A7}$$

This can be used to confirm the stability of conventional Householder matrix manipulations.

What we must show is that when the WY representation is updated (say by Method 1) then properties (A1)-(A3) remain in force. To this end, suppose

$$W_+ = [\ W\ ,\ fl(Qu)\ ]$$
$$Y_+ = [\ Y\ ,\ v\ ]$$

and

$$Q_+ = I + W_+Y_+^T$$

where u and v satisfy (A6) and (A7) and $(W_+\ ,\ Y_+)$ is the computed WY representation of the approximate orthogonal matrix QP. Setting $A = u$ in (A4) and (A5) gives

$$fl(Qu) = Qu + e \tag{A8}$$

with

$$\| e \|_2 = O(\mathbf{u})\ \| \mathrm{u} \|_2 = \mathrm{O}(\mathbf{u}) \ . \tag{A9}$$

Here we used the fact that $\| u \|_2 = 2 + O(\mathbf{u}) = \mathrm{O}(1)$. It follows from (A1), (A3), (A8), (A9), and the definition of W_+ that

$$\| W_+ \|_2 \le \| W \|_2 + \| fl(Qu) \|_2 = O(1) \ .$$

Likewise, (A2), (A6), and the definition of Y_+ imply that

$$\| Y_+ \|_2 \le \| Y \|_2 + \| v \|_2 = O(1) \ .$$

All that remains for us to show is that Q_+ satisfies (A3). Using (A8) and the definition of Q_+ we have

$$\begin{aligned} Q_+ &= I + W_+Y_+^T = I + WY^T + fl(Qu)v^T \\ &= Q + (Qu + e)v^T = Q(I + uv^T) + ev^T = Q(P + Q^{-1}ev^T) \ . \end{aligned}$$

Using (A9) it is easy to show that if

$$\bar{P} = P + Q^{-1} e v^T$$

then

$$\| \bar{P}^T \bar{P} - I \|_2 = O(\mathbf{u}) \ .$$

This coupled with (A3) ensures that $\| Q_+^T Q_+ - I \|_2 = O(\mathbf{u})$.

We mention that if W and Y are updated using Method 2, then a similar error analysis goes through.

References

G. Dietrich (1976), " A New Formulation of the Hypermatrix Householder-QR Decomposition", *Computer Methods in Applied Mechanics and Engineering,* 9 , 273-280.

J. Dongarra, J.D. Bunch , C.B.Moler, and G.W.Stewart (1979), *Linpack User's Guide*, SIAM Publications, Philadelphia.

J. Dongarra and S. Eisenstat (1984), "Squeezing the Most Out of an Algorithm in Cray Fortran ", *ACM Transactions on Mathematical Software* , 10, 216-230.

J. Dongarra, F.G. Gustavson, and A. Karp (1984), "Implementing Linear Algebra Algorithms on Dense Matrices on a Vector Pipeline Machine", *SIAM Review*, 26, 91-112.

J. Dongarra, L. Kaufman, S. Hammarling (1985). "Squeezing the Most out of Eigenvalue Solvers on High-Performance Computers", Technical Memorandum 46, Mathematics and Computer Science Division, Argonne National Laboratory, Argonne , IL 60439.

G.H. Golub and C. Van Loan (1983), *Matrix Computations* , The Johns Hopkins University Press, Baltimore Md.

D. Madura, R. Broussard, and D. Strickland (1985), "FPS-164 MAX: Parallel Multiprocessing for Linear Algebra Operation", *Proceedings of the 1985 Array Processing Conference,* , New Orleans., pp. 33-50.

H.R. Schwartz (1968), "Tridiagonalization of a Symmetric Band Matrix " , *Numerische Mathematik* , 12 , 231-241.

B.T.Smith, J.M.Boyle, Y.Ikebe, V.C. Klema, and C.B. Moler (1970), *Matrix Eigensystem Routines: EISPACK Guide*, second edition, Springer-Verlag, New York.

G.W. Stewart(1983), "An Algorithm for Computing the CS Decomposition of a Partitioned Orthonormal Matrix", *Numerische Mathematik,* 40, 297-306.

C. Van Loan (1985), "Computing the CS and the Generalized Singular Value Decomposition ", *Numerische Mathematik*, 46, 479-491.