An Improved Algorithm for Computing the Singular Value Decomposition

TONY F. CHAN
Yale University

The most well-known and widely used algorithm for computing the Singular Value Decomposition (SVD) $A = U\Sigma V^T$ of an $m \times n$ rectangular matrix A is the Golub–Reinsch algorithm (GR-SVD). In this paper, an improved version of the original GR-SVD algorithm is presented. The new algorithm works best for matrices with $m \gg n$, but is more efficient even when m is only slightly greater than n (usually when $m \geq 2n$) and in some cases can achieve as much as 50 percent savings. If the matrix U is explicitly desired, then n^2 extra storage locations are required, but otherwise no extra storage is needed. The two main modifications are: (1) first triangularizing A by Householder transformations before bidiagonalizing it (this idea seems to be widely known among some researchers in the field, but as far as can be determined, neither a detailed analysis nor an implementation has been published before), and (2) accumulating the left Givens transformations in GR-SVD on an $n \times n$ array instead of on an $m \times n$ array. A PFORT-verified FORTRAN implementation is included. Comparisons with the EISPACK SVD routine are given.

Categories and Subject Descriptors: G.1.3 **[Numerical Analysis]**: Numerical Linear Algebra—*eigenvalues*

General Terms: Algorithms, Performance

Additional Key Words and Phrases: Householder transformations, singular values

The Algorithm. An Improved Algorithm for Computing the Singular Value Decomposition. *ACM Trans. Math. Softw. 8*, 1 (Mar. 1982), 84–88.

1. INTRODUCTION

Let A be a real $m \times n$ matrix, with $m \geq n$. It is well known [5, 6] that the following decomposition of A always exists:

$$A = U\Sigma V^T, \tag{1.1}$$

where U is an $m \times n$ matrix and consists of n orthonormalized eigenvectors associated with the n largest eigenvalue of AA^T, V is an $n \times n$ matrix and consists of the orthonormalized eigenvectors of A^TA, and Σ is a diagonal matrix consisting

This work was supported by NSF Grant DCR 75-13497 and NASA Ames Contract NCA 2-OR745-520. The computing time was provided by the Stanford Linear Accelerator Center (SLAC).
Author's address: Department of Computer Science, Yale University, 10 Hillhouse Avenue, P.O. Box 2158 Yale Station, New Haven, CT 06520.

of the "singular values" of A, which are the nonnegative square roots of the eigenvalues of A^TA.

Thus,

$$U^TU = V^TV = VV^T = I_n$$

and

$$\Sigma = \text{diag}(\sigma_1, \ldots, \sigma_n). \tag{1.2}$$

It is usually assumed for convenience that

$$\sigma_1 >= \sigma_2 >= \cdots >= \sigma_n >= 0.$$

The decomposition (1.1) is called the *Singular Value Decomposition* (SVD) of A.

Remarks

(1) If $\text{rank}(A) = r$, then $\sigma_{r+1} = \sigma_{r+2} = \cdots = \sigma_n = 0$.
(2) There is no loss of generality in assuming that $m \geq n$, for if $m < n$, then we can instead compute the SVD of A^T. If the SVD of A^T is equal to $U\Sigma V^T$, then the SVD of A is equal to $V\Sigma U^T$.

The SVD plays a very important role in linear algebra. It has applications in such areas as least squares problems [5, 6, 11], in computing the pseudoinverse [6], in computing the Jordan canonical form [7], in solving integral equations [9], in digital image processing [1], and in optimization [2]. Many of the applications often involve large matrices. It is therefore important that the computational procedures for obtaining the SVD be as efficient as possible.

2. THE GOLUB-REINSCH ALGORITHM (GR-SVD)

We use the same notations as in [5] and [3].

This algorithm consists of two phases. In the first phase one constructs two finite sequences of Householder transformations

$$P^{(k)}, \qquad k = 1, 2, \ldots, n$$

and

$$Q^{(k)}, \qquad k = 1, 2, \ldots, n-2,$$

such that

$$P^{(n)} \cdots P^{(1)} A Q^{(1)} \cdots Q^{(n-2)} = \overbrace{\begin{bmatrix} \times & \times & & 0 \\ & \ddots & \ddots & \\ 0 & & \ddots & \\ & & & \times \ \times \\ \hline & & 0 & \end{bmatrix}}^{n} \left.\begin{matrix} \\ \\ \\ \\ \\ \end{matrix}\right\} \begin{matrix} n \\ = J^{(0)} \\ (m-n) \end{matrix}, \tag{2.1}$$

an upper bidiagonal matrix. Specifically, $P^{(i)}$ zeros out the subdiagonal elements in column i and $Q^{(j)}$ zeros out the appropriate elements in row j.

Because all the transformations introduced are orthogonal, the singular values of $J^{(0)}$ are the same as those of A. Thus, if

$$J^{(0)} = G\Sigma H^{\mathrm{T}}$$

is the SVD of $J^{(0)}$, then

$$A = PG\Sigma H^{\mathrm{T}}Q^{\mathrm{T}}$$

so that

$$U = PG, \qquad V = QH \tag{2.2}$$

with

$$P = P^{(1)} \cdots P^{(n)}, \qquad Q = Q^{(1)} \cdots Q^{(n-2)}.$$

The second phase is to iteratively diagonalize $J^{(0)}$ by the QR method so that

$$J^{(0)} \to J^{(1)} \to \cdots \to \Sigma, \tag{2.3}$$

where

$$J^{(i+1)} = (S^{(i)})^{\mathrm{T}} J^{(i)} T^{(i)},$$

where $S^{(i)}$ and $T^{(i)}$ are products of Givens transformations and are therefore orthogonal.

The matrices $T^{(i)}$ are chosen so that the sequence $M^{(i)} = (J^{(i)})^{\mathrm{T}} J^{(i)}$ converges to a diagonal matrix, while the matrices $S^{(i)}$ are chosen so that all $J^{(i)}$ are of bidiagonal form. The products of the $T^{(i)}$ and the $S^{(i)}$ are exactly the matrices H^{T} and G^{T}, respectively, in eq. (2.2). For more details, see [5].

It has been reported in [5] that the average number of iterations on $J^{(i)}$ in (2.3) is usually less than $2n$. In other words, $J^{(2n)}$ in eq. (2.3) is usually a good approximation to a diagonal matrix.

We briefly describe how the computation is usually implemented. Assume for simplicity that we can destroy A and return U in the storage for A. In the first phase the $P^{(i)}$ are stored in the lower part of A, and the $Q^{(i)}$ are stored in the upper triangular part of A. After the bidiagonalization the $Q^{(i)}$ are accumulated in the storage provided for V, the two diagonals of $J^{(0)}$ are copied to two other linear arrays, and the $P^{(i)}$ are accumulated in A.

In the second phase, for each i,

$S^{(i)}$ is applied to P from the right, and

$(T^{(i)})^{\mathrm{T}}$ is applied to Q^T from the left

in order to accumulate the transformations.

3. THE MODIFIED ALGORITHM (MOD-SVD)

Our original motivation for this algorithm is to find an improvement of GR-SVD when $m \gg n$. In that case two improvements are possible:

(1) In eq. (2.1), each of the transformations $P^{(i)}$ and $Q^{(i)}$ has to be applied to a submatrix of size $(m - i + 1) \times (n - i + 1)$ (see Figure 1). Now, since most entries of this submatrix are ultimately going to be zeros, it is intuitive that *if* it can

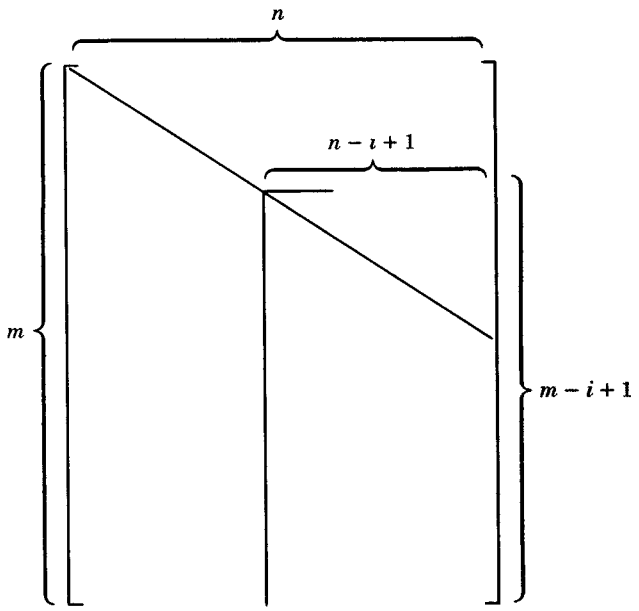

Fig. 1. $P^{(i)}$ and $Q^{(i)}$ affect the shaded portion of the matrix.

somehow be arranged that the $Q^{(i)}$ does not have to be applied to the subdiagonal part of this submatrix, then we will be saving a great amount of work when $m \gg n$.

This can indeed be done by first transforming A into upper triangular form by Householder transformations on the left:

$$L^{\mathrm{T}}[A] \rightarrow \begin{bmatrix} \\ 0 \end{bmatrix} \equiv \begin{bmatrix} R \\ \hline 0 \end{bmatrix}$$

where R is $n \times n$ upper triangular and L is orthogonal, and then proceed to bidiagonalize R. The important difference is that this time we will be working with a much smaller matrix, R, than A (if $n^2 \ll mn$), and so it is conceivable that the work required to bidiagonalize R is much less than that originally done by the right transformations when $m \gg n$.

The question still remains as to how to bidiagonalize R. An obvious way is to treat R as an input matrix to GR-SVD, using alternating left and right Householder transformations. In fact, it can be easily verified that if the SVD of R is equal to $X\Sigma Y^{\mathrm{T}}$, then the SVD of A is given by

$$A = L\begin{bmatrix} X \\ \hline 0 \end{bmatrix}\Sigma Y^{\mathrm{T}}. \tag{3.1}$$

We can identify U with $L[X/0]$ and V with Y. Notice that in order to obtain U, we have to form the extra product $L[X/0]$. If U is not needed explicitly (e.g., in least squares), then we do not have to accumulate any left transformations, and in that case, for $m \geq$ n, it seems likely that we will make a substantial saving.

It is also possible to take advantage of the structure of R to bidiagonalize it. This is discussed in Section 4.

(2) The second improvement of GR-SVD that can be made is the following: In GR-SVD, if U is wanted explicitly, each of the $S^{(i)}$ is applied to the $m \times n$ matrix P from the right to accumulate U. If $m \gg n$, then this accumulation may involve a large amount of work, because a single Givens transformation affects two columns of P (of length m) and each $S^{(i)}$ is the product of on the average $n/2$ Givens transformations. Therefore, in such cases it would seem more efficient to first accumulate all $S^{(i)}$ on an $n \times n$ array, say Z, and later form the matrix product PZ after $J^{(i)}$ has converged to Σ.

In essence, improvement (1) works best when U is not needed, improvement (2) works best when U is needed, and both work best when $m \gg n$. These improvements are both incorporated in the following modified GR-SVD algorithm:

MOD-SVD

(1) $L^T[A] \rightarrow [R/0]$ where R is $n \times n$ upper triangular.
(2) Find the SVD of R by GR-SVD, $R = X\Sigma Y^T$.
(3) Form $A = L[X/0]\Sigma Y^T$, the SVD of A.

The idea of transforming A to upper triangular form when $m \gg n$ and then calculating the SVD of R is mentioned in Lawson and Hanson [11, pp. 119 and 122] in the context of least squares problems where U is not explicitly required.

4. SOME COMPUTATIONAL DETAILS

(1) As in most GR-SVD implementations, the input matrix A is first copied into the array U. Thereafter, the array A is never referenced and all operations are performed on the array U. It should be obvious that when U is not needed, MOD-SVD does not require any extra storage. When U is needed, we can store L^T in the lower part of array U, copy R into another $n \times n$ array Z, and ask GR-SVD to return X in Z. Therefore we need at most n^2 extra storage locations (array Z), which is relatively small (when $m \gg n$) compared to mn locations already needed for array U.

(2) The next question is how to form $L[X/0]$ without using extra storage. This can be done by noting that

$$L\begin{bmatrix} X \\ \hline 0 \end{bmatrix} = L\begin{bmatrix} I \\ \hline 0 \end{bmatrix} X;$$

so we can first accumulate $L[I/0]$ in the space provided for U in place, and then do a matrix multiplication by X.

Another possibility is to actually accumulate the Householder transformations L on $[X/0]$. With the usual implementation, this requires mn instead of n^2 extra storage locations, but slightly less work ($2mn^2 - n^3$ versus $2mn^2 - n^3/3$ multiplications). Our current implementation uses the first method. A potential improvement has been suggested by Kauffman [10].

(3) The question arises whether it is possible to bidiagonalize R in a way that takes advantage of the zeros that are already in R. The usual Householder transformations cannot take advantage of this structure. One way is to use Givens

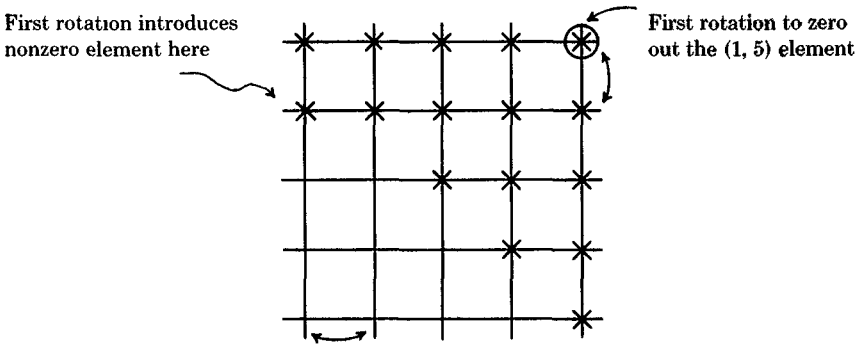

Figure 2

transformations to zero out the elements at the upper right-hand corner of R, one column or one row at a time. Pictorially (for $n = 5$), to zero out the (1, 5) element, we do two Givens transformations, as shown in Figure 2.

It turns out, however, by simple counting and verification by experiments, that this method takes about the same operations ($4n^3/3$ multiplications) as the previous method to bidiagonalize R, provided that we do not have to accumulate transformations. If we do need to accumulate either the left or right transformations, then this method will require more work ($4n^3$ versus $4n^3/3$ multiplications), mainly because it requires two rotations to zero out each element and these rotations have to be accumulated.

So it seems that taking advantage of the zero structure of R in this fashion actually makes the method less efficient. We have to note, however, that Givens transformations involve fewer additions and array accesses than Householder transformations per multiplication (see Section 5). Therefore this method may turn out to be more competitive on modern computers where the time taken for floating-point additions and multidimensional array indexings are not negligible compared to that for multiplications. Also, the use of fast Givens [4] may result in substantial improvement in efficiency.

5. OPERATION COUNTS

In Section 3 we indicated that MOD-SVD should be more efficient than GR-SVD when $m \gg n$. In this section we study the relative efficiency between GR-SVD and MOD-SVD as a function of m and n. We do this by computing the asymptotic operation counts for each algorithm.

In the following operation counts, we only keep the highest order terms in m and n, and so the results are correct for relatively large m and n.

GR-SVD

(1) *Bidiagonalization* (using Householder transformations)
$J = P^{(n)} \cdots P^{(1)} A Q^{(1)} \cdots Q^{(n-2)}$ $\quad 2(mn^2 - n^3/3)$ mult.

accumulate $P = P^{(1)} \cdots P^{(n)}$ $mn^2 - n^3/3$ mult.
accumulate $Q = Q^{(1)} \cdots Q^{(n-2)}$ $2n^3/3$ mult.

(2) *Diagonalization* (using Givens transformations)
accumulate $S^{(i)}$ on P Cmn^2 $(C = 4)$ mult.
accumulate $T^{(i)}$ on Q Cn^3 $(C = 4)$ mult.

MOD-SVD

(1) *Triangularization* (using Householder transformations) $L^T[A] \rightarrow [R/O]$ $mn^2 - n^3/3$ mult.
(2) GR-SVD of R, $R = X\Sigma Y^T$ depends on whether accumulations are needed
(3) Form $L[I/0]X$ (using Householder transformations) $2mn^2 - n^3/3$ mult.

Some comments are in order:

(1) The entries Cmn^2 and Cn^3 in the diagonalization phase of GR-SVD are obtained by assuming that the iterative phase of the SVD takes on the average two complete QR iterations per singular value [5], [11, p. 122]. We have checked this experimentally and found it to be quite accurate. It is assumed that slow Givens is used throughout the calculation. If fast Givens [4] had been used, then the entries would become approximately $2mn^2$ and $2n^3$, instead.

(2) As with most operation counts, we have used the number of multiplications as a measure of work. For the Householder transformations, each multiplication also invokes one addition and approximately two array addressings. For the Givens transformations, each multiplication invokes one-half an addition and one array addressing. On many large computers today, a floating-point multiplication is not much slower than a floating-point addition. Also, array indexing (involving integer arithmetic) is usually quite expensive. In such cases, a Householder multiplication actually involves more work than a Givens multiplication because of the extra additions and array indexings. Therefore, the operation counts given for the diagonalization phase of GR-SVD may be misleading because it may actually involve relatively less work. The total effect, however, can be accounted for by using a smaller value for C. For example, if one Givens "multiplication" takes half the work needed by a Householder "multiplication," then the effect on the *relative* efficiency can be accounted for by setting $C = 2$ instead of $C = 4$. On older or nonscientific machines where multiplications take much more time than additions and array addressings, the operation count based on multiplications alone is usually a good measure of relative efficiency.

(3) The application of $S^{(i)T}$ and $T^{(i)}$ on $J^{(i)}$ is actually of order $O(n^2)$ and is therefore not included in the previous counts.

(4) We have to distinguish between four cases in the comparison:

Case a: Both U and V are required explicitly.
Case b: Only U is required explicitly.
Case c: Only V is required explicitly.
Case d: Only Σ is required explicitly.

Table I. Total Operation Counts of GR-SVD and MON-SVD for Each of the Cases a, b, c, and d

Case	GR-SVD	MOD-SVD
a	$(3 + C)mn^2 + (C - 1/3)n^3$	$3mn^2 + (2C + 2)n^3$
b	$(3 + C)mn^2 - n^3$	$3mn^2 + (C + 4/3)n^3$
c	$2mn^2 + Cn^3$	$mn^2 + (C + 5/3)n^3$
d	$2mn^2 - 2n^3/3$	$mn^2 + n^3$

Table II. Ratio of Operation Count of MOD-SVD to that of GR-SVD, $r = m/n$

Case	Ratio	Crossover point r^*	limit as $r \to \infty$
a	$[3r + (2C + 2)]/[(3 + C)r + (C - 1/3)]$	$(C + 7/3)/C$	$3/(3 + C)$
b	$[3r + (C + 4/3)]/[(3 + C)r - 1]$	$(C + 7/3)/C$	$3/(3 + C)$
c	$[r + (C + 5/3)]/[2r + C]$	5/3	1/2
d	$[r + 1]/[2r - 2/3]$	5/3	1/2

These four cases do arise in applications. We will mention a few here:

Case a arises in the computation of pseudoinverse [5].
Case b is Case c for A^T.
Case c arises in least squares applications [5, 11] and in the solution of homogeneous linear equations [5].
Case d arises in the estimation of the condition number of a matrix and in the determination of the rank of a matrix [13].

The total operation count for each case is given in Table I.

Using Table I, we can compute the ratio of the operation counts of MOD-SVD to that of GR-SVD for each of the four cases. The results are given in Table II, where the ratio is expressed as a function of $r = m/n$. These ratios are plotted in Figure 3a–d for $C = 2, 3, 4$. The crossover point r^* is the value of r that makes the ratio equal to 1. If $r > r^*$, then MOD-SVD is more efficient than GR-SVD. We see that, in all four cases, MOD-SVD becomes more efficient than GR-SVD when r starts to get bigger than 2, approximately, and the savings can be as much as 50 percent when r is about 10. On the other hand, when r is about 1, GR-SVD is more efficient. This agrees with our earlier conjectures. However, the important thing is that all the ratios decrease quite fast as r becomes large. If we assume that it is equally likely to encounter matrices with any value of $r >= 1$ (this is not an unreasonable assumption for designers of general mathematical software, for example), then MOD-SVD is obviously preferable. In any case, these ratios give indications as to when one of the methods is more efficient, at least when m and n are large enough so that our operation counts apply.

In the context of least squares applications, we can also compare the operation counts of GR-SVD and MOD-SVD to those of the orthogonal triangularization methods (OTLS) [8], which are often used for such problems. Analogous to Table I, the ratios of operation counts are now

$$\text{OTLS: GR-SVD} = [r - 1/3]/[2r + C]$$

$$\text{OTLS: MOD-SVD} = [r - 1/3]/[r + C + 5/3].$$

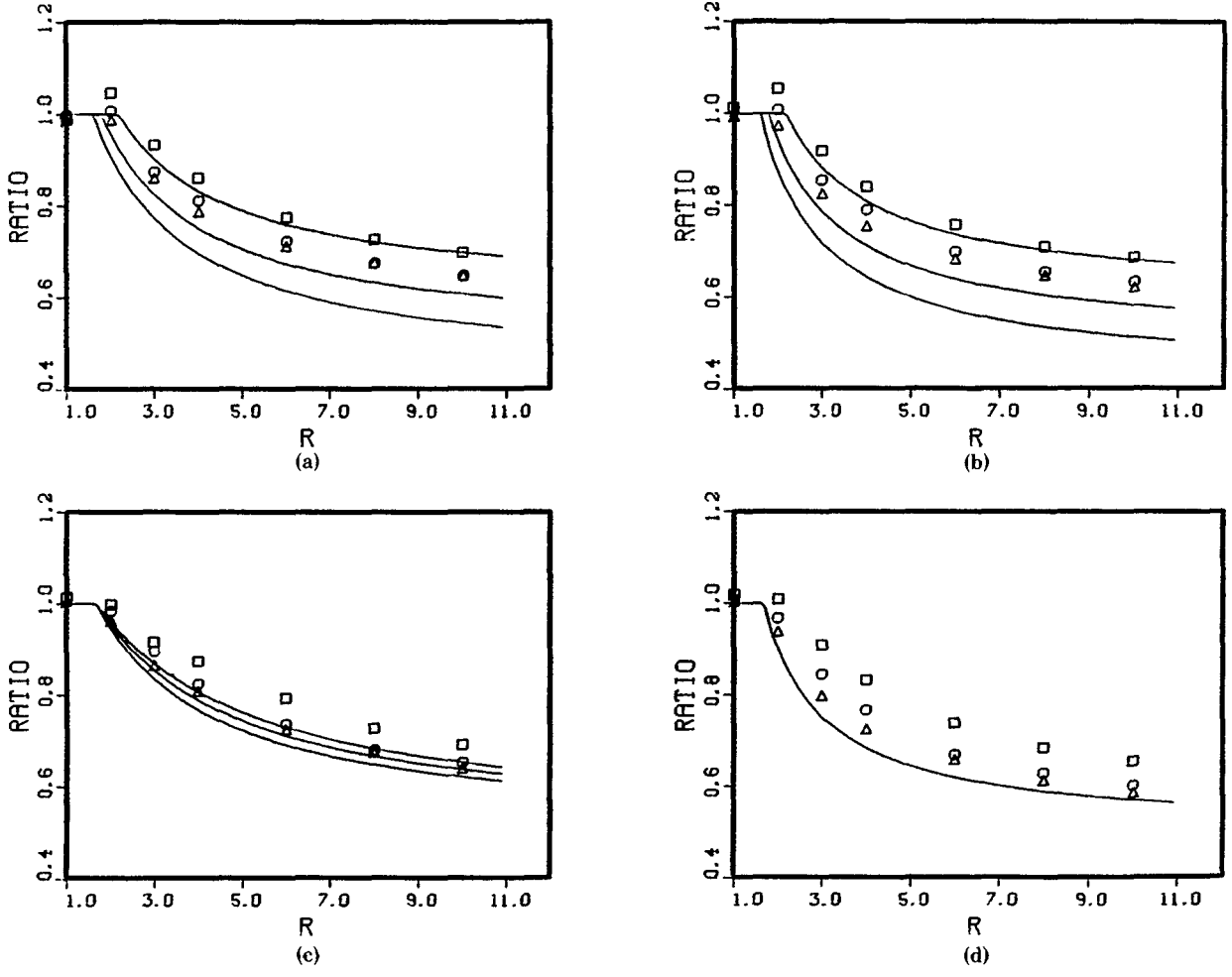

Fig. 3. Ratio of execution time: HYBSVD/EISPACK SVD. □: $n = 10$; ○: $n = 20$; △: $n = 40$, solid lines: operation counts.

One sees from these ratios that for m nearly equal to $n(r \approx 1)$, the two SVD algorithms require much more work than OTLS. However, when r is bigger than about 3, MOD-SVD requires only about 3 times more work than OTLS. It may therefore become economically feasible to solve the least squares problems at hand by MOD-SVD instead of OTLS. The reward is that the SVD returns much more useful information about the problem than OTLS [11].

It is easy to see that as r becomes arbitrarily large, MOD-SVD is as efficient as OTLS, since the bulk of the work is in the triangularization of the data matrix A. However, GR-SVD can be at most half as efficient as OTLS.

6. A HYBRID ALGORITHM

On the basis of the results of earlier sections, we can implement a hybrid method for computing the SVD of a rectangular matrix A, which automatically chooses to use the more efficient algorithm between GR-SVD and MOD-SVD. For each of the four Cases a, b, c, and d, if the input matrix A has a value of $r(= m/n)$, which is less than the crossover point r^* for that case, we use GR-SVD; otherwise we use MOD-SVD. The crossover points depend on the value of C used. As noted before, the value of C to be used depends on the relative efficiencies of floating-point multiplications, floating-point additions, and array indexings on the particular machine concerned. However, C can be determined once for any particular machine and compiler combination. For example, if floating-point multiplications take much more time than floating-point additions and array indexings on the machine in question, then we should use C approximately equal to 4. In most situations, one can probably do just as well by setting the crossover point equal to a fixed value ≈ 2, since this point is not sensitive to C at all.

In the algorithm (see p. 84) we give the codes of a PFORT [3a] verified FORTRAN subroutine called HYBSVD which implements the previously mentioned hybrid algorithm. HYBSVD will need to call a standard Golub–Reinsch SVD subroutine during part of its computation, and so we have included such a routine, called GRSVD, in the package to be used with HYBSVD. The routine GRSVD is actually a slightly modified version of the subroutine SVD in the EISPACK [12] package. The main modification that we have made is to eliminate the requirement in subroutine SVD that the row dimension of V declared in the calling program be equal to that of A. This minimized the storage requirements of GRSVD at the cost of one more argument in the argument list.

There is one additional feature implemented in HYBSVD (and also in GRSVD). In least squares applications, where we have overdetermined linear system $Ax = b$, the left transformations U^T have to be accumulated on the right-hand-side vectors b (there may be more than one b). This can be done by putting the vectors b in the matrix argument B when calling HYBSVD and setting IRHS to the number of b's. This is analogous to routine MINFIT in EISPACK.

The calling sequences and usages of HYBSVD are explained in the comments in the beginning of the subroutine.

7. COMPUTATIONAL RESULTS

The conclusions in Section 5 hold only if m and n are both large. In this section some computational experiments are carried out to see if the conclusions are still valid for matrices with realistic sizes.

We computed the SVD of some randomly generated matrices using both HYBSVD and the SVD routine in EISPACK [12].

All tests were run on the IBM 370/168's at the Stanford Linear Accelerator Center (SLAC). Long precision was used throughout the calculation. The mantissa of a floating-point number is represented by 56 bits (approximately 16 decimal digits). The FORTRAN H optimizing compiler (OPT = 2) was used throughout.

For each of the four cases, we fixed some values for n and computed the SVD of a sequence of randomly generated matrices with different values of r. The execution times taken by HYBSVD and EISPACK SVD were then compared, together with the accuracies of the computed answers. Since we are working in a multiprogramming environment, the execution times we measured cannot be taken as the actual computing time taken, although the timing was done with a lot of care; for example, by averaging over large samples. Thus, keeping these points in mind, we can still expect a qualitative agreement with the analysis based on operation counts.

On the IBM 370/168's at SLAC, a floating-point multiplication takes only about 1.5 times the work taken for a floating-point addition. Also, the cost of array indexing is not negligible. Therefore, as noted in Section 5, we should use a value for C that is considerably less than 4 for the purpose of comparing the relative efficiency of the two algorithms on the basis of computational results.

The results of the computations are plotted in Figure 3a–d. In general, they agree very well qualitatively with the asymptotic results we obtained by operation counts, the best fit being with $c \cong 3$. We observe that the larger n is, the better the agreement, as it should be. However, even when n is small, the theoretical results based on asymptotic operation counts still describe very well the qualitative behavior of the computational results in many cases. The computational results also show that large savings in work are indeed realizable for reasonably sized matrices, and that indeed HYBSVD becomes more efficient than GRSVD when $r \gtrsim 2$.

We also checked the accuracies of some of the computed results. The singular values returned by both procedures HYBSVD and EISPACK SVD agree to within a few units of the machine precision in almost all the cases that we have tested. The matrices U and V also agree to the same precision, but the signs of the corresponding columns may be reversed. However, the SVD is only unique to within such a sign change, so this is acceptable [13].

Note Added in Proof: Subsequent comparison tests were performed with routine SSVDC of LINPACK [3a]. The results were similar.

8. CONCLUSIONS

We have presented an improved algorithm and its FORTRAN implementation HYBSVD, for computing the SVD. We have demonstrated that the HYBSVD routine works substantially better than the EISPACK SVD routine for matrices that have many more rows than columns ($m \gtrsim 2n$), and since it uses the EISPACK SVD routine when $m \lesssim 2n$, it is as efficient as the EISPACK SVD routine in those cases. It is also as accurate as the EISPACK SVD routine. We

have also seen that the cost of solving a least squares problem by HYBSVD can often be less than twice that of the usual orthogonal triangularization algorithms. It may therefore become economically feasible to solve many least squares problems by SVD.

The author hopes that the improved algorithm can be included in future versions of mathematical software packages, such as EISPACK.

ACKNOWLEDGMENTS

The author would like to thank John Gregg Lewis, Gene Golub, Charles Van Loan, and Bill Coughran for their helpful discussions. The following persons also helped at one time or another: C. Lawson, R. Hanson, M. Gentleman, J. Oliger, P. Gill, J. Dennis, J. Bolstad, and J. S. Pang. Finally, the author thanks SLAC for providing the computing time that was used.

REFERENCES

1. ANDREWS, H.C, AND PATTERSON, C.L. Singular value decompositions and digital image processing. *IEEE Trans Acoust., Speech, Signal Processing ASSP-24*, 1 (Feb 1976).
2. BARTELS, R.H., GOLUB, G.H, AND SAUNDERS, M.A. Numerical techniques in mathematical programming. In *Nonlinear Programming*, Academic Press, New York, pp. 123–176.
3. CHAN, T.F. On computing the singular value decomposition. Rep. STAN-SC-77-588, Computer Science Dep. Stanford Univ., Stanford, Calif., 1977.

3a. DONGARRA, J. J., BUNCH, J. R., MOLER, C. B., AND STEWART, G. W. *LINPACK Users's Guide* SIAM, Philadelphia, 1979.

4. GENTLEMAN, W M Least sqaures computations by Givens transformations without square roots. Rep. CSRR-2062, Univ. of Waterloo, Waterloo, Ont, Canada.
5. GOLUB, G.H., AND REINSCH, C Singular value decomposition and least squares solutions. In *Handbook for Automatic Computation*, II, *Linear Algebra* J.H. Wilkinson and C. Reinsch (Eds.), Springer-Verlag, New York, 1971
6. GOLUB, G.H., AND KAHAN, W. Calculating the singular values and pseudoinverse of a matrix *SIAM J Numer. Anal 2*, 3 (1965), 205–224.
7. GOLUB, G.H, AND WILKINSON, J.H. Ill-conditioned Eigensystems and the computation of the Jordan canonical form. *SIAM Rev. 18*, 4 (Oct. 1976).
8. GOLUB, G H, AND BUSINGER, P.A. Linear least squares solution by Householder transformations *Numer. Math. 7* (Handbook Series Linear Algebra), 1965, pp. 269–276.
9. HANSON, R.J. A numerical method for solving Fredholm integral equations of the first kind using singular values. *SIAM J. Numer. Anal 8*, 3 (1971), 616–626.
10. KAUFFMAN, L. Application of Householder transformations to a sparse matrix. Computer Science Tech. Rep. No. 63, Bell Laboratories, Murray Hill, N.J., Nov 1977.
11 LAWSON, C.L, AND HANSON, R.J *Solving Least Squares Problems.* Prentice-Hall, Englewood Cliffs, N.J., 1974.
12. SMITH, B.T., ET AL. *Matrix Eigensystem Routines—EISPACK Guide*, 2nd ed. (*Lecture Notes in Computer Science Series*), Springer-Verlag, New York, 1976.
13 STEWART, G.W. *Introduction to Matrix Computations* Academic Press, New York, 1973.

Received October 1976; revised July 1978; accepted December 1980