CS 339 Project:
The SVD of a Matrix Product

Fabian R. Lischka*

August 29, 2003

Id: SVDproduct.tex,v 1.5 2003/08/29 18:05:54 frl Exp

Contents

*Scientific Computing and Computational Mathematics, Stanford University

1 Introduction

The computation of the SVD of a matrix A is often preceded by the reduction of A to bidiagonal form $B = U_B^T A V_B$. We consider the case that A is a product of square matrices $A = A_K \cdots A_2 A_1$. The idea pursued here, as presented in [2], is to compute the bidiagonal form of A implicitly, using Householder reflections, without ever forming the product explicitly. The flop count (about $(4 + 4K)N^3$ for the bidiagonalization) is very favorable compared to other methods that require about 3 to 10 times more work.

Using a few simple test cases with known singular values, we show that the accuracy of computed singular values—particular small ones—is vastly superior to computing the SVD from the explicitly formed product.

The algorithm with all the basic building blocks has been implemented here, and this note contains a description and all the source code.

2 The Normal SVD

The first thing to realize when computing the singular values of a matrix A is that singular values are invariant with respect to arbitrary multiplications (from either or both sides) with orthogonal matrices.[1] Hence, a reasonable first step in computing the SVD is the reduction of A to much simpler form. In particular, using Householder reflections from left and from right, we can bring A to bidiagonal form (main diagonal and one superdiagonal) $B = U_B^T A V_B$. We have then reduced the number of relevant elements from N^2 to $2N - 1$!

The total flop count for this bidiagonalization is $16/3N^3$ for square matrices, or $8/3N^3$ if U, V are not explicitly formed.

Then, in a second step B will be iteratively decomposed into $B = U_\Sigma \Sigma V_\Sigma^T$, using Golub-Kahan SVD steps, which apply the QR algorithm implicitly to the symmetric tridiagonal matrix $T = B^T B$. Therefore, we next describe the QR algorithm for the symmetric eigenvalue problem. The SVD of A is finally given by $(U_B U_\Sigma)^T A V_B V_\Sigma = \Sigma$.

2.1 Bidiagonalization

Given $A \in \mathbb{R}^{M \times N}$, we seek orthogonal U_B, V_B and bidiagonal B such that $B = U_B^T A V_B$. To achieve this, we apply a series of Householder reflections alternating from left and right.

2.1.1 Householder Reflections

Householder reflectors are orthogonal and symmetric matrices[2] that can be used to reflect a vector onto the first coordinate axis (and hence eliminate all vector entries but the first). The construction is simple with a few subtleties in the implementation. Conceptually, we modify a projector. As is well known, a projector $P = \frac{\mathbf{v}\mathbf{v}^T}{\mathbf{v}^T\mathbf{v}}$ projects a vector onto the line spanned by

[1] The singular vectors can be recovered by multiplication with the same orthogonal matrices.

[2] This implies that they are their own inverse.

$\mathbf{v}$, while $P^\perp = I - \frac{\mathbf{v}\mathbf{v}^T}{\mathbf{v}^T\mathbf{v}}$ projects $\mathbf{v}$ onto the hyperplane orthogonal to $\mathbf{v}$. Now, if we extend the perpendicular line beyond that hyperplane, we obtain the reflection $H = I - 2\frac{\mathbf{v}\mathbf{v}^T}{\mathbf{v}^T\mathbf{v}}$. For a given vector $\mathbf{x}$, we now aim to find $\mathbf{v}$ such that $H\mathbf{x} = \|\mathbf{x}\|_2 \, \mathbf{e}_1$. This is easily done, the vector $\mathbf{v}$ we choose is just $\mathbf{x} - \|\mathbf{x}\|_2 \, \mathbf{e}_1$. Then, reflecting $\mathbf{x}$ across the hyperplane orthogonal to $\mathbf{v}$ will bring it onto $\mathbf{e}_1$.

We want to avoid cancellation in computing this difference when $\mathbf{x}$ is already close to the first coordinate axis. There are two ways of dealing with this case: Either, one then reflects not onto the positive coordinate axis $\|\mathbf{x}\|_2 \, \mathbf{e}_1$, but onto the negative part $-\|\mathbf{x}\|_2 \, \mathbf{e}_1$, so choose $\mathbf{v} = \mathbf{x} + \|\mathbf{x}\|_2 \, \mathbf{e}_1$ if e. g. the first element x_1 of $\mathbf{x}$ is positive, and the original if $x_1 \leq 0$. (In other words, reflect to that half of the first coordinate axis that is further away). Alternatively, compute $x_1 - \|\mathbf{x}\|_2$ using $x_1 - \|\mathbf{x}\|_2 = \frac{x_1^2 - \|\mathbf{x}\|_2^2}{x_1 + \|\mathbf{x}\|_2} = \frac{-\sum_{i=2}^n x_i^2}{x_1 + \|\mathbf{x}\|_2}$.

To facilitate computation and storage, we can norm $\mathbf{v}$ to have $v_1 = 1$, and compute $\beta = \frac{2}{\mathbf{v}^T\mathbf{v}}$, so that $H = I - \beta\mathbf{v}\mathbf{v}^T$.

The flop count of determining the Household vector $\mathbf{v}$ is around $3N$. An implementation in Matlab can be found on page 17.

2.1.2 Computing the Bidiagonalization

We can now proceed to bring A into bidiagonal form like this: First, multiply A with U_1 from left, eliminating the column below A_{11}. Then, multiply $A(1:M, 2:N)$ with V_1 from the right, eliminating the first row beyond A_{12}. Then, $A(2:M, 2:N)$ with U_2 from left, eliminating the column below A_{22}. Then, $A(2:M, 3:N)$ with V_2 from the right, eliminating the first row beyond A_{23}. After $N-2$ double steps like this, everything below $A_{N-2,N-2}$ and beyond $A_{N-2,N-1}$ is eliminated, we can now apply one more U_{N-1} from the left to eliminate the column below $A_{N_1,N-1}$, and if $M > N$, apply one last step U_N from the left to erase the remaining last column below the square.

In total, we have $2N-2$ householder multiplications working on progressively smaller submatrices, for a total flopcount (without explicitly accumulating U and V) of $4N^2(M - N/3)$, that is $8/3N^3$ for square matrices. Accumulating U and V costs $4(M^2N - N^2M + N^3/3)$ and $4N^3/3$ flops, respectively, and hence doubles the workload for square matrices.

An implementation can be found on page 19.

Note that there is an alternative algorithm that is faster for $M \gg N$. It first computes a QR-decomposition of A, and then bidiagonalizes the square matrix R, for a flop count of $2MN^2 + 2N^3$, so theoretically cheaper for $M \geq 5/3\,N$. We will not pursue this algorithm here.

2.2 The QR Algorithm for the Symmetric Eigenvalue Problem

Given A symmetric, real, we compute $T = Q^T AQ$ tridiagonal and similar to A, using Householder tridiagonalization. Now we perform QR steps on T:

$$\begin{aligned} QR &= T \\ (\text{so } R &= Q^T T) \\ T_+ &= RQ = Q^T TQ \end{aligned}$$

or, with shift μ,

$$\begin{aligned} QR &= T - \mu I \\ (\text{so } R &= Q^T T - \mu Q^T) \\ T_+ &= RQ + \mu I = Q^T TQ \end{aligned}$$

We either perform these steps explicitly, or, appealing to the implicit-Q theorem, implicitly. From the equations above it is apparent that T_+ is similar to T and hence has the same characteristic polynomial, hence eigenvalues.[3]

2.2.1 Tridiagonalization

Tridiagonalization of a symmetric matrix A proceeds similar as bidiagonalization above. However, here we apply the same transformation on the left and right, since we aim to preserve not only singular values, but eigenvalues (hence we can achieve only upper Hessenberg form, or tridiagonal form in the symmetric case).

An optimal implementation requires $4/3\,N^3$ flops without, and twice that, $8/3\,N^3$ flops with accumulating Q. However, the implementation on page 18 requires somewhat more, as it does not exploit symmetry in A when performing the rank-one-update, thus operating on the whole square, instead of only a triangle.

[3] The explanation why the algorithm converges is wide beyond the scope of this note. However, one can intuitively regard the QR algorithm as a power iteration performed with many vectors simultaneously that are coerced to remain orthogonal. Hence, one sees that the eigenvector corresponding to the largest eigenvalue will emerge, and the neighbouring vector (being orthogonal) will converge to the eigenvector associated with the second largest eigenvalue, etc. And given that, keeping in mind the Rayleigh quotient, it is apparent why the diagonal elements converge to the eigenvalues, while the off diagonals go to zero.

2.2.2 Explicit QR with Shift

When computing an iteration as above, T being tridiagonal allows for a cheaper QR decomposition using Givens rotations (and the essential uniqueness of the QR decomposition), as shown on page 24. We can use (orthogonal!) Givens rotations from the left to eliminate the subdiagonal elements of T, and obtain an uppertriangular matrix R, hence we have $R = G_{N-1}^T \dots G_2^T G_1^T T$, so $QR = T$ with $Q = G_1 G_2 \dots G_{N-1}$.

Remarkably, the total flop count for computing R alone is linear, around $25N$, but accumulating Q requires around $3N^2$ flops - still a vast improvement over the cubic order of full QR ($4/3/, N^3$ without Q, $8/3/, N^3$ with Q).

A simple listing for an explicit QR step can be found on page 25, the QR decomposition for symmetric tridiagonal matrices is on page 24.

2.2.3 Implicit QR with Shift

By the implicit Q theorem, we know that for symmetric real A the transformation to tridiagonal $T = Q^T A Q$ is essentially unique (modulo signs), *given the first column* $\mathbf{q}_1$ *of* Q. Now, in the case above, we seek $T_+ = RQ + \mu I = Q^T T Q$, and find it by explicitly computing the required Q. However, by the implicit Q theorem, if we find *any* orthogonal G such that $G^T T G$ is tridiagonal, and the first column of G is "correct", we know that we have actually found the "correct" $Q = G$ (modulo signs), so we do not need to bother about the QR decomposition. Two questions remain: What is the "correct" first column of G? And how do we find the rest of G?

Finding the correct first column

For the "correct" G, we require $GR = T - \mu I$ for some upper triangular R, in other words, the first column of G is just a scaled version of the first column on the RHS $T - \mu I$, that is

$$\mathbf{g}_1 = \frac{\mathbf{t}_1 - \mu \mathbf{e}_1}{\|\mathbf{t_1} - \mu \mathbf{e_1}\|_2}.$$

So, at the danger of multiple redundant repetition, if we find G with this first column such that G orthogonal and $G^T T G$ tridiagonal, we are done.

Finding the rest of G

Suppose now that we compose all of G as a series of Givens rotations, that is $G = G_1 G_2 \dots G_{N_1}$ with G_k acting (only) on k and $k+1$ - both rows

and columns, of course, since we will multiply from both sides. Then

$$T_+ = G_{N-1}^T \dots G_2^T G_1^T T G_1 G_2 \dots G_{N-1}$$
$$G = G_1 G_2 \dots G_{N-1}$$

Consider the first column of G - it will be determined exclusively by G_1. Hence, we can choose the rotation matrix G_1 such that its first column is parallel to the first column of $T - \mu I$, as explicated above. What effect will that have on the intermediate $T_1 = G_1^T T G_1$? It will fill out two elements outside the sub/superdiagonals, namely t_{13} and t_{31} - the famous "bulge".

But, note that we can eliminate these elements by judicious choice of G_2! Then they will reappear, one element further down — at position t_{24} and t_{42} — and can be chased down with further Givens rotations. After all $N-1$ rotations, we end up with a tridiagonal $T_+ = G^T T G$, G orthogonal, and G_1 chosen such that $\mathbf{g}_1$ parallel to $T - \mu \mathbf{e}_1$, as required - hence, by the implicit Q theorem, we have found "the same" T_+ as if we had used the explicit QR step (note that the signs of the off-diagonal elements might differ somewhat, by they would have depended on the particular choice of Q (ie the signs of the diagonal of R) anyway).

The flop count for an implicit symmetric QR step is about $32N$ without accumulating Q, and if Q has $M \geq N$ rows (if it is passed in from the outside to be updated, for example), about $6MN$ for accumulating Q.

An implementation of the implicit QR step can be found on page 25.

2.2.4 Finding the Correct Shift

One reasonable choice for the shift is $mu = T_{nn}$. However, Wilkinson gives heuristic reasons why to prefer the eigenvalue of

$$T(n-1:n, n-1:n) = \begin{pmatrix} a_{n-1} & b_{n-1} \\ b_{n-1} & a_n \end{pmatrix}$$

that is closer to a_n. Now, these eigenvalues are given by

$$\lambda_\pm = \frac{a_{n-1} + a_n}{2} \pm \sqrt{b_{n-1}^2 + \left(\frac{a_{n-1} - a_n}{2}\right)^2},$$

and the one closer to a_n is given by

$$\lambda_n = \frac{a_{n-1} + a_n}{2} - \operatorname{sign}(d) \sqrt{b_{n-1}^2 + d^2},$$

where $d = (a_{n-1} - a_n)/2$. To avoid cancellation, use

$$\lambda_n = a_n - \frac{b_{n-1}^2}{d + \operatorname{sign}(d)\sqrt{b_{n-1}^2 + d^2}}.$$

For $d = 0$, the eigenvalues are given by $\lambda_\pm = a_n \pm |b_{n-1}|$. To avoid cancellation, we use the one that is absolutely greater. The flop count for determining mu is around 15 and a square root, an implementation is on page 22.

2.3 The Complete Symmetric QR Algorithm

The whole algorithm then proceeds as follows:

1. Compute tridiagonal $T = Q^T A Q$
2. Set negligible off-diagonal elements to zero
3. Determine a submatrix T_2 of T such that T_2 unreduced, and T diagonal below T_2
4. Apply an implicit QR step on T_2, update Q if desired
5. Repeat from step 2. until T diagonal

The flop count is about $4/3\,N^3$ without accumulating Q, and $9N^3$ with accumulating Q. For details, we refer to [1, section 8.2]. An implementation is on page 26.

2.4 The Relation Between SVD and the Symmetric Eigenvalue Problem

For $A \in \mathbb{R}^{M\times N}$, the singular values of A are equal to the square roots of the eigenvalues of $A^T A$, since with

$$\begin{aligned} A &= U\Sigma V^T, \\ A^T A &= V\Sigma U^T U\Sigma V^T = V\Sigma^2 V^T. \end{aligned}$$

Note that $A^T A$ is symmetric, so one could compute it, and then apply the symmetric QR algorithm from the last section to find its eigenvalues. However, explicitly computing $A^T A$ leads to a squaring of the condition number, and a subsequent loss of information (even if $A^T A$ is not necessary for computing U).

2.4.1 The Golub-Kahan SVD Step

A better way is to apply the QR steps implicitly. To do this, we proceed as follows: First, determine bidiagonal $B = U_B^T A V_B$. Then $T = B^T B$ is tridiagonal, symmetric. The implicit QR step would now compute mu from the bottom 2×2 submatrix of T, and then find $T_+ = Q^T T Q$ such that $QR = T - \mu I$. Similarly, we seek a new bidiagonal B, $B_+ = U^T B V$. Then $T_+ = B_+^T B_+ = V^T B^T U U^T B V = V^T B^T B V = V^T T V$. Again, we can now invoke the implicit-Q theorem:

If we find orthogonal U, V with

1. $T_+ = V^T T V$ is tridiagonal—or, sufficiently: $B_+ = U^T B V$ is bidiagonal

2. First column of V is proportional to first column of $T - \mu I$,

then we have our next B_+ and hence also T_+, without ever explicitly computing T or T_+.

These steps are known as Golub-Kahan SVD steps [1, section 8.6.2]. If we terminate this second part of the algorithm with $\Sigma = U_\Sigma B V_\Sigma$, then our SVD of A is given by $(U_B U_\Sigma)^T A V_B V_\Sigma = \Sigma$.

2.4.2 The Complete SVD Algorithm

The whole algorithm then schematically proceeds as follows:

1. Compute bidiagonal $B = U_B^T A V_B$

2. Set negligible off-diagonal elements to zero

3. Determine a submatrix B_2 of B such that B_2 unreduced, and B diagonal below B_2

4. Apply an Golub-Kahan SVD step on T_2, and update U, V if desired

5. Repeat from step 2. until B diagonal

The total flop count for a Golub-Kahan SVD of a square matrix is about $8/3\, N^3$ if U, V are not desired, and $21 N^3$ if they are.

An implementation of the Golub-Kahan SVD (also known as Golub-Reinsch) SVD can be found on page 32. While it is certainly not production quality (cancellations are mostly avoided, but overflow is not guarded against), it appears to deliver comparable accuracy on a range of test matrices to the MATLAB implementation svd().

As the algorithm is structured here, it shows very nicely the separation of the first step, computing the bidiagonalization of A, and the second, iteratively diagonalizing B. This structure will now be exploited for the computation of the SVD of a product of square matrices.

3 The SVD for a Matrix Product

We now consider the SVD for products $A = A_K \cdots A_k \cdots A_1$. The crucial idea is that instead of computing the product, and then bidiagonalize and proceed to find the SVD, we will implicitly bidiagonalize it, that is find the bidiagonal form of the product without ever explicitly computing the product. Once the bidiagonal form is computed, we can use any desired technique to find the SVD from there.

3.1 Bidiagonalizing a Matrix Product

We aim to find orthogonal U, V such that $U_B^T A V_B = B$ is bidiagonal, and without reference to the actual product A. We note that we can intersperse the decomposed identity $I = QQ^T$ between the A_k, such that $B = U_B^T A V_B = U_B A_K Q_K Q_K^T A_{K_1} \cdots Q_2^T A_2 Q_1 Q_1^T A_1 V_B$. How can we choose these intermediate matrices? Well, by choosing Q_1 judiciously, we can make $Q_1^T A_1$ tridiagonal. Then we can choose Q_2 as to make $Q_2^T A_2 Q_1$ tridiagonal, and so on, up to a choice of U_B as to make $U_B^T A_K Q_K$ tridiagonal.

However, then all $U_B^T A_K Q_K, \ldots, Q_2^T A_2 Q_1, Q_1^T A_1 Q_0$ are tridiagonal, hence their product, and then we can choose (intermediate) rotations from the right such as to make the whole product bidiagonal.

Having run through the iteration as described below in more detail, we can then compute B—it depends only on the diagonal and superdiagonal of all the tridiagonal matrices formed above. In particular, suppose we have the diagonal **q** and superdiagonal **e** of the product of all tridiagonal matrices T to the right of matrix k, then we can obtain **q** and **e** of the product up to and including T_k through this nice recurrence:

```
d = diag( T_k );
e = e .* d(1:N−1) + q(2:N) .* diag( T_k, 1 );
q = q .* d;
```

3.1.1 Implementation

For implementation purposes and to define notation, we note down a few invariances. The indices are $k = 1, \ldots, K$ for the matrices, $i = 1, \ldots, N$ for

the rows, similarly j for columns, and $t = 0, 1 \ldots, N$ for (time) steps. Initial values are designated e.g. $A_k^{(0)}$, after the first step of the algorithm we would have $A_k^{(1)}$, etc.

$$
\begin{aligned}
Q_K^{(0)} &= I \\
Q_0^{(1)} &= I \\
A_k^{(t)} &= Q_k^{(t)*} \cdots Q_k^{(2)*} Q_k^{(1)*} A_k^{(0)} Q_{k-1}^{(1)} \cdots Q_{k-1}^{(t)} \\
U^{(t)} &= Q_K^{(1)} \cdots Q_K^{(t)} \\
V^{(t)} &= Q_0^{(1)} \cdots Q_0^{(t)} \\
A_K^{(0)} \cdots A_1^{(0)} &= A \\
A_K^{(t)} \cdots A_1^{(t)} &= Q_K^{(t)*} \cdots Q_K^{(1)*} A_K^{(0)} A_{K-1}^{(0)} \cdots A_2^{(0)} \cdot A_1^{(0)} Q_{k-1}^{(1)} \cdots Q_{k-1}^{(t)} \\
&= U^{(t)*} A V^{(t)}
\end{aligned}
$$

Now, all $Q_k^{(t)}$ are Householder reflections, hence orthogonal and symmetric. Hence, all $U^{(t)}$ and $V^{(t)}$ are orthogonal. Furthermore, we will choose the $Q_k^{(t)}$ such that all $A_k^{(t)}$ are upper triangular up to (and including) column t, in other words, the elements below $a_{11}, a_{22}, \ldots, a_{tt}$ are zero.

Then, the product $A_K^{(t)} \cdots A_1^{(t)}$ of these matrices is also upper triangular up to (and including) column t, so in particular, $U^{(t)*} A V^{(t)}$ is upper triangular. Last but not least, $Q_0^{(t+1)}$ will be chosen such that $U^{(t)*} A V^{(t+1)}$ is bidiagonal up to and including column t, that is all elements to the right of $a_{12}, a_{23}, \ldots, a_{tt+1}$ are zero.

A complete implementation is on page 20. It requires around $(4+4K)N^3$ flops.

3.2 Accuracy

We tested the algorithm against a few cases in which the singular values are explicitly known, in an attempt to replicate the argument in [2]. In particular, we chose two cases: A product of matrices with very high dynamical range of singular values, and with orthogonal transformations chosen such $(\cdots A_2 Q Q^T A_1)$ that they all canceled out in the product, so that the final matrix product was $U \Sigma^K V$, and hence had singular values σ_k^K.

These matrices were generated with this code snippet:

```
S = diag(2.^(-1:-1:-N));
```

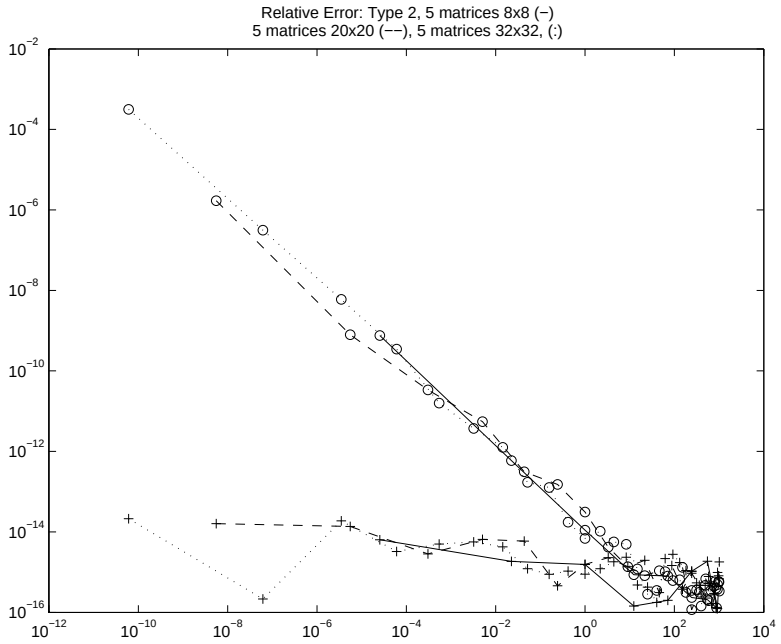

Figure 1: The explicit algorithm (o) has far higher relative errors than the implicit one (+). Here, the product of 5 Toeplitz matrices is plotted for different matrix sizes.

```
[V,X]=qr(randn(N));   % generate random orthogonal V
for k=1:K
        [U,X]=qr(randn(N));
        A (:,:, k) = U'*S*V;
        V=U;
end;
true = 2.^(K*(-1:-1:-N))';
```

In the next case, we chose symmetric tridiagonal Toeplitz matrices. If the columns contains a, and the sub- and superdiagonal b, then the eigenvalues (and singular values) are known to be

$$a + 2b\cos\left(\frac{i\pi}{N+1}\right), \quad i = 1 \ldots N,$$

and since these matrices are normal, the singular values of the power are just the powers of the singular values.

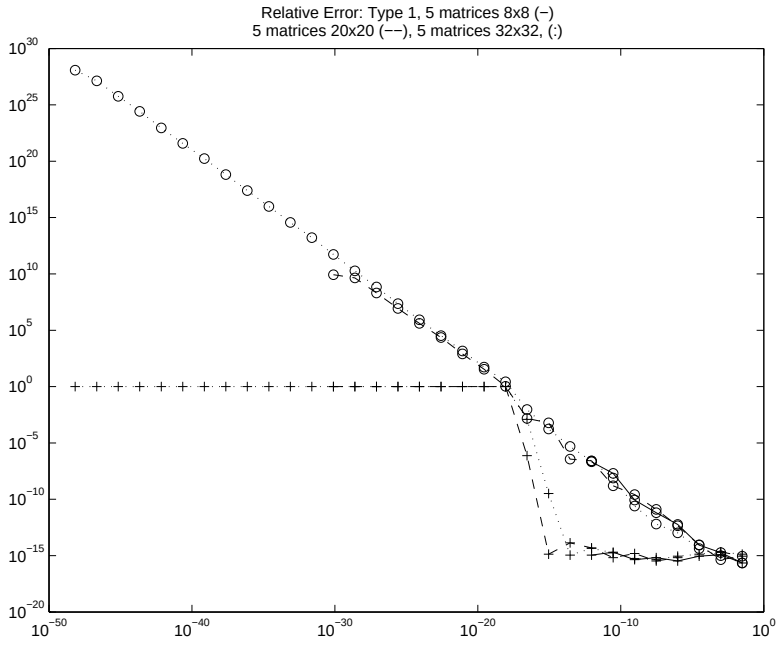

Figure 2: The implicit relative errors (+) are still far better, but display a plateau at 1—the computed singular value is zero. 5 matrices of type one are plotted for different sizes.

In the figures, we compare the (log) relative error of the implicitly computed singular values (marked with a +) with the relative error of the explic-

itly computed singular values (marked with a o), ie. with the true singular value σ_i, we plot $\frac{|\sigma_i^{impl} - \sigma_i|}{\sigma_i}$ for $i = 1 \ldots N$ versus σ_i, and similarly with the explicitly computed ones. We observe that the relative accuracy of the explicitly computed singular values plummets precipitously—these values are useless.[4]

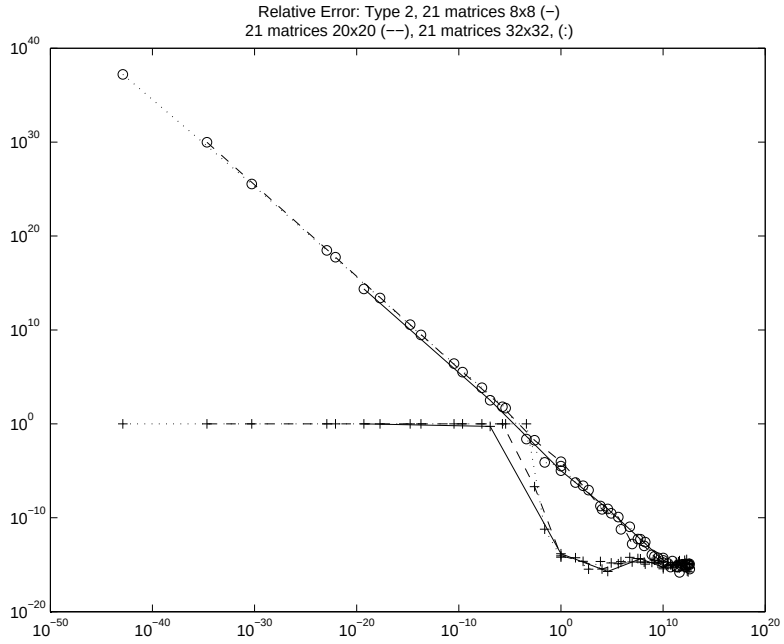

Figure 3: The explicit algorithm (o) has still far higher relative errors of the computed singular values than the implicit one (+).

Another observation to make is that the explicitly computed singular values deteriorate when we increase the number of matrices in the product, see figure 4 on page 15. The implicitly computed singular values are far more robust with respect to the number of factors.

[4]We also notice a peculiarity with the implicitly computed singular values: the relative error reaches a plateau at 1, for example in figure 2 on page 13. This is because the implicitly computed singular value is exactly zero. We have investigated this, but not reached a conclusion yet. The exact zero appears already after the bidiagonalization, so it is not an artefact of the tolerances used in the GK SVD step, where we deliberately set small values to zero. Whatever the reason for this plateau is, obviously a relative error of 1—as big as it is—is quite a bit better than relative errors of 10^{40}, as displayed by the explicit calculation.

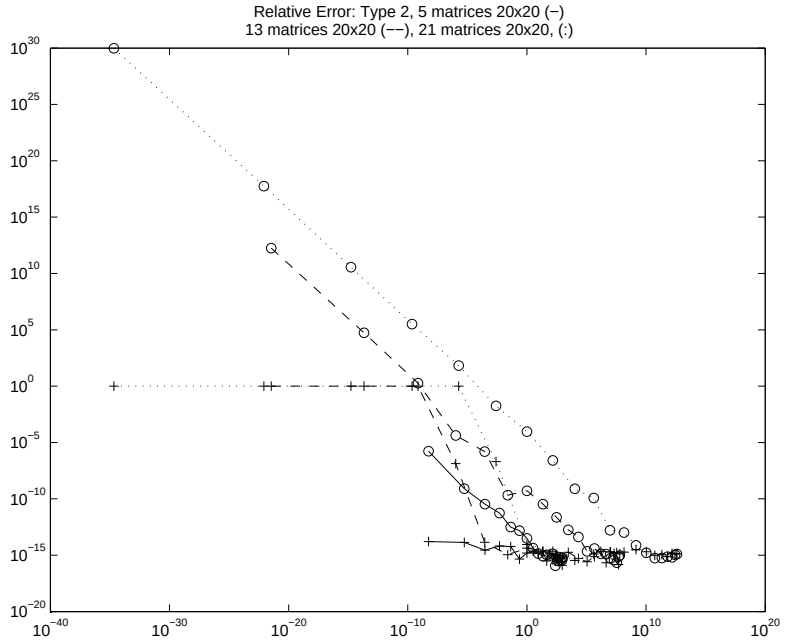

Figure 4: When the relative errors are plotted for constant matrix size 20, but more and more matrices, we observe that the explicitly computed singular values (o) deteriorate, but the implicitly computed ones (+) don't.

4 Testing

All algorithms were tested with respect to internal consistency (that is, are matrices that are supposed to be orthogonal really orthogonal? When the algorithm computes a decomposition, does the composition of the returned matrices recover the original matrix?), as well as by comparison to the reference solutions computed by MATLAB.

The test matrices were generated with *gentestmat*, page 33. They include totally random matrices, as well as matrices designed to have a very large dynamical range of singular values, and matrices that are badly conditioned - either singular or very close to singular. In addition, for the QR algorithm and the GK SVD algorithm, we also construct bi- or tridiagonal matrices with a certain pattern of zeros and non-zeros on the superdiagonal, and iterate through all possible combinations, in order to verify correct decomposition into diagonal and unreduced matrices when deflating the problem.

This facilitated finding certain bugs. For example, if the '$\leq$' in line 21 in algorithm *gksvdsteps* on page 29 is replaced by a '$<$', the algorithm runs into an infinite loop in certain cases (namely if the surrounding off-diagonal elements are (or have been set to) zero).[5]

[5] Of course, careful reading of [1, page 455] would have prevented this in the first place...

A Source Code

A.1 Bidiagonalization

A.1.1 *house* finds the Householder vector

```
function [ v, beta, mu ] = house( x )
% HOUSE computes v with v(1) = 1 and beta and mu such
% that H = I - beta v v' (symmetric, orthogonal) reflects x onto
% (the positive part of) the first coordinate axis: H*x = mu*e_1,
% with mu = 2-norm( x ) = sqrt( x'x ) >= 0, and v = x - mu * e_1.
%
% $Id: house.m,v 1.6 2003/08/27 09:21:47 frl Exp $

sigma = x(2:end)'*x(2:end);                    % flops: 2N
v     = x;                                     % mem copy: N
if isempty( sigma ) | sigma == 0
    if x(1) >= 0
        beta = 0;
    else
        % Note: we make this choice so that the invariance H*x = mu*e_1
        % always holds (so in this case, H*x = -x = abs(x(1))*e_1)
        % otherwise we get sign errors in certain cases in routines that
        % use house and rely on that behaviour (eg. if we don't multiply
        % the first column, but just set it [mu 0 0 0 0]')
        beta = 2;
    end;
    v(1) = 1;
    mu   = abs( x(1) );    % == norm( x );
else     % note: here, we always choose v = x - norm(x) * e1
    mu = sqrt( x(1)^2 + sigma ); % == norm( x )
    if x(1) <= 0
        v(1) = x(1) - mu;
    else
        v(1) = -sigma/( x(1) + mu ); % this is also x(1) - mu,
        % but because of potential cancellation written as
        % (x(1)^2 - mu^2)/(x(1)+mu)
    end;
    beta = 2*v(1)^2/( sigma + v(1)^2 );        % flops: 5
    v = v./ v(1);                              % flops: N
end;

% HH reflector now: I - beta * v * v'
% flop count: about 3N (+ 10)
```

Listing 1: **house** finds the Householder vector

A.1.2 *symtridhh* reduces a symmetric matric to tridiagonal form

```
function [ A, Q ] = symtridhh( A )
% SYMTRIDHH brings a symmetric real matrix A in similar tridiagonal form
% using similar householder transformations, T = Q'AQ

% reference: Golub, Van Loan; 3rd ed; ch. 8.3.1 and 5.1.6.
% $Id: symtridhh.m,v 1.7 2003/08/24 22:21:15 frl Exp $

if any(any(A ~= A'))
    error('Input must be symmetric!')
end
N = size( A, 1 );

for k = 1:(N-2)
    [ v, beta, mu ] = house( A( (k+1):N, k ) );                    % flops: 3(N-k)
    p = A( (k+1):N, (k+1):N )*(beta*v);                            % flops: 2(N-k)^2
    w = p - (beta/2*p'*v)*v;                                       % flops: 4(N-k)
    A( k, (k+2):N ) = 0              ;   A(k,k+1) = mu;            % mem: N-k
    if nargout == 2
        % collect the v
        A( k+1:N, k )  = v           ;   A( k+1, k ) = beta;
    else
        % else leave A symmetric, tridiagonal
        A( (k+2):N, k ) = 0          ;   A(k+1,k) = mu;            % mem: N-k
    end;
    % FIX: take advantage of symmetry here
    A( (k+1):N, (k+1):N ) = A( (k+1):N, (k+1):N ) - v*w' - w*v';
    % flops: without taking advantage of sym:                      % flops: 3(N-k)^2
end;

if nargout == 2
    % accumulate Q
    Q = eye( N );
    for k = (N-2):-1:1
        v           = A( k+1:N, k );
        beta        = v( 1 );
        v( 1 )      = 1;
        Q(k+1:N, k+1:N) = Q( k+1:N, k+1:N ) - beta * v * v' * Q( k+1:N, k+1:N );
        % and fix the A
        A(k+1:N, k) = A(k, k+1:N )';
    end;
end;

% total flops: sum k=1:N of 5(N-k)^2 = sum k=1:N of 5N^2
% total flops: = 5/3 N^3 + lower order terms
```

Listing 2: **symtridhh** reduces a symmetric matric to tridiagonal form using Householder reflections, preserving eigenvalues

A.1.3 *bidighh* reduces a matrix to bidiagonal form

```
function [ B, U, V ] = bidighh( A );
% BIDIGHH computes bidiagonal B=U'AV using householder reflections
% suppose A MxN, M >= N, A real. This routine finds orthogonal U MxM, V NxN
% such that B = U'AV is bidiagonal (the diagonal and one superdiagonal).
% The diagonal is stored in the  first  col of B, the superdiagonal in the
% second col of B, and B(end,2) == 0

% reference: Golub, Van Loan; 3rd ed; 5.4.3
% $Id: bidighh.m,v 1.14 2003/08/29 15:15:34 frl Exp $

M = size( A, 1 );    % rows
N = size( A, 2 );    % cols
if M < N
    error( 'A must have M >= N' );
end;

B      = A;                % B will be M x N
betas  = zeros( N, 1 );
gammas = zeros( N-2, 1 );

for k = 1:N
    % determine HH reflector for column B(k:M,k)
    [ v, beta, mu ] = house( B(k:M,k) );              % flops: 3(M-k)
    betas( k )      = beta;
    % B(k,k)          = mu;                           % = norm(x)
    B(k:M,k:N)      = B(k:M,k:N) - beta * v * v' * B(k:M,k:N);
                                                      % flops: 4*(N-k)*(M-k)
    B((k+1):M,k)    = v(2:end);
    % or, if you don't want to  collect  it , B((k+1):M,k) = 0;
    % B((k+1):M,k)  = 0;
    if k < N - 1
                [ v, beta, mu ] = house( B(k,(k+1):N)' );   % flops: 3(N-k)
                gammas( k )     = beta;
                % B(k,k+1)        = mu;                     % = norm(x)
        % note that we need to multiply from the right here:
                B(k:M,(k+1):N) = B(k:M,(k+1):N) - beta * B(k:M,(k+1):N) * v * v';
                                                      % flops: 4*(N-k)*(M-k)
                B(k,(k+2):N)    = v(2:end)';
        % or, if you don't want to  collect  it , B(k,(k+2):N) = 0;
        % B(k,(k+2):N)  = 0;
    end
```

```
end;

% total flops (without computing U, V)
% sum k = 1:N of 8(N−k)(M−K) + 6(M−k)
% = 4 MN^2 − 4/3 N^3 = 4 N^2 (M−N/3), or,
% for M=N, 8/3 N^3

if nargout > 1
    % compute U and V
    v = zeros( M, 1 );
    U = eye( M );
    for k = N:−1:1
        % HH reflector now: I − beta ∗ v ∗ v'. Note: v(1) = 1
        v(k)          = 1;
        v(k+1:M)    = B((k+1):M,k);
        U(k:M,k:M) = U(k:M,k:M) − betas( k ) ∗ v(k:M) ∗ v(k:M)' ∗ U(k:M,k:M);
        % flop count: 4∗(M−k)^2
        B(k+1:M,k) = 0;
    end;
    V = eye( N );
    for k = N−1:−1:2
        v(k)          = 1;
        v(k+1:N)    = B(k−1,(k+1):N)';
        V(k:N,k:N) = V(k:N,k:N) − gammas( k−1 ) ∗ v(k:N) ∗ v(k:N)' ∗ V(k:N,k:N);
        % flop count: 4∗(N−k)^2
        B(k−1,k+1:N)= 0;
    end;
end;
B = [ diag(B) [ diag(B,1) ; 0 ] ];

% flop count for V: 4/3 N^3
% flop count for U: 4( NM^2 − MN^2 + N^3/3 )
% total flops (with computing U,V)
% 4 NM^2 + 4/3 N^3, or for M==N, 16/3 N^3 = 5.3 N^3
```

Listing 3: **bidighh** reduces a matrix to bidiagonal form using Householder reflections, preserving singular values

A.1.4 *bidigprod* bidiagonalizes a matrix product implicitly

```
function [ B, U, V ] = bidigprod( A )
% BIDIGHH computes bidiagonal B=U' AAA V using householder reflections.
% suppose A NxNxK, real. Without ever computing the product
% AAA = A(:,:,K)∗...∗A(:,:,2)∗A(:,:,1), this routine finds
% orthogonal U NxN, V NxN such that B = U' AAA V is bidiagonal
% (the diagonal and one superdiagonal).
% B is returned in short format, ie the diagonal
```

```
% is stored in the first col of B, the superdiagonal in the
% second col of B, and B(end,2) == 0

% reference: Golub, Solna, van Dooren: Computing the SVD
% of a General Matrix Product/Quotient,
% SIAM J. MATRIX ANAL. APPL., Vol. 22, No. 1, pp 1-19
% $Id: bidigprod.m,v 1.4 2003/08/29 15:16:08 frl Exp $

N   = size( A, 1 );
if size( A, 2 ) ~= N
    error( 'matrices in A must be square (ie size(A,1)=size(A,2))' )
end;

computevectors = (nargout > 1);

K   = size( A, 3 );
if computevectors
    V   = eye( N );
    A (:,:, K+1) = V;   % this is U
end;

for t=1:N-1
    for k=1:K
        [ v, beta, mu ] = house( A( t:N, t, k ) );   % flops: 3(N-t)
        % I-beta vv' is Q^t_k, applied to t:N
        % multiply Ak from left ...
        A( t:N,t:N, k ) = A( t:N,t:N, k ) - beta*v*v' * A( t:N,t:N, k );
                                                     % (flops: 4(N-t)^2)
                                                     % flops: 4K(N-t)^2
        if k<K || computevectors % execute almost always
            % ...and Ak+1 from right - here we need to multiply all columns,
            % since we don't have the zeros needed in here
            A( :,t:N, k+1 ) = A( :,t:N, k+1 ) - A( :,t:N, k+1 )*v*v'*beta;
                                                     % flops: 4(K-1)N(t-N)
                                                     % with vector: +4N(t-N)
        end;
    end;

    if t < N-1
        % now, determine the t-th row of the product
        row = A( t, t:N, K );
        for k = K-1:-1:1
            row = row * A( t:N, t:N, k );            % (flops: 2(N-t)^2)
        end;
                                                     % flops: 2K(N-t)^2
        % and determine householder to eliminate it!
        % (the diagonal element remains untouched (and is thrown away))
        [ v, beta, mu ] = house( row( 1, 2:end )' );   % flops: 3(N-t)
        % multiply A( 1 ) and V from the right
```

```
        % need to multiply full columns (all rows), since might be filled
        A( :, t+1:N, 1 ) = A( :,t+1:N, 1 ) - A( :,t+1:N, 1 )*v*v'*beta;
                                                    % flops: 4N(t-N)
        if computevectors
            V( :,t+1:N ) = V( :,t+1:N )    - V( :, t+1:N ) *v*v'*beta;
                                                    % flops: 4N(t-N)
        end;
    end;
    % total flops per t:  with vectors (8+4K) N(t-N) + 6K (N-t)^2
    %                     = (4+4K) N^3
    %                     without vectors: (4K) N^3
end;
% now B = A(:,:,K+1)'*AA*V = A(:,:,K)*...*A(:,:,2)*A(:,:,1)


q   = ones(N,1);      % this will be the diagonal of the product B
e   = zeros(N-1,1); % superdiagonal of the product B
% note: to compute q, e, we only need diagonal and superdiag of A !
for k=1:K
    d   = diag( A(:,:,k) );
    e   = e .* d(1:end-1) + q(2:N) .* diag( A(:,:,k), 1 );
    q   = q .* d;
end;

if computevectors
    U = A(:,:,K+1);
end;
B = [ q [e ; 0] ];
```

Listing 4: **bidigprod** bidiagonalizes a matrix product implicitly using Householder reflections

A.2 The QR algorithm

A.2.1 *wilkinsonshift* determines the Wilkinson shift

```
function mu = wilkinsonshift( a1, b, a2 )
% WILKINSONSHIFT returns the eigenvalue of the symmetric matrix
% (a1, b; b, a2) that is closer to a2. Some care is taken to avoid
% cancellation.
% TS N-1,1  a1
% TS N ,1   a2
% TS N-1,2  b

% reference: Golub, Van Loan; 3rd ed; ch. 8.3.4
% $Id: wilkinsonshift.m,v 1.2 2003/08/26 18:57:30 frl Exp $
```

```
d         = ( a1 - a2 ) / 2;      % flops: 2
% mu      = a2 + d - sign( d ) * sqrt( d^2 + b^2 );
% note that ( d - sign(d)*sqrt(d^2+b^2) ) * ( d + sign(d)*sqrt(d^2+b^2) ) =
%             = d^2 - (d^2 + b^2) = - b^2, so we can avoid cancelation
%             between d and the square root, particularly when b^2 (which
%             is the last superdiagonal element) is small - which we hope!
if d == 0
    % then a1==a2 =: a, evals are a +- abs(b)
    % we choose absolutely larger to avoid cancellation
    if a2 > 0
        mu      = a2 + abs( b );
    else
        mu      = a2 - abs( b );
    end
else
    mu      = a2 - b^2 / ( d + sign(d)*sqrt( d^2 + b^2 ) );
end;

% flop count: around 14
```

Listing 5: **wilkinsonshift** determines the Wilkinson shift

A.2.2 *givens* determines a Givens rotation

```
function [ c, s ] = givens( a, b )
% GIVENS computes c and s such that [ c s; -s c ]' [ a b ]' = [ r  0 ]'
% protected against overflow

% reference: Golub, Van Loan; 3rd ed; 5.1.8
% $Id: givens.m,v 1.3 2003/08/19 20:48:07 frl Exp $

if b == 0
    c = 1; s = 0;
else
    if abs( b ) > abs( a )
        tau = -a/b; s = 1/sqrt( 1+tau^2 ); c = s*tau;
    else
        tau = -b/a; c = 1/sqrt( 1+tau^2 ); s = c*tau;
    end;
end;

% flop count: 5 and 1 sqrt
```

Listing 6: **givens** determines a Givens rotation to eliminate an element

A.2.3 *qrsymtrid* determines the QR decomposition

```
function [ Q, R ] = qrsymtrid( T )
% QR decomposition for symmetric tridiagonal matrices
% QRSYMTRID returns Q and R such that T=QR, Q is orthogonal (and upper
% Hessenberg), and R is upper triangular (of bandwidth 2, ie diagonal and
% two superdiagonals).

% reference: Golub, Van Loan; 3rd ed; ch. 8.3.3
% $Id: qrsymtrid.m,v 1.3 2003/08/21 20:25:55 frl Exp $

N = size( T, 1 );
if N ~= size( T, 2 ) || any(any(T ~= T'))
    error( 'input T must be symmetric' );
end;

R   = T;
Q   = eye( N );

% remove 0 on subdiagonal by premultiplication with orthogonal Givens
% matrices Q1', Q2', ... -> obtain upper triangular R = QN-1'...Q2' Q1' T
% so T = QR with Q = Q1 Q2 Q3 ... QN-1
for k=1:(N-1)
    % now want to find Givens rotation in k, k+1 such that
    % R( k+1, k ) = 0. This needs to be applied to row k, k+1
    % GIVENS computes c and s such that [ c s; -s c ]' [ a b ]' = [ r 0 ]'
    [ c, s ]          = givens( R( k,k ), R( k+1, k ) );  % flop count: 6
    G                 = [ c s; -s c ];
    K                 = min( N, k+2 );                    % flop count: 1, or so...
    R( k:k+1, k:K ) = G' * R( k:k+1, k:K );               % flop count: 3*6
    R( k+1, k )       = 0;     % should be close to zero anyway...
    % accumulate Q
    Q(1:k+1, k:k+1) = Q(1:k+1, k:k+1) * G;        % flop count: (k+1)*6
end;

% flop count: without Q accumulation, around 25 N
% with Q accumulation: sum k=1:N of 30 + 6 k
% total flop count 3 N^2 + 33 N
```

Listing 7: **qrsymtrid** determines the QR decomposition for symmetric tridiagonal matrices

A.2.4 *qrexstep* executes an explicit QR step

```
function T = qrexstep( T );
% explicit QR iteration for sym matrices with Wilkinson shift
% note: we assume T to be symmetric, no check here

% reference: Golub, Van Loan; 3rd ed; 8.3.4
```

```
% $Id: qrexstep.m,v 1.2 2003/08/26 18:58:03 frl Exp $

ident    = eye( size( T ) );
mu       = wilkinsonshift( T( end-1, end-1 ), T( end, end-1 ), T( end, end ) );
[Q,R]    = qrsymtrid( T - mu * ident );                    % flops: 3 N^2 + 33 N
% note: this can be optimized to take advantage of bandedness
T        = R*Q + mu * ident                                % flops:
```

Listing 8: **qrexstep** executes an explicit QR step with Wilkinson shift

A.2.5 *qrimstep* executes an implicit QR step

```
function [ TS, Q ] = qrimstep( TS, Q )
% symmetric implicit QR step with Wilkinson shift
% given a symmetric tridiagonal T, routine computes T+ = Q'TQ,
% where QR = T - mu I is a QR decomposition of T shifted by mu

% Q = G1 G2 .. G(N-1), where we determine Givens rotations ..
% .. G1 such that e1*G1 parallel to  first  col  of  T-mu I
% .. G2 , ...,  G(N-1) such that bulge above is chased down and out

% reference: Golub, Van Loan; 3rd ed; ch. 8.3.5
% $Id: qrimstep.m,v 1.7 2003/08/29 15:16:53 frl Exp $

N = size( TS, 1 );
% note: T in short format

% determine Wilkinson shift
mu = wilkinsonshift( TS( N-1, 1 ), TS( N-1, 2 ), TS( N, 1 ) );
    % flop count: 15

%   note: Givens(x) gives us G with G'x=scalar * e1,
%   so GG'x = x = scalar * G * e1, so we can chose x=(T-mu I)e1,
%   and use the normal Givens routine!
x       = TS(1, 1) - mu;
y       = TS(1, 2);

% note: later , we would want to pass a matrix in to be updated
% (instead of  starting  from the  identity , and then multiply ...)
% note: in flop  count, we assume Q has M rows
if nargin < 2
    Q   = eye( N );
end;

for k = 1:(N-1)
    % compute Givens rotation, around k,k+1, G = [c s; -s c]
    [ c s ]          = givens( x, y );                    % flops: 5 + 1 sqrt
    % rotate T: want T+ = G' T G, and Q+ = Q G
```

```
    G               = [ c s; −s c ];
    K               = min( N, k+2 );                    % flops: 1, or so ...
    % update Q
    Q(:, k:k+1 )    = Q(:, k:k+1 ) * G;                 % flops: 6 * M
    % for T in ordinary format, we want:
    % T( k:k+1, k−1:K ) = G' * T( k:k+1, k−1:K );
    % T( k−1:K, k:k+1 ) = T( k−1:K, k:k+1 ) * G;

    if k > 1
        % for debugging we can compute the zero'ed out element:
        % zerod = s * TS(k−1,2) + c * bulge
        TS( k−1, 2 ) = c * TS( k−1, 2 ) − s * bulge;    % flops: 3
    end;
    Tk1             = TS( k, 1 );
    Tk2             = TS( k, 2 );
    bulge           = −s * TS( k+1, 2 );                % flops: 1
    c2              = c^2;  s2 = s^2; sc2 = 2*c*s;
    TS( k, 1 )      = c2  * Tk1 + s2 * TS( k+1, 1 ) − sc2*Tk2;
    TS( k, 2 )      = c*s*( Tk1 − TS( k+1, 1 ) ) + ( c2 − s2 ) * Tk2;
    TS( k+1, 1 ) = s2  * Tk1 + c2 * TS( k+1, 1 ) + sc2*Tk2;
    TS( k+1, 2 ) = c   * TS( k+1, 2 );
    % last 4 lines: flops: +− 27, could be optimized (by storing c^2 etc.)
    % note: should seek to avoid cancellation above
    x               = TS( k, 2 );
    y               = bulge;        % which we want to eliminate...
end;
TS( N, 2 ) = 0;

% flop count: around 32 N for new TS, around 6 MN for accumulating Q
```

Listing 9: **qrimstep** executes an implicit QR step with Wilkinson shift

A.2.6 *symqrschur* computes the Eigen decomposition

```
function [ D, Q ] = symqrschur( A, tol )
% SYMQRSCHUR computes an approximate sym Schur decomposition Q'AQ = D
% given symmetric real A, and a tolerance tol,
% using implied QR steps with Wilkinson shift

% reference: Golub, Van Loan; 3rd ed; ch. 8.3.5, alg. 8.3.3
% $Id: symqrschur.m,v 1.4 2003/08/28 20:00:57 frl Exp $

% T = Q'*A*Q is (sym) tridiagonal
[ T, Q ] = symtridhh( A );
if nargin == 1
    tol = 1e−14;
end;
```

```
TS = stridl2s( T );  % TS(k,1) == T(k,k); TS(k,2) == T(k,k+1) == T(k+1,k)
N  = size( TS, 1 );
q  = 0;
while q < N
    q  = 0; p = N-1;
    for k = N-1:-1:1
        if abs( TS( k, 2 ) ) <= tol * ( abs( TS( k,1 ) ) + abs( TS( k+1, 1 ) ) )
            TS( k, 2 ) = 0;
            if q == N-k-1 % state one:   we are in D33
                q = N-k;
                p = k-1;
            % else          % state three:  we are in D11
            end;
        else
            if p == k      % state two:    we are in D22
                p = k-1;
            % else          % state three:  we are in D11
            end;
        end;
    end;
    % after this loop: TS(p+1,2)..TS(N-q-1,2) are non zero,
    % TS(N-q,2)..TS(N-1,2) are zero
    % ie, if all zero -> q=N, p=0; all nonzero -> q=0; p=0

    if q == N-1
        q = N;  % and abort
    else        % do some work: on D22, ie TS( p+1:N-q, : )
        [ TS( p+1:N-q, : ), Q(:, p+1:N-q ) ] = qrimstep( TS( p+1:N-q, : ), Q(:, p
            +1:N-q ) );
    end;
end;
D = TS( :,1 );
```

Listing 10: **symqrschur** computes the Eigen decomposition for symmetric matrices, using implicit QR steps

A.3 The SVD

A.3.1 *gksvdstep* executes a Golub-Kahan SVD step

```
function [ B, U, V ] = gksvdstep( B, U, V )
% Given an (upper) bidiagonal real B, NxN, with no zeros on the diagonal or
% superdiagonal, GKSVDSTEP (Golub-Kahan Singular Value Decomposition Step)
% returns B+ = U' B V, where U and V NxN are orthogonal matrices,
% and VV is the orthogonal matrix that would have been obtained by applying
% an implicit QR step with Wilkinson shift (QRIMSTEP) to T = B'B,
% ie T+ = B+'B+ = V'B'U U'BV = V' B'B V = V'TV
```

```
% and we want T+ triangular, and V*e1 proportional to T-mu*I
%
% B is given in short format, ie. the diagonal is in the first col B(1:N,1),
% the superdiagonal in the second col B(1:N-1, 2).

% reference: Golub, Van Loan; 3rd ed; ch. 8.6.2, alg. 8.6.1
% $Id: gksvdstep.m,v 1.3 2003/08/27 13:48:48 frl Exp $

N = size( B, 1 );

% determine the shift mu.
% mu should be the eigenvalue (closer to the lower right) of
% ( B(N-1,1)^2 + B(N-2,2)^2 , B(N-1,1) * B(N-1,2) )
% ( B(N-1,1) * B(N-1,2)     ,  B(N,1)^2 + B(N-1,2)^2 )
if N > 2
    mu = wilkinsonshift( B(N-1,1)^2+B(N-2,2)^2, B(N-1,1)*B(N-1,2), B(N,1)^2+B(
        N-1,2)^2 );
else if N ==2
        mu = wilkinsonshift( B(N-1,1)^2, B(N-1,1)*B(N-1,2), B(N,1)^2+B(N-1,2)
            ^2 );
    else
        error('Cannot perform GK SVD step on B with N < 2')
    end
end

x  = B( 1, 1 )^2 - mu;
y  = B( 1, 1 ) * B( 1, 2 );

if nargin < 3
    V = eye( N );
    if nargin < 2
        U = V;
    end;
end;

% B+ = U' B V = U(N-1)' ... U2' U1' B G1 V2 V3 ... V(N-1), with G1*e1 prop T-
    mu*I
for k = 1:N-1
    % compute Givens rotation, around k,k+1 FROM RIGHT, G = [c s; -s c]
    [ c s ]      = givens( x, y );                    % flops: 5 + 1 sqrt
    % rotate B: want B_(1/2) = B G, V+ = V G
    G            = [ c s; -s c ];
    % update V - we update the full column, since we don't know what was
    % passed in
    V(:, k:k+1) = V(:, k:k+1) * G;                    % flops: 6 * N
    % for B in ordinary format, we want:
    % B( k-1:K, k:k+1 ) = B( k-1:K, k:k+1 ) * G;

    if k > 1
```

```
        % for debugging we can compute the zero'ed out element:
        % zerod   = s * B(k-1,2) + c * bulge
        B( k-1, 2 ) =  c * B( k-1, 2 ) - s * bulge;       % flops: 3
    end;
    Bk          = B( k, 1 );
    bulge       = -s * B( k+1, 1 );                       % flops: 1
    B( k, 1 )   =  c * Bk - s * B( k, 2 );                % flops: 3
    B( k, 2 )   =  s * Bk + c * B( k, 2 );                % flops: 3
    B( k+1, 1 ) =  c * B( k+1, 1 );                       % flops: 1

    % now, bulge is at k+1, k
    x           = B( k, 1 );
    y           = bulge;        % which we want to eliminate...

    % compute Givens rotation, around k,k+1 FROM LEFT, G = [c s; -s c]
    [ c s ]     = givens( x, y );                         % flops: 5 + 1 sqrt
    % rotate B: want B_+ = G' B_(1/2), U+ = U G
    G           = [ c s; -s c ];
    % update U (from right, since transposed)
    % we update the full column, since don't know what was passed in
    U(:, k:k+1) = U(:, k:k+1 ) * G;                 % flops: 6 * K
    % for B in ordinary format, we want:
    % B = G' * B;

    % zerod       =  s * B( k, 1 ) + c * bulge
    B( k, 1 )   =  c * B( k, 1 ) - s * bulge;
    Bk2         = B( k, 2 );
    B( k, 2 )   =  c * Bk2       - s * B( k+1, 1 );
    bulge       = -s * B( k+1, 2 );
    B( k+1, 1 ) =  s * Bk2       + c * B( k+1, 1 );
    B( k+1, 2 ) =  c * B( k+1, 2 );
    % now, bulge is at k, k+2

    x           = B( k, 2 );
    y           = bulge;        % which we want to eliminate...
end;
B( N, 2 ) = 0;
```

Listing 11: **gksvdstep** executes a Golub-Kahan SVD step

A.3.2 *gksvdsteps* computes the SVD of a bidiagonal matrix

```
function [ D, U, V ] = gksvdsteps( B, U, V, tol )
% GKSVDSTEPS computes the SVD of a bidiagonal B, and
% overwrites U, V such that if previously B=U'AV, now D=U'AV
% this is an internal function, so no errorchecking on input
% both B and D in short format (as column vectors)
```

```
% reference: Golub, Van Loan; 3rd ed; ch. 8.6.2, alg. 8.6.2
% $Id: gksvdsteps.m,v 1.1 2003/08/28 20:50:52 frl Exp $

M = size( U, 1 );    % rows
N = size( V, 1 );    % cols

% partition B into an unreduced part in the middle, and a diagonal part at
% the bottom
q = 0;
while q < N
    q   = 0; p = N-1;
    for k = N-1:-1:1
        % note: below, we need <=, not <, otherwise we get caught in an
        % infitite loop, if these elements are exactly zero!
        if abs( B(k,2) ) <= tol*( abs( B(k,1) ) + abs( B(k+1,1) ) )
            B(k,2) = 0;
            if q == N-k-1 % state one
                q = N-k;
                p = k-1;
            % else          % state three
            end;
        else
            if p == k       % state two
                p = k-1;
            % else          % state three
            end;
        end;
    end;
    if q == N-1
        q=N;   % and abort...
        % NOW, with D11 = BL(1:p,1:p), D22 = BL(p+1:N-q,p+1:N-q),
        % D33 = BL(N-q+1:N, N-q+1:N), D33 is diagonal, D22 unreduced
    else        % do some work
        % first, determine if any element on the diagonal is zero
        % if so, rotate it aside
        k = p+1;
        smalldiag = tol * norm( B, 'inf' );
        while abs( B( k, 1 ) ) > smalldiag  && k < N-q
            k = k+1;
        end;
        % now, k == N-q, or abs( B(k,1) ) <= smalldiag, or both
        if abs( B( k, 1 ) ) <= smalldiag
            B( k, 1 ) = 0;
            % now, B( k, 1 ) approx 0, p+1 <= k <= N-q, B(N-q,2) == 0 (note:
            % even if q == 0, by the format we have chosen)
            if k < N-q
                % to do: if k<N-q, rotate that row away with Givens rotations from
                % left, G(k,j), j=k+1:N-q
                bulge = B(k,2);
```

```
                B(k,2)= 0;
                for j=k+1:N-q
                    [ c s ] = givens( B(j,1) , bulge );
                    % need to transpose (since mult from left), and
                    % transpose again, since need to eliminate upper
                    % element
                    % rotate B from left: B+ = G B, G=G(k,j)
                    % zerod = c * bulge + s * B(j,1)
                    B(j,1)   = -s * bulge + c * B(j,1);
                    bulge    =  s * B(j,2);
                    B(j,2)   =  c * B(j,2);
                    % rotate U from right (since transposed): U+ = U G'
                    G        = [ c s; -s c ];
                    U(:,[k j]) = U(:,[k j]) * G';          % flops: 6*M
                end % for j
            else % k==N-q
                % if k==N-q, then apply Givens from the
                % right, G(k,j), j=N-q-1:-1:p+1
                bulge = B(N-q-1,2);
                B(N-q-1,2)= 0;
                for j=N-q-1:-1:p+1
                    % rotate B from right, B+ = B G; G=G(k,j)
                    [ c s ]  = givens( B(j,1) , bulge );
                    % zerod   = s * B(j,1) + c * bulge
                    B(j,1)   = c * B(j,1) - s * bulge;
                    if j>p+1
                        bulge    = s * B(j-1,2);
                        B(j-1,2)= c * B(j-1,2);
                    end;
                    % rotate V from right
                    G        = [ c s; -s c ];
                    V(:,[j k]) = V(:,[j k]) * G;          % flops: 6*M
                end
            end % if k
        else     % no element on the diagonal is zero
            % do some real work - GK SVD step for p+1:N-q
            [ B(p+1:N-q,:), U(:,p+1:N-q), V(:,p+1:N-q) ] = gksvdstep( B(p+1:N-q
                ,:), U(:,p+1:N-q), V(:,p+1:N-q) );
        end;
    end; % if q
end; % while q<N

% now fix sign - want cols of V such that elements of D are non-negative
D = B( :,1 );
for k = 1:N
    if D(k) < 0
        D(k)    = -D(k);
        V(:,k)  = -V(:,k);
    end;
```

```
end;
```

Listing 12: **gksvdsteps** computes the SVD of a bidiagonal matrix using Golub-Kahan SVD steps

A.3.3 *gksvd* computes the SVD

```
function [ D, U, V ] = gksvd( A, tol )
% GKSVD computes the SVD of A
% Given A MxN, M>=N, and tol, this computes U MxM, V NxN orthogonal and DD
    diagonal
% such that A=U(DD+E)V', where two norm E is around unit roundoff x two-norm A
% D is returned as a Nx1 column vector, DD would be = [ diag(D); zeros( M-N, N ) ];

% reference: Golub, Van Loan; 3rd ed; ch. 8.6.2, alg. 8.6.2
% $Id: gksvd.m,v 1.9 2003/08/28 20:50:52 frl Exp $

if nargin == 1
    tol = 1e-14;
end;

M = size( A, 1 );    % rows
N = size( A, 2 );    % cols
if M < N
    error( 'A must have M >= N' );
end;

% BIDIGHH computes bidiagonal B=U'AV using householder reflections
[ B, U, V ] = bidighh( A );
% GKSVDSTEPS computes diagonal D and updates U, V such that D=U'AV
[ D, U, V ] = gksvdsteps( B, U, V, tol );
```

Listing 13: **gksvd** computes the SVD using bidiagonalization and Golub-Kahan SVD steps

A.3.4 *gksvdprod* computes the SVD of a matrix product

```
function [ D, U, V ] = gksvdprod( A, tol )
% GKSVDPROD computes the SVD of a product of square matrices
% Given A NxNxK, for AAA := A(:,:,K)*...*A(:,:,1) this computes
% U, V NxN orthogonal, and D Nx1 such that AAA=U diag(D) V'
% Note: AAA NxN is never explicitly computed!

% reference: Golub, Van Loan; 3rd ed; ch. 8.6.2, alg. 8.6.2
% reference: Golub, Solna, van Dooren: Computing the SVD
% of a General Matrix Product/Quotient,
```

```
% SIAM J. MATRIX ANAL. APPL., Vol. 22, No. 1, pp 1−19
% $Id: gksvdprod.m,v 1.1 2003/08/28 20:51:13 frl Exp $

if nargin == 1
    tol = 1e−14;
end;

% BIDIGPROD computes bidiagonal B=U' AAA V using householder reflections
[ B, U, V ] = bidigprod( A );
% GKSVDSTEPS computes diagonal D and updates U, V such that D=U'AV
[ D, U, V ] = gksvdsteps( B, U, V, tol );
```

Listing 14: **gksvdprod** computes the SVD of a matrix product using implicit bidiagonalization and Golub-Kahan SVD steps

A.4 Test Routines

A.4.1 *gentestmat* generates test matrices

```
function A = gentestmat( k, M, N, verbose )
% GENTESTMAT( k, M, N ) generates a MxN testmatrix of type k (M>=N)
% $Id: gentestmat.m,v 1.13 2003/08/26 15:31:49 frl Exp $

As      = { '−triu(ones(M))+2*diag(ones(M,1))',
              'uppertriang,structured,eval=1,only1evect,bad condition';
            'triu(10*rand(M)−1)−20*diag(ones(M,1))',
              'uppertriang,large negative diag';
            '2*rand(M,N)−1',
              'elements uniformly random −1..1';
            'exp(16*rand(M,N)−15).*(rand(M,N)−.5)',
              'elements of very different size';
            'U*diag(2.^(−1:−1:−N))*V',
              'very different singular values: 1/2..1/(2^N)';
            '10*randn(M,1)*rand(1,N)+1e−15*rand(M,N)',
              'rank 1 matrix with slight perturbations';
            '10*randn(M,1)*rand(1,N)',
              'rank 1 matrix (no perturbations)';
            'W*(diag(1e−6*ones(M,1))+diag(ones(M−1,1),1))*transpose(W)',
              '1 small eval, one evect';
            'W*(diag(1e+6*ones(M,1))+diag(ones(M−1,1),1))*transpose(W)',
              '1 big eval, one evect';
            'W*diag([1:M−1],1)*transpose(W)',
              'zero eigenvals, one evect',
            '−(rand(M,N)>(1−5/(M*N)))',
              'sparse matrix with about 5 negative ones, rest zero'
            'hilb(M)',
              'Hilbert matrix';
```

```
                ' full (␣gallery (␣''dorr'',␣M,␣0.02␣)␣)',
                    'Dorr␣matrix';
                'gallery (''kahan'',␣[␣M,␣N␣]␣)'
                    'Kahan␣matrix';
            };

if nargin == 2 || N > M
    N = M;
end;
if nargin < 4
    verbose = 1;
end;

% produce some random orthogonal matrices, needed for some types
[U,X]=qr( rand( M, N ), 0 );
[V,X]=qr( rand( N ) );
[W,X]=qr( rand( M ) );

if verbose
disp( sprintf( '\nGenerating␣%d␣x␣%d␣matrix,␣type␣%d␣(%s):␣generated␣by\n␣%s:\n'
    , M, N, k, As{k*2}, As{ k*2-1 } ) );
end;
A = eval( As{ k*2-1 } );
A = A(1:M, 1:N);
```

Listing 15: **gentestmat** generates test matrices

A.4.2 *testbidighh* tests bidiagonalization

```
% $Id: testbidighh.m,v 1.10 2003/08/29 09:58:52 frl Exp $

wtf = 0;
if wtf
    fid = fopen( 'bidighh.log', 'w' );
else
    fid = 1;
end;

tol = 1e-12;
N   = 30;
for M = 30:21:72
    fprintf( fid, '\nTesting␣BIDIGHH␣with␣various␣%g␣x␣%g␣matrices\n', M, N );
    fprintf( fid, 'Type:␣␣B-U''AV␣Res␣␣Relative␣Res␣␣␣SVs␣Rel␣Err␣␣␣␣␣␣␣U␣Orth
        ␣␣␣␣␣␣␣V␣Orth\n' );
    for Typ = 1:14
        TestMat = gentestmat( Typ, M, N, 0 );
        for trans = 0:(M==N)
            [BS,U,V]= bidighh( TestMat );
```

```
            B       = [ diag( BS(:,1) ) + diag( BS(1:end-1,2), 1 ); zeros( M-N, N )
                ];
            dev     = norm( B - U'*TestMat*V );
            reldev  = dev / norm( TestMat );
            svdB    = sort( svd( B ) );
            svdA    = sort( svd( TestMat ) );
            devsvd  = norm( svdA - svdB, 'inf' )/norm( svdA, 'inf' );
            Udev    = norm( U'*U - eye( M ) );
            Vdev    = norm( V'*V - eye( N ) );
            fprintf( fid , ' %2g: %12g, %12g, %12g, %12g, %12g\n', Typ, dev, reldev
                , devsvd, Udev, Vdev );
            if reldev > tol  ||  Udev > tol || Vdev > tol || devsvd > tol
                fprintf( fid ,   '^^^^^^^^^^^^^^^^^^^^ ^^^^^^^^^^^^^^^^^^^^\n' );
            end
            TestMat = TestMat';
        end;
    end;
end;

if wtf; fclose( fid ) ; end;
```

Listing 16: **testbidighh** tests bidiagonalization

A.4.3 *testsymqrschur* tests symmetric Schur decomposition

```
% $Id: testsymqrschur.m,v 1.12 2003/08/29 15:18:59 frl Exp $

wtf = 0;
if wtf
    fid = fopen( 'symqrschur.log', 'w' );
else
    fid = 1;
end;
tol = 1e-10;

fprintf( fid ,   '\n\nTEST SYMQRSCHUR\n' );
fprintf( fid ,   'We compute eigenvalues of a test matrix using eig(),and then using\n
    the Schur decomposition' );
fprintf( fid ,   'computed by symqrschur, and display the norm\nof the difference of
    the vector of eigenvalues' );
fprintf( fid ,   ' relative to the norm of the\nvector of eigenvalues itself . Also the
    norm of A-QDQ'', and Q''Q-I.\n' );
fprintf( fid ,   'All errors should be around 1e-15.\n' );
for N=5:25:55
    fprintf( fid ,   '\nTesting SYMQRSCHUR with various (symmetrized) %g x %g
        matrices\n', N, N );
    fprintf( fid ,   'Type: Eigval 2-norm  Relativ Res   Orthogonal\n' );
    for Typ = 1:14
```

```
        TestMat = gentestmat( Typ, N, N, 0 );
        TestMat = TestMat + TestMat';
        Ev      = eig( TestMat );
        [ D,Q ] = symqrschur( TestMat );
        ErrEv   = norm( sort( D ) - sort( Ev ) )/norm( Ev );
        Res     = norm( TestMat - Q*diag(D)*Q' );
        RelRes  = Res / norm( TestMat );
        Orth    = norm( Q'*Q - eye( N ) );
        fprintf( fid ,  ' %2g: %12g,  %12g, %12g\n', Typ, ErrEv, RelRes, Orth );
        if RelRes > tol || Orth > tol || ErrEv > tol
            fprintf( fid ,  '^^^^^^^^^^^^^^^^^^^^ ^^^^^^^^^^^^^^^^^^^^^^^^\n' );
        end
    end;
end

N = 4;
fprintf( fid ,  '\nTesting SYMQRSCHUR, all combinations of off-diagonal elements
    zero, %g x %g matrices\n', N, N );
fprintf( fid ,  'Type: Eigval 2-norm  Relativ Res   Orthogonal\n' );
for Typ=0:(2^N-2)
    Bits    = bitget( Typ, 1:N-1 );
    TS      = [ 1:N; Bits , 0 ]';
    TestMat = strids2l( TS );
    Ev      = eig( TestMat );
    [ D,Q ] = symqrschur( TestMat );
    ErrEv   = norm( sort( D ) - sort( Ev ) )/norm( Ev );
    Res     = norm( TestMat - Q*diag(D)*Q' );
    RelRes  = Res / norm( TestMat );
    Orth    = norm( Q'*Q - eye( N ) );
    fprintf( fid ,  ' %2g: %12g   %12g  %12g\n', Typ, ErrEv, RelRes, Orth );
    if RelRes > tol || Orth > tol || ErrEv > tol
        fprintf( fid ,  '^^^^^^^^^^^^^^^^^^^^ ^^^^^^^^^^^^^^^^^^^^^^^^\n' );
    end
end;

if wtf; fclose( fid ); end;
```

Listing 17: **testsymqrschur** tests symmetric Schur decomposition

A.4.4 *testgksvd* tests the Golub-Kahan SVD

```
% $Id: testgksvd.m,v 1.7 2003/08/29 15:18:59 frl Exp $

wtf = 0;
if wtf
    fid = fopen( 'gksvd.log', 'w' );
else
    fid = 1;
```

```
end;
tol = 1e-10;

fprintf( fid , 'TEST GKSVD\n' );
fprintf( fid , 'We compute singular values of a test matrix using the MATLAB svd
    (),\nand then using our implementation of' );
fprintf( fid , 'the Golub-Kahan SVD algorithm,\nand display the norm of the
    difference, relative to the norm of the svs.\n' );
fprintf( fid , 'We also test the norm of the residual A-UDV'', and\nthe
    orthogonality of U and V, namely norm U''U-I and V''V-I.\n' );
fprintf( fid , 'All errors should be around 1e-15.\n' );
for M=5:21:68
    N=min(30,M);
    fprintf( fid , '\nTesting GKSVD with various %g x %g matrices\n', M, N );
    fprintf( fid , 'Type:   svd 2-norm       RelRes       U Orth       V Orth
        \n' );
    for Typ = 1:14
        TestMat = gentestmat( Typ, M, N, 0 );
        [D,U,V] = gksvd( TestMat, 1e-14 );
        svds    = svd( TestMat );
        ErrSvd = norm( sort( D ) - sort( svds ) ) / norm( svds );
        Res     = norm( TestMat - U(:,1:N)*diag(D)*V' );
        RelRes = Res / norm( TestMat );
        UOrth  = norm( U'*U - eye( M ) );
        VOrth  = norm( V'*V - eye( N ) );
        fprintf( fid , '  %2g:  %12g, %12g, %12g, %12g\n', Typ, ErrSvd, RelRes,
            UOrth, VOrth );
        if RelRes > tol || UOrth > tol || VOrth > tol || ErrSvd > tol
            fprintf( fid , '^^^^^^^^^^^^^^^^^^^^ ^^^^^^^^^^^^^^^^^^^^^^^\n' );
        end
    end;
end

N = 4;
fprintf( fid , '\nTesting GKSVD, all combinations of off-diagonal elements zero, %
    g x %g matrices\n', N, N );
fprintf( fid , 'Type:   svd 2-norm       RelRes       U Orth       V Orth\n'
    );
for Typ=0:(2^N-1)
    Bits    = bitget( Typ, 1:N-1 );
    TestMat = -diag( 1:N ) + diag( Bits, 1 );
        [D,U,V] = gksvd( TestMat, 1e-14 );
        svds    = svd( TestMat );
        ErrSvd = norm( sort( D ) - sort( svds ) ) / norm( svds );
        Res     = norm( TestMat - U*diag(D)*V' );
        RelRes = Res / norm( TestMat );
        UOrth  = norm( U'*U - eye( N ) );
        VOrth  = norm( V'*V - eye( N ) );
```

```
        fprintf( fid , '␣␣%2g:␣%12g,␣%12g,␣%12g,␣%12g\n', Typ, ErrSvd, RelRes,
             UOrth, VOrth );
50      if RelRes > tol || UOrth > tol || VOrth > tol || ErrSvd > tol
            fprintf( fid , '^^^^^^^^^^^^^^^^^^^^␣^^^^^^^^^^^^^^^^^^^^^^^^\n' );
            err;
        end
end;
55
if wtf; fclose( fid ); end;
```

Listing 18: **testgksvd** tests the Golub-Kahan SVD

A.4.5 *testgksvdprod* tests the SVD of a matrix product

```
function testgksvdprod
% $Id: testgksvdprod.m,v 1.5 2003/08/29 16:48:32 frl Exp $

global err;
5   wtf = 0;
if wtf
    fid = fopen( 'gksvdprod.log', 'w' );
else
    fid = 1;
10  end;
tol = 1e-10;

fprintf( fid , 'TEST␣GKSVDPROD\n' );

15  if 0 % generate detailed error data
Layer = 1;
for Typ = 1:2
    for K = 5:8:21
        for N = 8:12:32
20          fprintf( fid , 'computing␣Type␣%g,␣%3g␣matrices␣%4g␣x␣%4g\n', Typ,
                K, N, N );
            [ A, true ] = gentestprod( Typ, N, K );
            impl    = sort( gksvdprod(A, 1e-16) );
                      Amul   = eye(size(A,1)); for k=1:size(A,3); Amul = A(:,:,k)*
                          Amul; end;
                      expl    = sort( svd(Amul) );
25          implrel = abs( (impl - true) ./ true );
            explrel = abs( (expl - true) ./ true );
            % err Nx5 contains in col 1 [Typ K N 0000000]'
            % col 2 true, col 3 impl, 4 expl, col 5 implrel, 6 explrel
            err (1:3,1,   Layer)  = [Typ;K;N];
30          err (1:N,2:6,Layer)  = [true impl expl implrel  explrel ];
            Layer = Layer + 1;
        end
```

```
    end
end
end

K = 4;
for M=5:21:47 % 47
    fprintf( fid , '\nTesting GKSVDPROD with various %g x %g matrices\n', M,
        M );
    fprintf( fid , 'Prod of types:   SVs 2-norm        RelRes        U Orth
        V Orth\n' );
    TestMats = zeros(M,M,14);
    for Typ = 1:14
        TestMats(:,:,Typ) = gentestmat( Typ, M, M, 0 );
    end;
    for k=1:(14-K)
        A       = TestMats(:,:,k:k+K);
        [D,U,V] = gksvdprod( A, 1e-16 );
        Amul    = eye(size(A,1)); for kk=1:size(A,3); Amul = A(:,:,kk)*Amul; end;
        expl    = sort( svd( Amul ) );
        impl    = sort( D );
        ErrSvd  = norm( impl - expl ) / norm( expl );
        Res     = norm( Amul - U*diag(D)*V' );
        RelRes  = Res / norm( Amul );
        UOrth   = norm( U'*U - eye( M ) );
        VOrth   = norm( V'*V - eye( M ) );
        fprintf( fid , '   %3g    %2g: %12g, %12g, %12g, %12g\n', k, k+K, ErrSvd
            , RelRes, UOrth, VOrth );
        if RelRes > tol || UOrth > tol || VOrth > tol || ErrSvd > tol
            fprintf( fid , '^^^^^^^^^^^^^^^^^^^^ ^^^^^^^^^^^^^^^^^^^^\n' );
        end
    end;
end

M = 8;
K = 6;
A   = zeros( M,M,K );
fprintf( fid , '\nTesting GKSVDPROD, random combinations of off-diagonal
    elements zero, %g x %g matrices\n', M, M );
fprintf( fid , 'Combinations:   SVs 2-norm        RelRes        U Orth        V
    Orth\n' );
for l=1:10
    for k = 1:K
        Bits    = bitget( floor( rand(1) * 2^M ), 1:M-1 );
        A(:,:,k)= -diag( 1:M ) + diag( Bits, 1 );
    end;
    [D,U,V] = gksvdprod( A, 1e-16 );
    Amul    = eye(size(A,1)); for kk=1:size(A,3); Amul = A(:,:,kk)*Amul; end;
    expl    = sort( svd( Amul ) );
    impl    = sort( D );
```

```
    ErrSvd = norm( impl − expl ) / norm( expl );
    Res    = norm( Amul − U∗diag(D)∗V' );
    RelRes = Res / norm( Amul );
    UOrth  = norm( U'∗U − eye( M ) );
    VOrth  = norm( V'∗V − eye( M ) );
    fprintf( fid , '      %2g   : %12g, %12g, %12g, %12g\n', K, ErrSvd, RelRes,
        UOrth, VOrth );
    if RelRes > tol || UOrth > tol || VOrth > tol || ErrSvd > tol
        fprintf( fid , '^^^^^^^^^^^^^^^^^^^^ ^^^^^^^^^^^^^^^^^^^^\n' );
    end
end;

function [ A, true ] = gentestprod( Typ, N, K )
A = zeros(N,N,K);
switch Typ
    case 1
        S = diag(2.^(−1:−1:−N));
        [V,X]=qr(randn(N));
        % AAAclose = V;
                for k=1:K
            [U,X]=qr(randn(N));
            A (:,:, k) = U'∗S∗V;
            V=U;
                end;
        true = 2.^(K∗(−1:−1:−N))';
        % AAAclose = U'∗diag(true)∗V;
    case 2
        a = 2;
        b = −1;
        Toep = toeplitz( [ a b zeros(1,N−2) ] );
        for k=1:K
            A (:,:, k) = Toep;
                end;
                true = (a + 2∗b∗cos((1:N)'/(N+1) ∗ pi )).^K;
end;
true = sort( true );

if wtf; fclose( fid ) ; end;
```

Listing 19: **testgksvdprod** tests the SVD of a matrix product

B Test Results

Listing 20: Testing **bidighh.m**

```
Testing BIDIGHH with various 30 x 30 matrices
```

Type:	B−U'AV Res	Relative Res	SVs Rel Err	U Orth	V Orth
1:	1.16311e−14,	6.38968e−16,	3.90346e−16,	1.32707e−15,	1.22236e−15
1:	7.85547e−15,	4.3155e−16,	3.90346e−16,	1.36103e−15,	1.53869e−15
2:	3.25612e−14,	4.72173e−16,	4.12146e−16,	1.7155e−15,	1.59236e−15
2:	3.3128e−14,	4.80392e−16,	3.09109e−16,	1.31389e−15,	1.45162e−15
3:	4.42127e−15,	7.32257e−16,	3.67753e−16,	1.46492e−15,	1.3982e−15
3:	3.9295e−15,	6.50809e−16,	5.88406e−16,	1.21291e−15,	1.39886e−15
4:	8.06037e−16,	4.81232e−16,	3.97705e−16,	1.55103e−15,	1.40591e−15
4:	6.6543e−16,	3.97285e−16,	5.30273e−16,	9.24012e−16,	1.12666e−15
5:	1.22988e−16,	2.45976e−16,	2.22045e−16,	1.24788e−15,	1.42615e−15
5:	1.08872e−16,	2.17743e−16,	4.44089e−16,	1.2785e−15,	1.54018e−15
6:	3.79939e−14,	2.78984e−16,	2.08696e−16,	1.27147e−15,	1.32726e−15
6:	4.35179e−14,	3.19546e−16,	2.08696e−16,	1.19938e−15,	1.35429e−15
7:	3.2602e−14,	1.57446e−16,	6.86291e−16,	1.31109e−15,	1.38764e−15
7:	5.14587e−14,	2.48512e−16,	5.49033e−16,	1.10876e−15,	1.73952e−15
8:	1.48016e−15,	1.48016e−15,	5.55111e−16,	1.49597e−15,	1.24095e−15
8:	1.54871e−15,	1.54871e−15,	6.66133e−16,	1.3185e−15,	1.418e−15
9:	1.69129e−09,	1.69129e−15,	5.82076e−16,	1.37127e−15,	1.39322e−15
9:	1.38221e−09,	1.38221e−15,	5.82076e−16,	1.34348e−15,	1.13086e−15
10:	2.3543e−14,	8.11829e−16,	4.90029e−16,	1.28475e−15,	1.85608e−15
10:	2.08388e−14,	7.18579e−16,	3.67522e−16,	1.73556e−15,	1.08046e−15
11:	0,	0,	0,	0,	0
11:	0,	0,	0,	0,	0
12:	1.15619e−15,	5.82026e−16,	2.23554e−16,	1.11069e−15,	1.1881e−15
12:	1.15619e−15,	5.82026e−16,	2.23554e−16,	1.11069e−15,	1.1881e−15
13:	4.52646e−14,	4.64032e−16,	4.37049e−16,	1.22379e−15,	1.62777e−15
13:	4.13584e−14,	4.23987e−16,	2.91366e−16,	1.71728e−15,	1.56896e−15
14:	2.2165e−15,	5.12971e−16,	1.54165e−16,	1.48598e−15,	1.3485e−15
14:	2.75157e−15,	6.36805e−16,	2.05554e−16,	2.43461e−15,	1.7354e−15

Testing BIDIGHH with various 51 x 30 matrices

Type:	B−U'AV Res	Relative Res	SVs Rel Err	U Orth	V Orth
1:	1.03023e−14,	5.65971e−16,	1.4638e−16,	1.33172e−15,	1.19153e−15
2:	3.00461e−14,	4.36643e−16,	1.54889e−16,	1.36523e−15,	1.13133e−15
3:	5.44524e−15,	7.82221e−16,	5.10355e−16,	1.56665e−15,	1.10475e−15
4:	1.25807e−15,	6.35918e−16,	6.73424e−16,	1.31304e−15,	1.67375e−15
5:	1.37209e−16,	2.74418e−16,	3.33067e−16,	1.49816e−15,	1.99688e−15
6:	5.29797e−14,	2.26477e−16,	1.21497e−16,	1.24499e−15,	1.29729e−15
7:	4.00305e−14,	1.80762e−16,	2.56682e−16,	2.05705e−15,	1.40396e−15
8:	1.35741e−15,	1.35741e−15,	4.44089e−16,	1.46949e−15,	1.26817e−15
9:	1.98582e−09,	1.98582e−15,	8.14907e−16,	1.59185e−15,	1.44837e−15
10:	3.72521e−14,	8.29152e−16,	3.16303e−16,	1.55865e−15,	1.11398e−15
11:	0,	0,	0,	0,	0
12:	8.36479e−16,	4.12397e−16,	2.18943e−16,	1.5675e−15,	1.61319e−15
13:	1.28675e−13,	5.06678e−16,	2.2383e−16,	1.38277e−15,	1.41686e−15
14:	2.13502e−15,	4.94114e−16,	2.56942e−16,	1.21958e−15,	1.30726e−15

Testing BIDIGHH with various 72 x 30 matrices

Type:	B−U'AV Res	Relative Res	SVs Rel Err	U Orth	V Orth

```
 1:  1.03043e−14,  5.66082e−16,  1.95173e−16,  1.36446e−15,  1.19153e−15
 2:  3.37155e−14,  4.58535e−16,  3.86538e−16,  1.22482e−15,  1.31456e−15
 3:  6.62674e−15,  8.50776e−16,  3.42087e−16,  1.31354e−15,  1.10513e−15
 4:  1.80178e−15,  8.09313e−16,  3.98948e−16,   1.5408e−15,   1.2826e−15
 5:  1.50519e−16,  3.01039e−16,  2.22045e−16,  1.50261e−15,  1.05411e−15
 6:  1.35264e−13,  4.88972e−16,  4.10973e−16,  2.01517e−15,  1.32859e−15
 7:  9.35737e−14,  3.60263e−16,  4.61432e−17,  1.47667e−15,  1.35241e−15
 8:  1.40582e−15,  1.40582e−15,  4.44089e−16,  1.32847e−15,  1.60646e−15
 9:  2.05817e−09,  2.05816e−15,  4.65661e−16,  1.94299e−15,  1.32995e−15
10:  5.45311e−14,   9.0768e−16,  4.73084e−16,  1.74485e−15,  1.45513e−15
11:             0,             0,             0,             0,             0
12:  1.41691e−15,   6.9176e−16,  1.08406e−16,  2.14733e−15,  1.21503e−15
13:  2.34462e−13,  4.86995e−16,  1.18068e−16,  1.15519e−15,  1.21381e−15
14:  2.13505e−15,   4.9412e−16,  2.05554e−16,  1.21935e−15,  1.22143e−15
```

Listing 21: Testing **symtridhh.m**

```
TEST SYMTRIDHH
We generate symmetrized testmatrices A, and compute tridiag T = Q'AQ,
with Q orthogonal. We then compute evals of A and T, and display
the norm of the difference ( relative to thenorm of the evals itself ).
We also consider the norm of redidual A−QTQ', and check
orthogonality , namely the norm of Q'Q−I.
All errors should be around 1e−15.

Testing SYMTRIDHH with various (symmetrized) 3 x 3 matrices
Type:   Eig 2−norm        RelRes           Orth
   1:  2.96059e−16,  1.19759e−16,  1.01465e−17
   2:  3.56259e−16,  1.72033e−15,  7.85078e−16
   3:  5.55987e−16,  1.31605e−16,  1.17881e−17
   4:  5.57659e−16,  9.27648e−17,  2.68632e−16
   5:  6.59283e−16,  1.60155e−15,  1.21832e−15
   6:  9.11721e−16,   2.9018e−16,  8.91626e−18
   7:  4.15676e−16,  4.05006e−16,  6.87298e−16
   8:  3.22409e−16,  2.26564e−16,  1.11291e−16
   9:  4.30369e−16,  1.32463e−15,  6.16847e−16
  10:  5.06497e−16,  8.22797e−16,  6.84196e−16
  11:  2.22045e−16,  1.42088e−16,  2.32287e−16
  12:  1.64346e−16,  1.41716e−16,   7.4254e−16
  13:             0,             0,             0
  14:  1.48052e−16,  2.23998e−16,  2.61029e−16

Testing SYMTRIDHH with various (symmetrized) 40 x 40 matrices
Type:   Eig 2−norm        RelRes           Orth
   1:   1.4584e−15,  1.17102e−15,  2.96644e−15
   2:   1.3886e−15,  1.07168e−15,   1.2189e−15
   3:  1.64518e−15,  1.07726e−15,  2.00745e−15
   4:  1.60898e−15,  9.65858e−16,  1.21956e−15
   5:  1.14063e−15,  5.03388e−16,  1.25668e−15
```

```
   6:  1.02648e−15,  7.06695e−16,  1.32482e−15
   7:  4.38545e−16,   7.8202e−16,  1.37105e−15
   8:  5.82761e−16,   1.2565e−15,  1.29653e−15
   9:  1.60193e−15,  3.38904e−15,  1.65435e−15
  10:  2.63217e−15,   1.0449e−15,  1.40833e−15
  11:  9.62474e−17,             0,             0
  12:  2.05082e−15,  6.01776e−16,  1.41359e−15
  13:             0,             0,             0
  14:  2.33872e−15,  1.61415e−15,  1.42786e−15

Testing SYMTRIDHH with various (symmetrized) 77 x 77 matrices
Type:   Eig 2−norm         RelRes          Orth
   1:  1.40815e−15,  1.19657e−15,  2.19079e−15
   2:  2.07201e−15,  6.29544e−16,  2.03332e−15
   3:  2.18977e−15,  1.33414e−15,   2.1724e−15
   4:  2.91409e−15,  1.22642e−15,  2.07284e−15
   5:  1.12263e−15,   9.3262e−16,  2.31048e−15
   6:  7.33699e−16,  5.72347e−16,  2.00698e−15
   7:  7.84042e−16,  3.62037e−16,  1.97703e−15
   8:  1.60016e−15,  2.34561e−15,  1.81197e−15
   9:  1.70559e−15,  5.28687e−15,  2.04223e−15
  10:  3.56296e−15,  9.98373e−16,    1.952e−15
  11:             0,             0,             0
  12:  1.14894e−15,  8.33801e−16,  1.76111e−15
  13:             0,             0,             0
  14:  1.47803e−15,  3.67251e−15,   3.5917e−15
```

Listing 22: Testing **symqrschur.m**

```
TEST SYMQRSCHUR
We compute eigenvalues of a test matrix using eig(),and then using
 the Schur decompositioncomputed by symqrschur, and display the norm
of the difference of the vector of eigenvaluesrelative to the norm of the
vector of eigenvalues itself . Also the norm of A−QDQ', and Q'Q−I.
All errors should be around 1e−15.

Testing SYMQRSCHUR with various (symmetrized) 5 x 5 matrices
Type: Eigval 2−norm Relativ Res  Orthogonal
   1:  1.05325e−16,   1.03336e−16,  1.16194e−16
   2:  4.31978e−16,   1.25925e−15,  1.10262e−15
   3:  1.62049e−15,   3.20683e−15,  1.96745e−15
   4:  1.83279e−16,   8.20152e−16,  9.26946e−16
   5:  3.01919e−16,   3.70511e−16,  9.14098e−16
   6:  5.00285e−16,   9.06193e−16,  7.76336e−16
   7:  2.41082e−16,    2.6901e−16,  4.55101e−16
   8:  4.62247e−16,   2.13728e−15,  1.23536e−15
   9:   1.1501e−15,   3.58771e−15,  1.56593e−15
```

```
 10:  6.80166e−16,   2.48341e−15,  2.17434e−15
 11:  2.39223e−16,   4.26977e−16,  3.10133e−16
 12:  1.87104e−16,   3.67036e−16,  5.76095e−16
 13:  4.91298e−16,     8.864e−16,   3.5439e−16
 14:   5.8554e−16,   1.29574e−15,  1.12105e−15

Testing SYMQRSCHUR with various (symmetrized) 30 x 30 matrices
Type: Eigval 2−norm Relativ Res   Orthogonal
  1:  8.14225e−16,   1.44968e−15,  1.60412e−15
  2:  8.77555e−16,   9.02361e−15,  3.40118e−15
  3:  1.09293e−15,   1.26153e−14,   3.3599e−15
  4:  1.79687e−15,   2.98163e−15,  3.02307e−15
  5:   1.3569e−15,   2.78376e−15,  6.73324e−15
  6:  5.04215e−16,   8.58818e−16,  3.86545e−15
  7:  7.30539e−16,   1.11851e−15,  4.14843e−15
  8:  1.02323e−15,   5.01782e−15,  3.70165e−15
  9:  6.73467e−16,   1.27762e−14,  2.90393e−15
 10:  2.05541e−15,   7.33445e−15,  4.38636e−15
 11:   4.6665e−16,   1.35997e−15,  7.25095e−16
 12:  1.67693e−15,   3.05752e−15,  3.15112e−15
 13:  1.12267e−15,   1.35535e−14,  3.50638e−15
 14:  2.36628e−15,   1.50356e−14,  5.32755e−15

Testing SYMQRSCHUR with various (symmetrized) 55 x 55 matrices
Type: Eigval 2−norm Relativ Res   Orthogonal
  1:  2.08774e−15,   2.14561e−15,  4.44727e−15
  2:   1.1805e−15,   8.38271e−15,  4.52572e−15
  3:  1.87731e−15,   5.94202e−15,  4.18512e−15
  4:  2.52331e−15,   8.45543e−15,  5.10079e−15
  5:    2.045e−15,    8.5375e−15,  6.26478e−15
  6:  7.01819e−16,   6.52831e−16,   7.5917e−15
  7:  8.08698e−16,   1.51094e−15,  5.79424e−15
  8:  1.04836e−15,   1.64218e−14,  4.18316e−15
  9:  1.15779e−15,   1.93888e−14,  4.14482e−15
 10:  3.02955e−15,   1.80621e−14,  6.32801e−15
 11:  5.21393e−16,   1.35997e−15,  6.85214e−16
 12:  1.05548e−15,   2.78322e−15,   9.1343e−15
 13:  9.77878e−16,    7.3179e−15,  4.78031e−15
 14:  2.39259e−15,   1.59616e−14,  5.96344e−15

Testing SYMQRSCHUR, all combinations of off−diagonal elements zero, 4 x 4 matrices
Type: Eigval 2−norm Relativ Res   Orthogonal
  0:              0              0              0
  1:  1.59745e−16   3.61914e−16  2.34172e−16
  2:  1.75542e−16    4.2101e−16  2.34172e−16
  3:  1.79625e−16   5.71586e−16  3.82964e−16
  4:  2.22045e−16   3.91189e−16  2.34172e−16
  5:   2.6537e−16   3.91189e−16  2.34172e−16
  6:  4.58548e−16   8.03103e−16  4.49518e−16
```

```
 7:  2.22237e−16   7.25453e−16  6.41224e−16
 8:            0             0            0
 9:  1.59745e−16   3.61914e−16  2.34172e−16
10:  1.75542e−16    4.2101e−16  2.34172e−16
11:  1.79625e−16   5.71586e−16  3.82964e−16
12:  2.22045e−16   3.91189e−16  2.34172e−16
13:   2.6537e−16   3.91189e−16  2.34172e−16
14:  4.58548e−16   8.03103e−16  4.49518e−16
```

Listing 23: Testing **gksvd.m**

```
TEST GKSVD
We compute singular values of a test matrix using the MATLAB svd(),
and then using our implementation ofthe Golub−Kahan SVD algorithm,
and display the norm of the difference , relative to the norm of the svs.
We also test the norm of the residual A−UDV', and
the orthogonality of U and V, namely norm U'U−I and V'V−I.
All errors should be around 1e−15.

Testing GKSVD with various 5 x 5 matrices
Type:  svd 2−norm       RelRes        U Orth        V Orth
 1:  5.87519e−16,  1.04787e−15,  7.3336e−16,  1.25795e−15
 2:  4.01541e−16,  1.45647e−15,  1.10424e−15,  1.27946e−15
 3:  6.76308e−16,  1.01623e−15,  1.38124e−15,  1.01521e−15
 4:  9.76752e−16,  8.93439e−16,  1.01021e−15,  9.40162e−16
 5:  2.91824e−16,  6.86821e−16,   1.1621e−15,  1.11282e−15
 6:  2.04015e−16,  2.00839e−16,  4.08378e−16,  4.39983e−16
 7:  1.92095e−16,   3.9175e−16,  8.92481e−16,   5.934e−16
 8:  2.86141e−16,  2.80871e−15,  7.24404e−16,  9.61336e−16
 9:  6.13808e−16,  2.08016e−15,  9.67864e−16,  1.35548e−15
10:  2.30991e−16,  4.92288e−16,   6.2181e−16,  5.15877e−16
11:  9.93014e−17,  2.22045e−16,  2.44416e−16,            0
12:   1.4498e−16,  4.48324e−16,  1.01027e−15,  8.19679e−16
13:  1.45095e−16,  8.68968e−16,   1.8206e−15,  6.50239e−16
14:  5.94255e−16,  9.56924e−16,  5.32616e−16,   9.6473e−16

Testing GKSVD with various 26 x 26 matrices
Type:  svd 2−norm       RelRes        U Orth        V Orth
 1:  1.52383e−15,  1.22263e−14,  3.18496e−15,  4.79706e−15
 2:  9.89758e−16,  1.15055e−14,  2.82807e−15,  2.85118e−15
 3:  9.78009e−16,  4.46019e−15,  2.88455e−15,   3.5885e−15
 4:  2.01854e−15,  6.79349e−15,  4.92374e−15,  3.77009e−15
 5:  1.59412e−15,  6.20694e−15,  4.13707e−15,  3.18288e−15
 6:  4.00402e−16,  7.16269e−16,  3.00348e−15,  1.40716e−15
 7:  1.77872e−16,  2.23186e−16,  2.01633e−15,  1.22069e−15
 8:  5.19853e−16,  1.96668e−14,  2.42661e−15,  2.68644e−15
 9:  7.52035e−16,  6.46906e−15,  3.14527e−15,  3.02264e−15
10:  1.75764e−15,  5.80753e−15,  2.60547e−15,  3.54454e−15
11:            0,            0,            0,            0
```

```
  12:  6.6764e−15,  7.07091e−15,  1.99596e−15,  1.79261e−15
  13:  1.29021e−15,  4.63315e−15,  2.92636e−15,  2.75782e−15
  14:  9.99883e−16,  8.17569e−15,  3.11266e−15,  3.40552e−15

Testing GKSVD with various 47 x 30 matrices
Type:   svd 2−norm       RelRes        U Orth        V Orth
   1:  2.09489e−15,  1.24293e−14,  5.12528e−15,  6.05448e−15
   2:  1.40148e−15,  8.98824e−15,  4.77663e−15,  5.55994e−15
   3:  1.94103e−15,  7.96737e−15,   5.624e−15,  4.15033e−15
   4:  2.34941e−15,  8.14611e−15,  6.32582e−15,  4.46316e−15
   5:  1.03754e−15,  5.37301e−15,  3.75205e−15,  3.86511e−15
   6:  3.31876e−16,  2.59627e−16,  2.56042e−15,  1.82067e−15
   7:  1.78272e−16,  2.41443e−16,  2.17324e−15,   1.1974e−15
   8:   6.4645e−16,  1.94766e−14,  3.76622e−15,  2.82941e−15
   9:  8.20157e−16,  1.61693e−14,  4.55499e−15,  3.69324e−15
  10:  1.34557e−15,   8.1102e−15,  4.20163e−15,  2.82183e−15
  11:             0,             0,             0,             0
  12:   1.1337e−14,  1.21113e−14,  2.24126e−15,  2.27147e−15
  13:  1.10332e−15,  4.40336e−15,  3.34795e−15,   3.8608e−15
  14:  1.16148e−15,  1.18339e−14,  2.70124e−15,  3.19951e−15

Testing GKSVD with various 68 x 30 matrices
Type:   svd 2−norm       RelRes        U Orth        V Orth
   1:  2.20044e−15,  1.24324e−14,  5.12526e−15,  6.05448e−15
   2:  7.26913e−16,  1.02812e−14,  2.58297e−15,  3.24464e−15
   3:  1.68435e−15,  7.20611e−15,  3.89819e−15,  4.04023e−15
   4:  1.32471e−15,  5.55433e−15,  3.72813e−15,  4.10325e−15
   5:  1.29926e−15,  1.43529e−14,  3.77122e−15,  4.36284e−15
   6:  4.62945e−16,  2.70363e−16,  2.43413e−15,  1.33213e−15
   7:  4.97419e−16,  6.59939e−16,  2.75997e−15,   1.5038e−15
   8:  6.47094e−16,  7.07282e−15,  2.73219e−15,  2.54129e−15
   9:   6.9036e−16,  1.14755e−14,   3.5923e−15,  3.26913e−15
  10:  1.21047e−15,  7.81022e−15,  2.88975e−15,  3.47656e−15
  11:             0,             0,             0,             0
  12:  2.20529e−15,  2.24672e−15,   2.7638e−15,  2.14134e−15
  13:  9.92265e−16,  4.13392e−15,  2.93278e−15,  3.08038e−15
  14:  1.13985e−15,  1.18384e−14,  2.69701e−15,  3.11648e−15

Testing GKSVD, all combinations of off−diagonal elements zero, 4 x 4 matrices
Type:   svd 2−norm       RelRes        U Orth        V Orth
   0:             0,             0,             0,             0
   1:  8.22155e−17,  2.67334e−16,  2.34172e−16,   2.3346e−16
   2:  7.97608e−17,  1.30236e−16,  3.39548e−16,  2.18373e−17
   3:  1.11022e−16,  1.92686e−16,  4.33868e−16,  3.50914e−16
   4:  1.78351e−16,  3.16573e−16,  1.23617e−16,  1.22554e−16
   5:  1.93295e−16,  3.16573e−16,  2.34172e−16,   2.3346e−16
   6:  2.83052e−16,   4.0698e−16,   3.3799e−16,  3.46887e−16
   7:  1.98038e−16,  4.56998e−16,  7.64141e−16,  5.34681e−16
   8:             0,             0,             0,             0
```

```
   9:  8.22155e−17,  2.67334e−16,  2.34172e−16,   2.3346e−16
  10:  7.97608e−17,  1.30236e−16,  3.39548e−16,  2.18373e−17
  11:  1.11022e−16,  1.92686e−16,  4.33868e−16,  3.50914e−16
  12:  1.78351e−16,  3.16573e−16,  1.23617e−16,  1.22554e−16
  13:  1.93295e−16,  3.16573e−16,  2.34172e−16,   2.3346e−16
  14:  2.83052e−16,   4.0698e−16,   3.3799e−16,  3.46887e−16
  15:  1.98038e−16,  4.56998e−16,  7.64141e−16,  5.34681e−16
```

Listing 24: Testing **gksvdprod.m**

```
TEST GKSVDPROD

Testing GKSVDPROD with various 5 x 5 matrices
Prod of types:    SVs 2−norm       RelRes         U Orth         V Orth
      1 ...   5:  3.13509e−16,  3.92705e−16,  1.20404e−15,  7.48128e−16
      2 ...   6:  2.52103e−16,  3.81329e−16,  1.83994e−15,  1.13721e−15
      3 ...   7:  6.04172e−16,  8.60865e−16,  2.62176e−16,  2.84868e−16
      4 ...   8:  2.79227e−16,  6.65333e−16,  1.41244e−15,  5.95482e−16
      5 ...   9:  4.51382e−16,  6.19393e−16,  9.20577e−16,  2.93687e−16
      6 ... 10:  1.99437e−15,  1.89229e−15,  9.92088e−16,  6.30896e−16
      7 ... 11:  4.05534e−16,  3.68961e−16,  7.01927e−16,  1.05206e−15
      8 ... 12:  2.50234e−16,  8.05426e−16,  9.30751e−16,   6.9201e−16
      9 ... 13:  3.88823e−16,  1.05393e−15,  1.59781e−15,  5.80918e−16
     10 ... 14:  1.19038e−16,    5.227e−16,  7.67048e−16,  5.10022e−16

Testing GKSVDPROD with various 26 x 26 matrices
Prod of types:    SVs 2−norm       RelRes         U Orth         V Orth
      1 ...   5:  8.67234e−16,  1.89524e−15,  2.52212e−15,  2.30427e−15
      2 ...   6:  1.75064e−16,  5.46424e−16,  2.96705e−15,  1.51283e−15
      3 ...   7:  3.03559e−16,  4.21144e−16,  2.92513e−15,  1.36347e−15
      4 ...   8:  2.72803e−16,  4.16389e−16,  2.66612e−15,  1.64363e−15
      5 ...   9:   7.4645e−16,  6.01465e−16,  2.83426e−15,  1.28927e−15
      6 ... 10:  6.49162e−16,  6.24111e−16,  2.22304e−15,  1.22372e−15
      7 ... 11:  1.28343e−15,  1.63018e−15,   1.5989e−15,  1.53967e−15
      8 ... 12:  1.96405e−15,   2.4361e−15,   1.5664e−15,  2.03085e−15
      9 ... 13:  2.02709e−15,  3.40895e−15,  1.33472e−15,  1.22484e−15
     10 ... 14:  7.45202e−16,  3.25605e−15,  1.65452e−15,  1.25866e−15

Testing GKSVDPROD with various 47 x 47 matrices
Prod of types:    SVs 2−norm       RelRes         U Orth         V Orth
      1 ...   5:  4.12439e−16,  5.06397e−16,  4.33516e−15,  5.19376e−15
      2 ...   6:  4.85653e−16,  1.00507e−15,  4.64492e−15,  1.73628e−15
      3 ...   7:  1.00014e−15,  1.38172e−15,  4.88242e−15,  2.56302e−15
      4 ...   8:  3.99377e−16,  7.53281e−16,  4.19321e−15,   2.1563e−15
      5 ...   9:  6.13696e−16,  7.36813e−16,  4.47864e−15,   1.9233e−15
      6 ... 10:  6.76974e−16,  9.63124e−16,  3.20063e−15,  1.81933e−15
      7 ... 11:  1.68318e−15,  1.35994e−15,     0.680988,   0.000115343
^^^^^^^^^^^^^^^^^^^ ^^^^^^^^^^^^^^^^^^^^^^
      8 ... 12:  1.15866e−15,  2.39725e−15,  1.85728e−15,  1.81418e−15
```

```
      9 ... 13:   1.80766e−15,  4.49616e−15,  2.00546e−15,  1.89891e−15
     10 ... 14:   7.53974e−16,  2.41885e−15,  2.28486e−15,  0.000832749
^^^^^^^^^^^^^^^^^^^ ^^^^^^^^^^^^^^^^^^^^^^^

Testing GKSVDPROD, random combinations of off−diagonal elements zero, 8 x 8
     matrices
Combinations:  SVs 2−norm       RelRes         U Orth         V Orth
         6   :  4.07443e−16,  9.56096e−16,  1.79258e−15,  1.34187e−15
         6   :  7.78901e−16,  1.57269e−15,  1.49449e−15,  1.42406e−15
         6   :  5.79042e−16,  1.13349e−15,  1.30182e−15,   1.798e−15
         6   :  1.34814e−15,  1.45932e−15,  1.84454e−15,  2.29575e−15
         6   :  4.32222e−16,   5.5043e−16,  1.49742e−15,  1.99867e−15
         6   :  4.98742e−16,  6.99294e−16,  1.20548e−15,  1.21745e−15
         6   :   6.1119e−16,   7.3617e−16,  2.05642e−15,  2.93709e−15
         6   :   1.3176e−15,   2.1998e−15,  1.41065e−15,  2.03242e−15
         6   :  1.16823e−15,  2.31999e−15,   1.4639e−15,  2.06746e−15
         6   :  4.17254e−16,  7.54149e−16,  8.27521e−16,  1.72915e−15
```

References

[1] Gene H. Golub and Charles F. Van Loan. *Matrix Computations*. The Johns Hopkins University Press, Baltimore, MD, USA, third edition, 1996.

[2] Gene H. Golub, Knut Sølna, and Paul Van Dooren. Computing the SVD of a general matrix product/quotient. *SIAM Journal on Matrix Analysis and Applications*, 22(1):1–19, 2000.