1 Introduction

Suppose $A$ is an $m \times n$ matrix. Then there exist two (different) unitary matrices: an $m \times m$ matrix $U$ and a $n \times n$ matrix $V$, and a $m \times n$ “diagonal” matrix $\Sigma$ with non-negative numbers on its diagonal and zeros elsewhere, so that

$$A = U \Sigma V^H$$

where the $^H$ denotes the Hermitian (or conjugate transpose) of a matrix, and the diagonal entries of $\Sigma$ are $\Sigma_{kk}$, with $k = \min(m, n)$. The triple of matrices $(U, \Sigma, V)$ is called the “singular value decomposition” (SVD) and the diagonal entries of $\Sigma$ are called the “singular values” of $A$. The columns of $U$ and $V$ are called the left and right “singular vectors” of $A$ respectively. You can get more information from a very nice Wikipedia article at http://en.wikipedia.org/wiki/Singular_value_decomposition. You can also find a very clear description of Beltrami’s original proof of the existence of the SVD in a simple case beginning in Section 2 (p. 5) of G. W. Stewart’s U. of Maryland report TR-92-31 (1992) at http://purl.umn.edu/1868.

As you have already seen, applications of the SVD include

- The right singular vectors corresponding to vanishing singular values of $A$ span the nullspace of $A$,
- The right singular vectors corresponding to positive singular values of $A$ span the domain of $A$.
- The left singular vectors corresponding to positive singular values of $A$ span the range of $A$.
- The rank of $A$ is the number of positive singular values of $A$.
- The singular values characterize the “relative importance” of some basis vectors in the domain and range spaces over others.
- Computing the Moore-Penrose “pseudo-inverse” of $A$, and making it possible to solve the system $Ax = b$ in the least-squares sense.

In this lab, you will see how the SVD can be computed, and get the flavor of the standard algorithm used for it.

This lab will take two sessions. You may find it convenient to print the pdf version of this lab rather than the web page itself.

2 Two algorithms

The easiest algorithm for SVD is to note the relationship between it and the eigenvalue decomposition: singular values are the square roots of the eigenvalues of $A^H A$ or $AA^H$. Indeed, if $A = U \Sigma V^H$, then

$$A^H A = (V \Sigma U^H)(U \Sigma V^H) = V(\Sigma^2)V^H, \text{ and }$$
$$AA^H = (U \Sigma V^H)(V \Sigma U^H) = U(\Sigma^2)U^H.$$
So, if you can solve for eigenvalues and eigenvectors, you can find the SVD.

Unfortunately, this is not a good algorithm because forming the product \( A^H A \) roughly squares the condition number, so that the eigenvalue solution is not likely to be accurate. Of course, it will work fine for small matrices with small condition numbers and you can find this algorithm presented in many web pages. Don’t believe everything you see on the internet.


The algorithm \textit{implicitly} computes the product \( AA^T \) and then uses a sequence of “Jacobi rotations” to diagonalize it. A Jacobi rotation is a \( 2 \times 2 \) matrix rotation that annihilates the off-diagonal term of a symmetric \( 2 \times 2 \) matrix. You can find more in a nice Wikipedia article located at \texttt{http://en.wikipedia.org/wiki/Jacobi_rotation}.

Given a \( 2 \times 2 \) matrix \( M \) with
\[
M^T M = \begin{bmatrix} \alpha & \gamma \\ \gamma & \beta \end{bmatrix}
\]
It is possible to find a “rotation by angle \( \theta \) matrix”
\[
\Theta = \begin{bmatrix} \cos \theta & -\sin \theta \\ \sin \theta & \cos \theta \end{bmatrix}
\]
with the property that \( \Theta^T M^T M \Theta = D \) is a diagonal matrix. \( \Theta \) can be found by multiplying out the matrix product, expressing the off-diagonal matrix component in terms of \( t = \tan \theta \), and setting it to zero to arrive at an equation
\[
t^2 + 2\zeta t - 1 = 0
\]
where \( \zeta = (\beta - \alpha)/(2\gamma) \). The quadratic formula gives \( t = \tan \theta \) and \( \sin \theta \) and \( \cos \theta \) can be recovered. There is no need to recover \( \theta \) itself.

The following algorithm repeatedly passes through the implicitly-constructed matrix \( AA^T \), choosing pairs of indices \( i < j \), constructing the \( 2 \times 2 \) submatrix from the intersection of rows and columns, and using Jacobi rotations to annihilate the off-diagonal term. Unfortunately, the Jacobi rotation for the pair \( i = 2, j = 10 \) messes up the rotation from \( i = 1, j = 10 \), so the process must be repeated until convergence. At convergence, the matrix \( \Sigma \) of the SVD has been implicitly generated, and the right and left singular vectors are recovered by multiplying all the Jacobi rotations together. The following algorithm carries out this process.

\textbf{Algorithm}

Given a convergence criterion \( \epsilon \), a matrix \( U \), that starts out as \( U = A \) and ends up as a matrix whose columns are left singular vectors, and another matrix \( V \) that starts out as the identity matrix and ends up as a matrix whose columns are right singular vectors.

\textbf{repeat}

\textbf{for all pairs} \( i < j \)

\textbf{(compute} \[
\begin{bmatrix} \alpha & \gamma \\ \gamma & \beta \end{bmatrix} \equiv \text{ the } (i,j) \text{ submatrix of } U^T U
\]
\[
\alpha = \sum_{k=1}^{n} U_{ki}^2 \\
\beta = \sum_{k=1}^{n} U_{kj}^2 \\
\gamma = \sum_{k=1}^{n} U_{ki} U_{kj} \\
\text{(compute the Jacobi rotation that diagonalizes} \begin{bmatrix} \alpha & \gamma \\ \gamma & \beta \end{bmatrix} )
\textbf{)}
\]
(a) Copy the following code skeleton to a function m-file named jacobi_svd.m. Complete the code so that it implements the above algorithm.

```matlab
function [U,S,V]=jacobi_svd(A) 
% [U S V]=jacobi_svd(A) 
% A is original matrix 
% Singular values come back in S (diag matrix) 
% orig matrix = U*S*V' 
% One-sided Jacobi algorithm for SVD 
% lawn15: Demmel, Veselic, 1989, 
% Algorithm 4.1, p. 32 

TOL=1.e-8;
n=size(A,1);
U=eye(n);
V=eye(n);
converge=TOL+1;
while converge>TOL
    converge=0;
    for j=2:n
        ...
    endfor
endfor
endfor
until all |c|/\sqrt{\alpha\beta} \leq \epsilon
```

The computed singular values are the norms of the columns of the final $U$ and the computed left singular vectors are the normalized columns of the final $U$. As mentioned above, the columns of $V$ are the computed right singular vectors.

**Exercise 1:**
for i=1:j-1
  % compute [alpha gamma;gamma beta]=(i,j) submatrix of U'*U
  alpha=??? %might be more than 1 line
  beta=??? %might be more than 1 line
  gamma=??? %might be more than 1 line
  converge=max(converge,abs(gamma)/sqrt(alpha*beta));

  % compute Jacobi rotation that diagonalizes
  % [alpha gamma;gamma beta]
  zeta=(beta-alpha)/(2*gamma);
  t=sign(zeta)/(abs(zeta)+sqrt(1+zeta^2));
  c=???
  s=???

  % update columns i and j of U
  t=U(:,i);
  U(:,i)=c*t-s*U(:,j);
  U(:,j)=s*t+c*U(:,j);

  % update matrix V of right singular vectors
  t=V(:,i);
  V(:,i)=???
  V(:,j)=???
end
end
end

% the singular values are the norms of the columns of U
% the left singular vectors are the normalized columns of U
for j=1:n
  singvals(j)=norm(U(:,j));
  U(:,j)=U(:,j)/singvals(j);
end
S=diag(singvals);

(b) Apply your version of jacobi_svd to the 2 \times 2 matrix A=U*S*V' generated from the following three matrices.

U=[0.6 0.8
   0.8 -0.6];
V=sqrt(2)/2*[1 1
                1 -1];
S=diag([5 4]);

It is easy to see that U and V are orthogonal matrices, so that the matrices U, S and V comprise the SVD of A. You may get a “division by zero” error, but this is harmless because it comes from gamma being zero, which will cause zeta to be infinite and t to be zero anyhow.

Your algorithm should essentially reproduce the matrices U, S and V. You might find that the diagonal entries of S are not in order, so that the U and V are similarly permuted, or you might observe that certain columns of U or V have been multiplied by (-1). Be sure, however, that an even number of factors of (-1) have been introduced.

If you do not get the right answers, you can debug your code in the following way. First, note that there is only a single term, i=1 and j=2 in the double for loop.
i. Bring up your code in the editor and click on the dash to the left of the the last line of code, the one that starts off “V(:,j)=”. This will cause a red dot to appear, indicating a “breakpoint.” (If you are not using the Matlab desktop, you can accomplish the same sort of thing using the keyboard command, placed just after the statement V(:,j)=.)

ii. Now, try running the code once again. You will find the code has stopped at the breakpoint.

iii. Take out a piece of paper and calculate the value of alpha. There are only two terms in this sum. Did you get the correct value? If not, fix it.

iv. Do the same for beta and gamma.

v. Similarly, check the values of s and c.

vi. Press the “Step” button to complete the calculation of V. Now multiply (you can do this at the Matlab command prompt) U*V'. Do you get A back? It can be shown that the algorithm always maintains the condition A=U*V' when it is correctly programmed. If not, you probably have something wrong in the two statements defining V, or, perhaps, you mis-copied the code updating U from the web page. Find the error. Remark: At this point in the iteration, U and V have not yet converged to their final values!

(c) Use your jacobi_svd to find the SVD of the matrix

\[ A_1 = \begin{bmatrix} 1 & 3 & 2 \\ 5 & 6 & 4 \\ 7 & 8 & 9 \end{bmatrix} \]

Compare the singular values with those you get from the Matlab svd function. You should find they agree almost to roundoff, despite the coarse tolerance inside jacobi_svd. You also will note that the values do not come out sorted, as they do from svd. It is a simple matter to sort them, but then you would have to permute the columns of U and V to match.

You now have an implementation of one SVD algorithm. This algorithm is a good one, with some realistic applications, and is one of the two algorithms available for the SVD in the GNU scientific library. (See http://www.gnu.org/software/gsl/ for more detail.) In the following sections, you will see a different algorithm.

3 The “standard” algorithm

The most commonly used SVD algorithm is found in Matlab and in the Lapack linear algebra library. (See http://www.netlib.org/lapack/.) It is a revised version of one that appeared in Golub and Van Loan. The revised version is presented by J. Demmel, W. Kahan, “Accurate Singular Values of Bidiagonal Matrices,” SIAM J. Sci. Stat. Comput., 11(1990) pp. 873-912. This paper can also be found at http://www.netlib.org/lapack/lawnspdf/lawn03.pdf.

The standard algorithm is a two-step algorithm. It first reduces the matrix to bidiagonal form and then finds the SVD of the bidiagonal matrix. Reduction to bidiagonal form is accomplished using Householder transformations, a topic you have already seen. Finding the SVD of a bidiagonal matrix is an iterative process that must be carefully performed in order to minimize both numerical errors and the number of iterations required. To accomplish these tasks, the algorithm chooses whether Golub and Van Loan’s original algorithm is better than Demmel and Kahan’s, or vice-versa. Further, other choices are made to speed up each iteration. There is not time to discuss all these details, so we will only consider a simplified version of Demmel and Kahan’s zero-shift algorithm.

In the Jacobi algorithm in the previous section, you saw how the two matrices U and V can be constructed by multiplying the various rotation matrices as the iterations progress. This procedure is the same for the standard algorithm, so, in the interest of simplicity, most of the rest of this lab will be concerned only with the singular values themselves.
The first step in the algorithm is to reduce the matrix to bidiagonal form. You have already seen how to use Householder matrices to reduce a matrix to upper-triangular form. Once that is done, the matrix can be transposed and Householder matrices can again be used to eliminate all non-zeros below the subdiagonal. You cannot reduce the matrix to diagonal form this way because the Householder matrices would change the diagonal entries and ruin the original factorization.

**Exercise 2**: Consider the following incomplete Matlab code, which is very similar to the h_factor function you wrote in Lab 7. (The matrices called Q and R there are called U and B here.)

```matlab
function [U,B,V]=bidiag_reduction(A)
    % [U B V]=bidiag_reduction(A)
    % Algorithm 6.5-1 in Golub & Van Loan, Matrix Computations
    % Johns Hopkins University Press
    % Finds an upper bidiagonal matrix B so that A=U*B*V'
    % with U,V orthogonal. A is an m x n matrix

    [m,n]=size(A);
    B=A;
    U=eye(m);
    V=eye(n);
    for k=1:n
        % eliminate non-zeros below the diagonal
        H=householder(B(:,k),k);
        B=H*B;
        U=U*H;
        % eliminate non-zeros to the right of the
        % superdiagonal by working with the transpose
        if k<n-1
            H=householder(B(k,:)',k+1);
            B=B*H';
            V=???
        end
    end
end
```

(a) Copy the above code to a file `bidiag_reduction.m`. Complete the statement with the question marks in it. Remember that the defining condition is always $A=U*B*V'$.

(b) Recover your version of `householder.m` from Lab 7 or use mine.

(c) Test your code on the matrix

```matlab
A1 = [ 1 3 2
      5 6 4
      7 8 9];
```

Be sure that the condition that $A1=U*B*V'$ is satisfied and that the matrix $B$ is bidiagonal.

(d) Test your code on a randomly-generated matrix

```matlab
A=rand(50,50);
```

Be sure that the condition that $A=U*B*V'$ is satisfied and that the matrix $B$ is bidiagonal.

An important piece of Demmel and Kahan’s algorithm is a very efficient way of generating a $2 \times 2$ “Givens” rotation matrix that annihilates the second component of a vector. The algorithm is presented on page 13 of their paper and is called “rot.”
Algorithm (Demmel, Kahan) \([c,s,r]=\text{rot}(f,g)\)

This algorithm computes the cosine, \(c\), and sine, \(s\), of a rotation angle that satisfies the following condition.

\[
\begin{bmatrix}
  c & s \\
  -s & c
\end{bmatrix}
\begin{bmatrix}
  f \\
  g
\end{bmatrix}
= \begin{bmatrix}
  r \\
  0
\end{bmatrix}
\]

if \(f = 0\) then
\[c = 0, \ s = 1, \text{ and } r = g\]
else if \(|f| > |g|\) then
\[t = g/f, \ t_1 = \sqrt{1 + t^2}
\[c = 1/t_1, \ s = tc, \text{ and } r = ft_1\]
else
\[t = f/g, \ t_1 = \sqrt{1 + t^2}
\[s = 1/t_1, \ c = ts, \text{ and } r = gt_1\]
endif

Exercise 3:

(a) Based on the above algorithm, write a Matlab function m-file named \texttt{rot.m} with the signature

\[
\text{function } [c,s,r]=\text{rot}(f,g)
\]

\[
% \text{rot}(f,g)
% \text{more comments}
% \text{your name and the date}
\]

(b) Test it on the vector \([1;0]\). You should get that \(c=1, \ s=0, \text{ and } r=1\).

(c) Test it on the vector \([1;2]\). Perform the matrix product

\[
\begin{bmatrix}
  c & s \\
  -s & c
\end{bmatrix} \begin{bmatrix}
  f \\
  g
\end{bmatrix}
\]

and check that the resulting vector is \([r;0]\).

The standard algorithm is based on repeated QR-type “sweeps” through the bidiagonal matrix. Suppose \(B\) is a bidiagonal matrix, so that it looks like the following.

\[
\begin{array}{ccccccc}
  d_1 & e_1 & 0 & 0 & \cdots \\
  0 & d_2 & e_2 & 0 & \cdots \\
  0 & 0 & d_3 & e_3 & \cdots \\
  \ddots & \ddots & \ddots & \ddots & \ddots \\
  \cdots & 0 & d_{n-1} & e_{n-1} \\
  \cdots & 0 & 0 & d_n
\end{array}
\]

In its simplest form, a sweep begins at the top left of the matrix and runs down the rows to the bottom. For row \(i\), the sweep first annihilates the \(e_i\) entry by multiplying on the right by a rotation matrix. This action introduces a non-zero value \(A_{i+1,i}\) immediately below the diagonal. This new non-zero is then annihilated by multiplying on the left by a rotation matrix but this action introduces a non-zero value \(B_{i,i+3}\), outside the upper diagonal, but the next row down. Conveniently, this new non-zero is annihilated by the same rotation matrix that annihilates \(e_{i+1}\). (The proof is a consequence of the fact that \(e_i\) was zero as a consequence of the first rotation, see the paper.) The algorithm continues until the final step on the bottom row that introduces no non-zeros. This process has been termed “chasing the bulge.”

The following Matlab code implements the above algorithm.
function [B]=msweep(B)
    % [B]=msweep(B)
    % Demmel & Kahan zero-shift QR downward sweep
    % B starts as a bidiagonal matrix and is returned as
    % a bidiagonal matrix

    n=size(B,1);
    for i=1:n-1
        [c s r]=rot(B(i,i),B(i,i+1));
        % construct matrix Q and multiply on the right by Q'
        % this annihilates both B(i-1,i+1) and B(i,i+1)
        % but makes B(i+1,i) non-zero
        Q=eye(n);
        Q(i:i+1,i:i+1)=[c s;-s c];
        B=B*Q';
        [c s r]=rot(B(i,i),B(i+1,i));
        % construct matrix Q and multiply on the left by Q
        % This annihilates B(i+1,i) but makes B(i,i+1) and
        % B(i,i+2) non-zero
        Q=eye(n);
        Q(i:i+1,i:i+1)=[c s;-s c];
        B=Q*B;
    end

    In this algorithm, there are two orthogonal (rotation) matrices, Q, employed. To see what their action
    is, consider the piece of B consisting of rows and columns i-1, i, i+1, and i+2. Multiplying on the right
    by the transpose of the first rotation matrix has the following consequence.

    \[
    \begin{pmatrix}
    \alpha & * & * & 0 \\
    0 & * & * & 0 \\
    0 & * & * & 0 \\
    0 & 0 & 0 & \gamma
    \end{pmatrix}
    \begin{pmatrix}
    1 & 0 & 0 & 0 \\
    0 & c & s & 0 \\
    0 & * & * & 0 \\
    0 & 0 & 0 & 1
    \end{pmatrix}
    =
    \begin{pmatrix}
    \alpha & \beta & 0 & 0 \\
    0 & * & * & 0 \\
    0 & * & * & 0 \\
    0 & 0 & 0 & \gamma
    \end{pmatrix}
    \]

    where asterisks indicate (possible) non-zeros and Greek letters indicate values that do not change. The fact
    that two entries are annihilated in the third column is a non-obvious consequence of the algorithm.

    The second matrix Q multiplies on the left and has the following consequence.

    \[
    \begin{pmatrix}
    1 & 0 & 0 & 0 \\
    0 & c & s & 0 \\
    0 & * & * & 0 \\
    0 & 0 & 0 & 1
    \end{pmatrix}
    \begin{pmatrix}
    \alpha & \beta & 0 & 0 \\
    0 & * & * & 0 \\
    0 & * & * & 0 \\
    0 & 0 & 0 & \gamma
    \end{pmatrix}
    =
    \begin{pmatrix}
    \alpha & \beta & 0 & 0 \\
    0 & * & * & 0 \\
    0 & * & * & 0 \\
    0 & 0 & 0 & \gamma
    \end{pmatrix}
    \]

    The important consequence of these two rotations is that the row with three non-zeros in it (the “bulge”) has moved from row \(i-1\) to row \(i\), while all other rows still have two non-zeros. This action is illustrated graphically in the following exercise.

    **Exercise 4:**

    (a) Copy the above code to a file named `msweep.m`. (The “m” stands for “matrix.”)
The following lines of code will set roundoff-sized matrix entries in $B$ to zero and the use Matlab’s \texttt{spy} routine to display all non-zeros. The \texttt{pause} statement makes the function stop and wait until a key is pressed. This will give you time to look at the plot.

\begin{verbatim}
% set almost-zero entries to true zero
% display matrix and wait for a key
B(find(abs(B)<1.e-13))=0;
spy(B)
disp('Plot completed. Strike a key to continue.')
pause
\end{verbatim}

Insert \textit{two} copies of this code into \texttt{msweep.m}, one after each of the statements that start off “$B=$”

Apply your \texttt{msweep} function to the matrix

\begin{verbatim}
B=[1 11 0 0 0 0 0 0 0 0
0 2 12 0 0 0 0 0 0 0
0 0 3 13 0 0 0 0 0 0
0 0 0 4 14 0 0 0 0 0
0 0 0 0 5 15 0 0 0 0
0 0 0 0 0 6 16 0 0 0
0 0 0 0 0 0 7 17 0 0
0 0 0 0 0 0 0 8 18 0
0 0 0 0 0 0 0 0 9 19
0 0 0 0 0 0 0 0 0 10];
\end{verbatim}

The sequence of plots shows the “bulge” migrates from the top of the matrix down the rows until it disappears off the end. Please include any one of the plots with your work.

Demmel and Kahan present a streamlined form of this algorithm in their paper. This algorithm is not only faster but is more accurate because there are no subtractions to introduce cancellation and roundoff inaccuracy. Their algorithm is presented below.

The largest difference in speed comes from the representation of the bidiagonal matrix $B$ as a pair of vectors, $d$ and $e$, the diagonal and superdiagonal entries, respectively. In addition, the intermediate product, $BQ'$, is not explicitly formed.

\textbf{Algorithm} (Demmel, Kahan)

This algorithm begins and ends with the two vectors $d$ and $e$, representing the diagonal and superdiagonal of a bidiagonal matrix. The vector $d$ has length $n$.

\begin{verbatim}
cold = 1
c = 1
for $i = 1$ to $n - 1$
\begin{verbatim}
  [c, s, r] = rot($c d_i$, $e_i$)
  if ($i \neq 1$) then $e_{i-1} = r s_{old}$
  \end{verbatim}
\begin{verbatim}
  [$c_{old}$, $s_{old}$, $d_i$] = rot($c_{old} r$, $d_{i+1}$)
\end{verbatim}
end for
\begin{verbatim}
h = c d_n
$e_{n-1} = h s_{old}$
d_n = h c_{old}
\end{verbatim}

\textbf{Exercise 5:}
\begin{enumerate}
  \item Based on the above algorithm, write a function m-file named \texttt{vsweep.m} (“v” for “vector”) with signature
  \end{enumerate}
function [d,e]=vsweep(d,e)
% [d e]=vsweep(d,e)
% comments

% your name and the date

(b) Use the same matrix as in the previous exercise to test vsweep. Compare the results of vsweep and msweep to be sure they are the same. You will probably want to remove the extra plotting statements from msweep.

B=[1 11 0 0 0 0 0 0 0 0
  0 2 12 0 0 0 0 0 0 0
  0 0 3 13 0 0 0 0 0 0
  0 0 0 4 14 0 0 0 0 0
  0 0 0 0 5 15 0 0 0 0
  0 0 0 0 0 6 16 0 0 0
  0 0 0 0 0 0 7 17 0 0
  0 0 0 0 0 0 0 8 18 0
  0 0 0 0 0 0 0 0 9 19
  0 0 0 0 0 0 0 0 0 10];
d=diag(B);
e=diag(B,1);

(c) Use the following code to compare results of msweep and vsweep and time them as well. Are the results the same? What are the times? You should find that vsweep is substantially faster.

n=200;
d=rand(n,1);
e=rand(n-1,1);
B=diag(d)+diag(e,1);
tic;B=msweep(B);mtime=toc
tic;[d e]=vsweep(d,e);vtime=toc
norm(d-diag(B))

It turns out that repeated sweeps tend to reduce the size of the superdiagonal entries. You can find a discussion of convergence in Demmel and Kahan, and the references therein, but the basic idea is to watch the values of e and when they become small, set them to zero. Demmel and Kahan prove that this does not perturb the singular values much. In the following exercise, you will see how the superdiagonal size decreases.

Exercise 6:

(a) Consider the same matrix you have been using:

d=[1;2;3;4;5;6;7;8;9;10];
e=[11;12;13;14;15;16;17;18;19];

Plot the absolute values $|e_i|$ (plot(abs(e))) for each of the first 15 iterations, $[d,e]=vsweep(d,e)$, all on the same plot (hold on). You should observe that some entries converge quite rapidly to zero, and they are all decreasing. Please include this plot with your summary.

(b) Plot $\|e\|$ versus iteration number for the first 40 iterations. Please include this plot with your summary.
(c) Plot $\|e\|$ versus iteration number on a semilog plot for the first 100 iterations. (Use `semilogy`.) You should see that the convergence eventually becomes linear. Please include this plot with your summary.

You should have observed that one entry converges very rapidly to zero, and that overall convergence to zero is asymptotically linear. It is often one of the two ends that converges most rapidly. Further, a judiciously chosen shift can make convergence even more rapid. We will not investigate these issues further here, but you can be sure that the software appearing, for example, in Lapack for the SVD takes advantage of these and other acceleration methods.

In the following exercise, the superdiagonal is examined during the iteration. As the values become small, they are set to zero, and the iteration continues with a smaller matrix. When all the superdiagonals are zero, the iteration is complete.

**Exercise 7:**

(a) Copy the following code to a function m-file named `mysvd.m`.

```matlab
function [d,iterations]=mysvd(d,e)
% [d,iterations]=mysvd(d,e)
% more comments
% your name and the date
TOL=100*eps;
n=length(d);
maxit=500*n^2;

% The following convergence criterion is discussed by
% Demmel and Kahan. First, estimate the smallest
% singular value.
lambda(n)=abs(d(n));
for j=n-1:-1:1
    lambda(j)=abs(d(j))*lambda(j+1)/(lambda(j+1)+abs(e(j)));
end
mu(1)=abs(d(1));
for j=1:n-1
    mu(j+1)=abs(d(j+1))*mu(j)/(mu(j)+abs(e(j)));
end
sigmaLower=min(min(lambda),min(mu));
thresh=max(TOL*sigmaLower,maxit*realmin);

iUpper=n-1;
iLower=1;
for iterations=1:maxit
    % reduce problem size when some zeros are
    % on the superdiagonal

    % how many zeros are near the bottom right?
    for i=iUpper:-1:1
        iUpper=i;
        if abs(e(i))>thresh
```
break;
end

% how many zeros are near the top left?
j=iUpper;
for i=iLower:iUpper
  if abs(e(i))>thresh
    j=i;
    break;
  end
end
iLower=j;

if (iUpper==iLower & abs(e(iUpper))<=thresh) | ... (iUpper<iLower)
  % all done, sort singular values
d=sort(abs(d));
  % sort results in ascending order, so reverse it
d(end:-1:1)=d;
  return
end

% do a sweep
[d(iLower:iUpper+1),e(iLower:iUpper)]= ...
  vsweep(d(iLower:iUpper+1),e(iLower:iUpper));
end
error('mysvd: too many iterations!')

(b) Consider the same matrix you have used before

d=[1;2;3;4;5;6;7;8;9;10];
e=[11;12;13;14;15;16;17;18;19];

What are the singular values? How many iterations did mysvd need? What is the largest difference between the singular values mysvd found and those that the Matlab function svd found for the same matrix? (You will have to expand the two diagonals into a matrix to use svd.)

(c) Use mysvd to compute the singular values of a randomly-generated matrix, d=rand(30,1) and e=rand(29,1). How many iterations does it take? What is the norm of the difference between the singular computed using mysvd and those computed using svd(diag(d)+diag(e,1))? 

Remark: In the above algorithm, you can see that the singular values are forced to be positive by taking absolute values. This is legal because if a negative singular value arises then multiplying both it and the corresponding column of $U$ by negative one does not change the unitary nature of $U$ and leaves the singular value positive.

You saw in the previous exercise that the number of iterations can get large. It turns out that the number of iterations is dramatically reduced by the proper choice of shift, and tricks such as choosing to run sweeps up from the bottom right to the top left instead of down as we did, depending on the matrix elements. Nonetheless, the presentation here should give you the flavor of the algorithm used.

In the following exercise, you will put your two functions, bidiag_reduction and mysvd to find the singular values of a full matrix.

Exercise 8:
(a) Given the matrix $A = \text{magic}(10)$, use $\text{bidiag\_reduction}$ to transform it into a bidiagonal matrix, $B$, and then use $\text{mysvd}$ to find its singular values. How do they compare with the singular values from the Matlab function $\text{svd}$?

(b) The singular values can be used to indicate the rank of a matrix. What is the rank of $A$? Why?