Chapter 23

Computing the Singular Value Decomposition

You should be familiar with

- The SVD theorem
- Jacobi rotations

In Chapter 15, we proved the singular value decomposition (SVD) by construction, discussed some information it provides about a matrix and showed how to use the SVD in image compression. In subsequent chapters, we applied the SVD to least squares and other problems. However, a significant issue remains. How do we efficiently compute the SVD? The chapter develops two algorithms for its computation. We begin with the one-sided Jacobi method, since it is based upon the use of Jacobi rotations, very similar to those we used in Chapter 19 in the computation of the eigenvalues of a symmetric matrix. We will then discuss the Demmel and Kahan Zero-shift QR Downward Sweep algorithm. This method involves using Householder reflections to transform any $m \times n$ matrix to a bidiagonal matrix. The bidiagonal matrix is then reduced to a diagonal matrix containing the singular values using bulge chasing, a technique presented in Section 18.8.

23.1 DEVELOPMENT OF THE ONE-SIDED JACOBI METHOD FOR COMPUTING THE REDUCED SVD

The SVD can be computed in the following way:

> *Find the singular values of A by computing the eigenvalues and orthonormal eigenvectors for A^TA. Place the square roots of the positive eigenvalues on the diagonal of the matrix $\widetilde{\Sigma}$ in order from greatest to least and fill all the other entries with zeros. These normalized eigenvectors form V. Find orthonormal eigenvectors of AA^T. These form the columns of U.*

This is a slow and potentially inaccurate means of finding the SVD. Roundoff errors can be introduced into the computation of A^TA that alter the correct eigenvalues. Here is an example.

Example 23.1. Let $A = \begin{bmatrix} 3.0556 & 3.0550 \\ 3.0550 & 3.0556 \end{bmatrix}$. The singular values of A are 6.1106 and 0.0006. Now compute A^TA using six-digit arithmetic and obtain

$$A^TA = \begin{bmatrix} 18.6697 & 18.6697 \\ 18.6697 & 18.6697 \end{bmatrix}.$$

The eigenvalues of A^TA are 6.1106 and 0.0000, as opposed to 6.1106 and 0.0006. ■

In this section, we will develop an algorithm for the reduced SVD based on Jacobi rotations, since we are already familiar with this approach to compute the eigenvalues of a symmetric matrix. The algorithm, known as the *one-sided Jacobi algorithm*, will generally give good results.

We need to avoid having to compute A^TA so we take an approach that will indirectly perform computations on A^TA while actually working with a sequence of seemingly different problems. We use a sequence of Jacobi rotations that will make columns $i, j, i < j$ of $AJ(i, j, c, s)$ orthogonal. The presence of $J(i, j, c, s)$ to the right of A is the reason the algorithm is called one-sided Jacobi. Consider the product

Numerical Linear Algebra with Applications. http://dx.doi.org/10.1016/B978-0-12-394435-1.00023-5

$$
\begin{array}{c}
\begin{array}{c} \\ i \\ \\ j \\ \\ \end{array}
\begin{array}{c}
\begin{array}{ccccccc} & & i & & j & & \end{array} \\
\left[\begin{array}{ccccccc}
a_{11} & & a_{1i} & \cdots & a_{1j} & \cdots & a_{1n} \\
\vdots & \ddots & \vdots & \cdots & \vdots & \cdots & \vdots \\
 & & a_{ii} & & \mathrm{a}_{ij} & \cdots & a_{in} \\
\vdots & & \vdots & \ddots & \vdots & & \\
 & & \mathrm{a}_{ji} & & \mathrm{a}_{jj} & & a_{jn} \\
\vdots & & \vdots & & \vdots & \ddots & \\
a_{n1} & & \mathrm{a}_{ni} & & a_{nj} & \cdots & a_{nn}
\end{array}\right] \\
A
\end{array}
\begin{array}{c} \\ i \\ \\ j \\ \\ \end{array}
\quad * \quad
\begin{array}{c} \\ i \\ \\ j \\ \\ \end{array}
\begin{array}{c}
\begin{array}{ccccccc} & & i & & j & & \end{array} \\
\left[\begin{array}{ccccccc}
1 & 0 & 0 & \cdots & \cdots & \cdots & 0 \\
\vdots & \ddots & & \cdots & \cdots & \cdots & \vdots \\
0 & & c & \cdots & s & \cdots & 0 \\
\vdots & & & \ddots & & & \\
0 & & -s & & c & & 0 \\
\vdots & & & & & \ddots & \\
0 & & & & & & 1
\end{array}\right] \\
J(i,j,c,s)
\end{array}
\end{array}
$$

which yields the matrix

$$
\begin{array}{c}
\begin{array}{c} \\ i \\ \\ j \\ \\ \end{array}
\begin{array}{c}
\begin{array}{ccccccc} & & i & & j & & \end{array} \\
\left[\begin{array}{ccccccc}
a_{11} & & ca_{1i} - sa_{1j} & \cdots & sa_{1i} + ca_{1j} & \cdots & a_{1n} \\
\vdots & \ddots & \vdots & \cdots & \vdots & \cdots & \vdots \\
a_{i1} & & ca_{ii} - sa_{ij} & & sa_{ii} + ca_{ij} & \cdots & a_{in} \\
\vdots & & \vdots & \ddots & \vdots & & \vdots \\
a_{j1} & & ca_{ji} - sa_{ji} & & sa_{ji} + ca_{jj} & & a_{jn} \\
\vdots & & \vdots & & \vdots & \ddots & \vdots \\
a_{n1} & & ca_{ni} - sa_{nj} & & sa_{ni} + sa_{nj} & \cdots & a_{nn}
\end{array}\right] \\
A
\end{array}
\end{array}
$$

Require that the vectors in columns i and j be orthogonal.

$$
\left\langle \begin{bmatrix} ca_{1i} - sa_{1j} \\ \vdots \\ ca_{ii} - sa_{ij} \\ \vdots \\ ca_{ji} - sa_{ji} \\ \vdots \\ ca_{ni} - sa_{nj} \end{bmatrix}, \begin{bmatrix} sa_{1i} + ca_{1j} \\ \vdots \\ sa_{ii} + ca_{ij} \\ \vdots \\ sa_{ji} + ca_{jj} \\ \vdots \\ sa_{ni} + ca_{nj} \end{bmatrix} \right\rangle = 0. \tag{23.1}
$$

Form the inner product in Equation 23.1 to obtain

$$
\begin{aligned}
&(ca_{1i} - sa_{1j})(sa_{1i} + ca_{1j}) + \cdots + (ca_{ii} - sa_{ij})(sa_{ii} + ca_{ij}) + \cdots + (ca_{ji} - sa_{ji})(sa_{ji} + ca_{jj}) + \cdots \qquad (23.2)\\
&\qquad + (ca_{ni} - sa_{nj})(sa_{ni} + ca_{nj}) = 0. \qquad (23.3)
\end{aligned}
$$

After some algebra, Equation 23.3 transforms to

$$
\left(c^2 - s^2\right) \sum_{k=1}^{n} a_{ki} a_{kj} + cs \left[\sum_{k=1}^{n} a_{ki}^2 - \sum_{k=1}^{n} a_{kj}^2 \right] = 0,
$$

and so

$$
\frac{c^2 - s^2}{cs} = \frac{\sum_{k=1}^{n} a_{kj}^2 - \sum_{k=1}^{n} a_{ki}^2}{\sum_{k=1}^{n} a_{ki} a_{kj}}. \tag{23.4}
$$

Proceed like we did with Equation 19.2, except that the right-hand side is different. The result is

$$
t^2 + 2\tau t - 1 = 0,
$$

where

$$
\tau = \frac{1}{2} \frac{\sum_{k=1}^{n} a_{kj}^2 - \sum_{k=1}^{n} a_{ki}^2}{\sum_{k=1}^{n} a_{ki} a_{kj}},
$$

and $s = ct$. Table 23.1 provides a summary of the required computations.

TABLE 23.1 Computation of c and s for the Jacobi One-Sided Method

$\tau = \frac{1}{2}\frac{\sum_{k=1}^{n} a_{kj}^2 - \sum_{k=1}^{n} a_{ki}^2}{\sum_{k=1}^{n} a_{ki}a_{kj}}$
$t = \left\{ \begin{array}{ll} \frac{1}{\tau+\sqrt{\tau^2+1}}, & \tau \geq 0 \\ \frac{-1}{-\tau+\sqrt{\tau^2+1}}, & \tau < 0 \end{array} \right)$
$c = \frac{1}{\sqrt{1+t^2}}$
$s = ct$

Now, what does this computation have to do with $A^{\mathrm{T}}A$? Require that the rotation $J(i,j,c,s)^{\mathrm{T}} A^{\mathrm{T}}AJ(i,j,c,s)$ zero-out the off diagonal entries at indices (i,j) and (j,i) of the symmetric matrix $A^{\mathrm{T}}A$. The entries of $A^{\mathrm{T}}A$ at indices (i,i), (i,j), (j,i), and (j,j) are shown in the following matrix:

$$A^{\mathrm{T}}A = \begin{array}{c} \\ i \\ \\ j \\ \\ \end{array} \begin{array}{c} \begin{array}{ccc} \quad i & \cdots & \quad j \end{array} \\ \left[\begin{array}{ccc} \cdots & \cdots & \cdots \\ \sum_{k=1}^{n} a_{ki}^2 & & \sum_{k=1}^{n} a_{ki}a_{kj} \\ \cdots & \cdots & \cdots \\ \sum_{k=1}^{n} a_{ki}a_{kj} & & \sum_{k=1}^{n} a_{kj}^2 \\ & \cdots & \end{array} \right] \end{array}$$

To zero-out $\left(A^{\mathrm{T}}A\right)_{ji}$ and $\left(A^{\mathrm{T}}A\right)_{ij}$ by forming $J(i,j,c,s)^{\mathrm{T}} A^{\mathrm{T}}AJ(i,j,c,s)$, proceed just as we did in Section 19.1, substituting $\sum_{k=1}^{n} a_{ki}^2$ for a_{ii}, $\sum_{k=1}^{n} a_{kj}^2$ for a_{jj}, and $\sum_{k=1}^{n} a_{ki}a_{kj}$ for a_{ji} and a_{ij}, and apply the results in Table 19.2. The values obtained are the same as those in Table 23.1. Choosing c and s so that columns i and j of $AJ(i,j,c,s)$ are orthogonal zeros-out the entries at indices (i,j) and (j,i) of $A^{\mathrm{T}}A$.

The algorithm now proceeds as follows. Start with A, and apply a sequence of right Jacobi rotations until the result is a matrix $\overline{U}$ with "nearly orthogonal" columns

$$AJ_1J_2J_3 \ldots J_k = \overline{U}. \tag{23.5}$$

Performing the Jacobi rotations given in Equation 23.5 is actually performing orthogonal similarity transformations on $A^{\mathrm{T}}A$, producing a matrix with the eigenvalues of $A^{\mathrm{T}}A$ on its diagonal.

$$J_k^{\mathrm{T}} \ldots J_2^{\mathrm{T}}J_1^{\mathrm{T}}A^{\mathrm{T}}AJ_1J_2 \ldots J_k \approx \Sigma^2, \tag{23.6}$$

$$\Sigma = \begin{bmatrix} \sigma_1 & & & 0 \\ & \sigma_2 & & \\ & & \ddots & \\ 0 & & & \sigma_n \end{bmatrix},$$

where the σ_i, $1 \leq i \leq n$, are the singular values of A. Let V be the orthogonal matrix $V = J_1J_2J_3 \ldots J_k$, so Equation 23.5 can be written as

$$AV = \overline{U}, \tag{23.7}$$

and

$$A = \overline{U}V^{\mathrm{T}}. \tag{23.8}$$

From Equation 23.8, $A^{\mathrm{T}} = V\overline{U}^{\mathrm{T}}$. Use this result in Equation 23.6 to obtain

$$J_k^{\mathrm{T}} \ldots J_2^{\mathrm{T}}J_1^{\mathrm{T}}V\overline{U}^{\mathrm{T}}AJ_1J_2 \ldots J_k \approx \Sigma^2.$$

Now, $AJ_1J_2\ldots J_k = AV = \overline{U}$ from Equation 23.7, so

$$J_k^{\mathrm{T}}\ldots J_2^{\mathrm{T}}J_1^{\mathrm{T}}V\overline{U}^{\mathrm{T}}\overline{U} \approx \Sigma^2$$

Since $V = J_1J_2J_3\ldots J_k$, we have

$$J_k^{\mathrm{T}}\ldots J_2^{\mathrm{T}}J_1^{\mathrm{T}}J_1J_2J_3\ldots J_k\overline{U}^{\mathrm{T}}\overline{U} \approx \Sigma^2$$

and

$$\overline{U}^{\mathrm{T}}\overline{U} = \Sigma^2. \tag{23.9}$$

Assuming that the columns of $\overline{U}$ are orthogonal, write it in the form $(\overline{u}_1\overline{u}_2\ldots\overline{u}_n)$, where the $\overline{u}_i$ are orthogonal, and Equation 23.9 can be written as follows:

$$\begin{bmatrix} \|\overline{u}_1\|_2^2 & & & 0 \\ & \|\overline{u}_2\|_2^2 & & \\ & & \ddots & \\ 0 & & & \|\overline{u}_n\|_2^2 \end{bmatrix} = \begin{bmatrix} \sigma_1^2 & & & 0 \\ & \sigma_2^2 & & \\ & & \ddots & \\ & & & \sigma_n^2 \end{bmatrix}. \tag{23.10}$$

Keep in mind that the columns of $\overline{U}$ are actually nearly orthogonal, so there will most likely be small entries off the diagonal. Equation 23.10 says that the 2-norm of the columns of $\overline{U}$ is approximately the singular values of A. Since $\frac{\overline{u}_i}{\sigma_i}$ is a unit vector, $U = \left(\frac{\overline{u}_1}{\sigma_1}\ \frac{\overline{u}_2}{\sigma_2}\ \cdots\ \frac{\overline{u}_n}{\sigma_n}\right)$ is an orthogonal matrix, and $\overline{U} = U\Sigma$. Note that $\overline{U} = U\Sigma$, and by using this in Equation 23.8, we have

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

the SVD of A.

We know that the Jacobi method applied to the symmetric matrix $A^{\mathrm{T}}A$ converges to a diagonal matrix containing its eigenvalues (Theorem 19.3). We stated to continue the Jacobi algorithm for the SVD until $AJ_1J_2J_3\ldots J_k$ is "nearly orthogonal." What test do we use to measure the extent of orthogonality, and will this test guarantee that the eigenvalues of $A^{\mathrm{T}}A$ are computed accurately? Let $\overline{u}_i$ and $\overline{u}_j$ be column vectors of $\overline{U}$. The error tolerance test is that the maximum value of expression 23.11 for all i, j in the current sweep is less than a prescribed tolerance.

$$\left|\left\langle \frac{u_i}{\|u_i\|_2}, \frac{u_j}{\|u_j\|_2} \right\rangle\right|. \tag{23.11}$$

This says that the inner product of the normalized columns of $\overline{U}$ should be small. A proof that this criterion leads to convergence can be found in the paper *Jacobi's method is more accurate than QR*, by Demmel and Veselić [82] and in a 1989 report by the same authors that can be found at http://www.netlib.org/lapack/lawnspdf/lawn15.pdf. This paper shows that Jacobi can compute small singular values with better relative accuracy than other commonly used methods.

23.1.1 Stability of Singular Value Computation

We have seen that the computation of the eigenvalues of a nonsymmetric matrix A can be ill-conditioned. A natural question to ask is whether the same is true for the computation of singular values. $A^{\mathrm{T}}A$ is symmetric, and so we know that the condition numbers of the eigenvalues of $A^{\mathrm{T}}A$ are one. However, theoretically we have to deal with a product of two matrices, and roundoff error will be present. Assuming that U and V have orthonormal columns, suppose we introduce errors δA into A, resulting in errors $\delta\Sigma$ in Σ. Then, $A + \delta A = U(\Sigma + \delta\Sigma)V^{\mathrm{T}}$, and $\Sigma + \delta\Sigma = U^{\mathrm{T}}(A + \delta A)V$. Orthogonal matrices preserve norms, so $\|\Sigma + \delta\Sigma\|_2 = \|A + \delta A\|_2$, and perturbations in A cause perturbations of roughly the same size in its singular values, so the computation of singular values is well conditioned. To this effect, see Ref. [19, pp. 366-367], where a proof of the following theorem is provided.

Theorem 23.1. *Let A and $A+E$ be $m\times n$ matrices, $m \geq n$. Let $\sigma_1 \geq \sigma_2 \geq \cdots \geq \sigma_n$ and $\tilde{\sigma}_1 \geq \tilde{\sigma}_2 \geq \cdots \tilde{\sigma}_n$ be, respectively, the singular values of A and $A + E$. Then $|\sigma_i - \tilde{\sigma}_i| \leq \|E\|_2$ for each i.*

Of course for Theorem 23.1 to be useful, E must be small. In addition, there are issues with small singular values. For a discussion of problems with small singular values and other singular value perturbation results, see Ref. [83].

23.2 THE ONE-SIDED JACOBI ALGORITHM

Algorithm 23.1 implements the one-sided Jacobi method for computing the reduced SVD. There are some notes you will need before reading the algorithm.

- Like the Jacobi algorithm for finding the eigenvalues of a real symmetric matrix, Algorithm 23.1 uses the cyclic-by-row method.
- Before performing an orthogonalization step, the norms of columns i and j of U are compared. If the norm of column i is less than that of column j, the two columns are switched. This necessitates swapping the same columns of V as well. This action assures that the singular values in S appear in decreasing order.
- The norms of the final columns of U are the approximation to the singular values. If a norm is less than the machine precision eps, it is assumed that the singular value is zero.
- If the matrix A has a row or column of zeros, the algorithm produces a decomposition $A = U\Sigma V^{\mathrm{T}}$, but U is not orthogonal, since it will have a row or column of zeros.

Algorithm 23.1 One-Sided Jacobi Algorithm

```
function JACOBISVD(A,tol,maxsweeps)
  % One-sided Jacobi method for computing the reduced SVD of
  % an m × n matrix.
  % Input: Matrix A, error tolerance tol, and the
  % maximum number of sweeps, maxsweeps.
  % Output: m × n orthogonal matrix U, n × n diagonal matrix Σ
  % containing the singular values of A in decreasing order, an
  % n × n orthogonal matrix V, and numsweeps, the number of sweeps required.
  % If the error tolerance is not obtained in maxsweeps, numsweeps = -1.
  U = A
  V = I
  singvals = 0
  tmp = [ 1  1  ...  1  1 ]^T
  errormeasure = tol + 1
  numsweeps = 0
  while (errormeasure ≥ tol) and (numsweeps ≤ maxsweeps) do
    numsweeps = numsweeps + 1
    for i = 1:n-1 do
      errormeasure = 0
      for j = i+1:n do
        normcoli = ‖U(:, i)‖_2
        normcolj = ‖U(:, j)‖_2
        if normcoli < normcolj then
          % Assure the singular values will appear in decreasing order in S.
          swap columns i and j of U and V
        end if
        α = Σ_{k=1}^{m} u_{ki}^2
        β = Σ_{k=1}^{m} u_{kj}^2
        γ = Σ_{k=1}^{m} u_{ki} u_{kj}
        if αβ ≠ 0 then
          errormeasure = max( errormeasure, |γ| / √(αβ) )
        end if
        % compute Jacobi rotation that makes columns i and j of U
        % orthogonal and also zeros-out (A^T A)_ij and (A^T A)_ji
```

```
            if γ ≠ 0 then
               ζ = (β−α)/(2γ)
               if ζ ≥ 0 then
                  t = 1/(|ζ|+√(1+ζ²))
               else
                  t = −1/(|ζ|+√(1+ζ²))
               end if
               c = 1/√(1+t²)   s = ct
            else
               c = 1
               s = 0
            end if
            % update columns i and j of U.
            t = U(:,i)
            U(:,i) = ct - s*U(:,j)
            U(:,j) = st + c*U(:,j)
            % update matrix V of right singular vectors.
            t = V(:,i)
            V(:,i) = ct - sV(:,j)
            V(:,j) = st + cV(:,j)
         end for
      end for
   end while
   % 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 do
      singvals_j = ‖U(:, j)‖_2
      if singvals_j > eps then
         U(:, j) = U(:, j)/singvals_j
      end if
   end for
   Σ = diag(singvals)
   if errormeasure ≥ tol then
      numsweeps = -1
   end if
end function
```

NLALIB: The function `jacobisvd` implements Algorithm 23.1. Its return values can be one of three forms:

a. `[U, S, V, maxsweeps] = jacobisvd(A,tol,maxsweeps)`
b. `S = jacobisvd(A,tol,maxsweeps)`
c. `jacobisvd(A,tol,maxsweeps)`

The default values of `tol` and `maxsweeps` are 1.0×10^{-10} and 10, respectively.

Example 23.2. The first part of the example finds the SVD for the Hanowa matrix of order 500. This matrix is often used as a test matrix for eigenvalue algorithms because all its eigenvalues lie on a line in the complex plane. We will apply `jacobisvd` to the matrix and compute $\|A - USV^{\mathrm{T}}\|_2$.

```
>> A = gallery('hanowa', 500);
[U S V] = jacobisvd(A, 1.0e-14);
norm(A - U*S*V')

ans =
     1.110223024625157e-16
```

For the second part, load the 20×20 matrix SMLSINGVAL.mat from the software distribution. It has singular values $\sigma_i, 1 \leq i \leq 15$ that range from 5.0 down to 1.0. The last five singular values are

$$\sigma_{16} = 1.0 \times 10^{-12}, \quad \sigma_{17} = 1.0 \times 10^{-13}, \quad \sigma_{18} = 1.0 \times 10^{-14}, \quad \sigma_{19} = 1.0 \times 10^{-15}, \quad \sigma_{20} = 0.5 \times 10^{-15}.$$

Compute the singular values of SMLSINGVAL and output the smallest six with 16 significant digits.

```
>> S = jacobisvd(SMLSINGVAL,1.0e-15);
>> for i = 15:20
      fprintf('S(%d) = %.16g\f',i,S(i));
   end
S(15) = 1.1000000000000001
S(16) = 0.0000000000010000
S(17) = 0.0000000000001000
S(18) = 0.0000000000000100
S(19) = 0.0000000000000010
S(20) = 0.0000000000000005
```

■

23.2.1 Faster and More Accurate Jacobi Algorithm

A variant of the one-sided Jacobi algorithm described in Refs. [84, 85] provides higher accuracy and speed than the algorithm we have described. The algorithm uses rank-revealing QR with column pivoting [2, pp. 276-280] that generates a decomposition $AP = QR$, where P is a permutation matrix. The algorithm described in the two papers delivers outstanding performance, and very rapidly computes the SVD of a dense matrix with high relative accuracy. The speed of the algorithm is comparable to the classical methods. The algorithm is said to be a preconditioned Jacobi SVD algorithm. Computation of singular values is well conditioned; however, there are some classes of matrices for which the computation of singular values appears ill-conditioned [84, p. 1323]. This is termed *artificial ill-conditioning*, and the algorithm handles this phenomenon correctly, while bidiagonalization-based methods do not. This algorithm is too complex for presentation in the text, but there are some interesting facets of the algorithm we can present.

After computing the QR decomposition of $m \times n$ matrix A with partial pivoting $m \geq n$, the SVD of A and the upper-triangular matrix R have the same singular values. Let

$$AP = QR, \tag{23.12}$$

and then

$$\begin{aligned} A^{\mathrm{T}}A &= \left(QRP^{\mathrm{T}}\right)^{\mathrm{T}}\left(QRP^{\mathrm{T}}\right) = \\ PR^{\mathrm{T}}Q^{\mathrm{T}}QRP^{\mathrm{T}} &= P\left(R^{\mathrm{T}}R\right)P^{\mathrm{T}}. \end{aligned}$$

P is an orthogonal matrix, so $A^{\mathrm{T}}A$ and $R^{\mathrm{T}}R$ have the same eigenvalues. As we will see, the only SVD computation is for the upper $n \times n$ submatrix of R.

The algorithm deals with two cases, $\mathrm{rank}\,(\mathrm{A}) = \mathrm{n}$, and $\mathrm{rank}\,(\mathrm{A}) = \mathrm{rA} < \mathrm{n}$. If $\mathrm{rank}\,(\mathrm{A}) = \mathrm{n}$, we can compute the SVD of A using the following steps:

a. Compute $AP = QR$ using column pivoting.
b. Let $U = I^{m \times m}$ and $V = I^{n \times n}$
c. Compute the SVD of $R^T\,(1:n, 1:n)$ using the enhanced Jacobi method:
$$\left[\ \hat{V},\ \hat{\Sigma},\ U\,(1:n, 1:n)\ \right] = \mathrm{enhanced\,Jacobi}\left(\mathrm{R}\,(1:\mathrm{n}, 1:\mathrm{n})^{\mathrm{T}}\right)$$
d. Form $U = QU$ and $V = P\hat{V}$.
e. Let Σ be the $m \times n$ zero matrix with $\hat{\Sigma}$ placed in its upper left-hand corner.

To see that this works, note that

$$\begin{aligned} R\,(1:n, 1:n)^{\mathrm{T}} &= \hat{V}\hat{\Sigma}U\,(1:n, 1:n)^{\mathrm{T}}, \\ R\,(1:n, 1:n) &= U\,(1:n, 1:n)\,\hat{\Sigma}\hat{V}^{\mathrm{T}}, \\ U\,(1:n, 1:n)^{\mathrm{T}}\,R\,(1:n, 1:n) &= \hat{\Sigma}\hat{V}^{\mathrm{T}}. \end{aligned}$$

Form

$$
\begin{aligned}
U\Sigma V^{\mathrm{T}} &= Q \begin{bmatrix} U(1:n,1:n) & 0 & \dots & 0 \\ 0 & 1 & & \\ \vdots & & \ddots & \\ 0 & & & 1 \end{bmatrix} \begin{bmatrix} \tilde{\Sigma} \\ 0 \\ \vdots \\ 0 \end{bmatrix} \hat{V}^{\mathrm{T}} P^{\mathrm{T}} \\
&= Q \begin{bmatrix} U(1:n,1:n) & 0 & \dots & 0 \\ 0 & 1 & & \\ \vdots & & \ddots & \\ 0 & & & 1 \end{bmatrix} \begin{bmatrix} \hat{\Sigma}\hat{V}^{\mathrm{T}} \\ 0 \\ \vdots \\ 0 \end{bmatrix} P^{\mathrm{T}} \\
&= Q \begin{bmatrix} R(1:n,1:n) \\ 0 \\ \vdots \\ 0 \end{bmatrix} P^{\mathrm{T}} = A.
\end{aligned}
$$

The case where rank $(A) = rA < n$ is somewhat more involved. Problem 23.9 asks you to implement a simplified version of this algorithm, using `jacobisvd` to compute the required decomposition for a submatrix of R. The problem provides the code to handle the rank-deficient case.

23.3 TRANSFORMING A MATRIX TO UPPER-BIDIAGONAL FORM

The Demmel and Kahan Zero-shift QR Downward Sweep algorithm for computing the SVD first reduces A to a bidiagonal matrix. The outline of an algorithm for transforming an $m \times n$ matrix to upper-bidiagonal form is easy to understand graphically. Let $k = \min(m-1, n)$. First, use premultiplication by a Householder matrix to zero-out $a_{21}, a_{31}, \dots, a_{m1}$. Now zero-out elements $a_{13}, a_{14}, \dots, a_{1n}$ of A using postmultiplication by a Householder matrix.

$$
A = \begin{bmatrix} * & * & * & * & * & * \\ * & * & * & * & * & * \\ * & * & * & * & * & * \\ * & * & * & * & * & * \\ * & * & * & * & * & * \\ * & * & * & * & * & * \end{bmatrix} \xrightarrow{H_{u_1}A} \begin{bmatrix} X & * & * & * & * & * \\ 0 & X & * & * & * & * \\ 0 & * & X & * & * & * \\ 0 & * & * & X & * & * \\ 0 & * & * & * & X & * \\ 0 & * & * & * & * & X \end{bmatrix} \xrightarrow{H_{u_1}AH_{v_1}} \begin{bmatrix} X & X & 0 & 0 & 0 & 0 \\ 0 & X & * & * & * & * \\ 0 & * & X & * & * & * \\ 0 & * & * & X & * & * \\ 0 & * & * & * & X & * \\ 0 & * & * & * & * & X \end{bmatrix} = A_1.
$$

Repeat the process by using Householder matrices to zero-out elements $a_{32}, a_{42}, \dots, a_{m2}$ and $a_{24}, a_{25}, \dots, a_{2n}$.

$$
A_1 = \begin{bmatrix} X & X & 0 & 0 & 0 & 0 \\ 0 & X & * & * & * & * \\ 0 & * & X & * & * & * \\ 0 & * & * & X & * & * \\ 0 & * & * & * & X & * \\ 0 & * & * & * & * & X \end{bmatrix} \xrightarrow{H_{u_2}H_{u_1}AH_{v_1}} \begin{bmatrix} X & X & 0 & 0 & 0 & 0 \\ 0 & X & * & * & * & * \\ 0 & 0 & X & * & * & * \\ 0 & 0 & * & X & * & * \\ 0 & 0 & * & * & X & * \\ 0 & 0 & * & * & * & X \end{bmatrix} \xrightarrow{H_{u_2}H_{u_1}AH_{v_1}H_{v_2}} \begin{bmatrix} X & X & 0 & 0 & 0 & 0 \\ 0 & X & X & 0 & 0 & 0 \\ 0 & 0 & X & * & * & * \\ 0 & 0 & * & X & * & * \\ 0 & 0 & * & * & X & * \\ 0 & 0 & * & * & * & X \end{bmatrix} = A_2.
$$

Execute the pre- and postmultiplication $k-1$ times, and finish with one more premultiplication. Through the series of Householder reflections, we have formed the upper-bidiagonal matrix B as follows:

$$
B = H_{u_k} H_{u_{k-1}} \dots H_{u_1} A H_{v_1} H_{v_2} \dots H_{v_{k-1}}.
$$

Since the Householder matrices are orthogonal, the singular values of B are the same as those of A.

Generating the postproduct by H_{v_i} requires an explanation. If we compute A^{T}, then $a_{i,i+1}, a_{i,i+2}, a_{i,i+3}, \dots, a_{i,n}$ are in column i, and we can compute a Householder reflection that zeros them out. By again taking the transpose, the required elements of row i are zero. Taking the transpose is inefficient, so we proceed as follows:

Let $B = A^{\mathrm{T}}$.

Compute Householder reflection H_{v_i} that zeros-out $b_{i+2,i}, b_{i+3,i}, \dots, b_{n,i}$ and form $H_{v_i}B = H_{v_i}A^{\mathrm{T}}$. Recalling that $H_{v_i}^{\mathrm{T}} = H_{v_i}$, take the transpose of $H_{v_i}A^{\mathrm{T}}$, and we have AH_{v_i}, a matrix in which the elements $a_{i,i+2}, a_{i,i+3}, \dots, a_{i,k}$ are zero. Compute AH_{v_i} implicitly using Equation 17.12.

Example 23.3. This example illustrates the conversion to bidiagonal form step by step for the matrix $A = \begin{bmatrix} 1 & 5 & 3 \\ 1 & 0 & -7 \\ 3 & 8 & 9 \end{bmatrix}$. Of course, the Householder matrices are not actually formed.

$$H_{u_1} = \begin{bmatrix} -0.3015 & -0.3015 & -0.9045 \\ -0.3015 & 0.9302 & -0.2095 \\ -0.9045 & -0.2095 & 0.3714 \end{bmatrix} \quad H_{u_1}A = \begin{bmatrix} -3.3166 & -8.7438 & -6.9348 \\ 0 & -3.1839 & -9.3015 \\ 0 & -1.5518 & 2.0955 \end{bmatrix} \quad H_{v_1} = \begin{bmatrix} 1.0000 & 0 & 0 \\ 0 & -0.7835 & -0.6214 \\ 0 & -0.6214 & 0.7835 \end{bmatrix}$$

$$H_{u_1}AH_{v_1} = \begin{bmatrix} -3.3166 & 11.1600 & 0 \\ 0 & 8.2745 & -5.3092 \\ 0 & -0.0863 & 2.6061 \end{bmatrix} \quad H_{u_2} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & -0.9999 & 0.0104 \\ 0 & 0.0104 & 0.9999 \end{bmatrix} \quad B = H_{u_2}H_{u_1}AH_{v_1} = \begin{bmatrix} -3.3166 & 11.1600 & 0 \\ 0 & -8.2750 & 5.3361 \\ 0 & 0 & 2.5506 \end{bmatrix}$$

■

Algorithm 23.2 describes the reduction to upper-bidiagonal form. Note that the function

$$[\, A,\ u \,] = \text{hzero2}\,(A, i, j, \text{row})$$

zeros-out the required column elements if row $= 0$ and the required row elements if row $= 1$. Its implementation is in the book software distribution.

Algorithm 23.2 Reduction of a Matrix to Upper-bidiagonal Form

```
function BIDIAG(A)
  % Reduces the m × n matrix A to bidiagonal form.
  % Input: matrix A
  % Output: matrix B in upper-bidiagonal form.
  k = min (m − 1, n)
  for i = 1:k do
    A = hzero2(A,i,i)
    if i ≤ k then
      A = hzero2 (A, i, i + 1, 1)
    end if
  end for
  return A
end function
```

NLALIB: The function `bidiag` implements Algorithm 23.2.

Remark 23.1. The book software distribution contains a function `bidiagdemo` that illustrates the algorithm. A press of the space bar graphically shows the location of the nonzero elements. NLALIB contains a 4×4 matrix SVALSDEMO that serves well for this purpose.

23.4 DEMMEL AND KAHAN ZERO-SHIFT *QR* DOWNWARD SWEEP ALGORITHM

We presented the one-sided Jacobi algorithm because it is based on ideas we have discussed previously and because research has proven it is capable of high accuracy. For the one-sided Jacobi method, our MATLAB implementation returned U, S, and V or a vector containing the singular values.

For many years, the Golub-Kahan-Reinsch algorithm has been the standard for SVD computation [25, 86]. It involves working implicitly with $A^{\mathrm{T}}A$. We will not discuss this algorithm but, instead, present the Demmel and Kahan zero-shift *QR* downward sweep algorithm, since it has excellent performance, and it reinforces our understanding of bulge chasing introduced in Section 18.8 [87]. A paper describing the algorithm can be accessed from the Internet at http://www.netlib.org/lapack/lawnspdf/lawn03.pdf. We will develop the algorithm to return only a vector of singular values. The algorithm executes in two stages. The first stage transforms an $m \times n$ matrix, $m \geq n$, into an upper-bidiagonal matrix using Householder reflections, and then this matrix is transformed into a diagonal matrix of singular values, again using products of orthogonal matrices.

$$\text{Stage 1} \quad A \Rightarrow B = \begin{bmatrix} b_{11} & b_{12} & & & 0 \\ & b_{22} & b_{23} & & \\ & & \ddots & \ddots & \\ & & & \ddots & b_{n-1,n} \\ 0 & & & & b_{nn} \end{bmatrix}$$

$$\text{Stage 2} \quad B \Longrightarrow \tilde{\Sigma} = \begin{bmatrix} \sigma_1 & & & & 0 \\ & \ddots & & & \\ & & \sigma_r & & \\ & & & 0 & \\ 0 & & & & 0 \end{bmatrix}$$

Phase 2 is similar to the implicit *QR* algorithm bulge chasing, and its ultimate aim is to eliminate the superdiagonal entries at indices $(1,\ 2)\,, (2,\ 3)\,, (3,\ 4)\,, \ldots, (n-1,\ n)$, leaving the singular values on the diagonal. In each pass, a rotation is applied on the right to zero-out an element of the superdiagonal. In the process, a nonzero element is introduced in a location where we don't want it (the bulge), but another element is zeroed-out as a side effect. The algorithm then applies a rotation on the left to remove the nonzero element created from the previous rotation but creates nonzeros in two other locations. After the last pass, the matrix remains in upper-bidiagonal form. By repeating the $k-1$ passes repeatedly, convergence to a diagonal matrix of singular values occurs. We will not attempt to explain why this algorithm works, but will just demonstrate the process. The interested reader should refer to http://www.netlib.org/lapack/lawnspdf/lawn03.pdf.

Actions in a Pass

Step $i = 1$:

Zero-out the entry at $(1,\ 2)$ *by multiplying on the right by a rotation matrix. This action introduces a non-zero value at* $(2,\ 1)$ *immediately below the diagonal.*
Multiply by a rotation on the left to zero-out $(2,\ 1)$. *This introduces nonzeros at indices*$(1,\ 2)$ *and* $(1,\ 3)$.

Steps $i = 2$ through $(k-2)$:

Multiply by a rotation on the right that zeros-out $(i, i+1)$ *and, as a side effect,* $(i-1,\ i+1)$. *This leaves a nonzero value at index* $(i+1,\ i)$.
Zero-out $(i+1,\ i)$. *This leaves nonzeros at indices* $(i,\ i+1)$ *and* $(i,\ i+2)$.

Step $i = k-1$:

Multiply by a rotation on the right that zeros-out $(i, i+1)$ *and, as a side effect,* $(i-1,\ i+1)$. *This leaves a nonzero value at index* $(i+1,\ i)$.
Zero-out $(i+1,\ i)$. *The matrix remains in upper-bidiagonal form.*

We use a 5×5 matrix to illustrate one pass of the algorithm.

$$A = \begin{bmatrix} * & * & 0 & 0 & 0 \\ 0 & * & * & 0 & 0 \\ 0 & 0 & * & * & 0 \\ 0 & 0 & 0 & * & * \\ 0 & 0 & 0 & 0 & * \end{bmatrix}.$$

Step 1*: Develop a Givens rotation, J_{r_1}, that zeros-out* (1, 2) *but generates a nonzero value at index* (2, 1).

$$A = \begin{bmatrix} * & 0 & 0 & 0 & 0 \\ X & * & * & 0 & 0 \\ 0 & 0 & * & * & 0 \\ 0 & 0 & 0 & * & * \\ 0 & 0 & 0 & 0 & * \end{bmatrix}.$$

Develop a rotation, J_{l_1}, that zeros-out (2, 1). *It introduces nonzeros at* (1, 2) *and* (1, 3).

$$A = \begin{bmatrix} * & X & X & 0 & 0 \\ 0 & * & * & 0 & 0 \\ 0 & 0 & * & * & 0 \\ 0 & 0 & 0 & * & * \\ 0 & 0 & 0 & 0 & * \end{bmatrix}.$$

Step 2*: Create a rotation, J_{r_2}, that zeros-out* (2, 3). *It leaves a nonzero at* (3, 1) *but zeros out* (1, 3).

$$A = \begin{bmatrix} * & * & 0 & 0 & 0 \\ 0 & * & 0 & 0 & 0 \\ X & 0 & * & * & 0 \\ 0 & 0 & 0 & * & * \\ 0 & 0 & 0 & 0 & * \end{bmatrix}.$$

Develop a rotation, J_{l_2}, that zeros-out (3, 1) *and leaves nonzeros at* (2, 3) *and* (2, 4)

$$A = \begin{bmatrix} * & * & 0 & 0 & 0 \\ 0 & * & X & X & 0 \\ 0 & 0 & * & * & 0 \\ 0 & 0 & 0 & * & * \\ 0 & 0 & 0 & 0 & * \end{bmatrix}.$$

Step 3: *Compute a rotation J_{r_3} to zero-out* (3, 4) *that leaves a nonzero at* (4, 3) *but zeros-out* (2, 4).

$$A = \begin{bmatrix} * & * & 0 & 0 & 0 \\ 0 & * & * & 0 & 0 \\ 0 & 0 & * & 0 & 0 \\ 0 & 0 & X & * & * \\ 0 & 0 & 0 & 0 & * \end{bmatrix},$$

Develop a rotation, J_{l_3}, that zeros-out (4, 3), *leaving nonzeros at* (3, 4) *and* (3, 5).

$$A = \begin{bmatrix} * & * & 0 & 0 & 0 \\ 0 & * & * & 0 & 0 \\ 0 & 0 & * & X & X \\ 0 & 0 & 0 & * & * \\ 0 & 0 & 0 & 0 & * \end{bmatrix}.$$

Step 4: *Multiply by a rotation J_{r_4} that zeros-out* (4, 5), *leaves a nonzero at* (5, 4), *and zeros-out* (3, 5).

$$A = \begin{bmatrix} * & * & 0 & 0 & 0 \\ 0 & * & * & 0 & 0 \\ 0 & 0 & * & * & 0 \\ 0 & 0 & 0 & * & 0 \\ 0 & 0 & 0 & X & * \end{bmatrix}.$$

As the final operation, multiply by a rotation J_{l_4} that zeros-out (5, 4) *and places a nonzero value at* (4, 5).

$$A = \begin{bmatrix} * & * & 0 & 0 & 0 \\ 0 & * & * & 0 & 0 \\ 0 & 0 & * & * & 0 \\ 0 & 0 & 0 & * & * \\ 0 & 0 & 0 & 0 & * \end{bmatrix}.$$

A pictorial view of a downward sweep is useful, and Figure 23.1 depicts each step graphically using a 4×4 matrix.

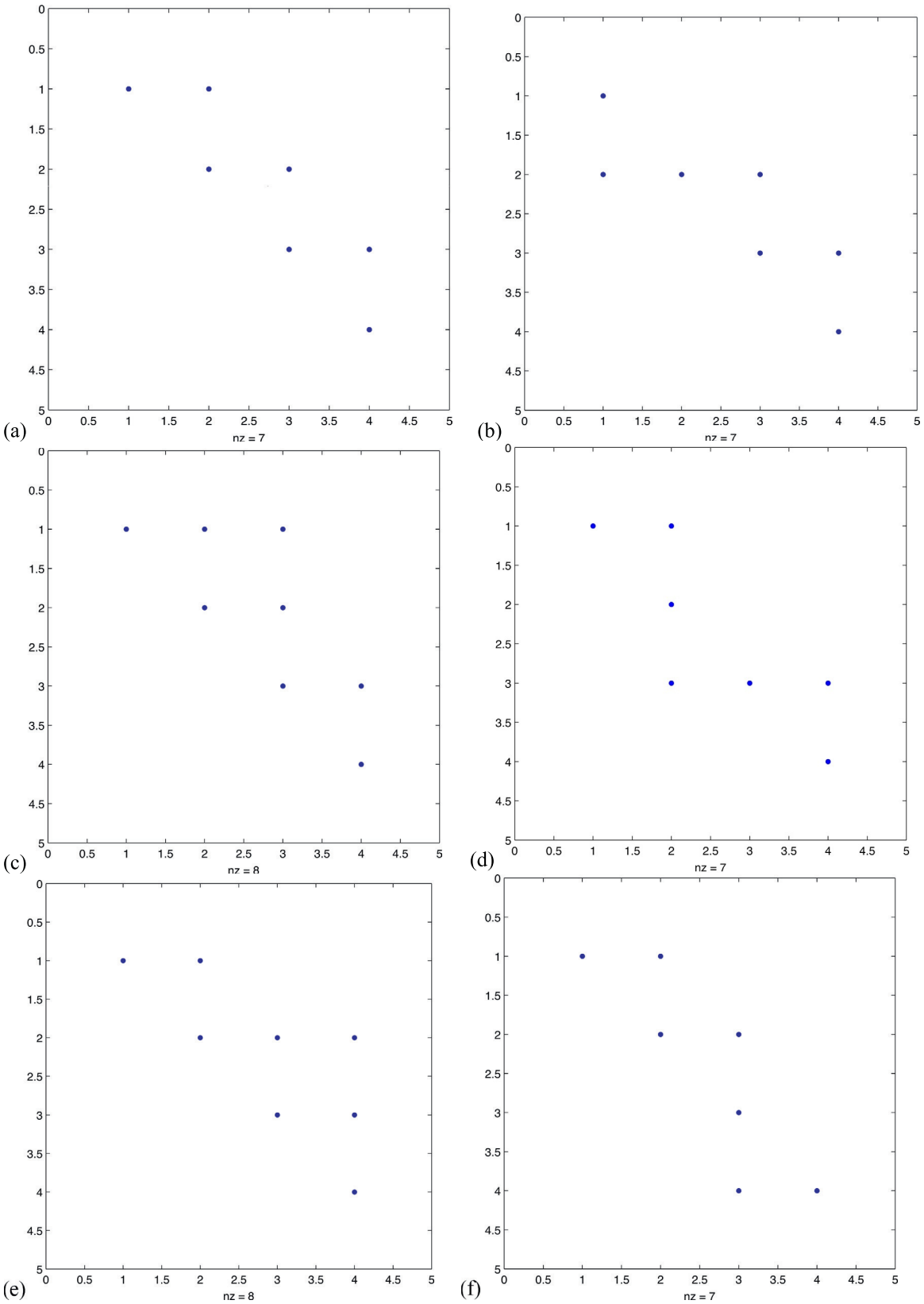
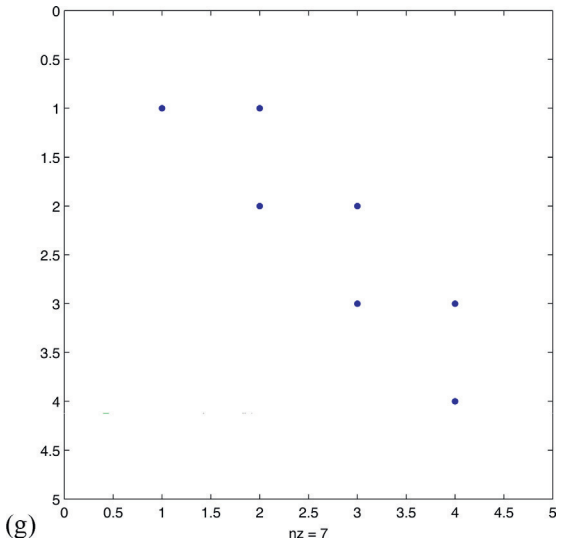


FIGURE 23.1 Demmel and Kahan zero-shift *QR* downward sweep.

Continued



(g)

FIGURE 23.1, CONT'D

The algorithm as described in Ref. [87] is of production quality. It describes very fast rotations to speed up the algorithm, switches between downward and upward sweeps depending on conditions, applies sophisticated convergence criteria, and so forth. A basic version of the algorithm is not difficult to understand, and we present it in Algorithm 23.3. Note that it uses deflation for efficiency and accuracy and features a function givensmulpsvd that performs the right-hand side product with a rotation matrix.

Algorithm 23.3 Demmel and Kahan Zero-Shift *QR* Downward Sweep.

```
function SINGVALS(A, tol)
  % Computes the singular values of an m × n matrix.
  % Input: real matrix A and error tolerance tol.
  % Output: a vector S of the singular values of A ordered
  % largest to smallest.
  if m < n then A = A^T
    tmp = m
    m = n
    n = tmp
  end if

  A = bidiag(A)
  k = min(m, n)

  while k ≥ 2 do
    % convergence test
    if |a_{k-1,k}| < tol(|a_{k-1,k-1}| + |a_{kk}|) then
      a_{k-1,k} = 0
      k = k-1
    else
      for i = 1:k-1 do % Compute the Givens parameters for a rotation
        % that will zero-out A(i,i+1) and A(i-1,i+1),
        % but makes A(i+1,i) non-zero.
        [ c, s ] = givensparms(a_{ii}, a_{i,i+1})
        % Apply the rotation by performing a postproduct.
        A(1:k,1:k) = givensmulpsvd(A(1:k,1:k),i,i+1,c,s)
        % Compute the Givens parameters for a rotation
```

```
            % that will zero-out a_{i+1,i} to correct the result
            % of the previous rotation. The rotation makes
            % a_{i,i+2} and A(i,i+1) non-zero.
            [ c, s ] = givensparms(a_{ii}, a_{i+1,i})
            % Apply the rotation as a preproduct.
            A(1:k,1:k) = givensmul(A(1:k,1:k),i,i+1,c,s)
          end for
        end if
      end while
      return S = diag(A)
    end function
```

NLALIB: The function `singvals` implements Algorithm 23.3.

Remark 23.2. The book software distribution contains a function `singvalsdemo(A)` that demonstrates convergence to the diagonal matrix of singular values. Initially, a graphic showing the bidiagonal matrix appears. A press of the space bar creates graphics like those in Figure 23.1. Continue pressing the space bar and see convergence taking place. At the conclusion, the function returns the computed singular values. NLALIB contains a 4×4 matrix SVALSDEMO that serves well with `singvalsdemo`.

Example 23.4. The matrix

$$A = gallery\,(5) = \begin{bmatrix} -9 & 11 & -21 & 63 & -252 \\ 70 & -69 & 141 & -421 & 1684 \\ -575 & 575 & -1149 & 3451 & -13,801 \\ 3891 & -3891 & 7782 & -23,345 & 93,365 \\ 1024 & -1025 & 2048 & -6144 & 24,572 \end{bmatrix}$$

is particularly interesting. Apply the function `eigb` to A:

```
>> A = gallery(5);
>> eigb(A)

ans =
  0.021860170045529 + 0.015660137292070i
  0.021860170045529 - 0.015660137292070i
 -0.008136735236891 + 0.025992813783568i
 -0.008136735236891 - 0.025992813783568i
 -0.027446869619491 + 0.000000000000000i
```

All the eigenvalues but one are complex; however, the characteristic equation of A is $p\,(\lambda) = \lambda^5$, so in fact all its eigenvalues are 0. To explain the results, compute the condition numbers of the eigenvalues.

```
>> eigcond(A)

ans =
   1.0e+10 *

   2.196851076143216
   2.146816343479836
   2.146816260054680
   2.068763020180955
   2.068762702670772
```

All the eigenvalues of A are ill-conditioned, so the failure of `eigb` is not unexpected. The MATLAB function `eig` fails as well.

MATLAB finds the rank of a matrix by computing the SVD and determining the number of singular values larger than the default tolerance **max(size(A))*eps(norm(A))**. As we have shown, the computation of singular values is well conditioned. The following sequence computes the singular values using `singvals`. Since all the eigenvalues of A are 0, A is not invertible, so it must have at least one zero singular value, and the last computed singular value is approximately 8.713×10^{-14}.

```
>> A = gallery(5);
>> rank(A)

ans =
     4

>> S = singvals(A,1.0e-15);
>> S(5)

ans =
     8.713452466443371e-14

>> max(size(A))*eps(norm(A))

ans =
     7.275957614183426e-11
```

Because `S(5)` is less than max(size(A))*eps(norm(A)), the rank is determined to be four. ■

23.5 CHAPTER SUMMARY

The One-Sided Jacobi Method for Computing the SVD

The use of Jacobi rotations is one of the first methods for computing the SVD but was replaced by the Golub-Kahan-Reinsch and Demmel-Kahan algorithms. Recently, the one-sided Jacobi method, with proper stopping conditions, was shown to compute small singular values with high relative accuracy. The method uses a sequence of postproducts of Jacobi rotation matrices that cause $AJ_1J_2 \ldots J_k$ to have approximately orthogonal columns. The norms of the columns of this matrix are the singular values. It turns out that the sequence of one-sided products implicitly reduces $A^{\mathrm{T}}A$ to a diagonal matrix using orthogonal similarity transformations, where the diagonal entries are the eigenvalues of $A^{\mathrm{T}}A$. Thus, it is never necessary to compute $A^{\mathrm{T}}A$ and deal with the computational time and rounding errors this will cause.

Transforming a Matrix to Upper-Bidiagonal Form

The first step in the standard algorithms for computing the SVD first reduce the matrix to upper-bidiagonal form using a sequence of Householder matrices. A left multiplication by a Householder matrix zeros-out elements below the diagonal, and a right multiplication zeros-out elements $(i, i+2)$ through (i, n) of row i.

The Demmel and Kahan Zero-Shift *QR* Downward Sweep Algorithm

The first step of this algorithm is reducing the matrix to upper-bidiagonal form. The algorithm then continues by bulge chasing that converges to a diagonal matrix of singular values. In the book software, the function `singvals` estimates singular values by continually chasing the bulge downward. In the production quality algorithm, chasing varies from downward to upward as convergence conditions change.

23.6 PROBLEMS

23.1 Let $A = \begin{bmatrix} 1 & 1 \\ 0.000001 & 0 \\ 0 & 0.000001 \end{bmatrix}$. Find the singular values of A using exact arithmetic and show that A has rank 2 but is close to a matrix of rank 1. Use the Symbolic Toolbox if available.

23.2 Let $A = \begin{bmatrix} 1 & 2 & -1 \\ 3 & 0 & 4 \\ -1 & 5 & 6 \end{bmatrix}$.

a. Use bidiag and convert A to an upper-bidiagonal matrix.

b. Carry out one downward sweep of the Demmel-Kahan algorithm.

23.3 The computation of singular values is well conditioned, but the same is not true of singular vectors. Singular vectors corresponding to close singular values are ill-conditioned. This exercise derives from an example in Ref. [83].

Let

$$A = \begin{bmatrix} 1 & 0 \\ 0 & 1+\epsilon \end{bmatrix},$$

and show that the right singular vectors of A are

$$V = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}.$$

Let

$$\hat{A} = \begin{bmatrix} 1 & \epsilon \\ \epsilon & 1 \end{bmatrix}$$

be a perturbation of A. Show that the right singular vectors of $\hat{A}$ are

$$\hat{V} = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}.$$

23.4 Develop an algorithm that takes a tridiagonal matrix A and transforms it to an upper bidiagonal matrix B using orthogonal matrices U and V such that $UAV = B$. HINT: Use Givens rotations with bulge chasing. First, eliminate $(2, 1)$ and locate the bulge. Remove it with a column rotation, and look for the next bulge. Eliminate it with a row rotation, and so forth. To determine the pattern of rotations, experiment with a 4×4 matrix.

23.5

a. Let A be an upper-bidiagonal matrix having a multiple singular value. Prove that A must have a zero on either its diagonal or superdiagonal.

b. Is part (a) true for a lower-bidiagonal matrix.

c. Assume that the diagonal and superdiagonal of a bidiagonal matrix are nonzero. Show that the singular values of the matrix are distinct.

23.6 The NLA implementation of the Demmel and Kahan zero-shift *QR* downward sweep algorithm does not compute U and V of the SVD.

a. Upon convergence of the singular values, some will be negative. Recall that each column $V(:, i)$ is an eigenvector of $A^{\mathrm{T}}A$ corresponding to singular value σ_i^2, so that $A^{\mathrm{T}}A\,(V(:, i)) = \sigma_i^2 V(:, i)$. If $\sigma_i < 0$, show that it is necessary to negate $V(:, i)$.

b. Show how to modify the algorithm so it computes the full SVD for A, $m \geq n$. You will need to maintain the products of the right and left Householder reflections used to bidiagonalize A. During bulge sweeping, maintain the products of the left and right Givens rotations.

23.6.1 MATLAB Problems

23.7 Randomly generate a matrix A of order 16×4 by using the MATLAB command `rand(16,4)`. Then verify using the MATLAB command rank that rank(A) = 4. Now run the following MATLAB commands:

```
[U S V] = svd(A);
S(4,4) = 0;
B = U*S*V';
```

What is the rank of B? Explain.

23.8 The execution of `A = gallery('kahan',n,theta)` returns an $n \times n$ upper-triangular matrix that has interesting properties regarding estimation of condition number and rank.The default value of theta is 1.2.

Let $n = 90$, and compute singvals(A). What is the smallest singular value? Verify that this is correct to five significant digits using `svd`. Try to compute the inverse of A. What is the true rank of `A`? What is the result of computing the rank using MATLAB? If there is a difference, explain.

23.9 This problem asks for a simplified implementation the modified Jacobi SVD algorithm presented in Section 23.2.1. When an SVD is required, use jacobisvd. The *QR* factorization used is rank revealing, so compute the rank of A as follows:

```
[Q,R,P] = qr(A);
rA = 0 ;
for i = 1 : n
    if abs(R(i,i)) > max(size(A))*eps(norm(A))
        rA = rA + 1 ;
    end
end
```

Section 23.2.1 presented the algorithm for the case of full rank. Use the following code when $rA < n$:

```
[Q1,R1] = qr(R(1:rA,1:n)') ;
[U(1:rA,1:rA),S,V(1:rA,1:rA)] = jacobisvd(R1(1:rA,1:rA)',tol,maxsweeps);
U = Q * U;
V = P*Q1*V;
```

Name the function `svdj`, and test it using the matrices wilkinson(21), gallery(5), a 10×6 matrix with full rank, and a 10×6 rank deficient matrix.

23.10

a. Implement the algorithm described in Problem 23.4 as the function `tritobidiag`.

b. In a loop that executes five times, generate a random 100×100 tridiagonal matrix A as indicated, and compute its singular values using S1 = svd(A). Use `tritobidiag` to transform A to a matrix B in upper-bidiagonal form. Compute its singular values using `S2 = svd(B)`. Check the result by computing $\|S1 - S2\|_2$.

```
>> a = randn(99,1);
>> b = randn(100,1);
>> c = randn(99,1);
>> A = trid(a,b,c);
```

23.11 Using your results from Problem 23.6, modify `singvals` so it optionally returns the full SVD $A = U\tilde{\Sigma}V^{\mathrm{T}}$. Name the function `svd0shift`, and test it with `gallery(5)`, the rosser matrix, and a random 50×30 matrix.