Chapter 15

Computing Eigenvalues and Singular Values

In MATLAB and other modern programming environments, eigenvalues and singular values are obtained using high-level functions, e.g., `eig(A)` and `svd(A)`. These functions implement algorithms from the LAPACK subroutine library [1]. To give an orientation about what is behind such high-level functions, in this chapter we briefly describe some methods for computing eigenvalues and singular values of dense matrices and large sparse matrices. For more extensive treatments, see e.g., [4, 42, 79].

The functions `eig` and `svd` are used for dense matrices, i.e., matrices where most of the elements are nonzero. Eigenvalue algorithms for a dense matrix have two phases:

1. Reduction of the matrix to compact form: tridiagonal in the symmetric case and Hessenberg in the nonsymmetric case. This phase consists of a finite sequence of orthogonal transformations.

2. Iterative reduction to diagonal form (symmetric case) or triangular form (nonsymmetric case). This is done using the QR algorithm.

For large, sparse matrices it is usually not possible (or even interesting) to compute all the eigenvalues. Here there are special methods that take advantage of the sparsity. Singular values are computed using variations of the eigenvalue algorithms.

As background material we give some theoretical results concerning perturbation theory for the eigenvalue problem. In addition, we briefly describe the power method for computing eigenvalues and its cousin inverse iteration.

In linear algebra textbooks the eigenvalue problem for the matrix $A \in \mathbb{R}^{n\times n}$ is often introduced as the solution of the polynomial equation

$$\det(A - \lambda I) = 0.$$

In the computational solution of general problems, this approach is useless for two reasons: (1) for matrices of interesting dimensions it is too costly to compute the determinant, and (2) even if the determinant and the polynomial could be computed, the eigenvalues are extremely sensitive to perturbations in the coefficients of the polynomial. Instead, the basic tool in the numerical computation of eigenvalues are *orthogonal similarity transformations*. Let V be an orthogonal matrix. Then make the transformation (which corresponds to a change of basis)

$$A \longrightarrow V^T A V. \tag{15.1}$$

It is obvious that the eigenvalues are preserved under this transformation:

$$Ax = \lambda x \quad \Leftrightarrow \quad V^T A V y = \lambda y, \tag{15.2}$$

where $y = V^T x$.

15.1 Perturbation Theory

The QR algorithm for computing eigenvalues is based on orthogonal similarity transformations (15.1), and it computes a sequence of transformations such that the final result is diagonal (in the case of symmetric A) or triangular (for nonsymmetric A). Since the algorithm is iterative, it is necessary to decide when a floating point number is small enough to be considered as zero numerically. To have a sound theoretical basis for this decision, one must know how sensitive the eigenvalues and eigenvectors are to small perturbations of the data, i.e., the coefficients of the matrix.

Knowledge about the sensitivity of eigenvalues and singular values is useful also for a more fundamental reason: often matrix elements are measured values and subject to errors. Sensitivity theory gives information about how much we can trust eigenvalues, etc., in such situations.

In this section we give a couple of perturbation results, without proofs,[37] first for a symmetric matrix $A \in \mathbb{R}^{n\times n}$. Assume that eigenvalues of $n \times n$ matrices are ordered

$$\lambda_1 \geq \lambda_2 \geq \cdots \geq \lambda_n.$$

We consider a perturbed matrix $A + E$ and ask how far the eigenvalues and eigenvectors of $A + E$ are from those of A.

Example 15.1. Let

$$A = \begin{pmatrix} 2 & 1 & 0 & 0 \\ 1 & 2 & 0.5 & 0 \\ 0 & 0.5 & 2 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}, \qquad A + E = \begin{pmatrix} 2 & 1 & 0 & 0 \\ 1 & 2 & 0.5 & 0 \\ 0 & 0.5 & 2 & 10^{-15} \\ 0 & 0 & 10^{-15} & 1 \end{pmatrix}.$$

This is a typical situation in the QR algorithm for tridiagonal matrices: by a sequence of orthogonal similarity transformations, a tridiagonal matrix is made to converge toward a diagonal matrix. When are we then allowed to consider a small off-diagonal floating point number as zero? How much can the eigenvalues of A and $A + E$ deviate? ∎

[37]For proofs, see, e.g., [42, Chapters 7, 8].

Theorem 15.2. *Let $A \in \mathbb{R}^{n\times n}$ and $A+E$ be symmetric matrices. Then*

$$\lambda_k(A) + \lambda_n(E) \le \lambda_k(A+E) \le \lambda_k(A) + \lambda_1(E), \qquad k = 1, 2, \dots, n,$$

and

$$|\lambda_k(A+E) - \lambda_k(A)| \le \|E\|_2, \qquad k = 1, 2, \dots, n.$$

From the theorem we see that, loosely speaking, if we perturb the matrix elements by ϵ, then the eigenvalues are also perturbed by $O(\epsilon)$. For instance, in Example 15.1 the matrix E has the eigenvalues $\pm 10^{-15}$ and $\|E\|_2 = 10^{-15}$. Therefore the eigenvalues of the two matrices differ by 10^{-15} at the most.

The sensitivity of the eigenvectors depends on the separation of eigenvalues.

Theorem 15.3. *Let $[\lambda, q]$ be an eigenvalue-eigenvector pair of the symmetric matrix A, and assume that the eigenvalue is simple. Form the orthogonal matrix $Q = \begin{pmatrix} q & Q_1 \end{pmatrix}$ and partition the matrices Q^TAQ and Q^TEQ,*

$$Q^TAQ = \begin{pmatrix} \lambda & 0 \\ 0 & A_2 \end{pmatrix}, \qquad Q^TEQ = \begin{pmatrix} \epsilon & e^T \\ e & E_2 \end{pmatrix}.$$

Define

$$d = \min_{\lambda_i(A) \ne \lambda} |\lambda - \lambda_i(A)|$$

and assume that $\| E \|_2 \le d/4$. Then there exists an eigenvector $\hat{q}$ of $A+E$ such that the distance between q and $\hat{q}$, measured as the sine of the angle between the vectors, is bounded:

$$\sin(\theta(q, \hat{q})) \le \frac{4 \| e \|_2}{d}.$$

The theorem is meaningful only if the eigenvalue is simple. It shows that eigenvectors corresponding to close eigenvalues can be sensitive to perturbations and are therefore more difficult to compute to high accuracy.

Example 15.4. The eigenvalues of the matrix A in Example 15.1 are

```
0.8820, 1.0000, 2.0000, 3.1180.
```

The deviation between the eigenvectors of A and $A+E$ corresponding to the smallest eigenvalue can be estimated by

$$\frac{4\|e\|_2}{|0.8820 - 1|} \approx 1.07 \cdot 10^{-14}.$$

Since the eigenvalues are well separated, the eigenvectors of this matrix are rather insensitive to perturbations in the data. ■

To formulate perturbation results for nonsymmetric matrices, we first introduce the concept of an *upper quasi-triangular matrix*: $R \in \mathbb{R}^{n\times n}$ is called upper quasi-triangular if it has the form

$$R = \begin{pmatrix} R_{11} & R_{12} & \cdots & R_{1m} \\ 0 & R_{22} & \cdots & R_{2m} \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & R_{mm} \end{pmatrix},$$

where each R_{ii} is either a scalar or a 2×2 matrix having complex conjugate eigenvalues. The eigenvalues of R are equal to the eigenvalues of the diagonal blocks R_{ii} (which means that if R_{ii} is a scalar, then it is an eigenvalue of R).

Theorem 15.5 (real Schur decomposition[38]). *For any (symmetric or nonsymmetric) matrix $A \in \mathbb{R}^{n\times n}$ there exists an orthogonal matrix U such that*

$$U^T AU = R, \tag{15.3}$$

where R is upper quasi-triangular.

Partition U and R:

$$U = \begin{pmatrix} U_k & \widehat{U} \end{pmatrix}, \qquad R = \begin{pmatrix} R_k & S \\ 0 & \widehat{R} \end{pmatrix},$$

where $U_k \in \mathbb{R}^{n\times k}$ and $R_k \in \mathbb{R}^{k\times k}$. Then from (15.3) we get

$$AU_k = U_k R_k, \tag{15.4}$$

which implies $\mathcal{R}(AU_k) \subset \mathcal{R}(U_k)$, where $\mathcal{R}(U_k)$ denotes the range of U_k. Therefore U_k is called an *invariant subspace* or an *eigenspace* of A, and the decomposition (15.4) is called a *partial Schur decomposition.*

If A is symmetric, then R is diagonal, and the Schur decomposition is the same as the *eigenvalue decomposition* $U^T AU = D$, where D is diagonal. If A is nonsymmetric, then some or all of its eigenvalues may be complex.

Example 15.6. The Schur decomposition is a standard function in MATLAB. If the matrix is real, then R is upper quasi-triangular:

```
>> A=randn(3)
A = -0.4326    0.2877    1.1892
    -1.6656   -1.1465   -0.0376
     0.1253    1.1909    0.3273

>> [U,R]=schur(A)
U = 0.2827    0.2924    0.9136
    0.8191   -0.5691   -0.0713
   -0.4991   -0.7685    0.4004
```

[38]There is complex version of the decomposition, where U is unitary and R is complex and upper triangular.

```
   R = -1.6984     0.2644   -1.2548
             0     0.2233    0.7223
             0    -1.4713    0.2233
```

If we compute the eigenvalue decomposition, we get

```
>> [X,D]=eig(A)

X = 0.2827             0.4094 - 0.3992i   0.4094 + 0.3992i
    0.8191            -0.0950 + 0.5569i  -0.0950 - 0.5569i
   -0.4991             0.5948             0.5948

D =  -1.6984              0                   0
          0        0.2233+1.0309i             0
          0               0            0.2233-1.0309i
```

The eigenvectors of a nonsymmetric matrix are not orthogonal. ∎

The sensitivity of the eigenvalues of a nonsymmetric matrix depends on the norm of the strictly upper triangular part of R in the Schur decomposition. For convenience we here formulate the result using the complex version of the decomposition.[39]

Theorem 15.7. *Let $U^H AU = R = D + N$ be the complex Schur decomposition of A, where U is unitary, R is upper triangular, and D is diagonal, and let τ denote an eigenvalue of a perturbed matrix $A + E$. Further, let p be the smallest integer such that $N^p = 0$. Then*

$$\min_{\lambda_i(A)} |\lambda_i(A) - \tau| \leq \max(\eta, \eta^{1/p}),$$

where

$$\eta = \| E \|_2 \sum_{k=0}^{p-1} \| N \|_2^k.$$

The theorem shows that the eigenvalues of a highly nonsymmetric matrix can be considerably more sensitive to perturbations than the eigenvalues of a symmetric matrix; cf. Theorem 15.2.

Example 15.8. The matrices

$$A = \begin{pmatrix} 2 & 0 & 10^3 \\ 0 & 2 & 0 \\ 0 & 0 & 2 \end{pmatrix}, \qquad B = A + \begin{pmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 10^{-10} & 0 & 0 \end{pmatrix}$$

have the eigenvalues

[39]The notation U^H means transposed and conjugated.

```
2, 2, 2,
```

and

```
2.00031622776602, 1.99968377223398, 2.00000000000000,
```

respectively. The relevant quantity for the perturbation is $\eta^{1/2} \approx 3.164 \cdot 10^{-04}$. ■

The nonsymmetric version of Theorem 15.3 is similar: again the angle between the eigenvectors depends on the separation of the eigenvalues. We give a simplified statement below, where we disregard the possibility of a complex eigenvalue.

Theorem 15.9. *Let $[\lambda, q]$ be an eigenvalue-eigenvector pair of A, and assume that the eigenvalue is simple. Form the orthogonal matrix $Q = \begin{pmatrix} q & Q_1 \end{pmatrix}$ and partition the matrices $Q^T AQ$ and $Q^T EQ$:*

$$Q^T AQ = \begin{pmatrix} \lambda & v^T \\ 0 & A_2 \end{pmatrix}, \qquad Q^T EQ = \begin{pmatrix} \epsilon & e^T \\ \delta & E_2 \end{pmatrix}.$$

Define

$$d = \sigma_{min}(A_2 - \lambda I),$$

and assume $d > 0$. If the perturbation E is small enough, then there exists an eigenvector $\hat{q}$ of $A + E$ such that the distance between q and $\hat{q}$ measured as the sine of the angle between the vectors is bounded by

$$\sin(\theta(q, \hat{q})) \leq \frac{4 \| \delta \|_2}{d}.$$

The theorem says essentially that if we perturb A by ϵ, then the eigenvector is perturbed by ϵ/d.

Example 15.10. Let the tridiagonal matrix be defined as

$$A_n = \begin{pmatrix} 2 & -1.1 & & & \\ -0.9 & 2 & -1.1 & & \\ & \ddots & \ddots & \ddots & \\ & & -0.9 & 2 & -1.1 \\ & & & -0.9 & 2 \end{pmatrix} \in \mathbb{R}^{n \times n}.$$

For $n = 100$, its smallest eigenvalue is 0.01098771, approximately. The following MATLAB script computes the quantity d in Theorem 15.9:

```
% xn is the eigenvector corresponding to
% the smallest eigenvalue
[Q,r]=qr(xn);
H=Q'*A*Q; lam=H(1,1);
A2=H(2:n,2:n);
d=min(svd(A2-lam*eye(size(A2))));
```

We get $d = 1.6207 \cdot 10^{-4}$. Therefore, if we perturb the matrix by 10^{-10}, say, this may change the eigenvector by a factor $4 \cdot 10^{-6}$, approximately. ■

15.2 The Power Method and Inverse Iteration

The power method is a classical iterative method for computing the largest (in magnitude) eigenvalue and the corresponding eigenvector. Its convergence can be very slow, depending on the distribution of eigenvalues. Therefore it should never be used for dense matrices. Usually for sparse matrices one should use a variant of the Lanczos method or the Jacobi–Davidson method; see [4] and Section 15.8.3. However, in some applications the dimension of the problem is so huge that no other method is viable; see Chapter 12.

Despite its limited usefulness for practical problems, the power method is important from a theoretical point of view. In addition, there is a variation of the power method, inverse iteration, that is of great practical importance.

In this section we give a slightly more general formulation of the power method than in Chapter 12 and recall a few of its properties.

The power method for computing the largest eigenvalue

```
% Initial approximation x
for k=1:maxit
  y=A*x;
  lambda=y'*x;
  if norm(y-lambda*x) < tol*abs(lambda)
    break  % stop the iterations
  end
  x=1/norm(y)*y;
end
```

The convergence of the power method depends on the distribution of eigenvalues of the matrix A. Assume that the largest eigenvalue in magnitude is simple and that λ_i are ordered $|\lambda_1| > |\lambda_2| \geq \cdots \geq |\lambda_n|$. The rate of convergence is determined by the ratio $|\lambda_2/\lambda_1|$. If this ratio is close to 1, then the iteration is very slow.

A stopping criterion for the power iteration can be formulated in terms of the residual vector for the eigenvalue problem: if the norm of the residual $r = A\hat{x} - \hat{\lambda}\hat{x}$ is small, then the eigenvalue approximation is good.

Example 15.11. Consider again the tridiagonal matrix

$$A_n = \begin{pmatrix} 2 & -1.1 & & & \\ -0.9 & 2 & -1.1 & & \\ & \ddots & \ddots & \ddots & \\ & & -0.9 & 2 & -1.1 \\ & & & -0.9 & 2 \end{pmatrix} \in \mathbb{R}^{n\times n}.$$

The two largest eigenvalues of A_{20} are 3.9677 and 3.9016, approximately. As initial approximation we chose a random vector. In Figure 15.1 we plot different error measures during the iterations: the relative residual $\| Ax^{(k)} - \lambda^{(k)}x^{(k)} \|/\lambda_1$ ($\lambda^{(k)}$ denotes

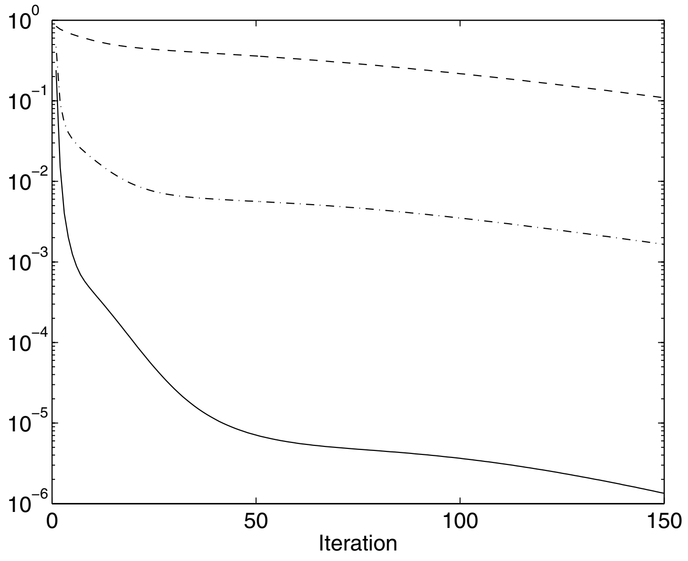

Figure 15.1. *Power iterations for A_{20}. The relative residual $\| Ax^{(k)} - \lambda^{(k)}x^{(k)} \|/\lambda_1$ (solid line), the absolute error in the eigenvalue approximation (dash-dotted line), and the angle (in radians) between the exact eigenvector and the approximation (dashed line).*

the approximation of λ_1 in the kth iteration), the error in the eigenvalue approximation, and the angle between the exact and the approximate eigenvector. After 150 iterations the relative error in the computed approximation of the eigenvalue is 0.0032.

We have $\lambda_2(A_{20})/\lambda_1(A_{20}) = 0.9833$. It follows that

$$0.9833^{150} \approx 0.0802,$$

which indicates that the convergence is quite slow, as seen in Figure 15.1. This is comparable to the reduction of the angle between the exact and the approximate eigenvector during 150 iterations: from 1.2847 radians to 0.0306. ■

If we iterate with A^{-1} in the power method,

$$x^{(k)} = A^{-1}x^{(k-1)},$$

then, since the eigenvalues of A^{-1} are $1/\lambda_i$, the sequence of eigenvalue approximations converges toward $1/\lambda_{min}$, where λ_{min} is the eigenvalue of smallest absolute value. Even better, if we have a good enough approximation of one of the eigenvalues, $\tau \approx \lambda_k$, then the shifted matrix $A - \tau I$ has the smallest eigenvalue $\lambda_k - \tau$. Thus, we can expect very fast convergence in the "inverse power method." This method is called *inverse iteration*.

Inverse iteration

```
% Initial approximation x and eigenvalue approximation tau
[L,U]=lu(A - tau*I);
for k=1:maxit
  y=U\(L\x);
  theta=y'*x;
  if norm(y-theta*x) < tol*abs(theta)
    break  % stop the iteration
  end
  x=1/norm(y)*y;
end
lambda=tau+1/theta; x=1/norm(y)*y;
```

Example 15.12. The smallest eigenvalue of the matrix A_{100} from Example 15.10 is $\lambda_{100} = 0.01098771187192$ to 14 decimals accuracy. If we use the approximation $\lambda_{100} \approx \tau = 0.011$ and apply inverse iteration, we get fast convergence; see Figure 15.2. In this example the convergence factor is

$$\left| \frac{\lambda_{100} - \tau}{\lambda_{99} - \tau} \right| \approx 0.0042748,$$

which means that after four iterations, the error is reduced by a factor of the order $3 \cdot 10^{-10}$. ■

To be efficient, inverse iteration requires that we have good approximation of the eigenvalue. In addition, we must be able to solve linear systems $(A - \tau I)y = x$ (for y) cheaply. If A is a band matrix, then the LU decomposition can be obtained easily and in each iteration the system can be solved by forward and back substitution (as in the code above). The same method can be used for other sparse matrices if a sparse LU decomposition can be computed without too much fill-in.

15.3 Similarity Reduction to Tridiagonal Form

The QR algorithm that we will introduce in Section 15.4 is an iterative algorithm, where in each step a QR decomposition is computed. If it is applied to a dense matrix $A \in \mathbb{R}^{n\times n}$, then the cost of a step is $O(n^3)$. This prohibitively high cost can be reduced substantially by first transforming the matrix to compact form, by an orthogonal similarity transformation (15.1),

$$A \longrightarrow V^T A V,$$

for an orthogonal matrix V. We have already seen in (15.2) that the eigenvalues are preserved under this transformation,

$$Ax = \lambda x \quad \Leftrightarrow \quad V^T A V y = \lambda y,$$

where $y = V^T x$.

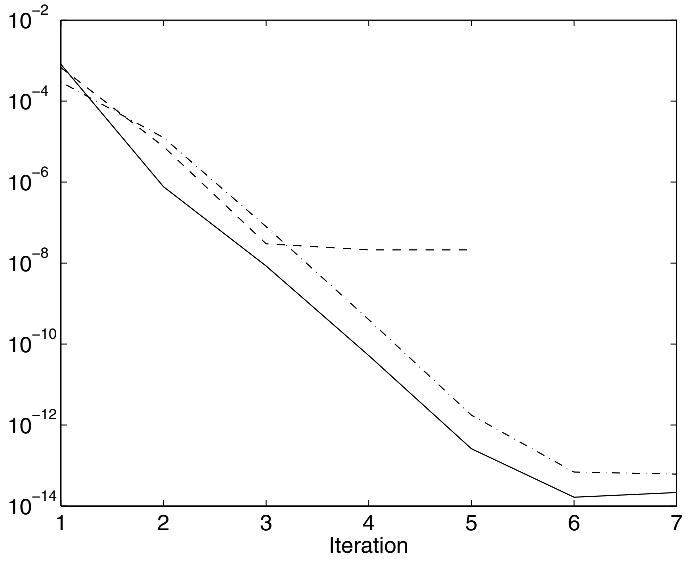

Figure 15.2. *Inverse iterations for A_{100} with $\tau = 0.011$. The relative residual $\| Ax^{(k)} - \lambda^{(k)} x^{(k)} \| / \lambda^{(k)}$ (solid line), the absolute error in the eigenvalue approximation (dash-dotted line), and the angle (in radians) between the exact eigenvector and the approximation (dashed line).*

Let $A \in \mathbb{R}^{n\times n}$ be symmetric. By a sequence of Householder transformations it can be reduced to tridiagonal form. We illustrate the procedure using an example with $n = 6$. First we construct a transformation that zeros the elements in positions 3 through n in the first column when we multiply A from the left:

$$H_1 A = H_1 \begin{pmatrix} \times & \times & \times & \times & \times & \times \\ \times & \times & \times & \times & \times & \times \\ \times & \times & \times & \times & \times & \times \\ \times & \times & \times & \times & \times & \times \\ \times & \times & \times & \times & \times & \times \\ \times & \times & \times & \times & \times & \times \end{pmatrix} = \begin{pmatrix} \times & \times & \times & \times & \times & \times \\ * & * & * & * & * & * \\ 0 & * & * & * & * & * \\ 0 & * & * & * & * & * \\ 0 & * & * & * & * & * \\ 0 & * & * & * & * & * \end{pmatrix}.$$

Elements that are changed in the transformation are denoted by $*$. Note that the elements of the first row are not changed. In an orthogonal similarity transformation we shall multiply by the same matrix transposed from the right. Since in the left multiplication the first row was not touched, the first column will remain unchanged:

$$H_1 A H_1^T = \begin{pmatrix} \times & \times & \times & \times & \times & \times \\ \times & \times & \times & \times & \times & \times \\ 0 & \times & \times & \times & \times & \times \\ 0 & \times & \times & \times & \times & \times \\ 0 & \times & \times & \times & \times & \times \\ 0 & \times & \times & \times & \times & \times \end{pmatrix} H_1^T = \begin{pmatrix} \times & * & 0 & 0 & 0 & 0 \\ \times & * & * & * & * & * \\ 0 & * & * & * & * & * \\ 0 & * & * & * & * & * \\ 0 & * & * & * & * & * \\ 0 & * & * & * & * & * \end{pmatrix}.$$

Due to symmetry, elements 3 through n in the first row will be equal to zero.

In the next step we zero the elements in the second column in positions 4 through n. Since this affects only rows 3 through n and the corresponding columns, this does not destroy the zeros that we created in the first step. The result is

$$H_2 H_1 A H_1^T H_2^T = \begin{pmatrix} \times & \times & 0 & 0 & 0 & 0 \\ \times & \times & * & 0 & 0 & 0 \\ 0 & * & * & * & * & * \\ 0 & 0 & * & * & * & * \\ 0 & 0 & * & * & * & * \\ 0 & 0 & * & * & * & * \end{pmatrix}.$$

After $n-2$ such similarity transformations the matrix is in tridiagonal form:

$$V^T A V = \begin{pmatrix} \times & \times & 0 & 0 & 0 & 0 \\ \times & \times & \times & 0 & 0 & 0 \\ 0 & \times & \times & \times & 0 & 0 \\ 0 & 0 & \times & \times & \times & 0 \\ 0 & 0 & 0 & \times & \times & \times \\ 0 & 0 & 0 & 0 & \times & \times \end{pmatrix},$$

where $V = H_1^T H_2^T \cdots H_{n-2}^T = H_1 H_2 \cdots H_{n-2}$.

In summary, we have demonstrated how a symmetric matrix can be reduced to tridiagonal form by a sequence of $n-2$ Householder transformations:

$$A \longrightarrow T = V^T A V, \qquad V = H_1 H_2 \cdots H_{n-2}, \tag{15.5}$$

Since the reduction is done by similarity transformations, the tridiagonal matrix T has the same eigenvalues as A.

The reduction to tridiagonal form requires $4n^3/3$ flops if one takes advantage of symmetry. As in the case of QR decomposition, the Householder transformations can be stored in the subdiagonal part of A. If V is computed explicitly, this takes $4n^3/3$ additional flops.

15.4 The QR Algorithm for a Symmetric Tridiagonal Matrix

We will now give a sketch of the QR algorithm for a symmetric, tridiagonal matrix. We emphasize that our MATLAB codes are greatly simplified and are intended only to demonstrate the basic ideas of the algorithm. The actual software (in LAPACK) contains numerous features for efficiency, robustness, and numerical stability.

The procedure that we describe can be considered as a continuation of the similarity reduction (15.5), but now we reduce the matrix T to diagonal form:

$$T \longrightarrow \Lambda = Q^T T Q, \qquad Q = Q_1 Q_2 \cdots, \tag{15.6}$$

where $\Lambda = \mathrm{diag}(\lambda_1\, \lambda_2\, \ldots, \lambda_n)$. The matrices Q_i will be orthogonal, but here they will be constructed using plane rotations. However, the most important difference between (15.5) and (15.6) is that there does not exist a finite algorithm[40] for computing Λ. We compute a sequence of matrices,

$$T_0 := T, \qquad T_i = Q_i^T T_{i-1} Q_i, \qquad i = 1, 2, \ldots, \tag{15.7}$$

such that it converges to a diagonal matrix,

$$\lim_{i\to\infty} T_i = \Lambda.$$

We will demonstrate in numerical examples that the convergence is very rapid, so that *in floating point arithmetic the algorithm can actually be considered as finite.* Since all the transformations in (15.7) are similarity transformations, the diagonal elements of Λ are the eigenvalues of T.

We now give a first version of the QR algorithm for a symmetric tridiagonal matrix $T \in \mathbb{R}^{n\times n}$.

QR iteration for symmetric T: Bottom eigenvalue

```
for i=1:maxit   % Provisional simplification
  mu=wilkshift(T(n-1:n,n-1:n));
  [Q,R]=qr(T-mu*I);
  T=R*Q+mu*I
end

function mu=wilkshift(T);
  % Compute the Wilkinson shift
  l=eig(T);
  if abs(l(1)-T(2,2))<abs(l(2)-T(2,2))
    mu=l(1);
  else
    mu=l(2);
  end
```

We see that the QR decomposition of a shifted matrix $QR = T - \tau I$ is computed and that the shift is then added back $T := RQ + \tau I$. The shift is the eigenvalue of the 2×2 submatrix in the lower right corner that is closest to t_{nn}. This is called the *Wilkinson shift.*

We applied the algorithm to the matrix

$$T_6 = \begin{pmatrix} 2 & -1 & 0 & 0 & 0 & 0 \\ -1 & 2 & -1 & 0 & 0 & 0 \\ 0 & -1 & 2 & -1 & 0 & 0 \\ 0 & 0 & -1 & 2 & -1 & 0 \\ 0 & 0 & 0 & -1 & 2 & -1 \\ 0 & 0 & 0 & 0 & -1 & 2 \end{pmatrix}. \tag{15.8}$$

After the first step the result was (slightly edited for readability)

[40]By a finite algorithm, we mean an algorithm for computing the diagonalization of T *in the field of real numbers, i.e., in exact arithmetic*, using a finite number of operations.

```
T = 1.0000    0.7071         0         0         0         0
    0.7071    2.0000    1.2247         0         0         0
         0    1.2247    2.3333   -0.9428         0         0
         0         0   -0.9428    1.6667    0.8660         0
         0         0         0    0.8660    2.0000   -0.5000
         0         0         0         0   -0.5000    3.0000
```

We see first that the triangular structure is preserved and that the off-diagonal elements in the lower right corner have become smaller. We perform three more steps and look more closely at that submatrix:

```
 2.36530292572181  -0.02609619264716
-0.02609619264716   3.24632297453998

 2.59270689576885   0.00000366571479
 0.00000366571479   3.24697960370634

 2.77097818052654   0.00000000000000
-0.00000000000000   3.24697960371747
```

Thus, after four iterations the off-diagonal element has become zero in working precision and therefore we have an eigenvalue in the lower right corner.

When the eigenvalue has been found, we can deflate the problem and continue working with the upper $(n-1) \times (n-1)$ submatrix, which is now

```
0.4374    0.3176         0         0         0
0.3176    0.7961   -0.4395         0         0
     0   -0.4395    1.3198    0.2922         0
     0         0    0.2922    3.4288    0.5902
     0         0         0    0.5902    2.7710
```

We now apply the same algorithm to this matrix. Tracing its lower right submatrix during three subsequent steps, we have

```
 3.74629910763238  -0.01184028941948
-0.01184028941948   2.44513898239641

 3.68352336882524   0.00000009405188
 0.00000009405188   2.44504186791263

 3.54823766699472   0.00000000000000
-0.00000000000000   2.44504186791263
```

After these three iterations we again have an eigenvalue at the lower right corner. The algorithm now proceeds by deflating this eigenvalue and reducing the dimension of the active matrix by one. A preliminary implementation of the algorithm is given below.

QR iteration for symmetric T

```
function [D,it]=qrtrid(T);
  % Compute the eigenvalues of a symmetric tridiagonal
  % matrix using the QR algorithm with explicit
  % Wilkinson shift
    n=size(T,1); it=0;
    for i=n:-1:3
      while abs(T(i-1,i)) > ...
                (abs(T(i,i))+abs(T(i-1,i-1)))*C*eps
        it=it+1;
        mu=wilkshift(T(i-1:i,i-1:i));
        [Q,R]=qr(T(1:i,1:i)-mu*eye(i));
        T=R*Q+mu*eye(i);
      end
      D(i)=T(i,i);
    end
    D(1:2)=eig(T(1:2,1:2))';
```

For a given submatrix `T(1:i,1:i)` the QR steps are iterated until the stopping criterion

$$\frac{|t_{i-1,i}|}{|t_{i-1,i-1}|+|t_{i,i}|} < C\mu$$

is satisfied, where C is a small constant and μ is the unit round-off. From Theorem 15.2 we see that considering such a tiny element as a numerical zero leads to a very small (and acceptable) perturbation of the eigenvalues. In actual software, a slightly more complicated stopping criterion is used.

When applied to the matrix T_{100} (cf. (15.8)) with the value $C = 5$, 204 QR steps were taken, i.e., approximately 2 steps per eigenvalue. The maximum deviation between the computed eigenvalues and those computed by the MATLAB `eig` function was $2.9 \cdot 10^{-15}$.

It is of course inefficient to compute the QR decomposition of a tridiagonal matrix using the MATLAB function `qr`, which is a Householder-based algorithm. Instead the decomposition should be computed using $n-1$ plane rotations in $O(n)$ flops. We illustrate the procedure with a small example, where the tridiagonal matrix $T = T^{(0)}$ is 6×6. The first subdiagonal element (from the top) is zeroed by a rotation from the left in the $(1,2)$ plane, $G_1^T(T^{(0)} - \tau I)$, and then the second subdiagonal is zeroed by a rotation in $(2,3)$, $G_2^T G_1^T(T^{(0)} - \tau I)$. Symbolically,

$$\begin{pmatrix} \times & \times & & & & \\ \times & \times & \times & & & \\ & \times & \times & \times & & \\ & & \times & \times & \times & \\ & & & \times & \times & \times \\ & & & & \times & \times \end{pmatrix} \longrightarrow \begin{pmatrix} \times & \times & + & & & \\ 0 & \times & \times & + & & \\ & 0 & \times & \times & & \\ & & \times & \times & \times & \\ & & & \times & \times & \times \\ & & & & \times & \times \end{pmatrix}.$$

Note the fill-in (new nonzero elements, denoted $+$) that is created. After $n-1$ steps we have an upper triangular matrix with three nonzero diagonals:

$$R = G_{n-1}^T \cdots G_1^T (T^{(0)} - \tau I) = \begin{pmatrix} \times & \times & + & & & \\ 0 & \times & \times & + & & \\ & 0 & \times & \times & + & \\ & & 0 & \times & \times & + \\ & & & 0 & \times & \times \\ & & & & 0 & \times \end{pmatrix}.$$

We then apply the rotations from the right, $RG_1 \cdots G_{n-1}$, i.e., we start with a transformation involving the first two columns. Then follows a rotation involving the second and third columns. The result after two steps is

$$\begin{pmatrix} \times & \times & \times & & & \\ + & \times & \times & \times & & \\ & + & \times & \times & \times & \\ & & & \times & \times & \times \\ & & & & \times & \times \\ & & & & & \times \end{pmatrix}.$$

We see that the zeroes that we introduced below the diagonal in the transformations from the left are systematically filled in. After $n-1$ steps we have

$$T^{(1)} = RG_1G_2 \cdots G_{n-1} + \tau I = \begin{pmatrix} \times & \times & \times & & & \\ + & \times & \times & \times & & \\ & + & \times & \times & \times & \\ & & + & \times & \times & \times \\ & & & + & \times & \times \\ & & & & + & \times \end{pmatrix}.$$

But we have made a similarity transformation: with $Q = G_1G_2 \cdots G_{n-1}$ and using $R = Q^T(T^{(0)} - \tau I)$, we can write

$$T^{(1)} = RQ + \tau I = Q^T(T^{(0)} - \tau I)Q + \tau I = Q^T T^{(0)} Q, \tag{15.9}$$

so we know that $T^{(1)}$ is symmetric,

$$T^{(1)} = \begin{pmatrix} \times & \times & & & & \\ \times & \times & \times & & & \\ & \times & \times & \times & & \\ & & \times & \times & \times & \\ & & & \times & \times & \times \\ & & & & \times & \times \end{pmatrix}.$$

Thus we have shown the following result.

Proposition 15.13. *The QR step for a tridiagonal matrix*

$$QR = T^{(k)} - \tau_k I, \qquad T^{(k+1)} = RQ + \tau_k I,$$

is a similarity transformation

$$T^{(k+1)} = Q^T T^{(k)} Q, \tag{15.10}$$

and the tridiagonal structure is preserved. The transformation can be computed with plane rotations in $O(n)$ flops.

From (15.9) it may appear as if the shift plays no significant role. However, it determines the value of the orthogonal transformation in the QR step. Actually, the shift strategy is absolutely necessary for the algorithm to be efficient: if no shifts are performed, then the QR algorithm usually converges very slowly, in fact as slowly as the power method; cf. Section 15.2. On the other hand, it can be proved [107] (see, e.g., [93, Chapter 3]) that the shifted QR algorithm has very fast convergence.

Proposition 15.14. *The symmetric QR algorithm with Wilkinson shifts converges cubically toward the eigenvalue decomposition.*

In actual software for the QR algorithm, there are several enhancements of the algorithm that we outlined above. For instance, the algorithm checks all off-diagonals if they are small: when a negligible off-diagonal element is found, then the problem can be split in two. There is also a divide-and-conquer variant of the QR algorithm. For an extensive treatment, see [42, Chapter 8].

15.4.1 Implicit shifts

One important aspect of the QR algorithm is that the shifts can be performed *implicitly*. This is especially useful for the application of the algorithm to the SVD and the nonsymmetric eigenproblem. This variant is based on the *implicit Q theorem*, which we here give in slightly simplified form.

Theorem 15.15. *Let A be symmetric, and assume that Q and V are orthogonal matrices such that $Q^T AQ$ and $V^T AV$ are both tridiagonal. Then, if the first columns of Q and V are equal, $q_1 = v_1$, then Q and V are essentially equal: $q_i = \pm v_i$, $i = 2, 3, \ldots, n$.*

For a proof, see [42, Chapter 8].

A consequence of this theorem is that if we determine and apply the first transformation in the QR decomposition of $T - \tau I$, and if we construct the rest of the transformations in such a way that we finally arrive at a tridiagonal matrix, then we have performed a shifted QR step as in Proposition 15.13. This procedure is implemented as follows.

Let the first plane rotation be determined such that

$$\begin{pmatrix} c & s \\ -s & c \end{pmatrix} \begin{pmatrix} \alpha_1 - \tau \\ \beta_1 \end{pmatrix} = \begin{pmatrix} \times \\ 0 \end{pmatrix}, \tag{15.11}$$

where α_1 and β_1 are the top diagonal and subdiagonal elements of T. Define

$$G_1^T = \begin{pmatrix} c & s & & & \\ -s & c & & & \\ & & 1 & & \\ & & & \ddots & \\ & & & & 1 \end{pmatrix},$$

and apply the rotation to T. The multiplication from the left introduces a new nonzero element in the first row, and, correspondingly a new nonzero is introduced in the first column by the multiplication from the right:

$$G_1^T T G_1 = \begin{pmatrix} \times & \times & + & & & \\ \times & \times & \times & & & \\ + & \times & \times & \times & & \\ & & \times & \times & \times & \\ & & & \times & \times & \times \\ & & & & \times & \times \end{pmatrix},$$

where $+$ denotes a new nonzero element. We next determine a rotation in the $(2,3)$-plane that annihilates the new nonzero and at the same time introduces a new nonzero further down:

$$G_2^T G_1^T T G_1 B_2 = \begin{pmatrix} \times & \times & 0 & & & \\ \times & \times & \times & + & & \\ 0 & \times & \times & \times & & \\ & + & \times & \times & \times & \\ & & & \times & \times & \times \\ & & & & \times & \times \end{pmatrix}.$$

In an analogous manner we "chase the bulge" downward until we have

$$\begin{pmatrix} \times & \times & & & & \\ \times & \times & \times & & & \\ & \times & \times & \times & & \\ & & \times & \times & \times & + \\ & & & \times & \times & \times \\ & & & + & \times & \times \end{pmatrix},$$

where by a final rotation we can zero the bulge and at the same time restore the tridiagonal form.

Note that it was only in the determination of the first rotation (15.11) that the shift was used. The rotations were applied only to the *unshifted* tridiagonal matrix. Due to the implicit QR theorem, Theorem 15.15, this is equivalent to a shifted QR step as given in Proposition 15.13.

15.4.2 Eigenvectors

The QR algorithm for computing the eigenvalues of a symmetric matrix (including the reduction to tridiagonal form) requires about $4n^3/3$ flops if only the eigenvalues are computed. Accumulation of the orthogonal transformations to compute the matrix of eigenvectors takes another $9n^3$ flops approximately.

If all n eigenvalues are needed but only a few of the eigenvectors are, then it is cheaper to use inverse iteration (Section 15.2) to compute these eigenvectors, with the computed eigenvalues $\hat{\lambda}_i$ as shifts:

$$(A - \hat{\lambda}_i I)x^{(k)} = x^{(k-1)}, \qquad k = 1, 2, \ldots .$$

The eigenvalues produced by the QR algorithm are so close to the exact eigenvalues (see below) that usually only one step of inverse iteration is needed to get a very good eigenvector, even if the initial guess for the eigenvector is random.

The QR algorithm is ideal from the point of view of numerical stability. There exist an exactly orthogonal matrix Q and a perturbation E such that the computed diagonal matrix of eigenvalues $\hat{D}$ satisfies exactly

$$Q^T(A+E)Q = \hat{D}$$

with $\| E \|_2 \approx \mu \| A \|_2$, where μ is the unit round-off of the floating point system. Then, from Theorem 15.2 we know that a computed eigenvalue $\hat{\lambda}_i$ differs from the exact eigenvalue by a small amount: $\|\hat{\lambda}_i - \lambda_i\|_2 \leq \mu \| A \|_2$.

15.5 Computing the SVD

Since the singular values of a matrix A are the eigenvalues squared of A^TA and AA^T, it is clear that the problem of computing the SVD can be solved using algorithms similar to those of the symmetric eigenvalue problem. However, it is important to avoid forming the matrices A^TA and AA^T, since that would lead to loss of information (cf. the least squares example on p. 54).

Assume that A is $m \times n$ with $m \geq n$. The first step in computing the SVD of a dense matrix A is to reduce it to upper bidiagonal form by Householder transformations from the left and right,

$$A = H \begin{pmatrix} B \\ 0 \end{pmatrix} W^T, \qquad B = \begin{pmatrix} \alpha_1 & \beta_1 & & & \\ & \alpha_2 & \beta_2 & & \\ & & \ddots & \ddots & \\ & & & \alpha_{n-1} & \beta_{n-1} \\ & & & & \alpha_n \end{pmatrix}. \tag{15.12}$$

For a description of this reduction, see Section 7.2.1. Since we use orthogonal transformations in this reduction, the matrix B has the same singular values as A. Let σ be a singular value of A with singular vectors u and v. Then $Av = \sigma u$ is equivalent to

$$\begin{pmatrix} B \\ 0 \end{pmatrix} \tilde{v} = \sigma \tilde{u}, \qquad \tilde{v} = W^T v, \qquad \tilde{u} = H^T u,$$

from (15.12).

It is easy to see that the matrix B^TB is tridiagonal. The method of choice for computing the singular values of B is the tridiagonal QR algorithm with implicit shifts applied to the matrix B^TB, without forming it explicitly.

Let $A \in \mathbb{R}^{m\times n}$, where $m \geq n$. The thin SVD $A = U_1\Sigma V^T$ (cf. Section 6.1) can be computed in $6mn^2 + 20n^3$ flops.

15.6 The Nonsymmetric Eigenvalue Problem

If we perform the same procedure as in Section 15.3 to a nonsymmetric matrix, then due to nonsymmetry, no elements above the diagonal are zeroed. Thus the final result is a *Hessenberg matrix*:

$$V^TAV = \begin{pmatrix} \times & \times & \times & \times & \times & \times \\ \times & \times & \times & \times & \times & \times \\ 0 & \times & \times & \times & \times & \times \\ 0 & 0 & \times & \times & \times & \times \\ 0 & 0 & 0 & \times & \times & \times \\ 0 & 0 & 0 & 0 & \times & \times \end{pmatrix}.$$

The reduction to Hessenberg form using Householder transformations requires $10n^3/3$ flops.

15.6.1 The QR Algorithm for Nonsymmetric Matrices

The “unrefined” QR algorithm for tridiagonal matrices given in Section 15.4 works equally well for a Hessenberg matrix, and the result is an upper triangular matrix, i.e., the R factor in the Schur decomposition. For efficiency, as in the symmetric case, the QR decomposition in each step of the algorithm is computed using plane rotations, but here the transformation is applied to more elements.

We illustrate the procedure with a small example. Let the matrix $H \in \mathbb{R}^{6\times 6}$ be upper Hessenberg, and assume that a Wilkinson shift τ has been computed from the bottom right 2×2 matrix. For simplicity we assume that the shift is real. Denote $H^{(0)} := H$. The first subdiagonal element (from the top) in $H - \tau I$ is zeroed by a rotation from the left in the $(1, 2)$ plane, $G_1^T(H^{(0)} - \tau I)$, and then the second subdiagonal is zeroed by a rotation in $(2, 3)$, $G_2^TG_1^T(H^{(0)} - \tau I)$. Symbolically,

$$\begin{pmatrix} \times & \times & \times & \times & \times & \times \\ \times & \times & \times & \times & \times & \times \\ & \times & \times & \times & \times & \times \\ & & \times & \times & \times & \times \\ & & & \times & \times & \times \\ & & & & \times & \times \end{pmatrix} \longrightarrow \begin{pmatrix} \times & \times & \times & \times & \times & \times \\ 0 & \times & \times & \times & \times & \times \\ & 0 & \times & \times & \times & \times \\ & & \times & \times & \times & \times \\ & & & \times & \times & \times \\ & & & & \times & \times \end{pmatrix}.$$

After $n - 1$ steps we have an upper triangular matrix:

$$R = G_{n-1}^T \cdots G_1^T (H^{(0)} - \tau I) = \begin{pmatrix} \times & \times & \times & \times & \times & \times \\ 0 & \times & \times & \times & \times & \times \\ & 0 & \times & \times & \times & \times \\ & & 0 & \times & \times & \times \\ & & & 0 & \times & \times \\ & & & & 0 & \times \end{pmatrix}.$$

We then apply the rotations from the right, $RG_1 \cdots G_{n-1}$, i.e., we start with a transformation involving the first two columns. Then follows a rotation involving the second and third columns. The result after two steps is

$$\begin{pmatrix} \times & \times & \times & \times & \times & \times \\ + & \times & \times & \times & \times & \times \\ & + & \times & \times & \times\times & \times \\ & & & \times & \times & \times \\ & & & & \times & \times \\ & & & & & \times \end{pmatrix}.$$

We see that the zeroes that we introduced in the transformations from the left are systematically filled in. After $n-1$ steps we have

$$H^{(1)} = RG_1G_2 \cdots G_{n-1} + \tau I = \begin{pmatrix} \times & \times & \times & \times & \times & \times \\ + & \times & \times & \times & \times & \times \\ & + & \times & \times & \times & \times \\ & & + & \times & \times & \times \\ & & & + & \times & \times \\ & & & & + & \times \end{pmatrix}.$$

But we have made a similarity transformation: with $Q = G_1G_2 \cdots G_{n-1}$ and using $R = Q^T(H^{(0)} - \tau I)$, we can write

$$H^{(1)} = RQ + \tau I = Q^T(H^{(0)} - \tau I)Q + \tau I = Q^T H^{(0)} Q, \tag{15.13}$$

and we know that $H^{(1)}$ has the same eigenvalues as $H^{(0)}$.

The convergence properties of the nonsymmetric QR algorithm are almost as nice as those of its symmetric counterpart [93, Chapter 2].

Proposition 15.16. *The nonsymmetric QR algorithm with Wilkinson shifts converges quadratically toward the Schur decomposition.*

As in the symmetric case there are numerous refinements of the algorithm sketched above; see, e.g., [42, Chapter 7], [93, Chapter 2]. In particular, one usually uses implicit double shifts to avoid complex arithmetic.

Given the eigenvalues, selected eigenvectors can be computed by inverse iteration with the upper Hessenberg matrix and the computed eigenvalues as shifts.

15.7 Sparse Matrices

In many applications, a very small proportion of the elements of a matrix are nonzero. Then the matrix is called *sparse*. It is quite common that less than 1% of the matrix elements are nonzero.

In the numerical solution of an eigenvalue problem for a sparse matrix, usually an iterative method is employed. This is because the transformations to compact form described in Section 15.3 would completely destroy the sparsity, which leads to excessive storage requirements. In addition, the computational complexity of the reduction to compact form is often much too high.

In Sections 15.2 and 15.8 we describe a couple of methods for solving numerically the eigenvalue (and singular value) problem for a large sparse matrix. Here we give a brief description of one possible method for storing a sparse matrix.

To take advantage of sparseness of the matrix, only the nonzero elements should be stored. We describe briefly one storage scheme for sparse matrices, *compressed row storage*.

Example 15.17. Let

$$A = \begin{pmatrix} 0.6667 & 0 & 0 & 0.2887 \\ 0 & 0.7071 & 0.4082 & 0.2887 \\ 0.3333 & 0 & 0.4082 & 0.2887 \\ 0.6667 & 0 & 0 & 0 \end{pmatrix}.$$

In compressed row storage, the nonzero entries are stored in a vector, here called `val` (we round the elements in the table to save space here), along with the corresponding column indices in a vector `colind` of equal length:

`val`	0.67	0.29	0.71	0.41	0.29	0.33	0.41	0.29	0.67
`colind`	1	4	2	3	4	1	3	4	1
`rowptr`	1	3	6	9	10				

The vector `rowptr` points to the positions in `val` that are occupied by the first element in each row. ■

The compressed row storage scheme is convenient for multiplying $y = Ax$. The extra entry in the `rowptr` vector that points to the (nonexistent) position after the end of the `val` vector is used to make the code for multiplying $y = Ax$ simple.

Multiplication $y = Ax$ for sparse A

```
function y=Ax(val,colind,rowptr,x)
  % Compute y = A * x, with A in compressed row storage
  m=length(rowptr)-1;
  for i=1:m
    a=val(rowptr(i):rowptr(i+1)-1);
    y(i)=a*x(colind(rowptr(i):rowptr(i+1)-1));
  end
  y=y';
```

It can be seen that compressed row storage is inconvenient for multiplying $y = A^T z$. However, there is an analogous *compressed column storage* scheme that, naturally, is well suited for this.

Compressed row (column) storage for sparse matrices is relevant in programming languages like Fortran and C, where the programmer must handle the sparse storage explicitly [27]. MATLAB has a built-in storage scheme for sparse matrices, with overloaded matrix operations. For instance, for a sparse matrix `A`, the MATLAB statement `y=A*x` implements sparse matrix-vector multiplication, and internally MATLAB executes a code analogous to the one above.

In a particular application, different sparse matrix storage schemes can influence the performance of matrix operations, depending on the structure of the matrix. In [39], a comparison is made of sparse matrix algorithms for information retrieval.

15.8 The Arnoldi and Lanczos Methods

The QR method can be used to compute the eigenvalue and singular value decompositions of medium-size matrices. (What a medium-size matrix is depends on the available computing power.) Often in data mining and pattern recognition the matrices are very large and sparse. However, the eigenvalue, singular value, and Schur decompositions of sparse matrices are usually dense: almost all elements are nonzero.

Example 15.18. The Schur decomposition of the link graph matrix in Example 1.3,

$$P = \begin{pmatrix} 0 & \frac{1}{3} & 0 & 0 & 0 & 0 \\ \frac{1}{3} & 0 & 0 & 0 & 0 & 0 \\ 0 & \frac{1}{3} & 0 & 0 & \frac{1}{3} & \frac{1}{2} \\ \frac{1}{3} & 0 & 0 & 0 & \frac{1}{3} & 0 \\ \frac{1}{3} & \frac{1}{3} & 0 & 0 & 0 & \frac{1}{2} \\ 0 & 0 & 1 & 0 & \frac{1}{3} & 0 \end{pmatrix},$$

was computed in MATLAB: `[U,R]=schur(A)`, with the result

```
U =
   -0.0000   -0.4680   -0.0722   -0.0530    0.8792   -0.0000
   -0.0000   -0.4680   -0.0722   -0.3576   -0.2766   -0.7559
   -0.5394    0.0161    0.3910    0.6378    0.0791   -0.3780
   -0.1434   -0.6458   -0.3765    0.3934   -0.3509    0.3780
   -0.3960   -0.2741    0.6232   -0.4708   -0.1231    0.3780
   -0.7292    0.2639   -0.5537   -0.2934    0.0773   -0.0000
```

```
R =
   0.9207    0.2239   -0.2840    0.0148   -0.1078    0.3334
        0    0.3333    0.1495    0.3746   -0.3139    0.0371
        0         0   -0.6361   -0.5327   -0.0181   -0.0960
        0         0         0   -0.3333   -0.1850    0.1751
        0         0         0         0   -0.2846   -0.2642
        0         0         0         0         0    0.0000
```

We see that almost all elements of the orthogonal matrix are nonzero. ∎

Therefore, since the storage requirements become prohibitive, it is usually out of the question to use the QR method. Instead one uses methods that do not transform the matrix itself but rather use it as an operator, i.e., to compute matrix vector products $y = Ax$. We have already described one such method in Section 15.2, the power method, which can be used to compute the largest eigenvalue and the corresponding eigenvector. Essentially, in the power method we compute a sequence of vectors, $Ax_0, A^2x_0, A^3x_0, \ldots$, that converges toward the eigenvector. However, as soon as we have computed one new power, i.e., we have gone from $y_{k-1} = A^{k-1}x$ to $y_k = A^k x$, we throw away y_{k-1} and all the information that was contained in the earlier approximations of the eigenvector.

The idea in a Krylov subspace method is to use the information in the sequence of vectors $x_0, Ax_0, A^2x_0, \ldots, A^{k-1}$, organized in a subspace, the *Krylov subspace*,

$$\mathcal{K}_k(A, x_0) = \text{span}\{x_0,\ Ax_0,\ A^2x_0,\ \ldots,\ A^{k-1}x_0\},$$

and to extract as good an approximation of the eigenvector as possible from this subspace. In Chapter 7 we have already described the Lanczos bidiagonalization method, which is a Krylov subspace method that can be used for solving approximately least squares problems. In Section 15.8.3 we will show that it can also be used for computing an approximation of some of the singular values and vectors of a matrix. But first we present the Arnoldi method and its application to the problem of computing a partial Schur decomposition of a large and sparse matrix.

15.8.1 The Arnoldi Method and the Schur Decomposition

Assume that $A \in \mathbb{R}^{n\times n}$ is large, sparse, and nonsymmetric and that we want to compute the Schur decomposition (Theorem 15.5) $A = URU^T$, where U is orthogonal and R is upper triangular. Our derivation of the Arnoldi method will be analogous to that in Chapter 7 of the LGK bidiagonalization method. Thus we will start from the existence of an orthogonal similarity reduction to upper Hessenberg form (here $n = 6$):

$$V^TAV = H = \begin{pmatrix} \times & \times & \times & \times & \times & \times \\ \times & \times & \times & \times & \times & \times \\ 0 & \times & \times & \times & \times & \times \\ 0 & 0 & \times & \times & \times & \times \\ 0 & 0 & 0 & \times & \times & \times \\ 0 & 0 & 0 & 0 & \times & \times \end{pmatrix}. \qquad (15.14)$$

In principle this can be computed using Householder transformations as in Section 15.3, but since A is sparse, this would cause the fill-in of the zero elements. Instead we will show that columns of V and H can be computed in a recursive way, using only matrix-vector products (like in the LGK bidiagonalization method).

Rewriting (15.14) in the form

$$AV = \begin{pmatrix} Av_1 & Av_2 & \dots & Av_j & \cdots \end{pmatrix} \tag{15.15}$$

$$= \begin{pmatrix} v_1 & v_2 & \dots & v_j & v_{j+1} & \cdots \end{pmatrix} \begin{pmatrix} h_{11} & h_{12} & \cdots & h_{1j} & \cdots \\ h_{21} & h_{22} & \cdots & h_{2j} & \cdots \\ & h_{32} & \cdots & h_{3j} & \cdots \\ & & \ddots & \vdots & \\ & & & h_{j+1,j} & \\ & & & & \ddots \end{pmatrix} \tag{15.16}$$

and reading off the columns one by one, we see that the first is

$$Av_1 = h_{11}v_1 + h_{21}v_2,$$

and it can be written in the form

$$h_{21}v_2 = Av_1 - h_{11}v_1.$$

Therefore, since v_1 and v_2 are orthogonal, we have $h_{11} = v_1^T Av_1$, and h_{21} is determined from the requirement that v_2 has Euclidean length 1. Similarly, the jth column of (15.15)–(15.16) is

$$Av_j = \sum_{i=1}^{j} h_{ij}v_i + h_{j+1,j}v_{j+1},$$

which can be written

$$h_{j+1,j}v_{j+1} = Av_j - \sum_{i=1}^{j} h_{ij}v_i. \tag{15.17}$$

Now, with $v_1, v_2, \dots, v_j$ given, we can compute v_{j+1} from (15.17) if we prescribe that it is orthogonal to the previous vectors. This gives the equations

$$h_{ij} = v_i^T Av_j, \qquad i = 1, 2, \dots, j.$$

The element $h_{j+1,j}$ is obtained from the requirement that v_{j+1} has length 1.

Thus we can compute the columns of V and H using the following recursion:

Arnoldi method

```
1. Starting vector $v_1$, satisfying $\|v_1\|_2 = 1$.
2. for $j = 1, 2, \dots$
    (a) $h_{ij} = v_i^T A v_j, \quad i = 1, 2, \dots, j$.
    (b) $v = Av_j - \sum_{i=1}^{j} h_{ij} v_i$.
    (c) $h_{j+1,j} = \|v\|_2$.
    (d) $v_{j+1} = (1/h_{j+1,j})\, v$.
3. end
```

Obviously, in step j only one matrix-vector product Av_j is needed.

For a large sparse matrix it is out of the question, mainly for storage reasons, to perform many steps in the recursion. Assume that k steps have been performed, where $k \ll n$, and define

$$V_k = \begin{pmatrix} v_1 & v_2 & \cdots & v_k \end{pmatrix}, \qquad H_k = \begin{pmatrix} h_{11} & h_{12} & \cdots & h_{1k-1} & h_{1k} \\ h_{21} & h_{22} & \cdots & h_{2k-1} & h_{2k} \\ & h_{32} & \cdots & h_{3k-1} & h_{3k} \\ & & \ddots & \vdots & \vdots \\ & & & h_{k,k-1} & h_{kk} \end{pmatrix} \in \mathbb{R}^{k \times k}.$$

We can now write the first k steps of the recursion in matrix form:

$$AV_k = V_k H_k + h_{k+1,k} v_{k+1} e_k^T, \tag{15.18}$$

where $e_k^T = \begin{pmatrix} 0 & 0 & \dots & 0 & 1 \end{pmatrix} \in \mathbb{R}^{1 \times k}$. This is called the *Arnoldi decomposition.*

After k steps of the recursion we have performed k matrix-vector multiplications. The following proposition shows that we have retained all the information produced during those steps (in contrast to the power method).

Proposition 15.19. *The vectors $v_1, v_2, \dots, v_k$ are an orthonormal basis in the Krylov subspace $\mathcal{K}(A, v_1) = \operatorname{span}\{v_1, Av_1, \dots, A^{k-1}v_1\}$.*

Proof. The orthogonality of the vectors follows by construction (or is verified by direct computation). The second part can be proved by induction. $\square$

The question now arises of how well we can approximate eigenvalues and eigenvectors from the Krylov subspace. Note that if Z_k were an eigenspace (see (15.4)), then we would have $AZ_k = Z_k M$ for some matrix $M \in \mathbb{R}^{k \times k}$. Therefore, to see how much V_k deviates from being an eigenspace, we can check how large the residual $AV_k - V_k M$ is for some matrix M. Luckily, there is a recipe for choosing the optimal M for any given V_k.

Theorem 15.20. *Let $V_k \in \mathbb{R}^{n\times k}$ have orthonormal columns, and define $R(M) = AV_k - V_k M$, where $M \in \mathbb{R}^{k\times k}$. Then*

$$\min_M \|R(M)\|_F = \min_M \|AV_k - V_k M\|_F$$

has the solution $M = V_k^T A V_k$.

Proof. See, e.g., [93, Theorem 4.2.6]. □

From the Arnoldi decomposition (15.18) we immediately get the optimal matrix

$$M = V_k^T (V_k H_k + h_{k+1,k} v_{k+1} e_k^T) = H_k,$$

because v_{k+1} is orthogonal to the previous vectors. It follows, again from the Arnoldi decomposition, that the optimal residual is given by

$$\min_M \|R(M)\|_F = \|AV_k - V_k H_k\|_F = |h_{k+1,k}|,$$

so the residual norm comes for free in the Arnoldi recursion.

Assuming that V_k is a good enough approximation of an eigenspace, how can we compute an approximate partial Schur decomposition $AU_k = U_k R_k$? Let

$$H_k = Z_k \widehat{R}_k Z_k^T$$

be the Schur decomposition of H_k. Then, from $AV_k \approx V_k H_k$ we get the approximate partial Schur decomposition of

$$A\widehat{U}_k \approx \widehat{U}_k \widehat{R}_k, \qquad \widehat{U}_k = V_k Z_k.$$

It follows that the eigenvalues of $\widehat{R}_k$ are approximations of the eigenvalues of A.

Example 15.21. We computed the largest eigenvalue of the matrix A_{100} defined in Example 15.10 using the power method and the Arnoldi method. The errors in the approximation of the eigenvalue are given in Figure 15.3. It is seen that the Krylov subspace holds much more information about the eigenvalue than is carried by the only vector in the power method. ■

The basic Arnoldi method sketched above has two problems, both of which can be dealt with efficiently:

- In exact arithmetic the v_j vectors are orthogonal, but in floating point arithmetic orthogonality is lost as the iterations proceed. Orthogonality is repaired by explicitly *reorthogonalizing the vectors*. This can be done in every step of the algorithm or selectively, when nonorthogonality has been detected.
- The amount of work and the storage requirements increase as the iterations proceed, and one may run out of memory before sufficiently good approximations have been computed. This can be remedied by restarting the Arnoldi

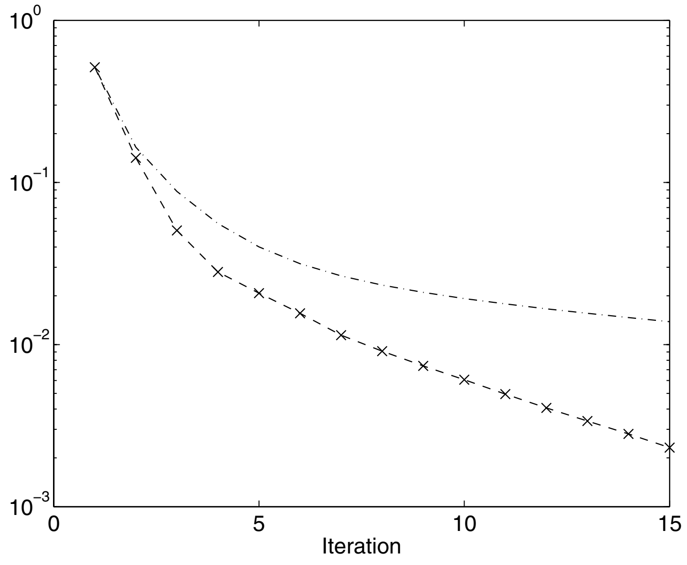

Figure 15.3. *The power and Arnoldi methods for computing the largest eigenvalue of A_{100}. The relative error in the eigenvalue approximation for the power method (dash-dotted line) and the Arnoldi method (dash-$\times$ line).*

procedure. A method has been developed that, given an Arnoldi decomposition of dimension k, reduces it to an Arnoldi decomposition of smaller dimension k_0, and in this reduction purges unwanted eigenvalues. This *implicitly restarted Arnoldi method* [64] has been implemented in the MATLAB function `eigs`.

15.8.2 Lanczos Tridiagonalization

If the Arnoldi procedure is applied to a symmetric matrix A, then, due to symmetry, the upper Hessenberg matrix H_k becomes tridiagonal. A more economical symmetric version of the algorithm can be derived, starting from an orthogonal tridiagonalization (15.5) of A, which we write in the form

$$\begin{aligned} AV &= \begin{pmatrix} Av_1 & Av_2 & \dots & Av_n \end{pmatrix} = VT \\ &= \begin{pmatrix} v_1 & v_2 & \dots & v_n \end{pmatrix} \begin{pmatrix} \alpha_1 & \beta_1 & & & & \\ \beta_1 & \alpha_2 & \beta_2 & & & \\ & \beta_2 & \alpha_3 & \beta_3 & & \\ & & \ddots & \ddots & \ddots & \\ & & & \beta_{n-2} & \alpha_{n-1} & \beta_{n-1} \\ & & & & \beta_{n-1} & \alpha_n \end{pmatrix}. \end{aligned}$$

By identifying column j on the left- and right-hand sides and rearranging the equation, we get

$$\beta_j v_{j+1} = A v_j - \alpha_j v_j - \beta_{j-1} v_{j-1},$$

and we can use this in a recursive reformulation of the equation $AV = VT$. The coefficients α_j and β_j are determined from the requirements that the vectors are orthogonal and normalized. Below we give a basic version of the *Lanczos tridiagonalization method* that generates a Lanczos decomposition,

$$A V_k = V_k T_k + \beta_k v_{k+1} e_k^T,$$

where T_k consists of the k first rows and columns of T.

Lanczos tridiagonalization

```
1. Put $\beta_0 = 0$ and $v_0 = 0$, and choose a starting vector $v_1$, satisfying $\|v_1\|_2 = 1$.
2. **for** $j = 1, 2, \dots$
    (a) $\alpha_j = v_j^T A v_j$.
    (b) $v = A v_j - \alpha_j v_j - \beta_{j-1} v_{j-1}$.
    (c) $\beta_j = \|v\|_2$.
    (d) $v_{j+1} = (1/\beta_j)\, v$.
3. **end**
```

Again, in the recursion the matrix, A is not transformed but is used only in matrix-vector multiplications, and in each iteration only one matrix-vector product need be computed. This basic Lanczos tridiagonalization procedure suffers from the same deficiencies as the basic Arnoldi procedure, and the problems can be solved using the same methods.

The MATLAB function `eigs` checks if the matrix is symmetric, and if this is the case, then the implicitly restarted Lanczos tridiagonalization method is used.

15.8.3 Computing a Sparse SVD

The LGK bidiagonalization method was originally formulated [41] for the computation of the SVD. It can be used for computing a partial bidiagonalization (7.11),

$$A Z_k = P_{k+1} B_{k+1},$$

where B_{k+1} is bidiagonal, and the columns of Z_k and P_{k+1} are orthonormal. Based on this decomposition, approximations of the singular values and the singular vectors can be computed in a similar way as using the tridiagonalization in the preceding section. In fact, it can be proved (see, e.g., [4, Chapter 6.3.3]) that the LGK bidiagonalization procedure is equivalent to applying Lanczos tridiagonalization to the symmetric matrix

$$\begin{pmatrix} 0 & A \\ A^T & 0 \end{pmatrix}, \tag{15.19}$$

with a particular starting vector, and therefore implicit restarts can be applied.

The MATLAB function `svds` implements the Lanczos tridiagonalization method for the matrix (15.19), with implicit restarts.

15.9 Software

A rather common mistake in many areas of computing is to underestimate the costs of developing software. Therefore, it would be very unwise not to take advantage of existing software, especially when it is developed by world experts and is available free of charge.

15.9.1 LAPACK

LAPACK is a linear algebra package that can be accessed and downloaded from Netlib at http://www.netlib.org/lapack/.

We quote from the Web page description:

> LAPACK is written in Fortran 77 and provides routines for solving systems of simultaneous linear equations, least-squares solutions of linear systems of equations, eigenvalue problems, and singular value problems. The associated matrix factorizations (LU, Cholesky, QR, SVD, Schur, generalized Schur) are also provided, as are related computations such as reordering of the Schur factorizations and estimating condition numbers. Dense and banded matrices are handled, but not general sparse matrices. In all areas, similar functionality is provided for real and complex matrices, in both single and double precision.
>
> LAPACK routines are written so that as much as possible of the computation is performed by calls to the Basic Linear Algebra Subprograms (BLAS).... Highly efficient machine-specific implementations of the BLAS are available for many modern high-performance computers.... Alternatively, the user can download ATLAS to automatically generate an optimized BLAS library for the architecture.

The basic dense matrix functions in MATLAB are built on LAPACK.

Alternative language interfaces to LAPACK (or translations/conversions of LAPACK) are available in Fortran 95, C, C++, and Java.

ScaLAPACK, a parallel version of LAPACK, is also available from Netlib at http://www.netlib.org/scalapack/. This package is designed for message passing parallel computers and can be used on any system that supports MPI.

15.9.2 Software for Sparse Matrices

As mentioned earlier, the eigenvalue and singular value functions in MATLAB are based on the Lanczos and Arnoldi methods with implicit restarts [64]. These algorithms are taken from ARPACK at http://www.caam.rice.edu/software/ARPACK/.

From the Web page:

> The package is designed to compute a few eigenvalues and corresponding eigenvectors of a general n by n matrix A. It is most appropriate for large sparse or structured matrices A where structured means that a matrix-vector product w ← Av requires order n rather than the usual order n^2 floating point operations.

An overview of algorithms and software for eigenvalue and singular value computations can be found in the book [4].

Additional software for dense and sparse matrix computations can be found at http://www.netlib.org/linalg/.

15.9.3 Programming Environments

We used MATLAB in this book as a vehicle for describing algorithms. Among other commercially available software systems, we would like to mention Mathematica, and statistics packages like SAS® and SPSS®,[41] which have facilities for matrix computations and data and text mining.

[41] http://www.wolfram.com/, http://www.sas.com/, and http://www.spss.com/.