Chapter 19

The Symmetric Eigenvalue Problem

You should be familiar with

- Properties of a symmetric matrix
- *QR* decomposition
- Givens rotations
- Orthogonal similarity transformations
- Householder reflections
- Deflation during transformations
- Using a shift during eigenvalue computation

Symmetric matrices appear naturally in many applications that include the numerical solution to ordinary and partial differential equations, the theory of quadratic forms, rotation of axes, matrix representation of undirected graphs, and principal component analysis in statistics. Symmetry allows the development of algorithms for solving certain problems more efficiently than the same problem can be solved for a general matrix. For instance, the computation of eigenvalues and their associated eigenvectors can be done accurately and with a lower flop count than their determination for a general matrix. We have invoked the spectral theorem many times in the book. It guarantees that every $n \times n$ symmetric matrix has n linearly independent eigenvectors, even in the presence of eigenvalues of multiplicity greater than 1. We have not proved the theorem, and we do so in this chapter by using Schur's triangularization, developed in Section 18.7.

We will discuss the Jacobi, the symmetric *QR* iteration, the Francis algorithm, and the bisection algorithm for computing the eigenvectors and eigenvalues of a real symmetric matrix. The Jacobi algorithm uses a modification of Givens rotations to create orthogonal similarity transformations that reduce the symmetric matrix into a diagonal matrix containing the eigenvalues, all the while computing the corresponding eigenvectors. The algorithm does not require that the matrix first be brought into upper Hessenberg form. The symmetric *QR* algorithm is an adaptation of the implicit single shift *QR* iteration for a general matrix, except that the shift is chosen to take advantage of the matrix symmetry. Note that a symmetric upper Hessenberg matrix is tridiagonal, and that a reduction to upper triangular form creates a diagonal matrix of eigenvalues. As a result, the symmetric *QR* iteration is faster than the iteration for a general matrix. The Francis algorithm is usually the method of choice for the computation of eigenvalues. First, the matrix is reduced to a tridiagonal matrix. Rather than using the *QR* decomposition, the Francis algorithm performs orthogonal similarity transformations to reduce the tridiagonal matrix to a diagonal matrix of eigenvalues. As the algorithm progresses, an eigenvector can be computed right along with its eigenvalue. The bisection algorithm for computing eigenvalues of a symmetric tridiagonal matrix is very different from the previous methods. By applying the bisection method for computing the roots of a nonlinear function along with some amazing facts about the eigenvalues and characteristic polynomials of a symmetric matrix, the algorithm can accurately compute one particular eigenvalue, all the eigenvalues in an interval, and so forth.

The chapter concludes with a summary of Cuppen's divide-and-conquer algorithm [58]. It is more than twice as fast as the *QR* algorithm if both eigenvalues and eigenvectors of a symmetric tridiagonal matrix are required. However, the algorithm is difficult to implement so that it is stable. In fact, it was 11 years before a proper implementation was discovered [59, 60].

19.1 THE SPECTRAL THEOREM AND PROPERTIES OF A SYMMETRIC MATRIX

We have used the spectral theorem a number of times in the book to develop important results, but have never presented a proof. Schur's triangularization allows to easily prove the spectral theorem.

Theorem 19.1 (Spectral theorem). *If A is a real $n \times n$ symmetric matrix, then it can be factored in the form $A = PDP^{\mathrm{T}}$, where P is an orthogonal matrix containing n orthonormal eigenvectors of A, and D is a diagonal matrix containing the corresponding eigenvalues.*

Numerical Linear Algebra with Applications. http://dx.doi.org/10.1016/B978-0-12-394435-1.00019-3

Proof. Since A is real and symmetric, it has real eigenvalues (Lemma 7.3), and applying Schur's triangularization $A = PTP^{\mathrm{T}}$, where T is an upper triangular matrix. We have $T = P^{\mathrm{T}}AP$, so $T^{\mathrm{T}} = P^{\mathrm{T}}A^{\mathrm{T}}P = P^{\mathrm{T}}AP$ and T is symmetric. Since T is upper triangular, $t_{ij} = 0$, $i > j$. The symmetry of T means that the elements $t_{ji} = 0$, $j < i$, and $D = T$ is a diagonal matrix. Since $A = PDP^{\mathrm{T}}$, A and D are similar matrices, and by Theorem 5.1 they have the same eigenvalues. The eigenvalues of D are the elements on its diagonal and thus are eigenvalues of A. Now, if $P = \begin{bmatrix} v_1 & v_2 & \dots & v_{n-1} & v_n \end{bmatrix}$, then

$$AP = \begin{bmatrix} Av_1 & Av_2 & \dots & Av_{n-1} & Av_n \end{bmatrix} = \begin{bmatrix} \lambda_1 v_1 & \dots & \lambda_{n-1} v_{n-1} & \lambda_n v_n \end{bmatrix} = \begin{bmatrix} v_1 & \dots & v_{n-1} & v_n \end{bmatrix} \operatorname{diag}(\lambda_1, \lambda_2, \dots, \lambda_{n-1}, \lambda_n)$$

so $Av_i = \lambda_i v_i$, $1 \le i \le n$, and the columns of P are orthonormal eigenvectors of A. □

Remark 19.1. Since a real symmetric matrix has n orthonormal eigenvectors, we see from the proof of Theorem 19.1 that we can arrange P so its columns contain eigenvectors that correspond to the eigenvalues in decreasing order of magnitude; in other words, D will have the form $\begin{bmatrix} \lambda_1 & & 0 \\ & \lambda_2 & \\ & & \ddots \\ 0 & & \lambda_n \end{bmatrix}$, where $|\lambda_1| \ge |\lambda_2| \ge \cdots \ge |\lambda_n|$.

19.1.1 Properties of a Symmetric Matrix

We begin by listing some properties of a symmetric matrix we already know and then developing a new result.

- The eigenvalues of a symmetric matrix are real, and the eigenvectors can be assumed to be real (Lemma 7.3).
- If A is symmetric and x, y are $n \times 1$ vectors, then $\langle Ax, y\rangle = \langle x, Ay\rangle$.
- If A is a symmetric matrix, $\|A\|_2 = \rho(A)$, where $\rho(A)$ is the spectral radius of A.
- A symmetric $n \times n$ matrix A is positive definite if and only if all its eigenvalues are positive.
- If A is a real symmetric matrix, then any two eigenvectors corresponding to distinct eigenvalues are orthogonal (Theorem 6.4).
- A symmetric matrix can be diagonalized with an orthogonal matrix (Theorem 19.1—the spectral theorem). Thus, even if there are multiple eigenvalues, there is always an orthonormal basis of n eigenvectors.

Another special property of a real symmetric matrix is that the eigenvalues are well conditioned. Thus, by using a good algorithm we can compute eigenvalues with assurance that the errors will be small.

Theorem 19.2. *The eigenvalues of a real symmetric matrix are well conditioned.*

Proof. We know by the spectral theorem that any real symmetric matrix can be diagonalized. If x is a normalized right eigenvector of A corresponding to eigenvalue λ, then $Ax = \lambda x$. We have $x^{\mathrm{T}}A^{\mathrm{T}} = x^{\mathrm{T}}A = \lambda x^{\mathrm{T}}$, so x is also a left eigenvector of A. The condition number, κ, of λ is

$$\kappa(\lambda) = \frac{1}{x^{\mathrm{T}}x} = 1,$$

and λ is perfectly conditioned. □

Since this chapter concerns the accurate computation of eigenvalues for a symmetric matrix, this is good news. Unfortunately, the same is not true for the eigenvectors of a symmetric matrix. They can be ill-conditioned (see Example 19.2 and Problem 19.21). The sensitivity of the eigenvalues of a symmetric matrix depends on the separation of the eigenvalues. If a matrix has an eigenvalue of multiplicity greater than 1, or if there is a cluster of closely spaced eigenvalues, the eigenvectors will be ill-conditioned.

19.2 THE JACOBI METHOD

As we have indicated, special algorithms that exploit symmetry have been developed for finding the eigenvalues and eigenvectors of a symmetric matrix. We will confine ourselves to real symmetric matrices and begin with the Jacobi method. The Jacobi method does not first transform the matrix A to upper Hessenberg form, but transforms A directly to a diagonal matrix using orthogonal similarity transformations. The strategy used in the Jacobi algorithm is to develop an iteration that will make the sum of the squares of the entries off the diagonal converge to 0, thus obtaining a diagonal matrix of eigenvalues.

In practical terms, the algorithm continues until $\sqrt{\sum_{i=1}^{n}\sum_{j=1,j\neq i}^{n} a_{ij}^2}$ is sufficiently small. Recall that $\|A\|_F^2 = \sum_{i=1}^{n}\sum_{j=1}^{n} a_{ij}^2$ is the square of the Frobenius norm, and so we can define the function off(A) as follows.

Definition 19.1. $\text{off}(A) = \sqrt{\sum_{i=1}^{n}\sum_{j=1,j\neq i}^{n} a_{ij}^2} = \sqrt{\|A\|_F^2 - \sum_{i=1}^{n} a_{ii}^2}$

The algorithm creates orthogonal matrices $J_0, J_1, \ldots J_{k-1}$ such that

$$\lim_{k\longrightarrow\infty} \text{off}(A_k) = 0,$$

where

$$\begin{cases} A_0 = A \\ A_k = J_{k-1}^{\mathrm{T}} A_{k-1} J_{k-1} \end{cases}$$

Since $A = A_0$ is symmetric and $\left(J_{k-1}^{\mathrm{T}} A_{k-1} J_{k-1}\right)^{\mathrm{T}} = J_{k-1}^{\mathrm{T}} A_{k-1}^{\mathrm{T}} J_{k-1} = J_{k-1}^{\mathrm{T}} A_{k-1} J_{k-1}$, A_k is symmetric. Each orthogonal matrix J_k is a Givens rotation, slightly different from the rotations used in Section 17.2. Those Givens rotations were of the form

$$J(i,j,c,s) = \begin{array}{c} \\ \\ i \\ \\ j \\ \\ \end{array}\!\!\begin{array}{c} \begin{array}{ccccccc} & & \text{i} & & \text{j} & & \end{array} \\ \begin{bmatrix} 1 & 0 & 0 & \cdots & \cdots & \cdots & 0 \\ \vdots & \ddots & & & \cdots & \cdots & \vdots \\ & & c & \cdots & \text{s} & \cdots & 0 \\ \vdots & & & \ddots & & & \\ & & \text{-s} & & \text{c} & & 0 \\ \vdots & & & & & \ddots & \\ 0 & & & & & & 1 \end{bmatrix} \end{array},$$

where the numbers c and s were chosen such that the product $J(i,j,c,s)A$ caused entry a_{ji} to become 0. For the Jacobi method, we choose c and s so that $J(i,j,c,s)^{\mathrm{T}} AJ(i,j,c,s)$ zeros out the pair of nonzero elements a_{ij} and a_{ji}.

Suppose we are at step k of the iteration, $A_k = J_{k-1}^{\mathrm{T}} A_{k-1} J_{k-1}$, and want to zero out nonzero entries $a_{ij}^{(k-1)}$ and $a_{ji}^{(k-1)}$ of A_{k-1}. To determine how we should choose c and s, look at the product $\overline{A} = J^{\mathrm{T}}(i,j,c,s)AJ(i,j,c,s)$.

$$\begin{array}{c} \\ \\ i \\ \\ j \\ \\ \end{array}\!\!\begin{array}{c} \begin{array}{ccccccc} & & \text{i} & & \text{j} & & \end{array} \\ \begin{bmatrix} 1 & \cdots & 0 & \cdots & 0 & \cdots & 0 \\ \vdots & \ddots & & & & & \\ 0 & & c & \cdots & -s & & 0 \\ \vdots & & & \ddots & & & \\ 0 & & s & & c & & 0 \\ \vdots & & \vdots & & \vdots & & \\ 0 & & 0 & & 0 & & 1 \end{bmatrix} \end{array} \quad J^{\mathrm{T}}(i,j,c,s)$$

$$\begin{array}{c} \\ \\ i \\ \\ j \\ \\ \end{array}\!\!\begin{array}{c} \begin{array}{ccccccc} & & \text{i} & & \text{j} & & \end{array} \\ \begin{bmatrix} a_{11} & \cdots & & \cdots & & \cdots & a_{1n} \\ \vdots & \ddots & & & & & \\ & & a_{ii} & \cdots & a_{ij} & & \\ \vdots & & & \ddots & & & \\ & & a_{ji} & & a_{jj} & & \\ \vdots & & \vdots & & \vdots & \ddots & \\ a_{1n} & & 0 & & 0 & & a_{nn} \end{bmatrix} \end{array} \quad A$$

$$\begin{array}{c} \\ \\ i \\ \\ j \\ \\ \end{array}\!\!\begin{array}{c} \begin{array}{ccccccc} & & \text{i} & & \text{j} & & \end{array} \\ \begin{bmatrix} 1 & \cdots & 0 & \cdots & 0 & \cdots & 0 \\ \vdots & \ddots & & & & & \\ 0 & & c & \cdots & s & & 0 \\ \vdots & & & \ddots & & & \\ 0 & & -\text{s} & & \text{c} & & 0 \\ \vdots & & \vdots & & \vdots & \ddots & \\ 0 & & 0 & & 0 & & 1 \end{bmatrix} \end{array} \quad J(i,j,c,s)$$

In forming the product $\overline{A} = J(i,j,c,s)^T AJ(i,j,c,s)$, patterns emerge, and are displayed in the following table.

TABLE 19.1 Jacobi Iteration Formulas

$\overline{a_{kl}} = a_{kl},\ k,l \neq i,j$
$\overline{a_{ik}} = \overline{a_{ki}} = ca_{ik} - sa_{jk},\ k \neq i,j$
$\overline{a_{jk}} = \overline{a_{kj}} = sa_{ik} + ca_{jk},\ k \neq i,j$
$\overline{a_{ij}} = \overline{a_{ji}} = \left(c^2 - s^2\right)a_{ij} + cs\left(a_{ii} - a_{jj}\right)$
$\overline{a_{ii}} = c^2 a_{ii} - 2csa_{ij} + s^2 a_{jj}$
$\overline{a_{jj}} = s^2 a_{ii} + 2csa_{ij} + c^2 a_{jj}$

We want $\overline{a_{ij}} = \overline{a_{ji}} = 0$, so let

$$\left(c^2 - s^2\right) a_{ij} + cs\left(a_{ii} - a_{jj}\right) = 0. \tag{19.1}$$

If $a_{ij} = 0$, let $c = 1$, and $s = 0$. This satisfies Equation 19.1. Otherwise, we must find c and s. Rewrite Equation 19.1 as

$$\frac{c^2 - s^2}{cs} = \frac{a_{jj} - a_{ii}}{a_{ij}} \tag{19.2}$$

Note that we have assumed $a_{ij} \neq 0$. Since $J(i, j, c, s)$ is orthogonal, we must have $c^2 + s^2 = 1$, and $c = \cos\theta$, $s = \sin\theta$ for some θ. Substitute these relations into Equation 19.2 to obtain

$$\frac{\cos^2\theta - \sin^2\theta}{\sin\theta\cos\theta} = \frac{a_{jj} - a_{ii}}{a_{ij}} \tag{19.3}$$

Noting that $\cos 2\theta = \cos^2\theta - \sin^2\theta$, and $\sin 2\theta = 2\sin\theta\cos\theta$, Equation 19.3 can be written as

$$\cot 2\theta = \frac{1}{2}\left(\frac{a_{jj} - a_{ii}}{a_{ij}}\right). \tag{19.4}$$

We need the following trigonometric identity:

$$\tan^2\theta + 2\tan\theta\cot 2\theta - 1 = 0. \tag{19.5}$$

Equation 19.5 is verified as follows:

$$\begin{aligned}
\tan^2\theta + 2\tan\theta\cot 2\theta - 1 &= \frac{\sin^2\theta}{\cos^2\theta} + 2\frac{\sin\theta}{\cos\theta}\frac{\cos 2\theta}{\sin 2\theta} - 1 \\
&= \frac{\sin^2\theta}{\cos^2\theta} + 2\frac{\sin\theta}{\cos\theta}\frac{\left(\cos^2\theta - \sin^2\theta\right)}{2\sin\theta\cos\theta} - 1 \\
&= \frac{\sin^2\theta}{\cos^2\theta} + \frac{\cos^2\theta - \sin^2\theta}{\cos^2\theta} - 1 = 0.
\end{aligned}$$

Noting that $\tan\theta = \dfrac{s}{c}$, we now use the identity 19.5 with Equations 19.2 and 19.4 to obtain

$$\frac{s^2}{c^2} + 2\frac{s}{c}\frac{1}{2}\left(\frac{a_{jj} - a_{ii}}{a_{ij}}\right) - 1 = \frac{s^2}{c^2} + \frac{s}{c}\left(\frac{c^2 - s^2}{cs}\right) - 1 = 1 - 1 = 0. \tag{19.6}$$

Let $t = \frac{s}{c}$, and we can write Equation 19.6 as

$$t^2 + 2\tau t - 1 = 0, \tag{19.7}$$

where $\tau = \frac{1}{2}\left(\dfrac{a_{jj} - a_{ii}}{a_{ij}}\right)$. An application of the quadratic equation to Equation 19.7 produces two roots:

$$t = -\tau \pm \sqrt{\tau^2 + 1}.$$

The solutions for t can be rewritten as follows:

$$\begin{aligned}
t_1 &= \left(-\tau + \sqrt{\tau^2 + 1}\right)\frac{\left(-\tau - \sqrt{\tau^2 + 1}\right)}{\left(-\tau - \sqrt{\tau^2 + 1}\right)} = \frac{-1}{-\tau - \sqrt{\tau^2 + 1}} = \frac{1}{\tau + \sqrt{\tau^2 + 1}}, \\
t_2 &= \left(-\tau - \sqrt{\tau^2 + 1}\right)\frac{\left(-\tau + \sqrt{\tau^2 + 1}\right)}{\left(-\tau + \sqrt{\tau^2 + 1}\right)} = \frac{-1}{-\tau + \sqrt{\tau^2 + 1}}.
\end{aligned}$$

When $\tau < 0$, choose t_2, and when $\tau > 0$, choose t_1. This choice means we are adding two positive numbers in the denominator, and so there is no cancelation error. Using t, we now must find values for c and s. When we find an appropriate value for c,

$$s = ct.$$

Now, $1 + \tan^2\theta = 1 + t^2 = \sec^2\theta = \frac{1}{\cos^2\theta} = \frac{1}{c^2}$, so

$$c = \frac{1}{\sqrt{1+t^2}}.$$

The following is a summary of our results to this point.

TABLE 19.2 Computation of c and s for the Jacobi Method

$\tau = \frac{a_{jj} - a_{ii}}{2a_{ij}}$
$t = \left\{ \begin{array}{ll} \frac{1}{\tau + \sqrt{\tau^2+1}} & \tau \geq 0 \\ \frac{-1}{-\tau + \sqrt{\tau^2+1}} & \tau < 0 \end{array} \right)$
$c = \frac{1}{\sqrt{1+t^2}}$
$s = ct$

The question now is the issue of convergence. We will investigate this question by looking at the off-diagonal entries as the iteration progresses. Before proceeding, we need to note the following:

- trace $(AB) =$ trace (BA) for any $n \times n$ matrices A and B (Theorem 1.2).
- If A is symmetric, trace $\left(A^2\right) = \sum_{i=1}^{n} \sum_{j=1}^{n} a_{ij}^2$ (Problem 19.2).
- If P is an orthogonal matrix, then $\left\|P^{\mathrm{T}}AP\right\|_F = \|A\|_F$, which follows from Lemma 15.1.

A technical lemma is necessary before we can prove convergence of the Jacobi method. It says that $\text{off}(\overline{A}_k) \leq \text{off}\left(\overline{A}_{k-1}\right)$ ($\text{off}(\overline{A}_i)$ is monotonically decreasing).

Lemma 19.1. *If* $\overline{A} = J(i,j,c,s)^{\mathrm{T}} A J(i,j,c,s)$, *then*

$$\text{off}(\overline{A})^2 = \text{off}(A)^2 - 2a_{ij}^2.$$

Proof. By looking at Table 19.1, it is evident that all the entries on the diagonal of $\overline{A}$ except $\overline{a}_{ii}$ and $\overline{a}_{jj}$ are the same as those of A. The elements at indices (i,i) and (j,j) on the diagonal must satisfy the following relation:

$$\begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix} \begin{bmatrix} a_{ii} & a_{ij} \\ a_{ij} & a_{jj} \end{bmatrix} \begin{bmatrix} \cos\theta & \sin\theta \\ -\sin\theta & \cos\theta \end{bmatrix} = \begin{bmatrix} \overline{a}_{ii} & 0 \\ 0 & \overline{a}_{jj} \end{bmatrix},$$

and so

$$\left\| \begin{bmatrix} \overline{a}_{ii} & 0 \\ 0 & \overline{a}_{jj} \end{bmatrix} \right\|_F^2 = \left\| \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix} \begin{bmatrix} a_{ii} & a_{ij} \\ a_{ij} & a_{jj} \end{bmatrix} \begin{bmatrix} \cos\theta & \sin\theta \\ -\sin\theta & \cos\theta \end{bmatrix} \right\|_F^2,$$

By Lemma 15.1,

$$\overline{a}_{ii}^2 + \overline{a}_{jj}^2 = a_{ii}^2 + a_{jj}^2 + 2a_{ij}^2.$$

Since all the other diagonal entries of $\overline{A}$ are identical to those of A, we have

$$\sum_{i=1}^{n} \overline{a}_{ii}^2 = \sum_{i=1}^{n} a_{ii}^2 + 2a_{ij}^2. \tag{19.8}$$

Since A and $\overline{A}$ are orthogonally similar, $\|\overline{A}\|_F^2 = \|A\|_F^2$ by Lemma 15.1. This and Equation 19.8 imply

$$\text{off}(\overline{A})^2 = \|\overline{A}\|_F^2 - \sum_{k=1}^{n} \overline{a}_{ii}^2 = \|A\|_F^2 - \sum_{i=1}^{n} a_{ii}^2 - 2a_{ij}^2 = \text{off}(A)^2 - 2a_{ij}^2. \qquad \square$$

Theorem 19.3. *At each iteration, choose for* a_{ij} *the off-diagonal element largest in magnitude. With this strategy, the Jacobi method converges.*

Proof. Consider $A_k = J_{k-1}^{\mathrm{T}} A_{k-1} J_{k-1}$. There are $n^2 - n$ entries off the diagonal, and since a_{ij} is largest in magnitude,

$$(\mathrm{off}\,(A_{k-1}))^2 \leq n\,(n-1)\,a_{ij}^2.$$

Write the equation in the form

$$a_{ij}^2 \geq \frac{(\mathrm{off}\,(A_{k-1}))^2}{n\,(n-1)}.$$

By Lemma 19.1, $\mathrm{off}\,(A_k)^2 = \mathrm{off}\,(A_{k-1})^2 - 2a_{ij}^2$, and so

$$\mathrm{off}(A_k)^2 \leq \mathrm{off}\,(A_{k-1})^2 - 2\frac{(\mathrm{off}\,(A_{k-1}))^2}{n\,(n-1)} = \left(1 - \frac{1}{N}\right)\mathrm{off}\,(A_{k-1})^2 \tag{19.9}$$

where $N = \frac{n(n-1)}{2}$. Equation 19.9 shows that after k Jacobi iteration steps,

$$\mathrm{off}(A_k) \leq \left(\sqrt{1 - \frac{1}{N}}\right)^k \mathrm{off}\,(A)\,,$$

which shows that the Jacobi iteration converges. □

The term $\mathrm{off}\,(A_k)$ decreases at a rate of $\sqrt{\left(1 - \frac{1}{N}\right)}$. This rate of convergence is considered linear. However, it can be shown that the average rate of convergence (asymptotic rate) is quadratic [2, pp. 479-480].

Remark 19.2. Each iteration of the Jacobi algorithm makes a_{ij} and a_{ji} zero, and the computation to do this can destroy pairs of zeros already created. However, as the iteration progresses, $\mathrm{off}\,(A_k)$ decreases, leaving approximations to the eigenvalues on the diagonal. There is a function in the software distribution, `eigsymjdemo`, that allows you to see this behavior happening.

19.2.1 Computing Eigenvectors Using the Jacobi Iteration

Our preceding discussion did not include the computation of eigenvectors, and it is easy to do. The iteration computes

$$J_k^{\mathrm{T}} J_{k-1}^{\mathrm{T}} \ldots J_2^{\mathrm{T}} J_1^{\mathrm{T}} A J_1 J_2 \ldots J_{k-1} J_k = (J_1 J_2 \ldots J_{k-1} J_k)^{\mathrm{T}} A\,(J_1 J_2 \ldots J_{k-1} J_k) \approx D,$$

where D is a diagonal matrix of eigenvalues, and each J_i is a Givens rotation. Thus, the matrix

$$J = J_1 J_2 \ldots J_{k-1} J_k$$

is an orthogonal matrix of eigenvectors. Starting with $J_0 = I$, maintain this product.

19.2.2 The Cyclic-by-Row Jacobi Algorithm

The algorithm we have presented is called the *classical Jacobi method*. The product $A_1 = J\,(i,j,c,s)^{\mathrm{T}} A_{k-1}$ affects only rows i and j of A_{k-1}, and $A_2 = A_1 J\,(i,j,c,s)$ affects only columns i and j of A_1. Thus, each Jacobi iteration costs $O\,(n)$ flops. To find the largest entry in magnitude requires searching $\frac{n(n-1)}{2}$ entries, so it is necessary to execute $O\left(n^2\right)$ comparisons for one Jacobi iteration. This is simply too expensive, so it is almost never used in practice.

There is a modification of the Jacobi algorithm that is designed to speed it up. It is known as the *cyclic-by-row Jacobi algorithm*. Compute $\mathrm{off}(A)$ by cycling through the $N = \frac{n(n-1)}{2}$ entries above the diagonal by rows left-to-right as follows:

$$(1,2)\,,(1,3)\,,\ldots,(1,n)\,,(2,3)\,,(2,4)\,,\ldots,(2,n)\,,\ldots,(n-1,n)\,.$$

One cycle of N Jacobi rotations is termed a *sweep*, and sweeps are performed until $\mathrm{off}(A_k)$ is sufficiently small. This algorithm has the same convergence properties as the classical Jacobi algorithm [9, pp. 270-271].

We are now ready to specify Algorithm 19.1, eigsymj, that computes the eigenvalues and corresponding eigenvectors of a real symmetric matrix. We assume the following functions are available:

- jacobics: computes c and s for the Jacobi rotation defined in Table 19.2.
- jacobimul: computes the Jacobi rotation $J\,(i,j,c,s)^{\mathrm{T}} AJ\,(i,j,c,s)$ for an iteration of the Jacobi algorithm as defined in Table 19.1.

- givensmulp: computes $AJ(i, j, c, s)$.
- off: computes $\text{off}(A) = \sqrt{\sum_{k=1}^{n} \sum_{p=1, k \neq p}^{n} a_{kp}^2}$.

NLALIB: The function `eigsymj` implements Algorithm 19.1. The supporting functions `jacobics`, `jacobimul`, `givensmulp`, and `off` are in the book software distribution.

Remark 19.3. The MATLAB implementation uses variable input and output arguments to make `eigsymj` as flexible as possible. Here are the possible calling formats:

- `eigsymj(A)`: returns a column vector of eigenvalues.
- `E = eigsymj(A)`: assigns E a column vector containing the eigenvalues.
- `[V, D] = eigsymj(A)`: assigns the columns of V the eigenvectors corresponding to the eigenvalues on the diagonal matrix D.
- `[V, D, numsweeps] = eigsymj(A)`: adds the number of sweeps required to the output.

Algorithm 19.1 Jacobi Method for Computing All Eigenvalues of a Real Symmetric Matrix

```
function EIGSYMJ(A,tol,maxsweeps)
  % executes the cyclic-by-row Jacobi method to approximate the eigenvalues
  % and eigenvectors of a real symmetric matrix A.
  % [V D numsweeps]=eigsymj(A,tol,maxsweeps) returns an orthogonal
  % matrix V and diagonal matrix D of eigenvalues such that
  % V^T AV = D. The algorithm returns when tol < off(A_k).
  % If the desired tolerance is not obtained within maxsweeps sweeps,
  % a value of -1 is returned for numsweeps.

  Print error message and return if A is not symmetric.

  desiredAccuracy=false
  numsweeps=1
  V=I
  while (numsweeps ≤ maxsweeps) and (not desiredAccuracy) do
    % execute a cycle of n(n-1)/2 Jacobi rotations.
    for i=1:n-1 do
      for j=i+1:n do
        % compute c and s so that a_ij = a_ji = 0.
        [ c  s ] = jacobics(A, i, j)
        % compute A = J(i, j, c, s)^T AJ(i, j, c, s)
        A=jacobimul(A,i,j,c,s)
        % multiply V on the right by the Givens rotation J(i,j,c,s).
        V=givensmulp(V,i,j,c,s);
      end for
    end for
    if off(A)<tol then
      desiredAccuracy=true
    end if
    numsweeps=numsweeps+1
  end while

  if desiredAccuracy=false then
    numsweeps=-1
  end if
  D=diag(diag(A))
  return [ V  D  numsweeps ]
end function
```

Example 19.1. Let $A = \begin{bmatrix} 1 & 5 & 2 \\ 5 & -1 & 3 \\ 2 & 3 & 4 \end{bmatrix}$. Two sweeps of the Jacobi algorithm produces the following sequence of matrices, with the data rounded to four decimal places.

$$\begin{bmatrix} 5.0990 & 0.0000 & 3.4487 \\ 0.0000 & -5.0990 & 1.0520 \\ 3.4487 & 1.0520 & 4.0000 \end{bmatrix}, \quad \begin{bmatrix} 8.0417 & 0.6829 & 0.0000 \\ 0.6829 & -5.0990 & 0.8003 \\ 0.0000 & 0.8003 & 1.0574 \end{bmatrix}, \quad \begin{bmatrix} 8.0417 & 0.6774 & 0.0866 \\ 0.6774 & -5.2014 & 0.0000 \\ 0.0866 & 0.0000 & 1.1597 \end{bmatrix},$$

$$\begin{bmatrix} 8.0762 & 0.0000 & 0.0865 \\ 0.0000 & -5.2359 & -0.0044 \\ 0.0865 & -0.0044 & 1.1597 \end{bmatrix}, \quad \begin{bmatrix} 8.0773 & -0.0001 & 0.0000 \\ -0.0001 & -5.2359 & -0.0044 \\ 0.0000 & -0.0044 & 1.1586 \end{bmatrix}, \quad \begin{bmatrix} 8.0773 & -0.0001 & 0.0000 \\ -0.0001 & -5.2359 & 0.0000 \\ 0.0000 & 0.0000 & 1.1586 \end{bmatrix}.$$

Notice that the first rotation made $a_{12} = a_{21} = 0$, but the second rotation made $a_{13} = a_{31} = 0$, while a_{12} and a_{21} became 0.6829. On the third rotation the entries a_{31} and a_{13} become 0.0866 when a_{32} and a_{23} become 0. Despite this behavior, off(A) for the final matrix is 7.80332×10^{-5}. The values on the diagonal of the final matrix are eigenvalues correct to four decimal places. ■

Example 19.2. This example demonstrates that a symmetric matrix can have ill-conditioned eigenvectors. The symmetric matrix EIGVECSYMCOND from the software distribution has a condition number of 70.867. The following MATLAB statements clearly demonstrate that the eigenvectors are ill-conditioned.

```
>> [V1,D1] = eigsymj(EIGVECSYMCOND,1.0e-14,20);
>> E = 1.0e-10*rand(25,1);
>> E = diag(E);
>> EIGVECSYMCONDP = EIGVECSYMCOND;
>> EIGVECSYMCONDP = EIGVECSYMCONDP + E;
>> [V2,D2] = eigsymj(EIGVECSYMCONDP,1.0e-14,20);
>> norm(D1-D2)

ans =
   1.091393642127514e-10

>> norm(V1-V2)

ans =

   0.962315272737213
```

■

19.3 THE SYMMETRIC *QR* ITERATION METHOD

In this section, we will develop the symmetric *QR* iteration method for computing the eigenvalues and eigenvectors of a real symmetric matrix. The method takes advantage of concepts we have already developed, namely, the orthogonal reduction of a matrix to upper Hessenberg form and the shifted Hessenberg *QR* iteration. However, adjustments are made to take advantage of matrix symmetry. A symmetric upper Hessenberg matrix is tridiagonal, so our first task is to develop an efficient way to take advantage of symmetry when reducing A to a tridiagonal matrix. We will use Householder matrices for the orthogonal similarity transformations.

We motivate the process using a general symmetric 3×3 matrix.

$$A = \begin{bmatrix} a & b & c \\ b & d & e \\ c & e & f \end{bmatrix}.$$

Recall that a Householder matrix is orthogonal and symmetric. Using the vector $x_1 = \begin{bmatrix} b \\ c \end{bmatrix}$, create a 2×2 Householder matrix $H_{u_1} = \begin{bmatrix} h_{11} & h_{12} \\ h_{12} & h_{22} \end{bmatrix}$ that zeros out c. Then,

$$H_{u_1}\begin{bmatrix} b \\ c \end{bmatrix} = \begin{bmatrix} h_{11}b + h_{12}c \\ h_{12}b + h_{22}c \end{bmatrix} = \begin{bmatrix} h_{11}b + h_{12}c \\ 0 \end{bmatrix}.$$

Embed H_{u_1} in the 3×3 identity matrix as the lower 2×2 submatrix to form

$$H_1 = \begin{bmatrix} 1 & 0 & 0 \\ 0 & h_{11} & h_{12} \\ 0 & h_{12} & h_{22} \end{bmatrix}.$$

Form the product

$$\begin{aligned} H_1 A &= \begin{bmatrix} 1 & 0 & 0 \\ 0 & h_{11} & h_{12} \\ 0 & h_{12} & h_{22} \end{bmatrix} \begin{bmatrix} a & b & c \\ b & d & e \\ c & e & f \end{bmatrix} = \begin{bmatrix} a & b & c \\ h_{11}b + h_{12}c & h_{11}d + h_{12}e & h_{11}e + h_{12}f \\ h_{12}b + h_{22}c & h_{12}d + h_{22}e & h_{12}e + h_{22}f \end{bmatrix} \\ &= \begin{bmatrix} a & b & c \\ h_{11}b + h_{12}c & h_{11}d + h_{12}e & h_{11}e + h_{12}f \\ 0 & h_{12}d + h_{22}e & h_{12}e + h_{22}f \end{bmatrix}. \end{aligned}$$

Now multiply on the right by H_1 to obtain

$$H_1 A H_1 = \begin{bmatrix} a & bh_{11} + ch_{12} & 0 \\ h_{11}b + ch_{12} & h_{11}(h_{11}d + h_{12}c) + h_{12}(h_{11}e + h_{12}f) & h_{12}(h_{11}d + h_{12}e) + h_{22}(h_{11}e + h_{12}f) \\ 0 & h_{12}(h_{11}d + h_{12}e) + h_{22}(h_{11}e + h_{12}f) & h_{12}(h_{12}e + h_{22}f) \end{bmatrix}.$$

The product is symmetric and tridiagonal.

The algorithm for an $n \times n$ symmetric matrix is a generalization of this 3×3 example, and results in the Algorithm 19.2, trireduce. If the details are not required, the reader can skip to Example 19.4 that demonstrates the use of trireduce.

For an $n \times n$ symmetric matrix, construct a Householder matrix that will zero out all the elements below a_{21}. For this, choose the vector $x_1 = \begin{bmatrix} a_{21} \\ a_{31} \\ \vdots \\ a_{n1} \end{bmatrix}$ and form the $(n-1) \times (n-1)$ Householder matrix $H_{u_1}^{(n-1)}$. Insert it into the identity matrix to create matrix H_1 so that

$$\begin{aligned} H_1 A &= \begin{bmatrix} 1 & 0 & 0 & \dots & 0 \\ 0 & & & & \\ 0 & & H_{u_1}^{(n-1)} & & \\ \vdots & & & & \\ 0 & & & & \end{bmatrix} \begin{bmatrix} a_{11} & a_{12} & a_{13} & \dots & a_{1n} \\ a_{21} & & & & \\ a_{31} & & & & \\ \vdots & & & & \\ a_{n1} & & & & \end{bmatrix} \\ &= \begin{bmatrix} a_{11} & a_{12} & a_{13} & \dots & a_{1n} \\ v_1 & & & & \\ 0 & & X & & \\ \vdots & & & & \\ 0 & & & & \end{bmatrix} \end{aligned}$$

The value $v_1 = \pm \|x_1\|_2$ (Section 17.8.1). The product zeros out all the entries in column 1 in the index range $(3,1) - (n,1)$, alters a_{21}, but leaves the first row of A unchanged. Noting that $H_1^{\mathrm{T}} = H_1$, form the orthogonal similarity transformation

$$A^{(1)} = H_1 A H_1 = \begin{bmatrix} a_{11} & v_1 & 0 & \dots & 0 \\ v_1 & a_{22}^{(1)} & & & \\ 0 & & X & & \\ \vdots & & & & \\ 0 & & & & \end{bmatrix}$$

that maintains symmetry and zeros out the elements at indices $(1,3)$-$(1,n)$.

The next step is to zero out the elements in $A^{(1)}$ at indices $(4,2)$-$(n,2)$ using the vector $x_2 = \begin{bmatrix} a_{32}^{(1)} \\ a_{42}^{(1)} \\ a_{52}^{(1)} \\ \vdots \\ a_{n2}^{(1)} \end{bmatrix}$ to construct the Householder $(n-2) \times (n-2)$ Householder matrix

$$H_2 = \begin{matrix} 1 & 0 & 0 & \dots & 0 \\ 0 & 1 & 0 & \dots & 0 \\ 0 & 0 & & & \\ \vdots & \vdots & & H_{u_2}^{(n-2)} & \\ 0 & 0 & & & \end{matrix}.$$

The identity matrix in the upper left corner maintains the tridiagonal structure already built. Now form $A_2 = H_2 A_1 H_2$ to create the matrix

$$\begin{matrix} a_{11} & v_1 & 0 & \dots & 0 \\ v_1 & a_{22}^{(1)} & v_2 & \dots & 0 \\ 0 & v_2 & & & \\ \vdots & \vdots & & \mathrm{X} & \\ 0 & 0 & & & \end{matrix},$$

where $v_2 = \pm \|x_2\|_2$. Continue by forming $H_3 A_2 H_3, \dots, H_{n-2} A_{n-3} H_{n-2}$ to arrive at a symmetric tridiagonal matrix T. The product $P = H_{n-2} H_{n-3} \dots H_2 H_1$ is an orthogonal matrix such that $T = PAP$.

An example using a 5×5 matrix will help in understanding the process. The example uses the MATLAB function

```
H = hsym(A,i)
```

in the software distribution. It builds the $n \times n$ Householder matrix with the embedded $(n-i) \times (n-i)$ Householder submatrix used to zero out elements $(i+2,i), \dots, (n,i)$.

Example 19.3. Let

$$A = \begin{bmatrix} 1 & 2 & -1 & 3 & 5 \\ 2 & -1 & -2 & 1 & 0 \\ -1 & -2 & 1 & -7 & 2 \\ 3 & 1 & -7 & 2 & -1 \\ 5 & 0 & 2 & -1 & 1 \end{bmatrix}.$$

```
>> H1 = hsym(A,1)

H1 =
            1            0            0            0            0
            0     -0.32026      0.16013     -0.48038     -0.80064
            0      0.16013      0.98058     0.058264     0.097106
            0     -0.48038     0.058264      0.82521     -0.29132
            0     -0.80064     0.097106     -0.29132      0.51447

>> H1*A

ans =
            1            2           -1            3            5
       -6.245     -0.48038       2.5621      -1.6013            0
            0       -2.063      0.44669      -6.6845            2
            0       1.1891      -5.3401       1.0535           -1
            0      0.31511       4.7666      -2.5775            1

>> A1 = H1*A*H1
```

```
A1 =
            1       -6.245            0            0            0
       -6.245       1.3333       2.3421     -0.94135       1.0999
            0       2.3421    -0.087586      -5.0817       4.6714
            0     -0.94135      -5.0817       0.2783      -2.2919
            0       1.0999       4.6714      -2.2919       1.4759

>> H2 = hsym(A1,2)

H2 =
            1            0            0            0            0
            0            1            0            0            0
            0            0     -0.85061      0.34189     -0.39947
            0            0      0.34189      0.93684     0.073798
            0            0     -0.39947     0.073798      0.91377

>> A2 = H2*A1*H2
    ...
A3 =
            1       -6.245            0            0            0
       -6.245       1.3333      -2.7534            0            0
            0      -2.7534       6.9609      -4.5858            0
            0            0      -4.5858      -3.9358      0.32451
            0            0            0      0.32451      -1.3584
>> P = H3*H2*H1;
>> P*A*P'

ans =
            1       -6.245            0            0            0
       -6.245       1.3333      -2.7534            0            0
            0      -2.7534       6.9609      -4.5858            0
            0            0      -4.5858      -3.9358      0.32451
            0            0            0      0.32451      -1.3584
```

■

19.3.1 Tridiagonal Reduction of a Symmetric Matrix

While the explanation provided shows how the algorithm works, the computation is not efficient. In particular, it should not be necessary to construct the entire matrix, H_i having the $i \times i$ identity matrix in the upper left-hand corner. The product $H_i A_{i-1} H_i$ should be done implicitly and take advantage of symmetry. Recall that a Householder matrix is of the form $H_u = I - \beta u u^{\mathrm{T}}$, where $\beta = \frac{2}{u^{\mathrm{T}} u}$. Since H_u is symmetric, $H_u A H_u^{\mathrm{T}} = H_u A H_u$, and

$$\begin{aligned} H_u A H_u &= \left(I - \beta u u^{\mathrm{T}}\right) A \left(I - \beta u u^{\mathrm{T}}\right) \\ &= A - \beta A u u^{\mathrm{T}} - u \beta u^{\mathrm{T}} A + \beta u u^{\mathrm{T}} \beta A u u^{\mathrm{T}} \end{aligned}$$

Define $p = \beta A u$, so

$$H_u A H_u = A - p u^{\mathrm{T}} - u p^{\mathrm{T}} + \beta u u^{\mathrm{T}} p u^{\mathrm{T}}.$$

Noting that $u^{\mathrm{T}} p$ is a real number, define $K = \frac{\beta u^{\mathrm{T}} p}{2}$ so that

$$H_u A H_u = A - p u^{\mathrm{T}} - u p^{\mathrm{T}} + 2 K u u^{\mathrm{T}} = A - (p - K u) u^{\mathrm{T}} - u \left(p^{\mathrm{T}} - K u^{\mathrm{T}}\right).$$

Define $q = p - K u$, and we have the final result

$$p = \beta A u \tag{19.10}$$

$$K = \frac{\beta u^{\mathrm{T}} p}{2}, \tag{19.11}$$

$$q = p - K u \tag{19.12}$$

$$H_u A H_u = A - q u^{\mathrm{T}} - u q^{\mathrm{T}}, \tag{19.13}$$

Equation 19.13 enables a faster computation for the transformation.

Our algorithm for reduction to a tridiagonal matrix replaces A by the tridiagonal matrix, so the remaining discussion will assume we are dealing with matrix A as it changes. As i varies from 1 to $n-2$, we know that after the product H_iAH_i

- the entries at indices $(i+2,i)\ldots(n,i)$ and $(i,i+2)\ldots(i,n)$ have value 0.
- $a_{i+1,i} = a_{i,i+1} = \pm \left\| \begin{bmatrix} a_{i+1,i} & a_{i+2,i} & \ldots & a_{n-1,i} & a_{ni} \end{bmatrix}^{\mathrm{T}} \right\|_2$.

Looking at Example 19.3, we see that each iteration H_iAH_i affects only the submatrix $A(i+1:n, i+1:n)$, so there is no need to deal with any other portion of the matrix. Our strategy is to assign the values $a_{i+1,i}$, $a_{i,i+1}$ and then perform the product

$$A(i+1:n, i+1:n) = A(i+1:n, i+1:n) - uq^{\mathrm{T}} - qu^{\mathrm{T}}$$

using Equation 19.13. After completion of the $n-2$ iterations, the algorithm places zeros above and below the three diagonals to eliminate small entries remaining due to roundoff error. If the orthogonal transforming matrix P is required, compute $P = H_{n-2}H_{n-3}\ldots H_2H_1I$ using the formula

$$P(2:n, i+1:n) = P(2:n, i+1:n) - \beta P(2:n, i+1:n)\, uu^{\mathrm{T}}$$

that affects only the portion of P that changes with each iteration. We put all this together in Algorithm 19.2.

Algorithm 19.2 Orthogonal Reduction of a Symmetric Matrix to Tridiagonal Form

```
function TRIREDUCE(A)
   % Compute a tridiagonal matrix orthogonally similar to the
   % symmetric matrix A.
   % T=trireduce(A) - assigns to T a symmetric tridiagonal matrix
   % orthogonally similar to A.
   % [P T]=trireduce(A) - assigns to T a symmetric tridiagonal
   % matrix orthogonally similar to A and an orthogonal matrix
   % P such that P^T AP = T.

   if P required then
      P=I
   end if
   for i=1:n-2 do
      [ u  β ] = houseparms(A(i+1:n, i))
      p = βA(i+1:n, i+1:n) u
      K = βu^T p / 2
      q = p - Ku
      a_{i+1,i} = ± ||A(i+1:n, i)||_2
      a_{i,i+1} = a_{i+1,i}
      A(i+1:n, i+1:n) = A(i+1:n, i+1:n) - uq^T - qu^T
      if P required then
         P(2:n, i+1:n) = P(2:n, i+1:n) - βP(2:n, i+1:n) uu^T
      end if
   end for
   Clear all elements above and below the diagonals.
   if P required then
      return [ P  A ]
   else
      return A
   end if
end function
```

NLALIB: The function `trireduce` implements Algorithm 19.2.

Example 19.4. Example 19.2 dealt with the matrix EIGVECSYMCOND. In this example, we apply `trireduce` to that matrix and compute $\left\| PTP^{\mathrm{T}} - EIGVECSYMCOND \right\|_2$. In addition, MATLAB code compares the eigenvalues obtained from A and T.

```
>> [P T] = trireduce(EIGVECSYMCOND);
>> norm(P*T*P'-EIGVECSYMCOND)

ans =

    1.955492905735752e-10

>> EA = sort(eigsymj(EIGVECSYMCOND,1.0e-10,20));
>> ET = sort(eigsymj(T,1.0e-10,20));
>> norm(EA - ET)

ans =
    5.799990492988926e-10
```

■

Efficiency

The reduction to tridiagonal form requires $\frac{4}{3}n^3$ flops, and the algorithm is stable [2, pp. 458-459].

19.3.2 Orthogonal Transformation to a Diagonal Matrix

The final step of the symmetric shifted QR iteration is the orthogonal reduction of the tridiagonal matrix to a diagonal matrix of eigenvalues. In Section 18.6.1, we discussed a shift strategy that involves computing the QR decomposition of the matrix $T_k - \sigma I = Q_k R_k$ and then forming $T_{k+1} = R_k Q_k + \sigma I$. We need to show that $R_k Q_k$ is tridiagonal and symmetric.

$T = T_0$ is initially symmetric and tridiagonal. Now,

$$T_k - \sigma I = Q_k R_k,$$

so

$$Q_k^{\mathrm{T}} \left(T_k - \sigma I \right) Q_k = R_k Q_k.$$

$R_k Q_k$ is upper Hessenberg by Theorem 18.4, and thus is tridiagonal. We have

$$\begin{aligned} \left(R_k Q_k \right)^{\mathrm{T}} &= \left(Q_k^{\mathrm{T}} \left(T_k - \sigma I \right) Q_k \right)^{\mathrm{T}} \\ &= Q_k^{\mathrm{T}} \left(T_k - \sigma I \right)^{\mathrm{T}} Q_k \\ &= Q_k^{\mathrm{T}} \left(T_k - \sigma I \right) Q_k \\ &= R_k Q_k, \end{aligned}$$

and therefore T_{k+1} is symmetric.

As discussed in Section 18.6.1, $\sigma = h_{kk}$, the Rayleigh quotient shift, is often used with a nonsymmetric matrix. For a symmetric matrix, σ is usually chosen using the *Wilkinson shift*, which is a properly chosen eigenvalue of the 2×2 lower right submatrix,

$$W_k = \begin{bmatrix} t_{k-1,k-1} & t_{k,k-1} \\ t_{k,k-1} & t_{kk} \end{bmatrix}.$$

The eigenvalues of W_k are the roots of the characteristic polynomial

$$\left(t_{k-1,k-1} - \lambda \right) \left(t_{kk} - \lambda \right) - t_{k,k-1}^2.$$

Using the quadratic formula, we have

$$\begin{aligned} \lambda &= \frac{\left(t_{k-1,k-1} + t_{kk} \right) \pm \sqrt{\left(t_{k-1,k-1} + t_{kk} \right)^2 - 4 \left(t_{k-1,k-1} t_{kk} - t_{k,k-1}^2 \right)}}{2} \\ &= \frac{\left(t_{k-1,k-1} + t_{kk} \right) \pm \sqrt{\left(t_{k-1,k-1} - t_{kk} \right)^2 + 4 t_{k,k-1}^2}}{2} \end{aligned}$$

$$
\begin{aligned}
&= \frac{2t_{kk} + \left(t_{k-1,k-1} - t_{kk}\right) \pm 2\sqrt{\left(\frac{t_{k-1,k-1}-t_{kk}}{2}\right)^2 + t_{k,k-1}^2}}{2} \\
&= t_{kk} + \left(\frac{t_{k-1,k-1} - t_{kk}}{2}\right) \pm \sqrt{\left(\frac{t_{k-1,k-1} - t_{kk}}{2}\right)^2 + t_{k,k-1}^2} \\
&= t_{kk} + r \pm \sqrt{r^2 + t_{k,k-1}^2},
\end{aligned}
$$

where $r = \dfrac{t_{k-1,k-1} - t_{kk}}{2}$. We want to choose the eigenvalue closest to t_{kk} as the shift, so if $r < 0$, choose "+", and if $r > 0$, choose "−". If $r = 0$, choose "−". Using the function

$$
\text{sign}\,(x) = \begin{cases} 1, & x > 0 \\ 0, & x = 0 \\ -1, & x < 0 \end{cases},
$$

the shift is

$$
\begin{cases} \sigma = t_{kk} + r - \text{sign}\,(r)\sqrt{r^2 + t_{k,k-1}^2}, & r \neq 0 \\ \sigma = t_{kk} - \sqrt{t_{k,k-1}^2}, & r = 0 \end{cases} \tag{19.14}
$$

The reason for using the Wilkinson shift has to do with the guarantee of convergence. The choice of $\sigma = t_{kk}$ can lead to divergence of the iteration, but convergence when the using the Wilkinson shift is guaranteed to be at least linear, but in most cases is cubic [61].

The MATLAB function `eigsymqr` computes the eigenvalues and, optionally, the eigenvectors of a real symmetric matrix. After applying `trireduce`, the eigenvalue computation in `eigsymqr` method is identical to that for a nonsymmetric matrix, except that the shift is given by Equation 19.14. The eigenvector computation is done by maintaining the orthogonal matrices involved in the transformations.

Example 19.5. The matrix SYMEIGTST in the book software distribution is a 21×21 symmetric matrix. The matrix is orthogonally similar to the famous symmetric tridiagonal 21×21 Wilkinson matrix used for testing eigenvalue computation. The MATLAB documentation for wilkinson(21) specifies that this matrix is a symmetric with pairs of nearly, but not exactly, equal eigenvalues. Its two largest eigenvalues are both about 10.746; they agree to 14, but not to 15, decimal places. The example uses `eigsymqr` to compute the eigenvalues.

```
>> load SYMEIGTST
eigsymqr(SYMEIGTST,1.0e-14,50)

ans =
  -1.125441522119974
   0.253805817096685
   0.947534367529301
          ...
   5.000244425001915
   6.000234031584169
   6.000217522257100
          ...
  10.746194182903356
  10.746194182903350
```

■

19.4 THE SYMMETRIC FRANCIS ALGORITHM

As noted in Section 18.8.1, the Francis algorithm has been a staple in eigenvalue computation for many years. The double-shift version will compute all the eigenvalues and eigenvectors of a general real matrix, and will find all the complex eigenvalues without using complex arithmetic. Since the eigenvalues of a symmetric matrix are real, the double-shift version is not necessary. We will present the single-shift version, called the *Francis iteration of degree one*.

The first step is to apply orthogonal similarity transformations that reduce the matrix A to a symmetric tridiagonal matrix, $T = Q^{\mathrm{T}}AQ$. The algorithm now executes the single-shift bulge chase discussed in Section 18.8.1. After k iterations, the sequence approximates a diagonal matrix.

$$(J_k J_{k-1} \dots J_2 J_1)\, T \left(J_1^{\mathrm{T}} J_2^{\mathrm{T}} \dots J_k^{\mathrm{T}}\right) = D. \tag{19.15}$$

If we let $P = J_k J_{k-1} \dots J_2 J_1$ in Equation 19.15 and use the fact that $T = Q^{\mathrm{T}}AQ$ there results

$$PQ^{\mathrm{T}}AQP^{\mathrm{T}} = D.$$

The orthogonal matrix PQ^{T} approximates the matrix of eigenvectors, and D approximates the corresponding eigenvalues.

The MATLAB function `eigsymb` finds the eigenvalues and, optionally, the eigenvectors of a symmetric matrix. The function `chase(T)` discussed in Section 18.8.1 performs the bulge chase. Like `eigsymqr`, `eigsymb` uses deflation so it only works on submatrices. The function just replaces the explicit single shift by the implicit single shift, so we will not present the algorithm.

Example 19.6. The Poisson matrix is a symmetric block tridiagonal sparse matrix of order n^2 resulting from discretizing Poisson's equation with the 5-point central difference approximation on an n-by-n mesh. We will discuss the equation in Section 20.5. This examples computes the 100×100 Poisson matrix, computes its eigenvalues and corresponding eigenvectors using eigsymb, and checks the result.

```
>> P = gallery('poisson',10);
>> P = full(P);  % convert from sparse to full matrix format
>> [V,D] = eigsymb(P,1.0e-14,100);
>> norm(V'*P*V-D)

ans =

     8.127291292857505e-14
```

■

19.4.1 Theoretical Overview and Efficiency

The flop count for the Francis algorithm is the sum of the counts for reduction to tridiagonal form and reduction to a diagonal matrix. These counts are as follows [2, pp. 458-464]:

- The cost of computing just the eigenvalues of A is approximately $\frac{4}{3}n^3$ flops.
- Finding all the eigenvalues and eigenvectors costs approximately $9n^3$ flops.

This is the same order of magnitude as the simpler-shifted Hessenberg QR iteration in presented in Section 18.7. However, eigenvectors are readily found, and the number of multiplications needed is smaller.

19.5 THE BISECTION METHOD

After a symmetric matrix has been reduced to tridiagonal form, T, the bisection method can be used to compute a subset of the eigenvalues; for instance, eigenvalue i of n, the largest 15% of the eigenvalues or the smallest 5. If important eigenvalues lie in an interval $a \leq \lambda \leq b$, the algorithm can find all of them. If needed, inverse iteration will compute the corresponding eigenvector(s).

The bisection algorithm for computing roots of a nonlinear function is a standard topic in any numerical analysis or numerical methods course (see Ref. [33, pp. 61-64]). If f is a continuous real-valued function on an interval left $\leq x \leq$ right with f (left) and f (right) having opposite signs (f (left) f (right) < 0), then there must be a value r in the interval such that $f(r) = 0$. r is a *root* of f. Let mid $= \frac{\text{left}+\text{right}}{2}$ be the middle point of the interval. After evaluating $v = f$ (left) f (mid), either you have found a root or know in which of the intervals (left, mid) or (mid, right) of length $\frac{\text{right}-\text{left}}{2}$ the root lies

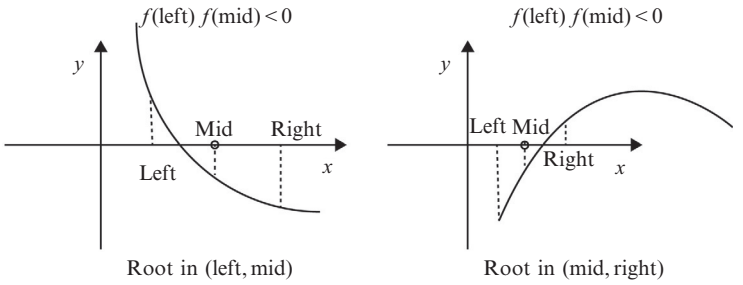

FIGURE 19.1 Bisection.

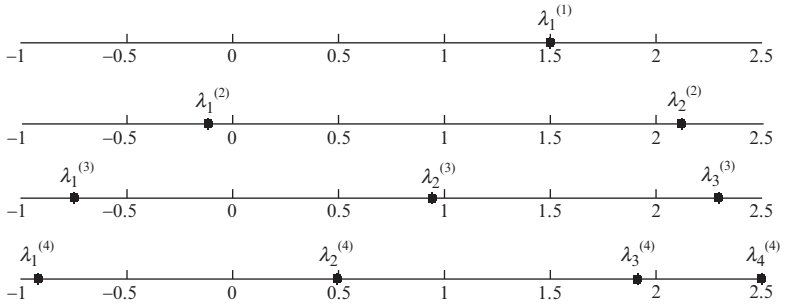

FIGURE 19.2 Interlacing.

(Figure 19.1). Move to the new interval and repeat the process until finding a root or isolating the root in a very small interval.

Outline of the Bisection Method

a. $v = 0$: mid is a root.
b. $v > 0$: root is in the interval mid $< r <$ right.
c. $v < 0$: root is in the interval left $< r <$ mid.

For any $n \times n$ matrix with real distinct eigenvalues, let $f(\mu) = \det(A - \mu I)$, and find two values $\mu_1 =$ left and $\mu_r =$ right for which

$$f\,(\text{left})\, f\,(\text{right}) < 0,$$

and apply the bisection algorithm until approximating a root. Of course, the root is an eigenvalue of A. This does not violate the fact that polynomial root finding is unstable, since the algorithm deals only with the polynomial value and never computes any coefficients. There are no problems like evaluation of the quadratic formula (see Section 8.4.2) or perturbing a particular coefficient of a polynomial (see Section 10.3.1). A determinant of an arbitrary matrix can be computed stably using Gaussian elimination with partial pivoting ($PA = LU \Rightarrow A = P^{\mathrm{T}}LU \Rightarrow \det(A) = (-1)^r u_{11}u_{22}\ldots u_{nn}$, where r is the number of row exchanges). What makes the use of bisection extremely effective when applied to a symmetric tridiagonal matrix is some extraordinary properties of its eigenvalues and $f\,(\mu)$.

Let T be an unreduced symmetric tridiagonal matrix (no lower diagonal element is zero). At the end of the section, we will discuss the method for a matrix with a zero on its lower diagonal. Assume

$$T = \begin{bmatrix} a_1 & b_1 & & & \\ b_1 & a_2 & b_2 & & \\ & b_2 & a_3 & \ddots & \\ & & \ddots & \ddots & b_{n-1} \\ & & & b_{n-1} & a_n \end{bmatrix}, \quad b_i \neq 0.$$

Let $T^{(1)}$ be the 1×1 matrix $[a_1]$, $T^{(2)}$ be the 2×2 matrix $\begin{bmatrix} a_1 & b_1 \\ b_1 & a_2 \end{bmatrix}$ and, in general, $T^{(k)} = T(1:k, 1:k)$ be the upper-left $k \times k$ submatrix of T, $1 \leq k \leq n$. The eigenvalues of $T^{(k)}$ are distinct (Problem 19.1), and assume they are $\lambda_1^{(k)} < \lambda_2^{(k)} < \cdots < \lambda_k^{(k)}$. The essence of the bisection algorithm is that the eigenvalues of two successive matrices $T^{(k)}$ and $T^{(k+1)}$ *strictly interlace* [9, pp. 103-104]. This means that

$$\lambda_i^{(k+1)} < \lambda_i^{(k)} < \lambda_{i+1}^{(k+1)}$$

for $k = 1, 2, \ldots, n-1$ and $i = 1, 2, \ldots, k-1$. This remarkable property enables us to know the precise number of eigenvalues in any interval on the real line.

Example 19.7. Let $T = \begin{bmatrix} \frac{3}{2} & 1 & 0 & 0 \\ 1 & \frac{1}{2} & 1 & 0 \\ 0 & 1 & \frac{1}{2} & 1 \\ 0 & 0 & 1 & \frac{3}{2} \end{bmatrix}$. Figure 19.2 shows the position of eigenvalues for $T^{(1)}$, $T^{(2)}$, $T^{(3)}$, and $T^{(4)} = T$. Note the strict interlacing. ■

In addition to depending on the interlacing property of $T^{(k)}$, the bisection algorithm requires the computation of the characteristic polynomials, $p_k(\mu)$, of $T^{(k)}$ for a specified μ. The characteristic polynomial of $T^{(1)}$ is $p_1(\mu) = a_1 - \mu$, and define $p_0(\mu) = 1$. Assume we know $p_i(\mu), 1 \leq i \leq k-1$ and want to compute

$$p_k(\mu) = \det\left(\begin{bmatrix} a_1 - \mu & b_1 & & & & & \\ b_1 & a_2 - \mu & b_2 & & & & \\ & b_2 & a_3 - \mu & \ddots & & & \\ & & \ddots & \ddots & \ddots & & \\ & & & \ddots & a_{k-1} - \mu & b_{k-1} & \\ & & & & b_{k-1} & a_k - \mu \end{bmatrix}\right).$$

Expand by minors across row k to obtain

$$\begin{aligned} p_k(\mu) &= (-1)^{k+(k-1)} b_{k-1} \det\left(T^{(k-2)} - \mu I\right) + (-1)^{2k} (a_k - \mu) \det\left(T^{(k-1)} - \mu I\right) \\ &= (a_k - \mu) p_{k-1}(\mu) - b_{k-1}^2 p_{k-2}(\mu). \end{aligned}$$

If you are unsure of the result, draw a 5×5 matrix and verify the equation. Putting things together, we have the three term recurrence relation

$$p_k(\mu) = \begin{cases} 1 & k = 0 \\ a_1 - \mu & k = 1 \\ (a_k - \mu) p_{k-1}(\mu) - b_{k-1}^2 p_{k-2}(\mu) & k \geq 2 \end{cases} \tag{19.16}$$

This recurrence relation is known as a *Sturm sequence* and enables the computation of $p_k(\mu)$ without using Gaussian elimination. The interlacing of eigenvalues gives rise to the following extraordinary result [27, pp. 203-208].

Theorem 19.4. *The number, $d(\mu)$, of disagreements in sign between consecutive numbers of the sequence $\{p_0(\mu), p_1(\mu), p_2(\mu), \ldots, p_n(\mu)\}$ is equal to the number of eigenvalues smaller than μ.*

Remark 19.4. If $p_k(\mu) = 0$, define the sign of $p_k(\mu)$ to be the opposite of that for $p_{k-1}(\mu)$. There cannot be two consecutive zero values in the sequence (Problem 19.13).

Example 19.8. Let T be the matrix of Example 19.7. Its characteristic polynomials are

$p_0(\mu)$	$p_1(\mu)$	$p_2(\mu)$	$p_3(\mu)$	$p_4(\mu)$
1	$\frac{3}{2} - \mu$	$\mu^2 - 2\mu - \frac{1}{4}$	$-\mu^3 + \frac{5}{2}\mu^2 + \frac{1}{4}\mu - \frac{13}{8}$	$\mu^4 - 4\mu^3 + \frac{5}{2}\mu^2 + 4\mu - \frac{35}{16}$

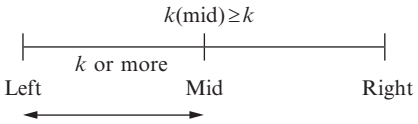

FIGURE 19.3 Bisection method: λ_k located to the left.

1. Let $\mu = 0$, and the sequence is $\left\{ \begin{array}{cccc} 1 & \frac{3}{2} & -\frac{1}{4} & -\frac{13}{8} & -\frac{35}{16} \end{array} \right\}$, with 1 sign change.
2. Let $\mu = 1$, and the sequence is $\left\{ \begin{array}{cccc} 1 & \frac{1}{2} & -\frac{5}{4} & \frac{1}{8} & \frac{21}{16} \end{array} \right\}$ with 2 sign changes.
3. Let $\mu = 3$. The sequence is $\left\{ \begin{array}{cccc} 1 & -\frac{3}{2} & \frac{11}{4} & -\frac{43}{8} & \frac{85}{16} \end{array} \right\}$ with 4 sign changes.

From (1), we see there is one negative eigenvalue, (2) tells us there is an eigenvalue in the range $0 \leq \mu < 1$, and from (3) we conclude that there must be 2 eigenvalues in the interval $1 \leq \mu < 3$.

In fact the eigenvalues are $\left\{ \begin{array}{ccccc} \frac{1}{2} - \sqrt{2} & \frac{1}{2} & \frac{1}{2} + \sqrt{2} & \frac{5}{2} \end{array} \right\}$. ■

Assume that the symmetric tridiagonal matrix T has eigenvalues $\lambda_1 \leq \lambda_2 \leq \cdots \lambda_n$ and that we wish to compute eigenvalue λ_i. The eigenvalue must be isolated in an interval (a, b) before generating a series of smaller and smaller intervals enclosing λ_i. The following lemma gives us starting values for a and b.

Lemma 19.2. *If $\|\cdot\|$ is a subordinate norm and $A \neq 0$, $\rho(A) \leq \|A\|$.*

Proof. Let λ_i be an eigenvalue of A with associated eigenvector v_i. Since $Av_i = \lambda_i v_i$,

$$\|Av_i\| = \|\lambda_i v_i\| = |\lambda_i| \, \|v_i\| ,$$

and

$$\|Av_i\| \leq \|A\| \, \|v_i\| ,$$

so

$$|\lambda_i| \, \|v_i\| \leq \|A\| \, \|v_i\| .$$

Since $\|v_i\| \neq 0$, we have $|\lambda_i| \leq \|A\|$ for all λ_i, and

$$\max_{1 \leq i \leq n} |\lambda_i| = \rho(A) \leq \|A\| .$$ □

Since the infinity norm can be computed quickly, Lemma 19.2 gives us starting values $a = -\|A\|_\infty$, $b = \|A\|_\infty$. We can now outline the bisection method for computing a particular λ_k, $1 \leq k \leq n$.

a. Let left $= -\|A\|_\infty$, right $= \|A\|_\infty$.

b. Compute mid $= \frac{\text{left+right}}{2}$.

c. Compute d (mid), the number of disagreements in sign between consecutive numbers in the sequence

$$p_0(\text{mid}), p_1(\text{mid}), p_2(\text{mid}), \ldots, p_n(\text{mid}),$$

properly handling a case where $p_k(\text{mid}) = 0$.

d. If d (mid) $\geq k$, then λ_k is in the interval [left, mid]. Let right = mid (Figure 19.3).
else
λ_k is in the interval [mid, right]. Let left = mid (Figure 19.4).

e. Repeat steps 2-4 until (right − left) < tol, where tol is an acceptable length for an interval enclosing the root.

We will not give the algorithm using pseudocode. The MATLAB function `bisection` with calling format `bisection(T,k,tol)` in the book software distribution implements the method. If `tol` is not given, it defaults to 1.0×10^{-12}.

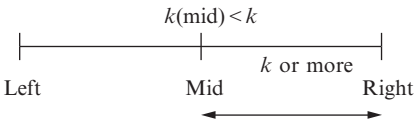

FIGURE 19.4 Bisection method: λ_k located to the right.

Example 19.9. Let A be the matrix of Example 19.7 and perform seven iterations of the bisection method to approximate $\lambda_2 = \frac{1}{2}$.

Iteration	Left	Right	Mid	d (mid)	Action
1	−2.5	2.5	0	1	left = 0
2	0	2.5	1.25	2	right = 1.725
3	0	1.25	0.625	2	right = 0.625
4	0	0.625	0.3125	1	left = 0.3125
5	0.3125	0.625	0.46875	1	left = 0.46875
6	0.46875	0.625	0.546875	2	right = 0.546875
7	0.46875	0.546875	0.507813	2	right = 0.507813

After seven iterations, the eigenvalue is isolated in the interval $0.46875 < \lambda_2 < 0.507813$. If the process completes a total of 16 iterations, $0.499954 < \lambda_2 < 0.500031$, and the approximation for λ_2 is

$$\frac{0.499954 + 0.500031}{2} = 0.49999. \qquad \blacksquare$$

Example 19.10. This example uses the bisection method to compute the two largest eigenvalues of the Wilkinson symmetric 21×21 tridiagonal matrix.

```
>> W = wilkinson(21);
>> lambda20 = bisection(W,20,1.0e-14)

lambda20 =
  10.746194182903317

>> lambda21 = bisection(W,21,1.0e-14)

lambda21 =
  10.746194182903395
```

■

19.5.1 Efficiency

Each evaluation of $\{p_0\,(\text{mid})\,, p_1\,(\text{mid})\,, p_2\,(\lambda\text{mid})\,, \ldots, p_n\,(\text{mid})\}$ costs $O\,(n)$ flops. If tol is the desired size of the subinterval containing the eigenvalue, since each iteration halves the search interval, the number of iterations, k, required is determined by

$$\frac{|\text{right} - \text{left}|}{2^k} < \text{tol}.$$

Thus,

$$k \approx \log_2 |\text{right} - \text{left}| - \log_2 \text{tol}$$

and the algorithm requires $O\,(kn)$ flops. If only a few eigenvalues are required, the bisection method is faster than finding all the eigenvalues using orthogonal similarity reduction to a diagonal matrix.

19.5.2 Matrix A Is Not Unreduced

If λ is an eigenvalue of multiplicity $m > 1$, the bisection algorithm for computing a root will find one occurrence of λ if m is odd (point of inflection) and will fail to find λ if m is even (tangent to horizontal axis) (Figure 19.5). This is not a problem, since the bisection method requires that A be unreduced, and a symmetric unreduced tridiagonal matrix has distinct eigenvalues (Problem 19.1). What happens if there are one or more zeros on the subdiagonal of A? We split the

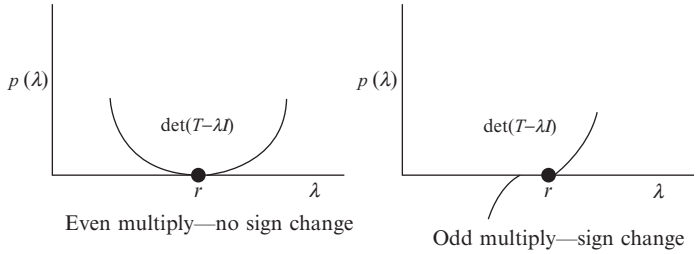

FIGURE 19.5 Bisection and multiple eigenvalues.

problem into finding eigenvalues of the unreduced matrices between the pairs of zeros. Lemma 19.3 shows how to handle the case when the subdiagonal contains one zero.

Lemma 19.3. *Suppose a tridiagonal matrix has the form*

$$T = \begin{bmatrix} a_1 & b_1 & & & & & & & \\ b_1 & a_2 & b_2 & & & & & & \\ & b_2 & \ddots & \ddots & & & & & \\ & & \ddots & a_{i-1} & b_{i-1} & & & & \\ & & & b_{i-1} & a_i & \mathbf{0} & & & \\ & & & & \mathbf{0} & a_{i+1} & b_{i+1} & & \\ & & & & & b_{i+1} & \ddots & & \\ & & & & & & \ddots & & b_{n-1} \\ & & & & & & & b_{n-1} & a_n \end{bmatrix}.$$

Then,

$$\det\left(T - \lambda I^{n\times n}\right) = \det\left(T\left(1:i,\, 1:i\right) - \lambda I^{i\times i}\right) \det\left(T\left(i+1:n,\, i+1:n\right) - \lambda I^{(n-i)\times(n-i)}\right).$$

The proof is left to the problems.

With one zero on the subdiagonal, Lemma 19.3 says to apply bisection to the unreduced $i \times i$ and $(n-i) \times (n-i)$ submatrices and form the union of the eigenvalues.

19.6 THE DIVIDE-AND-CONQUER METHOD

This presentation is a summary of the divide-and-conquer method. For more in-depth coverage of this algorithm, see Refs. [1, pp. 216-228], [19, pp. 359-363], and [26, pp. 229-232].

The recursive algorithm divides a symmetric tridiagonal matrix into submatrices and then applies the same algorithm to the submatrices. We will illustrate the method of splitting the problem into smaller submatrix problems using a 5×5 matrix

$$T = \begin{bmatrix} a_1 & b_1 & 0 & 0 & 0 \\ b_1 & a_2 & b_2 & 0 & 0 \\ 0 & b_2 & a_3 & b_3 & 0 \\ 0 & 0 & b_3 & a_4 & b_4 \\ 0 & 0 & 0 & b_4 & a_5 \end{bmatrix}.$$

Write A as a sum of two matrices as follows:

$$T = \begin{bmatrix} a_1 & b_1 & 0 & 0 & 0 \\ b_1 & a_2 - b_2 & 0 & 0 & 0 \\ 0 & 0 & a_3 - b_2 & b_3 & 0 \\ 0 & 0 & b_3 & a_4 & b_4 \\ 0 & 0 & 0 & b_4 & a_5 \end{bmatrix} + \begin{bmatrix} 0 & 0 & 0 & 0 & 0 \\ 0 & b_2 & b_2 & 0 & 0 \\ 0 & b_2 & b_2 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix}.$$

If we let

$$T_1 = \begin{bmatrix} a_1 & b_1 \\ b_1 & a_2 - b_2 \end{bmatrix}, \quad T_2 = \begin{bmatrix} a_3 - b_2 & b_3 & 0 \\ b_3 & a_4 & b_4 \\ 0 & b_4 & a_5 \end{bmatrix},$$

and

$$H = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 \\ 0 & b_2 & b_2 & 0 & 0 \\ 0 & b_2 & b_2 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix},$$

then

$$T = \begin{bmatrix} T_1 & \\ & T_2 \end{bmatrix} + H.$$

$T_1^{2\times 2}$ and $T_2^{3\times 3}$ are symmetric tridiagonal matrices, and $H^{5\times 5}$ has rank one. The rank-one matrix can be written more simply as $H = b_2 v v^{\mathrm{T}}$, where

$$v = \begin{bmatrix} 0 \\ 1 \\ 1 \\ 0 \\ 0 \end{bmatrix}.$$

In the $n \times n$ case, write T as the sum of a 2×2 block symmetric tridiagonal matrix and a rank one matrix, known as a *rank-one correction*.

$$T = \begin{array}{|c|c|}\hline T_1^{k\times k} & \\ \hline & T_2^{(n-k)\times(n-k)} \\ \hline \end{array} + \begin{array}{|c|c|}\hline & \\ \hline t_k & t_k \\ \hline t_k & t_k \\ \hline \end{array}$$

Note that $T_1(k,k) = a_k - b_k$ and $T_2(1,1) = a_{k+1} - b_k$, and the rank-one correction matrix can be written as $t_k v v^{\mathrm{T}}$, where

$$v = \begin{bmatrix} 0 & 0 & \ldots & 1 & 1 & 0 & \ldots & 0 \end{bmatrix}^{\mathrm{T}}.$$

The two entries of 1 are at indices k and $k+1$.

Suppose the divide-and-conquer algorithm is named dconquer and returns the eigenvalues and eigenvectors of a symmetric tridiagonal matrix using the format

[V, D] = dconquer(T).

The algorithm dconquer must be able to take any symmetric tridiagonal matrix T, split it into the sum of a 2×2 block symmetric tridiagonal matrix and a rank-one correction matrix and return the eigenvalues and eigenvectors of T. Here is how dconquer must function. Choose $k = \left\lfloor \frac{n}{2} \right\rfloor$ and form

$$T = \begin{bmatrix} T_1 & \\ & T_2 \end{bmatrix} + t_k v v^{\mathrm{T}}.$$

Now compute

$$\begin{bmatrix} V_1, & D_1 \end{bmatrix} = \text{dconquer}(T_1), \quad \begin{bmatrix} V_2, & D_2 \end{bmatrix} = \text{dconquer}(T_2),$$

and put $\begin{bmatrix} V_1, & D_1 \end{bmatrix}$, $\begin{bmatrix} V_2, & D_2 \end{bmatrix}$, and $t_k v v^{\mathrm{T}}$ together to obtain the matrices V and D. Each of the recursive calls dconquer (T_1) and dconquer (T_2) must divide their respective matrix as described and compute eigenvalues and eigenvectors for them. Continue this process until arriving at a set of 1×1 eigenvalue problems, each having a rank-one correction. This is called the *stopping condition*. These are easily solved, and a series of function returns solves all the problems encountered on the way to the stopping condition. Returning from the first recursive call gives the eigenvalues and eigenvectors of the initial matrix.

How do we find the eigenvalues of T from T_1, T_2 and the rank-one correction? By the spectral theorem,

$$T_1 = P_1 D_1 P_1^{\mathrm{T}}, \quad T_2 = P_2 D_2 P_2^{\mathrm{T}},$$

where P_1, P_2 are orthogonal matrices of eigenvectors, and D_1, D_2 are diagonal matrices of corresponding eigenvalues. Then,

$$\begin{aligned} T &= \begin{bmatrix} T_1 & 0 \\ 0 & T_2 \end{bmatrix} + t_k v v^{\mathrm{T}} = \begin{bmatrix} P_1 D_1 P_1^{\mathrm{T}} & 0 \\ 0 & P_2 D_2 P_2^{\mathrm{T}} \end{bmatrix} + t_k v v^{\mathrm{T}}. \\ \text{Let } D &= \begin{bmatrix} D_1 & 0 \\ 0 & D_2 \end{bmatrix}, u = \begin{bmatrix} P_1^{\mathrm{T}} & 0 \\ 0 & P_2^{\mathrm{T}} \end{bmatrix} v \text{ and form} \\ & \begin{bmatrix} P_1 & 0 \\ 0 & P_2 \end{bmatrix} \left(\begin{bmatrix} D_1 & 0 \\ 0 & D_2 \end{bmatrix} + t_k u u^{\mathrm{T}} \right) \begin{bmatrix} P_1^{\mathrm{T}} & 0 \\ 0 & P_2^{\mathrm{T}} \end{bmatrix} = \begin{bmatrix} P_1 & 0 \\ 0 & P_2 \end{bmatrix} \begin{bmatrix} D_1 & 0 \\ 0 & D_2 \end{bmatrix} \begin{bmatrix} P_1^{\mathrm{T}} & 0 \\ 0 & P_2^{\mathrm{T}} \end{bmatrix} \\ & + t_k \begin{bmatrix} P_1 & 0 \\ 0 & P_2 \end{bmatrix} \begin{bmatrix} P_1^{\mathrm{T}} & 0 \\ 0 & P_2^{\mathrm{T}} \end{bmatrix} v v^{\mathrm{T}} \begin{bmatrix} P_1 & 0 \\ 0 & P_2 \end{bmatrix} \begin{bmatrix} P_1^{\mathrm{T}} & 0 \\ 0 & P_2^{\mathrm{T}} \end{bmatrix} \\ &= \begin{bmatrix} P_1 D_1 P_1^{\mathrm{T}} & 0 \\ 0 & P_2 D_2 P_2^{\mathrm{T}} \end{bmatrix} + t_k v v^{\mathrm{T}} = T \end{aligned}$$

We have shown that T is similar to the matrix

$$D + t_k u u^{\mathrm{T}},$$

and so it has the same eigenvalues as T. It can be shown [1, p. 218] that its eigenvalues are roots of the function

$$f(\lambda) = 1 + t_k \sum_{i=1}^{n} \frac{u_i^2}{d_i - \lambda}. \tag{19.17}$$

The equation $f(\lambda) = 0$ is known as the *secular equation*, and finding the roots of f accurately is not an easy problem. Figure 19.6 is a graph of f for particular values of t_k, u, and d, where $d = \mathrm{diag}(D)$. It would seem reasonable to use Newton's method [33, pp. 66-71] to compute the roots, which lie between the singularities $\lambda = d_i$, called the *poles*. It is possible that the first iteration of Newton's method will take an initial approximation $\overline{\lambda}_0$ and produce a very large value $\overline{\lambda}_1$. Using the classical Newton's method can cause the algorithm to become unstable (Problem 19.28). The solution is to approximate

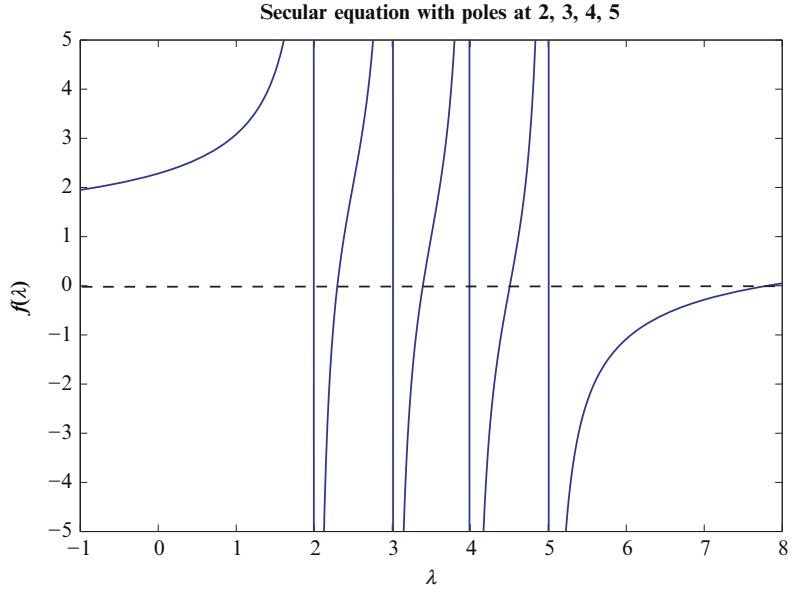

FIGURE 19.6 Secular equation.

$f(\lambda)$, $d_i < \lambda < d_{i+1}$ by another function $h(\lambda)$ that makes the root computation stable. The interested reader should see Ref. [1, pp. 221-223] for the details.

The algorithm can compute an eigenvector from its associated eigenvalue using only $O(n)$ flops, The computation of eigenvectors for the divide-and-conquer algorithm will not be discussed. See Ref. [1, pp. 224-226] for the technique used.

The cost of finding the eigenvalues is $O\left(n^2\right)$, the same as for the Francis algorithm. The computation of an eigenvector using the Francis method costs $O\left(n^2\right)$ flops, as opposed to $O(n)$ flops for divide-and-conquer. We stated in the introduction to this chapter that this algorithm is more than twice as fast as the *QR* algorithm if both eigenvalues and eigenvectors of a symmetric tridiagonal matrix are required.

19.6.1 Using dconquer

Implementing dconquer using a MATLAB function is difficult. It is possible to write a function, say *myMEX*, in the programming language C in such a way that the function can be called from MATLAB. The function *myMEX* must written so it conforms with what is termed the MEX interface and must be compiled using the MATLAB command `mex` (see Ref. [20]). This interface provides access to the input and output parameters when the function is called from MATLAB. Normally, the function serves an interface to a Fortran or C machine code library. For instance, LAPACK [62] is a set of functions written in Fortran that provide methods for solving systems of linear equations, least-squares solutions of linear systems of equations, eigenvalue problems, and executing various factorizations. One of the LAPACK functions, `dsyevd`, transforms A to a symmetric tridiagonal matrix, and then applies the divide-and-conquer algorithm to compute the eigenvalues and corresponding eigenvectors. It will compute the eigenvalues only, but if we want only eigenvalues there is no advantage to the divide-and-conquer algorithm.

Change into the directory "divide-and-conquer" and then into a subdirectory for your computer architecture, "windows" or "OSX_linux." There you will find a function, `dconquer.c`, that uses the MEX interface to call `dsyevd`. Open the file reame.pdf for instructions concerning compiling dconquer.c. Applying `mex` produces a file of the form "dconquer.mexw64" if the operating system is 64-bit Windows, or "dconquer.mexmaci64" for 64-bit OS X. Call `dconquer` using the format

```
[V, D] = dconquer(A);
```

Example 19.11. The following MATLAB sequence builds a 1000×1000 symmetric matrix, computes its eigenvalues and eigenvectors using `dconqer`, and times its execution. As evidence of accuracy, the code computes $\left\| VDV^{\mathrm{T}} - A \right\|_2$ and $\left\| V^{\mathrm{T}}V - I \right\|_2$.

```
>> A = randi([-1000000 1000000],1000,1000);
>> A = A + A';

>> tic;[Vdconquer Ddconquer] = dconquer(A);toc;
Elapsed time is 0.272290 seconds.
>> norm(Vdconquer*Ddconquer*Vdconquer'-A)

ans =
   3.0434e-07

>> norm(Vdconquer'*Vdconquer - eye(1000))

ans =
   8.7754e-15
```

■

Remark 19.5. In general, MEX is complicated and should be used sparingly. We applied it here to call a specific method whose complexity made it difficult to implement in an m-file.

19.7 CHAPTER SUMMARY

The Spectral Theorem and Properties of a Symmetric Matrix

At this point in the book, we have a great deal of machinery in place and have discussed the accurate computation of eigenvalues. We also are familiar with the Schur's triangularization theorem from Section 18.7, which says that any $n \times n$ matrix A, even a singular one, can be factored as $A = PTP^{\mathrm{T}}$, where P is an orthogonal matrix, and T is upper triangular. We use it to prove the spectral theorem for symmetric matrices, a result we have used without proof until this chapter.

This section also reviews some fundamental facts about real symmetric matrices and shows that the condition number of each eigenvalue is 1 (perfect). Unfortunately, this is not true for eigenvectors. It is possible for a symmetric matrix to have ill-conditioned eigenvectors.

The Jacobi Method

The Jacobi method directly reduces a symmetric matrix to diagonal form without an initial conversion to tridiagonal form. It works by zeroing out pairs of equal entries, a_{ij}, a_{ji} off the diagonal until the off $(A) = \sqrt{\|A\|_F^2 - \sum_{i=1}^n a_{ii}^2}$ is sufficiently small. We prove that the rate of convergence is linear. However, it can be shown that the average rate of convergence (asymptotic rate) is quadratic.

The Symmetric *QR* Iteration Method

The first step of this algorithm is the orthogonal similarity transformation of A to tridiagonal matrix T using Householder reflections. The algorithm presented takes advantage of matrix symmetry. The final step is the application of the *QR* iteration with the Wilkinson shift that reduces T to a diagonal matrix. As with a nonsymmetric matrix, Givens rotations are used to find the *QR* decomposition of each shifted matrix. This is an $O\left(n^3\right)$ algorithm.

The Symmetric Francis Algorithm

The single-shift Francis algorithm is the method of choice for computing the eigenvalues and associated eigenvectors of a symmetric matrix. The first phase is reduction to tridiagonal form, as with the symmetric *QR* iteration. The reduction to diagonal form uses orthogonal similarity transformations produced by Givens rotations applied to a Wilkinson-shifted matrix, and the *QR* algorithm is not used directly. Using deflation, iterations chase a bulge from the top off the bottom of the matrix until the current diagonal entry is sufficiently close to an eigenvalue. The method is still $O\left(n^3\right)$ but generally performs better than the symmetric *QR* iteration.

The Bisection Method

This algorithm is very different from the other algorithms we have discussed. Finding the roots of a nonlinear function is covered in a numerical analysis or numerical methods course. One of the methods, bisection, can be applied to compute the eigenvalues of a matrix, which are roots of the nonlinear function $p(\lambda) = \det(A - \lambda I)$, the characteristic polynomial of A. Using Gaussian elimination with partial pivoting, $p(\lambda)$ can be computed in a stable fashion. For a general matrix $A \in \mathbb{R}^{n\times n}$, this method of computing eigenvalues cannot compete with the methods we discussed in Chapter 18. For a symmetric matrix, upper Hessenberg form is a symmetric tridiagonal matrix. After transforming A to such a matrix T, let $T^{(k)}$ be the submatrix $T(1:k, 1:k)$ of T, and $\{p_0, p_1, p_2, \ldots, p_n\}$ be the characteristic polynomials of $T^{(k)}$, $p_0(\lambda) = 1$. The characteristic polynomials are evaluated by a simple recurrence relation. The eigenvalues of $T^{(k)}$ and $T^{(k+1)}$ strictly interlace, which means that $\lambda_i^{(k+1)} < \lambda_i^{(k)} < \lambda_{i+1}^{(k+1)}$. In turn, this property can be used to prove that the number of sign changes between consecutive numbers of the sequence $\{p_0(\lambda), p_1(\lambda), p_2(\lambda), \ldots, p_n(\lambda)\}$ is equal to the number of eigenvalues smaller than λ. This remarkable property enables accurate evaluation of the eigenvalues using the bisection technique.

The Divide-and-Conquer Method

The recursive divide-and-conquer is the fastest algorithm for computing both eigenvalues and eigenvectors of a symmetric matrix A. Like the *QR* algorithm, it first transforms the A into a tridiagonal matrix T. The computation then proceeds by writing T in the form

$$T = \begin{bmatrix} T_1 & \\ & T_2 \end{bmatrix} + t_k v v^{\mathrm{T}}.$$

T is the sum of a 2×2 block symmetric tridiagonal matrix and a rank-one correction. Using recursion, the algorithm finds the eigenvalues and eigenvectors of T. During the computation, the secular equation $f(\lambda) = 1 + t_k \sum_{i=1}^n \frac{u_i^2}{d_i - \lambda}$ must be solved. This secular function has singularities (poles) at the diagonal entries, d_i, of T. Rather than using Newton's method, the best approach to finding the eigenvalues located between poles is to find the roots of a function that approximates f.

19.8 PROBLEMS

In these problems, the term "by hand" means that you must show your work step by step. You can use MATLAB. For instance, if you are required to form $J(1,2,c,s)AJ(1,2,c,s)^{\mathrm{T}}$ with a 4×4 matrix do this:

```
[c, s] = givensparms(A(1,1),A(2,1));
J = eye(4);
J(1,1) = c, J(2,2) = c;
J(2,1) = -s, J(1,2) = s;
A1 = J*A*J';
```

19.1 Let A be an $n \times n$ symmetric tridiagonal matrix with its sub- and superdiagonals nonzero. Prove that the eigenvalues of A are distinct by answering parts (a)-(e).

a. If λ is an eigenvalue, show that the rank of $E = A - \lambda I$ is at most $n - 1$.
b. Consider the upper triangular $(n-1) \times n$ submatrix $E(2:n, 1:n)$. Show that E has rank $n - 1$.
c. Show that rank (E) = rank $(A - \lambda I) = n - 1$.
d. Show that the null space of E is spanned by an eigenvector corresponding to λ.
e. Prove that the symmetry of A implies that λ must be distinct.

19.2 If A is symmetric, show that trace $\left(A^2\right) = \sum_{i=1}^{n} \sum_{j=1}^{n} a_{ij}^2$.

19.3 The QR iteration with the Wilkinson shift and the symmetric Francis algorithm both begin by zeroing out $T(2,1)$ using an orthogonal similarity transformation. Show that the first column is the same for both algorithms.

19.4 For the matrix $T = \begin{bmatrix} 1 & 3 & 0 \\ 3 & 1 & 4 \\ 0 & 4 & 2 \end{bmatrix}$, execute the QR iteration three times using the Wilkinson shift to estimate the eigenvalue 6.3548. Just show the values of σ and $T = RQ + \sigma I$ for each iteration.

19.5 For the matrix $T = \begin{bmatrix} -2 & 4 & 0 & 0 \\ 4 & -6 & 4 & 0 \\ 0 & 4 & 4 & 1 \\ 0 & 0 & 1 & -6 \end{bmatrix}$, execute one bulge chase by hand assuming $\sigma = 0$.

19.6 Let $A = \begin{bmatrix} 2 & 1 & 0 \\ 1 & 2 & 1 \\ 0 & 1 & 6 \end{bmatrix}$. Perform four iterations of the bisection method by hand to estimate the eigenvalue between 2 and 3.

For Problems 19.7–19.9, you will find it useful to write a MATLAB function

```
sign_changes = d(T,lambda)
```

that computes d(λ).

19.7 Let $A = \begin{bmatrix} 1 & -1 & 0 & 0 \\ -1 & 1 & 2 & 0 \\ 0 & 2 & 2 & -1 \\ 0 & 0 & -1 & 1 \end{bmatrix}$.

a. Show that there must be an eigenvalue greater than or equal to 2 and an eigenvalue less than zero.
b. Show there are two eigenvalues between 0.12 and 2.

19.8 Let $A = \begin{bmatrix} 3 & -1 & 0 \\ -1 & 1 & -1 \\ 0 & -1 & 1 \end{bmatrix}$.

a. Show there must be one negative eigenvalue.
b. Show there must be one eigenvalue in the range $0 \leq \lambda < 2$.
c. Show there is one eigenvalue greater than or equal to 2.

19.9 How many eigenvalues of $A = \begin{bmatrix} 1 & -1 & 0 & 0 \\ -1 & 1 & -1 & 0 \\ 0 & -1 & 1 & -1 \\ 0 & 0 & -1 & 1 \end{bmatrix}$ lie in the interval $0 < \lambda < 2$?

19.10 Let

$$T = \begin{bmatrix} a_1 & b_1 & & & \\ b_1 & a_2 & \ddots & & \\ & \ddots & \ddots & b_{n-2} & \\ & & b_{n-2} & \ddots & 0 \\ & & & 0 & a_n \end{bmatrix},$$

where $a_i \neq 0$, $1 \leq i \leq n$, $b_i \neq 0$, $1 \leq i \leq n-2$.

a. Show that a_n is an eigenvalue of T with associated eigenvector $e_n = \begin{bmatrix} 0 & 0 & \dots & 0 & 1 \end{bmatrix}^{\mathrm{T}}$.

b. Explain how this result relates to our choice of convergence criteria for the QR and Francis methods.

19.11 Outline an algorithm for finding the eigenvalues of a reduced symmetric tridiagonal matrix (the subdiagonal contains one or more zeros).

19.12 Let $A = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$. Show that the QR algorithm with shift $\sigma = t_{k,k}$ fails, but the Wilkinson shift succeeds.

19.13 Let $p_0(\mu), p_1(\mu), \dots, p_{i-1}(\mu), p_i(\mu), \dots, p_n(\mu)$ be the Sturm sequence used by the bisection method, with $b_i \neq 0$, $1 \leq i \leq n$. Show it is not possible that $p_{i-1}(\mu) = 0$ and $p_i(\mu) = 0$; in other words, there cannot be two consecutive zero values in the sequence. Hint: What does Equation 19.16 say about $p_{i-2}(\mu)$?

19.14 Prove Lemma 19.3. Hint: Look at it as a problem involving block matrices.

19.15 Let A be a positive definite matrix and $A = R^{\mathrm{T}}R$ be the Cholesky decomposition. Using the singular value decomposition of R, show how to compute the eigenvalues and associated eigenvectors of A.

19.16 If A is positive definite, the Cholesky decomposition can be used to determine the eigenvalues of A. Here is an outline of the algorithm:

```
A = R^T R
A_1 = R R^T
for i = 1:maxiter do
   A_i = R_i^T R_i
   A_{i+1} = R_i R_i^T
end for
return A_maxiter
```

It can be shown that the sequence converges to a diagonal matrix of eigenvalues.

a. Prove that if A is positive definite, then any matrix similar to A is also positive definite. Hint: What can you say about the eigenvalues of a positive definite matrix?

b. Show that each matrix, A_i, is similar to A so that upon termination of the algorithm, A and A_{maxiter} have the same eigenvalues.

19.17 We know that if v is an eigenvector of A, $\frac{v^{\mathrm{T}}Av}{v^{\mathrm{T}}v}$ is called the Rayleigh quotient and that the corresponding eigenvalue $\lambda = \frac{v^{\mathrm{T}}Av}{v^{\mathrm{T}}v}$. Now, suppose v is an approximation to an eigenvector. How well does the Rayleigh quotient approximate λ? We can answer that question when A is symmetric.

a. Without loss of generality, assume that v is a good approximation to eigenvector v_1 of the symmetric matrix A. Argue that $v = c_1v_1 + c_2v_2 + \cdots + c_nv_n$, where the v_i are an orthonormal set of eigenvectors of A corresponding to eigenvalues λ_i.

b. Show that

$$\sigma = \frac{v^{\mathrm{T}}Av}{v^{\mathrm{T}}v} = \frac{\lambda_1c_1^2 + \lambda_2c_2^2 + \cdots \lambda_nc_n^2}{c_1^2 + c_2^2 + \cdots + c_n^2}.$$

c. Using the result of part (b), show that

$$\sigma = \lambda_1 \left[\frac{1 + \left(\frac{\lambda_2}{\lambda_1}\right)\left(\frac{c_2}{c_1}\right)^2 + \cdots + \left(\frac{\lambda_n}{\lambda_1}\right)\left(\frac{c_n}{c_1}\right)^2}{1 + \left(\frac{c_2}{c_1}\right)^2 + \cdots + \left(\frac{c_n}{c_1}\right)^2} \right],$$

and argue that σ is a close approximation to λ_1.

19.18 We have not discussed another approach to the symmetric eigenvalue problem, the *Rayleigh quotient iteration*. Basically, it is inverse iteration using the Rayleigh quotient as the shift. From Problem 19.17, we know that if v is a good approximation to an eigenvector of the symmetric matrix A, then the Rayleigh quotient $\frac{v^{\mathrm{T}}Av}{v^{\mathrm{T}}v}$ is a good approximation to the corresponding eigenvalue. Start with an initial approximation, v_0, to an eigenvector, compute the Rayleigh quotient, and begin inverse iteration. Unlike the inverse iteration algorithm discussed in Section 18.9, the shift changes at each iteration to the Rayleigh quotient determined by the next approximate eigenvector. A linear system must be solved for each iteration. The Rayleigh quotient iteration computes both an eigenvector and an eigenvalue, and convergence is almost always cubic [1, pp. 214-216]. This convergence rate is very unusual and means that the number of correct digits triples for every iteration.

```
function RQITER(A, v0, tol, maxiter)
   v0 = v0 / ||v0||_2
   σ0 = (v0^T A v0) / (v0^T v0)
   iter = 0
   i = 0
   while (||A vi − σi vi||_2 ≥ tol) and (iter ≤ maxiter) do
      i = i + 1
      vi = (A − σi−1 I)^−1 vi−1
      vi = vi / ||vi||_2
      σi = (vi^T A vi) / (vi^T vi)
      iter = iter + 1
   end while

   if iter > maxiter then
      iter = -1
   end if
   return [ vi, σi, iter ]
end function
```

Let

$$A = \begin{bmatrix} 4 & 6 & 7 \\ 6 & 12 & 11 \\ 7 & 11 & 4 \end{bmatrix}.$$

a. Let $v_0 = \begin{bmatrix} 0.9 & -0.8 & 0.4 \end{bmatrix}^{\mathrm{T}}$, and perform two iterations of rqiter by hand.

b. To obtain the starting approximation, perform two iterations of the power method starting with $v_0 = \begin{bmatrix} 1 & 1 & 1 \end{bmatrix}^{\mathrm{T}}$. Using the new v_0, perform one Rayleigh quotient iteration by hand. Start a second and comment on what happens. What eigenvalue did you approximate, and why?

19.8.1 MATLAB Problems

19.19

a. Implement the classical Jacobi method by modifying `eigsymj` to create a function `cjacobi(A,tol,maxiter)`. You will need to implement a function `jacobifind` that finds the largest off-diagonal entry in magnitude in the symmetric matrix. The calling format should be the same as `eigsymj`, except return the number of iterations required to attain the tolerance rather than the number of sweeps. The function `eigsymj` executes $\frac{n(n-1)}{2}$ Jacobi rotations per sweep, so for the classical Jacobi method the `maxiter` will be much larger than `maxsweeps`.

b. Create a random 100×100 symmetric matrix and time the execution of `cjacobi` and `eigsymj`. Compute $\left\| VDV^{\mathrm{T}} - A \right\|_2$ for each method. Compare the results.

19.20 Find all the eigenvalues of the following symmetric matrices using both the Jacobi method and the Francis algorithm. For each method, compute $\|\text{Emat} - E\|_2$, where Emat is the MATLAB result, and E is the result from the method. Sort the eigenvalues before computing the norms.

a. $A = \begin{bmatrix} -12 & 6 & 1 \\ 6 & 3 & 2 \\ 1 & 2 & 15 \end{bmatrix}$

b. $A = \begin{bmatrix} -8 & 16 & 23 & -13 \\ 16 & 9 & 2 & 3 \\ 23 & 2 & 1 & -23 \\ -13 & 3 & -23 & -7 \end{bmatrix}$

19.21 The MATLAB command

```
A = gallery('pei',n,alpha),
```

where `alpha` is a scalar, returns the symmetric matrix `alpha eye(n) + ones(n)`. The default for `alpha` is 1, and the matrix is singular for `alpha` equal to 0. The eigenvalues are

$$\lambda_1 = \lambda_2 = \cdots = \lambda_{n-1} = \text{alpha}, \quad \lambda_n = n + \text{alpha}.$$

For parts (a) and (b), use (1) `eigsymj`, (2) `eigsymqr`, (3) `eigsymb`, and (4) `dconquer`, to compute the eigenvalues. In each case, compute $\|E - E_i\|_2$, where E are the exact eigenvalues. Do these methods handle the eigenvalues of multiplicity $n - 1$ properly?

a. $n = 25$, alpha = 5

b. $n = 50$, alpha = 0

c. Let `A = gallery('pei',5,3)`, and compute the eigenvector matrix `V1` and the corresponding diagonal matrix `D1` of eigenvalues. Perturb `A(5,1)` by 1.0×10^{-8} and find matrices `V2` and `D2`. Compute $\|D1 - D2\|_2$, and list the eigenvectors `V1` and `V2`. Explain the results.

19.22 Use the function `eigsymj` to compute all the eigenvalues of the 50×50 Hilbert matrix. Use tol $= 1.0 \times 10^{-10}$. Compute the norm of difference between that solution and the one obtained using the MATLAB function `eig`. Given that the Hilbert matrices are very ill-conditioned, explain your result.

19.23 Problem 19.16 presents the basics for a method to compute the eigenvalues of a positive definite matrix.

a. Implement a function `choleig` that takes a positive definite matrix A and executes `maxiter` iterations. After each iteration, use the MATLAB function `spy` to show the location of the nonzero entries as follows:

```
A(abs(A)<1.e-7)=0;
spy(A);
pause;
```

This will allow you to "watch" convergence to a diagonal matrix of eigenvalues. The function should verify that A is positive definite.

b. Test `choleig` with the matrix

```
A = gallery('gcdmat',4);
```

c. Create a 3×3 positive definite matrix using the code provided, and then run `choleig` with `maxiter = 75` and watch convergence. Depending on the matrix, it may or may not graphically show as a diagonal matrix. In any case, check the results against `eig (A)`.

```
A = diag(randi([1 10],3,1));
B = randn(3,3);
[Q,R] = qr(B);
A = Q'*A*Q;
```

19.24 The term *eigenvector localization* applies to an eigenvector where the majority of its length is contributed by a small number of entries. This means that majority of its entries are zero or close to zero. This phenomenon is well known in several scientific applications such as quantum mechanics, DNA data, and astronomy. If the eigenvector

is normalized, one measure of this property is the *inverse participation ratio* (IPR) that is defined as

$$\text{IPR}\,(v) = \sum_{i=1}^{n} v_i^4.$$

The larger the value of IPR, the more localized the eigenvector.

a. If the eigenvector values are equally distributed throughout its indices, show that the IPR is $\frac{1}{n}$.

b. Show that if the eigenvector has only one nonzero entry, its IPR is 1.

c. If n is an integer, the following statements generate a random symmetric tridiagonal matrix. Create a 200×200 random symmetric tridiagonal matrix and plot the eigenvector number against IPR (v_i). What do you observe?

```
d = randn(n,1);
sd = randn(n-1,1);
A = trid(sd,d,sd);
```

d. Do part (c) with the 200×200 matrix $B = \begin{bmatrix} 1+2r & -r & 0 & \cdots & 0 \\ -r & 1+2r & -r & \cdots & 0 \\ 0 & \ddots & \ddots & \ddots & \vdots \\ \vdots & \vdots & -r & 1+2r & -r \\ 0 & 0 & 0 & -r & 1+2r \end{bmatrix}$ used in the solution to the heat equation in Section 12.2. Let $r = 0.5$.

19.25 Modify the function bisection so it computes all the eigenvalues in an interval $a < \lambda < b$. Name function `bisectinterval` having the declaration

```
lambda = bisectinterval(T,a,b,tol)
```

Test it using a random symmetric tridiagonal matrix and wilkinson(21).

19.26 This problem shows how shifts help when computing eigenvalues. Create functions that are simple modifications of `eigsymqr` and `eigsymb` as follows:

- `eigsymqr0`: Remove the shift from the code.
- `eigsymqrr`: Replace the Wilkinson shift by $\sigma = t_{k,k}$, the value we used for a general matrix.
- `eigsymb0`, `chase0`: Remove the shift from the chase code.
- `eigsymbr`, `chaser`: Replace the Wilkinson shift by $\sigma = t_{k,k}$.

In the functions `eigsymqr0` and `eigsymb0`, perform the following code replacement so termination occurs the first time the iteration does not converge to an eigenvalue within the allotted iterations.

Replace

```
if iter > maxiter
    fprintf('Failure of convergence. ');
    fprintf('Current eigenvalue approximation %g\n',T(k,k));
    break;
end
```

by

```
if iter > maxiter
    fprintf('Failure of convergence. ');
    varargout{1} = {};
    varargout{2} = {};
    return;
end
```

Generate a random 400×400 symmetric matrix (`A = randn(400,400), A = A + A'`). Time the use of the original functions `eigsymqr` and `eigsymb` to compute the eigenvalues and eigenvectors of A. Now do the same for each of the modified functions, and discuss the results. For the methods that converge, compute $\left\| VDV^{\mathrm{T}} - A \right\|_2$.

19.27

a. Implement the Rayleigh quotient iteration described in Problem 19.18 in the MATLAB function `rqiter`.

b. Using a random v_0, test `rqiter` with random matrices of dimensions 5×5 and 25×25. Use tol $= 1.0 \times 10^{-12}$, maxiter = 25.

19.28

a. Implement Newton's method with a MATLAB function having calling syntax

```
root = newton(f,df,x0,tol,maxiter);
```

The argument `f` is the function, `df` is $f'(x)$, `x0` is the initial approximation, `tol` is the required relative error bound $\left(\frac{|x_{i+1}-x_i|}{|x_i|} < \text{tol}\right)$, and `maxiter` is the maximum number of iterations to perform.

b. Graph $f(\lambda) = 1 + \frac{1}{1-\lambda} + \frac{1}{2-\lambda} + \frac{1}{3-\lambda}$ and use `newton` to estimate all roots of the f. How well did Newton's method do?

c. Now graph $g(\lambda) = 1 + \frac{0.005}{1-\lambda} + \frac{0.005}{2-\lambda} + \frac{0.005}{3-\lambda}$ over the interval $0.95 < \lambda < 1.05$. Describe the shape of the graph for all points except those very close to the pole at $\lambda = 1$.

d. Plot $g(\lambda)$ of the interval $1.004 < \lambda < 1.006$. Is there a root in the interval?

e. Try to compute the root using Newton's method with $x_0 = 1.01$. Explain the results.