CITED REFERENCES AND FURTHER READING:

Stoer, J., and Bulirsch, R. 1980, *Introduction to Numerical Analysis* (New York: Springer-Verlag), Chapter 6. [1]

Wilkinson, J.H., and Reinsch, C. 1971, *Linear Algebra*, vol. II of *Handbook for Automatic Computation* (New York: Springer-Verlag). [2]

Smith, B.T., et al. 1976, *Matrix Eigensystem Routines — EISPACK Guide*, 2nd ed., vol. 6 of Lecture Notes in Computer Science (New York: Springer-Verlag). [3]

IMSL Math/Library Users Manual (IMSL Inc., 2500 CityWest Boulevard, Houston TX 77042). [4]

NAG Fortran Library (Numerical Algorithms Group, 256 Banbury Road, Oxford OX27DE, U.K.), Chapter F02. [5]

Golub, G.H., and Van Loan, C.F. 1989, *Matrix Computations*, 2nd ed. (Baltimore: Johns Hopkins University Press), §7.7. [6]

Wilkinson, J.H. 1965, *The Algebraic Eigenvalue Problem* (New York: Oxford University Press). [7]

Acton, F.S. 1970, *Numerical Methods That Work*; 1990, corrected edition (Washington: Mathematical Association of America), Chapter 13.

Horn, R.A., and Johnson, C.R. 1985, *Matrix Analysis* (Cambridge: Cambridge University Press).

11.1 Jacobi Transformations of a Symmetric Matrix

The Jacobi method consists of a sequence of orthogonal similarity transformations of the form of equation (11.0.14). Each transformation (a *Jacobi rotation*) is just a plane rotation designed to annihilate one of the off-diagonal matrix elements. Successive transformations undo previously set zeros, but the off-diagonal elements nevertheless get smaller and smaller, until the matrix is diagonal to machine precision. Accumulating the product of the transformations as you go gives the matrix of eigenvectors, equation (11.0.15), while the elements of the final diagonal matrix are the eigenvalues.

The Jacobi method is absolutely foolproof for all real symmetric matrices. For matrices of order greater than about 10, say, the algorithm is slower, by a significant constant factor, than the QR method we shall give in §11.3. However, the Jacobi algorithm is much simpler than the more efficient methods. We thus recommend it for matrices of moderate order, where expense is not a major consideration.

The basic Jacobi rotation $\mathbf{P}_{pq}$ is a matrix of the form

$$\mathbf{P}_{pq} = \begin{bmatrix} 1 & & & & & & \\ & \cdots & & & & & \\ & & c & \cdots & s & & \\ & & \vdots & 1 & \vdots & & \\ & & -s & \cdots & c & & \\ & & & & & \cdots & \\ & & & & & & 1 \end{bmatrix} \tag{11.1.1}$$

Here all the diagonal elements are unity except for the two elements c in rows (and columns) p and q. All off-diagonal elements are zero except the two elements s and $-s$. The numbers c and s are the cosine and sine of a rotation angle ϕ, so $c^2 + s^2 = 1$.

A plane rotation such as (11.1.1) is used to transform the matrix $\mathbf{A}$ according to

$$\mathbf{A}' = \mathbf{P}_{pq}^T \cdot \mathbf{A} \cdot \mathbf{P}_{pq} \tag{11.1.2}$$

Now, $\mathbf{P}_{pq}^T \cdot \mathbf{A}$ changes only rows p and q of $\mathbf{A}$, while $\mathbf{A} \cdot \mathbf{P}_{pq}$ changes only columns p and q. Notice that the subscripts p and q do not denote components of $\mathbf{P}_{pq}$, but rather label which kind of rotation the matrix is, i.e., which rows and columns it affects. Thus the changed elements of $\mathbf{A}$ in (11.1.2) are only in the p and q rows and columns indicated below:

$$\mathbf{A}' = \begin{bmatrix} & \cdots & a'_{1p} & \cdots & a'_{1q} & \cdots & \\ \vdots & & \vdots & & \vdots & & \vdots \\ a'_{p1} & \cdots & a'_{pp} & \cdots & a'_{pq} & \cdots & a'_{pn} \\ \vdots & & \vdots & & \vdots & & \vdots \\ a'_{q1} & \cdots & a'_{qp} & \cdots & a'_{qq} & \cdots & a'_{qn} \\ \vdots & & \vdots & & \vdots & & \vdots \\ & \cdots & a'_{np} & \cdots & a'_{nq} & \cdots & \end{bmatrix} \tag{11.1.3}$$

Multiplying out equation (11.1.2) and using the symmetry of $\mathbf{A}$, we get the explicit formulas

$$\left.\begin{aligned} a'_{rp} &= c a_{rp} - s a_{rq} \\ a'_{rq} &= c a_{rq} + s a_{rp} \end{aligned}\right\} \qquad r \neq p,\ r \neq q \tag{11.1.4}$$

$$a'_{pp} = c^2 a_{pp} + s^2 a_{qq} - 2sc a_{pq} \tag{11.1.5}$$

$$a'_{qq} = s^2 a_{pp} + c^2 a_{qq} + 2sc a_{pq} \tag{11.1.6}$$

$$a'_{pq} = (c^2 - s^2) a_{pq} + sc(a_{pp} - a_{qq}) \tag{11.1.7}$$

The idea of the Jacobi method is to try to zero the off-diagonal elements by a series of plane rotations. Accordingly, to set $a'_{pq} = 0$, equation (11.1.7) gives the following expression for the rotation angle ϕ

$$\theta \equiv \cot 2\phi \equiv \frac{c^2 - s^2}{2sc} = \frac{a_{qq} - a_{pp}}{2a_{pq}} \tag{11.1.8}$$

If we let $t \equiv s/c$, the definition of θ can be rewritten

$$t^2 + 2t\theta - 1 = 0 \tag{11.1.9}$$

The smaller root of this equation corresponds to a rotation angle less than $\pi/4$ in magnitude; this choice at each stage gives the most stable reduction. Using the form of the quadratic formula with the discriminant in the denominator, we can write this smaller root as

$$t = \frac{\operatorname{sgn}(\theta)}{|\theta| + \sqrt{\theta^2 + 1}} \tag{11.1.10}$$

If θ is so large that θ^2 would overflow on the computer, we set $t = 1/(2\theta)$. It now follows that

$$c = \frac{1}{\sqrt{t^2+1}} \tag{11.1.11}$$

$$s = tc \tag{11.1.12}$$

When we actually use equations (11.1.4)–(11.1.7) numerically, we rewrite them to minimize roundoff error. Equation (11.1.7) is replaced by

$$a'_{pq} = 0 \tag{11.1.13}$$

The idea in the remaining equations is to set the new quantity equal to the old quantity plus a small correction. Thus we can use (11.1.7) and (11.1.13) to eliminate a_{qq} from (11.1.5), giving

$$a'_{pp} = a_{pp} - t a_{pq} \tag{11.1.14}$$

Similarly,

$$a'_{qq} = a_{qq} + t a_{pq} \tag{11.1.15}$$

$$a'_{rp} = a_{rp} - s(a_{rq} + \tau a_{rp}) \tag{11.1.16}$$

$$a'_{rq} = a_{rq} + s(a_{rp} - \tau a_{rq}) \tag{11.1.17}$$

where τ $(= \tan\phi/2)$ is defined by

$$\tau \equiv \frac{s}{1+c} \tag{11.1.18}$$

One can see the convergence of the Jacobi method by considering the sum of the squares of the off-diagonal elements

$$S = \sum_{r \neq s} |a_{rs}|^2 \tag{11.1.19}$$

Equations (11.1.4)–(11.1.7) imply that

$$S' = S - 2|a_{pq}|^2 \tag{11.1.20}$$

(Since the transformation is orthogonal, the sum of the squares of the diagonal elements increases correspondingly by $2|a_{pq}|^2$.) The sequence of S's thus decreases monotonically. Since the sequence is bounded below by zero, and since we can choose a_{pq} to be whatever element we want, the sequence can be made to converge to zero.

Eventually one obtains a matrix **D** that is diagonal to machine precision. The diagonal elements give the eigenvalues of the original matrix **A**, since

$$\mathbf{D} = \mathbf{V}^T \cdot \mathbf{A} \cdot \mathbf{V} \tag{11.1.21}$$

where

$$\mathbf{V} = \mathbf{P}_1 \cdot \mathbf{P}_2 \cdot \mathbf{P}_3 \cdots \tag{11.1.22}$$

the $\mathbf{P}_i$'s being the successive Jacobi rotation matrices. The columns of $\mathbf{V}$ are the eigenvectors (since $\mathbf{A} \cdot \mathbf{V} = \mathbf{V} \cdot \mathbf{D}$). They can be computed by applying

$$\mathbf{V}' = \mathbf{V} \cdot \mathbf{P}_i \tag{11.1.23}$$

at each stage of calculation, where initially $\mathbf{V}$ is the identity matrix. In detail, equation (11.1.23) is

$$\begin{aligned} v'_{rs} &= v_{rs} \qquad (s \neq p,\ s \neq q) \\ v'_{rp} &= c v_{rp} - s v_{rq} \\ v'_{rq} &= s v_{rp} + c v_{rq} \end{aligned} \tag{11.1.24}$$

We rewrite these equations in terms of τ as in equations (11.1.16) and (11.1.17) to minimize roundoff.

The only remaining question is the strategy one should adopt for the order in which the elements are to be annihilated. Jacobi's original algorithm of 1846 searched the whole upper triangle at each stage and set the largest off-diagonal element to zero. This is a reasonable strategy for hand calculation, but it is prohibitive on a computer since the search alone makes each Jacobi rotation a process of order N^2 instead of N.

A better strategy for our purposes is the *cyclic Jacobi method*, where one annihilates elements in strict order. For example, one can simply proceed down the rows: $\mathbf{P}_{12}, \mathbf{P}_{13}, \ldots, \mathbf{P}_{1n}$; then $\mathbf{P}_{23}, \mathbf{P}_{24}$, etc. One can show that convergence is generally quadratic for both the original or the cyclic Jacobi methods, for nondegenerate eigenvalues. One such set of $n(n-1)/2$ Jacobi rotations is called a *sweep*.

The program below, based on the implementations in [1,2], uses two further refinements:

- In the first three sweeps, we carry out the pq rotation only if $|a_{pq}| > \epsilon$ for some threshold value

$$\epsilon = \frac{1}{5} \frac{S_0}{n^2} \tag{11.1.25}$$

where S_0 is the sum of the off-diagonal moduli,

$$S_0 = \sum_{r<s} |a_{rs}| \tag{11.1.26}$$

- After four sweeps, if $|a_{pq}| \ll |a_{pp}|$ and $|a_{pq}| \ll |a_{qq}|$, we set $|a_{pq}| = 0$ and skip the rotation. The criterion used in the comparison is $|a_{pq}| < 10^{-(D+2)}|a_{pp}|$, where D is the number of significant decimal digits on the machine, and similarly for $|a_{qq}|$.

In the following routine the n×n symmetric matrix a is stored as a[1..n][1..n]. On output, the superdiagonal elements of a are destroyed, but the diagonal and subdiagonal are unchanged and give full information on the original symmetric matrix a. The vector d[1..n] returns the eigenvalues of a. During the computation, it contains the current diagonal of a. The matrix v[1..n][1..n] outputs the normalized eigenvector belonging to d[k] in its kth column. The parameter nrot is the number of Jacobi rotations that were needed to achieve convergence.

Typical matrices require 6 to 10 sweeps to achieve convergence, or $3n^2$ to $5n^2$ Jacobi rotations. Each rotation requires of order $4n$ operations, each consisting of a multiply and an add, so the total labor is of order $12n^3$ to $20n^3$ operations. Calculation of the eigenvectors as well as the eigenvalues changes the operation count from $4n$ to $6n$ per rotation, which is only a 50 percent overhead.

```
#include <math.h>
#include "nrutil.h"
#define ROTATE(a,i,j,k,l) g=a[i][j];h=a[k][l];a[i][j]=g-s*(h+g*tau);\
    a[k][l]=h+s*(g-h*tau);

void jacobi(float **a, int n, float d[], float **v, int *nrot)
Computes all eigenvalues and eigenvectors of a real symmetric matrix a[1..n][1..n]. On
output, elements of a above the diagonal are destroyed. d[1..n] returns the eigenvalues of a.
v[1..n][1..n] is a matrix whose columns contain, on output, the normalized eigenvectors of
a. nrot returns the number of Jacobi rotations that were required.
{
    int j,iq,ip,i;
    float tresh,theta,tau,t,sm,s,h,g,c,*b,*z;

    b=vector(1,n);
    z=vector(1,n);
    for (ip=1;ip<=n;ip++) {                     Initialize to the identity matrix.
        for (iq=1;iq<=n;iq++) v[ip][iq]=0.0;
        v[ip][ip]=1.0;
    }
    for (ip=1;ip<=n;ip++) {                     Initialize b and d to the diagonal
        b[ip]=d[ip]=a[ip][ip];                      of a.
        z[ip]=0.0;                              This vector will accumulate terms
    }                                               of the form ta_pq as in equa-
    *nrot=0;                                        tion (11.1.14).
    for (i=1;i<=50;i++) {
        sm=0.0;
        for (ip=1;ip<=n-1;ip++) {               Sum off-diagonal elements.
            for (iq=ip+1;iq<=n;iq++)
                sm += fabs(a[ip][iq]);
        }
        if (sm == 0.0) {                        The normal return, which relies
            free_vector(z,1,n);                     on quadratic convergence to
            free_vector(b,1,n);                     machine underflow.
            return;
        }
        if (i < 4)
            tresh=0.2*sm/(n*n);                 ...on the first three sweeps.
        else
            tresh=0.0;                          ...thereafter.
        for (ip=1;ip<=n-1;ip++) {
            for (iq=ip+1;iq<=n;iq++) {
                g=100.0*fabs(a[ip][iq]);
                After four sweeps, skip the rotation if the off-diagonal element is small.
                if (i > 4 && (float)(fabs(d[ip])+g) == (float)fabs(d[ip])
                    && (float)(fabs(d[iq])+g) == (float)fabs(d[iq]))
                    a[ip][iq]=0.0;
```

```
            else if (fabs(a[ip][iq]) > tresh) {
                h=d[iq]-d[ip];
                if ((float)(fabs(h)+g) == (float)fabs(h))
                    t=(a[ip][iq])/h;                t = 1/(2θ)
                else {
                    theta=0.5*h/(a[ip][iq]);        Equation (11.1.10).
                    t=1.0/(fabs(theta)+sqrt(1.0+theta*theta));
                    if (theta < 0.0) t = -t;
                }
                c=1.0/sqrt(1+t*t);
                s=t*c;
                tau=s/(1.0+c);
                h=t*a[ip][iq];
                z[ip] -= h;
                z[iq] += h;
                d[ip] -= h;
                d[iq] += h;
                a[ip][iq]=0.0;
                for (j=1;j<=ip-1;j++) {             Case of rotations 1 ≤ j < p.
                    ROTATE(a,j,ip,j,iq)
                }
                for (j=ip+1;j<=iq-1;j++) {          Case of rotations p < j < q.
                    ROTATE(a,ip,j,j,iq)
                }
                for (j=iq+1;j<=n;j++) {             Case of rotations q < j ≤ n.
                    ROTATE(a,ip,j,iq,j)
                }
                for (j=1;j<=n;j++) {
                    ROTATE(v,j,ip,j,iq)
                }
                ++(*nrot);
            }
        }
    }
    for (ip=1;ip<=n;ip++) {
        b[ip] += z[ip];
        d[ip]=b[ip];                                Update d with the sum of ta_pq,
        z[ip]=0.0;                                  and reinitialize z.
    }
  }
  nrerror("Too many iterations in routine jacobi");
}
```

Note that the above routine assumes that underflows are set to zero. On machines where this is not true, the program must be modified.

The eigenvalues are not ordered on output. If sorting is desired, the following routine can be invoked to reorder the output of `jacobi` or of later routines in this chapter. (The method, straight insertion, is N^2 rather than $N \log N$; but since you have just done an N^3 procedure to get the eigenvalues, you can afford yourself this little indulgence.)

```
void eigsrt(float d[], float **v, int n)
```
Given the eigenvalues `d[1..n]` and eigenvectors `v[1..n][1..n]` as output from `jacobi` (§11.1) or `tqli` (§11.3), this routine sorts the eigenvalues into descending order, and rearranges the columns of `v` correspondingly. The method is straight insertion.
```
{
    int k,j,i;
    float p;

    for (i=1;i<n;i++) {
        p=d[k=i];
```

```
        for (j=i+1;j<=n;j++)
            if (d[j] >= p) p=d[k=j];
        if (k != i) {
            d[k]=d[i];
            d[i]=p;
            for (j=1;j<=n;j++) {
                p=v[j][i];
                v[j][i]=v[j][k];
                v[j][k]=p;
            }
        }
    }
}
```

CITED REFERENCES AND FURTHER READING:

Golub, G.H., and Van Loan, C.F. 1989, *Matrix Computations*, 2nd ed. (Baltimore: Johns Hopkins University Press), §8.4.

Smith, B.T., et al. 1976, *Matrix Eigensystem Routines — EISPACK Guide*, 2nd ed., vol. 6 of Lecture Notes in Computer Science (New York: Springer-Verlag). [1]

Wilkinson, J.H., and Reinsch, C. 1971, *Linear Algebra*, vol. II of *Handbook for Automatic Computation* (New York: Springer-Verlag). [2]

11.2 Reduction of a Symmetric Matrix to Tridiagonal Form: Givens and Householder Reductions

As already mentioned, the optimum strategy for finding eigenvalues and eigenvectors is, first, to reduce the matrix to a simple form, only then beginning an iterative procedure. For symmetric matrices, the preferred simple form is tridiagonal. The *Givens reduction* is a modification of the Jacobi method. Instead of trying to reduce the matrix all the way to diagonal form, we are content to stop when the matrix is tridiagonal. This allows the procedure to be carried out *in a finite number of steps*, unlike the Jacobi method, which requires iteration to convergence.

Givens Method

For the Givens method, we choose the rotation angle in equation (11.1.1) so as to zero an element that is *not* at one of the four "corners," i.e., not a_{pp}, a_{pq}, or a_{qq} in equation (11.1.3). Specifically, we first choose $\mathbf{P}_{23}$ to annihilate a_{31} (and, by symmetry, a_{13}). Then we choose $\mathbf{P}_{24}$ to annihilate a_{41}. In general, we choose the sequence

$$\mathbf{P}_{23}, \mathbf{P}_{24}, \dots, \mathbf{P}_{2n}; \mathbf{P}_{34}, \dots, \mathbf{P}_{3n}; \dots ; \mathbf{P}_{n-1,n}$$

where $\mathbf{P}_{jk}$ annihilates $a_{k,j-1}$. The method works because elements such as a'_{rp} and a'_{rq}, with $r \neq p$ $r \neq q$, are linear combinations of the old quantities a_{rp} and a_{rq}, by