and the fact that each decomposition is unique implies

$$\mathbf{V} = \mathbf{U} = \mathbf{X}_R = \mathbf{X}_L^T \tag{11.0.29}$$

and

$$\lambda_i = w_i, \qquad i = 0, \ldots, N-1 \tag{11.0.30}$$

That is, the (left and right) eigenvectors are the columns of any of the matrices listed in equation (11.0.29), and the corresponding eigenvalues and singular values are identical.

From a general matrix $\mathbf{A}$, not necessarily even square, one can form the two symmetric matrices $\mathbf{A}^T \cdot \mathbf{A}$ and $\mathbf{A} \cdot \mathbf{A}^T$. You can work out from equation (11.0.27) that the eigenvalues of either of these two matrices are squares of singular values of $\mathbf{A}$. However, this doesn't tell you about the eigenvalues of $\mathbf{A}$: The matrix whose eigenvalues are the squares of the eigenvalues of $\mathbf{A}$ is the unrelated matrix $\mathbf{A} \cdot \mathbf{A}$, not $\mathbf{A}^T \cdot \mathbf{A}$ or $\mathbf{A} \cdot \mathbf{A}^T$.

CITED REFERENCES AND FURTHER READING:

Stoer, J., and Bulirsch, R. 2002, *Introduction to Numerical Analysis*, 3rd ed. (New York: Springer), Chapter 6.[1]

Wilkinson, J.H., and Reinsch, C. 1971, *Linear Algebra*, vol. II of *Handbook for Automatic Computation* (New York: Springer).[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).[3]

Anderson, E., et al. 1999, LAPACK User's Guide, 3rd ed. (Philadelphia: S.I.A.M.). Online with software at 2007+, http://www.netlib.org/lapack.[4]

IMSL Math/Library Users Manual (Houston: IMSL Inc.), see 2007+, http://www.vni.com/products/imsl.[5]

NAG Fortran Library (Oxford, UK: Numerical Algorithms Group), see 2007+, http://www.nag.co.uk, Chapter F02.[6]

Golub, G.H., and Van Loan, C.F. 1996, *Matrix Computations*, 3rd ed. (Baltimore: Johns Hopkins University Press), §7.7.[7]

Wilkinson, J.H. 1965, *The Algebraic Eigenvalue Problem* (New York: Oxford University Press).[8]

Acton, F.S. 1970, *Numerical Methods That Work*; 1990, corrected edition (Washington, DC: 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 (11.0.15). 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 (11.0.16), while the elements of the final diagonal matrix are the eigenvalues.

The Jacobi method is absolutely foolproof for all real symmetric matrices. In particular, it returns the small eigenvalues with better relative accuracy than methods that first reduce the matrix to tridiagonal form. For matrices of order greater than about 10, however, the algorithm is slower, by a significant constant factor, than the QR method we shall give in §11.4. 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 rows p and q, and columns p and q, as indicated below:

$$\mathbf{A}' = \begin{bmatrix} & & a'_{0p} & & a'_{0q} & & \\ & & \vdots & & \vdots & & \\ a'_{p0} & \cdots & a'_{pp} & \cdots & a'_{pq} & \cdots & a'_{p,n-1} \\ & & \vdots & & \vdots & & \\ a'_{q0} & \cdots & a'_{qp} & \cdots & a'_{qq} & \cdots & a'_{q,n-1} \\ & & \vdots & & \vdots & & \\ & & a'_{n-1,p} & & a'_{n-1,q} & & \end{bmatrix} \tag{11.1.3}$$

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

$$\left.\begin{aligned} a'_{rp} &= ca_{rp} - sa_{rq} \\ a'_{rq} &= ca_{rq} + sa_{rp} \end{aligned}\right\} \quad r \neq p,\ r \neq q \tag{11.1.4}$$

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

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

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

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

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

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

$$a'_{pp} = a_{pp} - ta_{pq} \tag{11.1.14}$$

Similarly,

$$a'_{qq} = a_{qq} + ta_{pq} \tag{11.1.15}$$

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

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

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 $\mathbf{D}$ that is diagonal to machine precision. The diagonal elements give the eigenvalues of the original matrix $\mathbf{A}$, 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: $\mathbf{P}_{01}, \mathbf{P}_{02}, ..., \mathbf{P}_{0,n-1}$; then $\mathbf{P}_{12}, \mathbf{P}_{13}$, etc. One can show that convergence is generally quadratic for either the original or the cyclic Jacobi method, 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 $|a_{qq}|$.

Typical matrices require six to ten sweeps to achieve convergence, or $3n^2$ to $5n^2$ Jacobi rotations. Each rotation requires of order $8n$ floating-point operations, so the total labor is of order $24n^3$ to $40n^3$ operations. Calculation of the eigenvectors as well as the eigenvalues changes the operation count from $8n$ to $12n$ per rotation, which is only a 50% overhead.

The following routine implements the Jacobi method. Simply create a `Jacobi` object using your symmetric matrix `a[0..n-1] [0..n-1]`:

```
Jacobi jac(a);
```

The vector `d[0..n-1]` then contains the eigenvalues of `a`. During the computation, it contains the current diagonal of `a`. The matrix `v[0..n-1][0..n-1]` outputs the normalized eigenvector belonging to `d[k]` in column `k`. The parameter `nrot` is the number of Jacobi rotations that were needed to achieve convergence.

eigen_sym.h

```
struct Jacobi {
```
Computes all eigenvalues and eigenvectors of a real symmetric matrix by Jacobi's method.
```
    const Int n;
    MatDoub a,v;
    VecDoub d;
    Int nrot;
    const Doub EPS;

    Jacobi(MatDoub_I &aa) : n(aa.nrows()), a(aa), v(n,n), d(n), nrot(0),
        EPS(numeric_limits<Doub>::epsilon())
```
Computes all eigenvalues and eigenvectors of a real symmetric matrix `a[0..n-1][0..n-1]`. On output, `d[0..n-1]` contains the eigenvalues of `a` sorted into descending order, while `v[0..n-1][0..n-1]` is a matrix whose columns contain the corresponding normalized eigenvectors. `nrot` contains the number of Jacobi rotations that were required. Only the upper triangle of `a` is accessed.
```
    {
        Int i,j,ip,iq;
        Doub tresh,theta,tau,t,sm,s,h,g,c;
        VecDoub b(n),z(n);
        for (ip=0;ip<n;ip++) {
            for (iq=0;iq<n;iq++) v[ip][iq]=0.0;
            v[ip][ip]=1.0;
        }
        for (ip=0;ip<n;ip++) {
            b[ip]=d[ip]=a[ip][ip];
            z[ip]=0.0;
        }
        for (i=1;i<=50;i++) {
            sm=0.0;
            for (ip=0;ip<n-1;ip++) {
                for (iq=ip+1;iq<n;iq++)
                    sm += abs(a[ip][iq]);
            }
            if (sm == 0.0) {
                eigsrt(d,&v);
                return;
            }
            if (i < 4)
                tresh=0.2*sm/(n*n);
```
Initialize to the identity matrix.

Initialize `b` and `d` to the diagonal of `a`.

This vector will accumulate terms of the form ta_{pq} as in equation (11.1.14).

Sum magnitude of off-diagonal elements.

The normal return, which relies on quadratic convergence to machine underflow.

On the first three sweeps...

```
            else
                tresh=0.0;                              ...thereafter.
            for (ip=0;ip<n-1;ip++) {
                for (iq=ip+1;iq<n;iq++) {
                    g=100.0*abs(a[ip][iq]);
                    After four sweeps, skip the rotation if the off-diagonal element is small.
                    if (i > 4 && g <= EPS*abs(d[ip]) && g <= EPS*abs(d[iq]))
                            a[ip][iq]=0.0;
                    else if (abs(a[ip][iq]) > tresh) {
                        h=d[iq]-d[ip];
                        if (g <= EPS*abs(h))
                            t=(a[ip][iq])/h;            t = 1/(2θ)
                        else {
                            theta=0.5*h/(a[ip][iq]);    Equation (11.1.10).
                            t=1.0/(abs(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=0;j<ip;j++)              Case of rotations 0 ≤ j < p.
                            rot(a,s,tau,j,ip,j,iq);
                        for (j=ip+1;j<iq;j++)           Case of rotations p < j < q.
                            rot(a,s,tau,ip,j,j,iq);
                        for (j=iq+1;j<n;j++)            Case of rotations q < j < n.
                            rot(a,s,tau,ip,j,iq,j);
                        for (j=0;j<n;j++)
                            rot(v,s,tau,j,ip,j,iq);
                        ++nrot;
                    }
                }
            }
            for (ip=0;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.
            }
        }
        throw("Too many iterations in routine jacobi");
    }
    inline void rot(MatDoub_IO &a, const Doub s, const Doub tau, const Int i,
        const Int j, const Int k, const Int l)
    {
        Doub g=a[i][j];
        Doub h=a[k][l];
        a[i][j]=g-s*(h+g*tau);
        a[k][l]=h+s*(g-h*tau);
    }
};
```

Note that the above routine assumes that underflows are set to zero. On machines where this is not true, the program must be modified. See §1.5.4 and/or find out about the `fesetenv` (Linux) or `__controlfp` (Microsoft) functions.

The Jacobi method does not order the eigenvalues itself. We incorporate the following routine to sort the eigenvalues into descending order. The same routine is used in `Symmeig` in the next section. (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.)

eigen_sym.h

```
void eigsrt(VecDoub_IO &d, MatDoub_IO *v=NULL)
Given the eigenvalues d[0..n-1] and (optionally) the eigenvectors v[0..n-1][0..n-1] as de-
termined by Jacobi (§11.1) or tqli (§11.4), this routine sorts the eigenvalues into descending
order and rearranges the columns of v correspondingly. The method is straight insertion.
{
    Int k;
    Int n=d.size();
    for (Int i=0;i<n-1;i++) {
        Doub p=d[k=i];
        for (Int j=i;j<n;j++)
            if (d[j] >= p) p=d[k=j];
        if (k != i) {
            d[k]=d[i];
            d[i]=p;
            if (v != NULL)
                for (Int j=0;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. 1996, *Matrix Computations*, 3rd 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).[1]

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

11.2 Real Symmetric Matrices

As already mentioned, the optimum strategy in most cases 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.

Here is a routine based on this strategy that finds all eigenvalues and eigenvectors of a real symmetric matrix. It is typically a factor of about five faster than the Jacobi routine of the previous section. The implementations of the functions `tred2` and `tqli` that reduce the matrix to tridiagonal form and then find the eigensystem are discussed in the next two sections.

There are two user interfaces, implemented as two constructors. The first constructor is the usual one:

```
Symmeig s(a);
```

It returns the eigenvalues of `a` in descending order in `s.d[0..n-1]`. The normalized eigenvector corresponding to `d[k]` is in the matrix column `s.z[0..n-1][k]`. Setting the default argument to `false` suppresses the computation of the eigenvectors: