

1 Eigenvalues and eigenvectors

1.1 Introduction

A non-zero column-vector \mathbf{v} is called the *eigenvector* of a matrix \mathbf{A} with the *eigenvalue* λ , if

$$\mathbf{A}\mathbf{v} = \lambda\mathbf{v} . \quad (1)$$

If an $n \times n$ matrix \mathbf{A} is real and symmetric, $\mathbf{A}^T = \mathbf{A}$, then it has n real eigenvalues $\lambda_1, \dots, \lambda_n$, and its (orthogonalized) eigenvectors $\mathbf{V} = \{\mathbf{v}_1, \dots, \mathbf{v}_n\}$ form a full basis,

$$\mathbf{V}\mathbf{V}^T = \mathbf{V}^T\mathbf{V} = \mathbf{1} , \quad (2)$$

in which the matrix is diagonal,

$$\mathbf{V}^T\mathbf{A}\mathbf{V} = \begin{bmatrix} \lambda_1 & 0 & \cdots & 0 \\ 0 & \lambda_2 & & \vdots \\ \vdots & & \ddots & \\ 0 & \cdots & & \lambda_n \end{bmatrix} . \quad (3)$$

Matrix diagonalization means finding all eigenvalues and (optionally) eigenvectors of a matrix.

Eigenvalues and eigenvectors enjoy a multitude of applications in different branches of science and technology.

1.2 Similarity transformations

Orthogonal transformations,

$$\mathbf{A} \rightarrow \mathbf{Q}^T\mathbf{A}\mathbf{Q} , \quad (4)$$

where $\mathbf{Q}^T\mathbf{Q} = \mathbf{1}$, and, generally, similarity transformations,

$$\mathbf{A} \rightarrow \mathbf{S}^{-1}\mathbf{A}\mathbf{S} , \quad (5)$$

preserve eigenvalues and eigenvectors. Therefore one of the strategies to diagonalize a matrix is to apply a sequence of similarity transformations (also called rotations) which (iteratively) turn the matrix into diagonal form.

A side effect of zeroing a given off-diagonal element A_{pq} by a Jacobi rotation is that other off-diagonal elements are changed. Namely, the elements of the rows and columns with indices p and q . However, after the Jacobi rotation the sum of squares of all off-diagonal elements is reduced. The algorithm repeatedly performs rotations until the off-diagonal elements become sufficiently small.

The convergence of the Jacobi method can be proved for two strategies for choosing the order in which the elements are zeroed:

1. *Classical method*: with each rotation the largest of the remaining off-diagonal elements is zeroed.
2. *Cyclic method*: the off-diagonal elements are zeroed in strict order, e.g. row after row.

Although the classical method allows the least number of rotations, it is typically slower than the cyclic method since searching for the largest element is an $O(n^2)$ operation. The count can be reduced by keeping an additional array with indexes of the largest elements in each row. Updating this array after each rotation is only an $O(n)$ operation.

A *sweep* is a sequence of Jacobi rotations applied to all non-diagonal elements. Typically the method converges after a small number of sweeps. The operation count is $O(n)$ for a Jacobi rotation and $O(n^3)$ for a sweep.

The typical convergence criterion is that the diagonal elements have not changed after a sweep. Other criteria can also be used, like the sum of absolute values of the off-diagonal elements is small, $\sum_{i < j} |A_{ij}| < \epsilon$, where ϵ is the required accuracy, or the largest off-diagonal element is small, $\max |A_{i < j}| < \epsilon$.

The eigenvectors can be calculated as $V = \mathbf{1}J_1J_2\dots$, where J_i are the successive Jacobi matrices. At each stage the transformation is

$$\begin{aligned} V_{ij} &\rightarrow V_{ij}, \quad j \neq p, q \\ V_{ip} &\rightarrow cV_{ip} - sV_{iq} \\ V_{iq} &\rightarrow sV_{ip} + cV_{iq} \end{aligned} \tag{11}$$

Alternatively, if only one (or few) eigenvector \mathbf{v}_k is needed, one can instead solve the (singular) system $(A - \lambda_k)\mathbf{v} = 0$.

1.2.2 QR/QL algorithm

An orthogonal transformation of a real symmetric matrix, $\mathbf{A} \rightarrow \mathbf{Q}^T \mathbf{A} \mathbf{Q} = \mathbf{R} \mathbf{Q}$, where \mathbf{Q} is from the QR-decomposition of \mathbf{A} , partly turns the matrix \mathbf{A} into

diagonal form. Successive iterations eventually make it diagonal. If there are degenerate eigenvalues there will be a corresponding block-diagonal sub-matrix. For convergence properties it is of advantage to use *shifts*: instead of $\text{QR}[\mathbf{A}]$ we do $\text{QR}[\mathbf{A} - s\mathbf{1}]$ and then $\mathbf{A} \rightarrow \mathbf{R}\mathbf{Q} + s\mathbf{1}$. The shift s can be chosen as \mathbf{A}_{nn} . As soon as an eigenvalue is found the matrix is deflated, that is the corresponding row and column are crossed out.

Accumulating the successive transformation matrices \mathbf{Q}_i into the total matrix $\mathbf{Q} = \mathbf{Q}_1 \dots \mathbf{Q}_N$, such that $Q^T \mathbf{A} \mathbf{Q} = \Lambda$, gives the eigenvectors as columns of the \mathbf{Q} matrix.

If only one (or few) eigenvector \mathbf{v}_k is needed one can instead solve the (singular) system $(\mathbf{A} - \lambda_k)\mathbf{v} = 0$.

Tridiagonalization. Each iteration of the QR/QL algorithm is an $O(n^3)$ operation. On a tridiagonal matrix it is only $O(n)$. Therefore the effective strategy is first to make the matrix tridiagonal and then apply the QR/QL algorithm. Tridiagonalization of a matrix is a non-iterative operation with a fixed number of steps.

1.3 Eigenvalues of updated matrix

In practice it happens quite often that the matrix A to be diagonalized is given in the form of a diagonal matrix, \mathbf{D} , plus an update matrix, \mathbf{W} ,

$$\mathbf{A} = \mathbf{D} + \mathbf{W}, \quad (12)$$

where the update \mathbf{W} is a simpler, in a certain sense, matrix which allows a more efficient calculation of the updated eigenvalues, as compared to general diagonalization algorithms.

The most common updates are

- symmetric rank-1 update,

$$\mathbf{W} = \mathbf{u}\mathbf{u}^T, \quad (13)$$

where \mathbf{u} is a column-vector;

- symmetric rank-2 update,

$$\mathbf{W} = \mathbf{u}\mathbf{v}^T + \mathbf{v}\mathbf{u}^T; \quad (14)$$

- symmetric row/column update – a special case of rank-2 update,

$$\mathbf{W} = \begin{bmatrix} 0 & \dots & u_1 & \dots & 0 \\ \vdots & \ddots & \vdots & \ddots & \vdots \\ u_1 & \dots & u_p & \dots & u_n \\ \vdots & \ddots & \vdots & \ddots & \vdots \\ 0 & \dots & u_n & \dots & 0 \end{bmatrix} \equiv \mathbf{e}(p)\mathbf{u}^T + \mathbf{u}\mathbf{e}(p)^T, \quad (15)$$

where $\mathbf{e}(p)$ is the unit vector in the p -direction.

1.3.1 Rank-1 update

We assume that the size- n real symmetric matrix A to be diagonalized is given in the form of a diagonal matrix plus a rank-1 update,

$$\mathbf{A} = \mathbf{D} + \sigma\mathbf{u}\mathbf{u}^T, \quad (16)$$

where \mathbf{D} is a diagonal matrix with diagonal elements $\{d_1, \dots, d_n\}$ and \mathbf{u} is a given vector. The diagonalization of such matrix can be done in $O(m^2)$ operations, where $m \leq n$ is the number of non-zero elements in the update vector \mathbf{u} , as compared to $O(n^3)$ operations for a general diagonalization [?].

The eigenvalue equation for the updated matrix reads

$$(\mathbf{D} + \sigma\mathbf{u}\mathbf{u}^T)\mathbf{q} = \lambda\mathbf{q}, \quad (17)$$

where λ is an eigenvalue and \mathbf{q} is the corresponding eigenvector. The equation can be rewritten as

$$(\mathbf{D} - \lambda\mathbf{1})\mathbf{q} + \sigma\mathbf{u}\mathbf{u}^T\mathbf{q} = 0. \quad (18)$$

Multiplying from the left with $\mathbf{u}^T(\mathbf{D} - \lambda\mathbf{1})^{-1}$ gives

$$\mathbf{u}^T\mathbf{q} + \mathbf{u}^T(\mathbf{D} - \lambda\mathbf{1})^{-1}\sigma\mathbf{u}\mathbf{u}^T\mathbf{q} = 0. \quad (19)$$

Finally, dividing by $\mathbf{u}^T\mathbf{q}$ leads to the (scalar) *secular equation* (or *characteristic equation*) in λ ,

$$1 + \sum_{i=1}^m \frac{\sigma u_i^2}{d_i - \lambda} = 0, \quad (20)$$

where the summation index counts the m non-zero components of the update vector \mathbf{u} . The m roots of this equation determine the (updated) eigenvalues¹.

¹Multiplying this equation by $\prod_{i=1}^m (d_i - \lambda)$ leads to an equivalent polynomial equation of the order m , which has exactly m roots.

Finding a root of a rational function requires an iterative technique, such as the Newton-Raphson method. Therefore diagonalization of an updated matrix is still an iterative procedure. However, each root can be found in $O(1)$ iterations, each iteration requiring $O(m)$ operations. Therefore the iterative part of this algorithm needs $O(m^2)$ operations.

Finding roots of this particular secular equation can be simplified by utilizing the fact that its roots are bounded by the eigenvalues d_i of the matrix \mathbf{D} . Indeed if we denote the roots as $\lambda_1, \lambda_2, \dots, \lambda_n$ and assume that $\lambda_i \leq \lambda_{i+1}$ and $d_i \leq d_{i+1}$, it can be shown that

1. if $\sigma \geq 0$,

$$d_i \leq \lambda_i \leq d_{i+1}, \quad i = 1, \dots, n-1, \quad (21)$$

$$d_n \leq \lambda_n \leq d_n + \sigma \mathbf{u}^T \mathbf{u}; \quad (22)$$

2. if $\sigma \leq 0$,

$$d_{i-1} \leq \lambda_i \leq d_i, \quad i = 2, \dots, n, \quad (23)$$

$$d_1 + \sigma \mathbf{u}^T \mathbf{u} \leq \lambda_1 \leq d_1. \quad (24)$$

1.3.2 Symmetric row/column update

The matrix \mathbf{A} to be diagonalized is given in the form

$$\mathbf{A} = \mathbf{D} + \mathbf{e}(p)\mathbf{u}^T + \mathbf{u}\mathbf{e}(p)^T = \begin{bmatrix} d_1 & \dots & u_1 & \dots & 0 \\ \vdots & \ddots & \vdots & \ddots & \vdots \\ u_1 & \dots & d_p & \dots & u_n \\ \vdots & \ddots & \vdots & \ddots & \vdots \\ 0 & \dots & u_n & \dots & d_n \end{bmatrix}, \quad (25)$$

where \mathbf{D} is a diagonal matrix with diagonal elements $\{d_i | i = 1, \dots, n\}$, $\mathbf{e}(p)$ is the unit vector in the p -direction, and \mathbf{u} is a given update vector where the p -th element can be assumed to equal zero, $u_p = 0$, without loss of generality. Indeed, if the element is not zero, one can simply redefine $d_p \rightarrow d_p + 2u_p$, $u_p \rightarrow 0$.

The eigenvalue equation for matrix \mathbf{A} is given as

$$(D - \lambda)\mathbf{x} + \mathbf{e}(p)\mathbf{u}^T \mathbf{x} + \mathbf{u}\mathbf{e}(p)^T \mathbf{x} = 0, \quad (26)$$

where \mathbf{x} is an eigenvector and λ is the corresponding eigenvalue. The component number p of this vector-equation reads

$$(d_p - \lambda)x_p + \mathbf{u}^T \mathbf{x} = 0, \quad (27)$$

while the component number $k \neq p$ reads

$$(d_k - \lambda)x_k + u_k x_p = 0, \quad (28)$$

Dividing the last equation by $(d_k - \lambda)$, multiplying from the left with $\sum_{k=1}^n u_k$, substituting $\mathbf{u}^T \mathbf{x}$ using equation (27) and dividing by x_p gives the secular equation,

$$-(d_p - \lambda) + \sum_{k \neq p}^n \frac{u_k^2}{d_k - \lambda} = 0, \quad (29)$$

which determines the updated eigenvalues.

$$(30)$$

1.3.3 Symmetric rank-2 update

A symmetric rank-2 update can be represented as two consecutive rank-1 updates,

$$\mathbf{u}\mathbf{v}^T + \mathbf{v}\mathbf{u}^T = \mathbf{a}\mathbf{a}^T - \mathbf{b}\mathbf{b}^T, \quad (31)$$

where

$$\mathbf{a} = \frac{1}{\sqrt{2}}(\mathbf{u} + \mathbf{v}), \quad \mathbf{b} = \frac{1}{\sqrt{2}}(\mathbf{u} - \mathbf{v}). \quad (32)$$

The eigenvalues can then be found by applying the rank-1 update method twice.

1.4 Singular Value Decomposition

Singular Value Decomposition (SVD) is a factorization of a (here real square, for simplicity) matrix \mathbf{A} in the form

$$\mathbf{A} = \mathbf{U}\mathbf{S}\mathbf{V}^T, \quad (33)$$

where \mathbf{S} is a diagonal matrix, and \mathbf{U} and \mathbf{V} are orthogonal matrices ($\mathbf{U}^T \mathbf{U} = \mathbf{V}^T \mathbf{V} = \mathbf{1}$).

One algorithm to perform SVD is the *one-sided Jacobi SVD algorithm* which is a generalization of the Jacobi eigenvalue algorithm. Instead of just Jacobi transformation one first performs Givens rotation and then Jacobi transformation.

For a 2×2 matrix the one-sided Jacobi SVD transformation is given as following: first, one makes a Givens rotation to symmetrize two off-diagonal elements,

$$\mathbf{A} \equiv \begin{bmatrix} w & x \\ y & z \end{bmatrix} \rightarrow \mathbf{A}' = \mathbf{G}^T \mathbf{A} = \begin{bmatrix} \cos(\theta) & -\sin(\theta) \\ \sin(\theta) & \cos(\theta) \end{bmatrix} \begin{bmatrix} w & x \\ y & z \end{bmatrix} = \begin{bmatrix} a & b \\ b & c \end{bmatrix}, \quad (34)$$

where the rotation angle $\theta = \text{atan2}(x - y, w + z)$; and, second, one makes a usual Jacobi transformation to eliminate the off-diagonal elements,

$$\mathbf{A}' \rightarrow \mathbf{J}^T \mathbf{A}' \mathbf{J} = \begin{bmatrix} \cos(\phi) & -\sin(\phi) \\ \sin(\phi) & \cos(\phi) \end{bmatrix} \begin{bmatrix} a & b \\ b & c \end{bmatrix} \begin{bmatrix} \cos(\phi) & \sin(\phi) \\ -\sin(\phi) & \cos(\phi) \end{bmatrix} = \begin{bmatrix} d_1 & 0 \\ 0 & d_1 \end{bmatrix}. \quad (35)$$

In summary, the elementary one-sided Jacobi SVD transformation of a matrix \mathbf{A} is given as

$$\mathbf{A} \rightarrow \mathbf{J}^T \mathbf{G}^T \mathbf{A} \mathbf{J}, \quad (36)$$

$$\mathbf{U} \rightarrow \mathbf{U} \mathbf{G} \mathbf{J}, \quad (37)$$

$$\mathbf{V} \rightarrow \mathbf{V} \mathbf{J}, \quad (38)$$

where the Givens rotation with matrix \mathbf{G} symmetrizes two off-diagonal elements and the Jacobi transformation with matrix \mathbf{J} eliminates these off-diagonal elements.

One applies this transformation to all off-diagonal elements in sweeps until the diagonal elements change no more, just like in the cyclic Jacobi eigenvalue algorithm.

Table 1: Jacobi diagonalization in C. The header `matrix.h` is supposed to define the self-explanatory functions `vector_get`, `vector_set`, `matrix_get`, `matrix_set`, and `matrix_set_identity`.

```

#include<math.h>
#include<matrix.h>

int jacobi(matrix* A, vector* e, matrix* V){
    // Jacobi diagonalization.
    // Upper triangle of A is destroyed.
    // e and V accumulate eigenvalues and eigenvectors
    int changed, rotations=0, n=A->size1;
    for(int i=0;i<n;i++)vector_set(e,i,matrix_get(A,i,i));
    matrix_set_identity(V);
    do{
        changed=0; int p,q;
        for(p=0;p<n;p++)for(q=p+1;q<n;q++){
            double app=vector_get(e,p);
            double aqq=vector_get(e,q);
            double apq=matrix_get(A,p,q);
            double phi=0.5*atan2(2*apq,aqq-app);
            double c = cos(phi);
            double s = sin(phi);
            double app1=c*c*app-2*s*c*apq+s*s*aqq;
            double aqq1=s*s*app+2*s*c*apq+c*c*aqq;
            if(app1!=app || aqq1!=aqq){
                changed=1; rotations++;
                vector_set(e,p,app1);
                vector_set(e,q,aqq1);
                matrix_set(A,p,q,0.0);
                for(int i=0;i<p;i++){
                    double aip=matrix_get(A,i,p);
                    double aiq=matrix_get(A,i,q);
                    matrix_set(A,i,p,c*aip-s*aiq);
                    matrix_set(A,i,q,c*aiq+s*aip);
                }
                for(int i=p+1;i<q;i++){
                    double api=matrix_get(A,p,i);
                    double aiq=matrix_get(A,i,q);
                    matrix_set(A,p,i,c*api-s*aiq);
                    matrix_set(A,i,q,c*aiq+s*api);
                }
                for(int i=q+1;i<n;i++){
                    double api=matrix_get(A,p,i);
                    double aqi=matrix_get(A,q,i);
                    matrix_set(A,p,i,c*api-s*aqi);
                    matrix_set(A,q,i,c*aqi+s*api);
                }
                for(int i=0;i<n;i++){
                    double vip=matrix_get(V,i,p);
                    double viq=matrix_get(V,i,q);
                    matrix_set(V,i,p,c*vip-s*viq);
                    matrix_set(V,i,q,c*viq+s*vip);
                }
            }
        }
    }while(changed!=0);
    return rotations;
}

```