Eigenvalues and eigenvectors

Introduction

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

$$A\mathbf{v} = \lambda\mathbf{v} \; . \tag{1}$$

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

$$VV^T = V^T V = \mathbf{1} \; , \tag{2}$$

in which the matrix is diagonal,

$$V^T A V = \begin{bmatrix} \lambda_1 & 0 & \cdots & 0 \\ 0 & \lambda_2 & & \vdots \\ \vdots & & \ddots & \\ 0 & \cdots & & \lambda_n \end{bmatrix} \; . \tag{3}$$

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.

Similarity transformations

Orthogonal transformations,

$$A \to Q^T A Q \; , \tag{4}$$

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

$$A \to S^{-1} A S \; , \tag{5}$$

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.

Jacobi eigenvalue algorithm

Jacobi eigenvalue algorithm is an iterative method to calculate the eigenvalues and eigenvectors of a real symmetric matrix by a sequence of Jacobi rotations.

Jacobi rotation is an orthogonal transformation which zeroes a pair of the off-diagonal elements of a (real symmetric) matrix A,

$$A \to A' = J(p,q)^T A J(p,q) \;\; : \;\; A'_{pq} = A'_{qp} = 0 \; . \tag{6}$$

The orthogonal matrix $J(p,q)$ which eliminates the element A_{pq} is called the Jacobi rotation matrix. It is equal identity matrix except for the four elements with indices pp, pq, qp, and qq,

$$J(p,q) = \begin{bmatrix} 1 & & & & & & \\ & \ddots & & & & 0 & \\ & & \cos\phi & \cdots & \sin\phi & & \\ & & \vdots & \ddots & \vdots & & \\ & & -\sin\phi & \cdots & \cos\phi & & \\ & 0 & & & & \ddots & \\ & & & & & & 1 \end{bmatrix} \begin{matrix} \\ \\ \leftarrow \text{row } p \\ \\ \leftarrow \text{row } q \\ \\ \end{matrix} \; . \tag{7}$$

Or explicitly,

$$\begin{aligned} &J(p,q)_{ij} = \delta_{ij} \; \forall \; ij \notin \{pq, qp, pp, qq\} \; ; \\ &J(p,q)_{pp} = \cos\phi = J(p,q)_{qq} \; ; \\ &J(p,q)_{pq} = \sin\phi = -J(p,q)_{qp} \; . \end{aligned} \tag{8}$$

After a Jacobi rotation, $A \rightarrow A' = J^T A J$, the matrix elements of A' become

$$\begin{aligned} A'_{ij} &= A_{ij} \; \forall \; i \neq p, q \wedge j \neq p, q \\ A'_{pi} &= A'_{ip} = cA_{pi} - sA_{qi} \; \forall \; i \neq p, q \; ; \\ A'_{qi} &= A'_{iq} = sA_{pi} + cA_{qi} \; \forall \; i \neq p, q \; ; \\ A'_{pp} &= c^2 A_{pp} - 2scA_{pq} + s^2 A_{qq} \; ; \\ A'_{qq} &= s^2 A_{pp} + 2scA_{pq} + c^2 A_{qq} \; ; \\ A'_{pq} &= A'_{qp} = sc(A_{pp} - A_{qq}) + (c^2 - s^2)A_{pq} \; , \end{aligned} \tag{9}$$

where $c \equiv \cos\phi$, $s \equiv \sin\phi$. The angle ϕ is chosen such that after rotation the matrix element A'_{pq} is zeroed,

$$\cot(2\phi) = \frac{A_{qq} - A_{pp}}{2A_{pq}} \; \Rightarrow \; A'_{pq} = 0 \; . \tag{10}$$

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

The eigenvectors can be calculated as $V = \mathbf{1}J_1 J_2 ...$, where J_i are the successive Jacobi matrices. At each stage the transformation is

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

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

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 QR[$\mathbf{A}$]

we do QR$[\mathbf{A} - s\mathbf{1}]$ and then $\mathbf{A} \to \mathbf{RQ} + s\mathbf{1}$. 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.

Eigenvalues of updated matrix

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

$$\mathbf{A} = \mathbf{D} + \mathbf{W} \,, \tag{12}$$

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,
$$\mathbf{W} = \mathbf{u}\mathbf{u}^T \,, \tag{13}$$
where $\mathbf{u}$ is a columnt-vector;
- symmetric rank-2 update,
$$\mathbf{W} = \mathbf{u}\mathbf{v}^T + \mathbf{v}\mathbf{u}^T \,; \tag{14}$$
- symmetric row/column update,
$$\mathbf{W} = \begin{bmatrix} 0 & \dots & u_1 & \dots & 0 \\ \vdots & \ddots & \vdots & \ddots & \vdots \\ u_1 & \dots & u_p & \dots & u_n \\ \vdots & \ddots & \vdots & \ddots & \vdots \\ 0 & \dots & u_n & \dots & 0 \end{bmatrix} \equiv \mathbf{e}_p \mathbf{u}^T + \mathbf{u}\mathbf{e}_p^T \,, \tag{15}$$
where $\mathbf{e}_p$ is the unit vector in the p-direction.

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,

$$\mathbf{A} = \mathbf{D} + \sigma \mathbf{u}\mathbf{u}^T \,, \tag{16}$$

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 $m \le n$ 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

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

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

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

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

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

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

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

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) eigenvalues[1].

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 $\lambda_1, \lambda_2, \ldots, \lambda_n$ and assume that $\lambda_i \le \lambda_{i+1}$ and $d_i \le d_{i+1}$, it can be shown that

1. if $\sigma \ge 0$,

$$d_i \le \lambda_i \le d_{i+1} \,, \qquad i = 1, \ldots, n-1 \,, \tag{21}$$
$$d_n \le \lambda_n \le d_n + \sigma \mathbf{u}^T\mathbf{u} \,; \tag{22}$$

2. if $\sigma \le 0$,

$$d_{i-1} \le \lambda_i \le d_i \,, \qquad i = 2, \ldots, n \,, \tag{23}$$
$$d_1 + \sigma \mathbf{u}^T\mathbf{u} \le \lambda_1 \le d_1 \,. \tag{24}$$

Symmetric rank-2 update

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

$$\mathbf{u}\mathbf{v}^T + \mathbf{v}\mathbf{u}^T = \mathbf{a}\mathbf{a}^T - \mathbf{b}\mathbf{b}^T \,, \tag{25}$$

where

$$\mathbf{a} = \frac{1}{\sqrt{2}}(\mathbf{u} + \mathbf{v}) \,, \; \mathbf{b} = \frac{1}{\sqrt{2}}(\mathbf{u} - \mathbf{v}) \,. \tag{26}$$

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

[1] Multiplying 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.

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);
      int i; double aip, api, aiq, aqi, vip, viq;
      for(i=0;i<p;i++){
        aip=matrix_get(A,i,p);
        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(i=p+1;i<q;i++){
        api=matrix_get(A,p,i);
        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(i=q+1;i<n;i++){
        api=matrix_get(A,p,i);
        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(i=0;i<n;i++){
        vip=matrix_get(V,i,p);
        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;
}
```