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 $\lambda_1, \dots, \lambda_n$, and its (orthogonalized) eigenvectors $V = \{\mathbf{v}_1, \dots, \mathbf{v}_n\}$ form a full basis,

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

in which the matrix is diagonal,

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

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

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

Similarity transformations

Orthogonal transformations,

$$A \to Q^TAQ \ , \tag{4}$$

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

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

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,

$$A \to A' = J(p,q)^TAJ(p,q) \ : \ A'_{pq} = A'_{qp} = 0 \ . \tag{6}$$

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,

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

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, $A \to A' = J^T A J$, 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 elemens 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

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

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

C++ implementation with armadillo matrices

Table 1: Jacobi diagonalization

```
#include<math.h>
#include<armadillo>
using namespace arma;

int jacobi(mat & A, vec & e, mat & V){
// Jacobi diagonalization.
// Upper triangle of A is destroyed.
// e and V accumulate eigenvalues and eigenvectors
int changed, rotations=0, n=A.n_rows;
e=A.diag();
V.eye();
do{
        changed=0; int p,q;
        for(p=0;p<n;p++)for(q=p+1;q<n;q++){
                double app=e(p), aqq=e(q), apq=A(p,q);
                double phi=0.5*atan2(2*apq,aqq-app);
                double c = cos(phi), 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++;
                        e(p)=app1;
                        e(q)=aqq1;
                        A(p,q)=0.0;
                        int i; double aip, api, aiq, aqi, vip, viq;
                        for(i=0;i<p;i++){
                                aip=A(i,p);
                                aiq=A(i,q);
                                A(i,p)=c*aip-s*aiq;
                                A(i,q)=c*aiq+s*aip;
                        }
                        for(i=p+1;i<q;i++){
                                api=A(p,i);
                                aiq=A(i,q);
                                A(p,i)=c*api-s*aiq;
                                A(i,q)=c*aiq+s*api;
                        }
                        for(i=q+1;i<n;i++){
                                api=A(p,i);
                                aqi=A(q,i);
                                A(p,i)=c*api-s*aqi;
                                A(q,i)=c*aqi+s*api;
                        }
                        for(i=0;i<n;i++){
                                vip=V(i,p);
                                viq=V(i,q);
                                V(i,p)=c*vip-s*viq;
                                V(i,q)=c*viq+s*vip;
                        }
                }
        }
}while(changed!=0);
return rotations;
}
```