1 Matrix Diagonalization

A vector $\mathbf{v}$ is called an *eigenvector* of a matrix A with an *eigenvalue* λ, if $A\mathbf{v} = \lambda\mathbf{v}$. Matrix diagonalization means finding all eigenvalues λ_i and (optionally) eigenvectors $\mathbf{v}_i$ of the matrix A. If A is hermitian, $A^\dagger = A$, then its eigenvalues are real and its eigenvectors $V = \{\mathbf{v}_1, \ldots, \mathbf{v}_n\}$ form a full basis where the matrix A is diagonal, $V^T AV = \Lambda$, where Λ is a diagonal matrix with eigenvalues λ_i along the diagonal. In the following we shall only consider real symmetric matrices which have real eigenvalues.

1.1 Similarity transformations

Orthogonal transformations, $A \to Q^T AQ$, where $Q^T Q = \mathbf{1}$, and, generally, similarity transformations, $A \to S^{-1}AS$, 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.

1.1.1 Jacobi rotation

Jacobi rotation is an orthogonal transformation where the transformation matrix, called Jacobi rotation matrix $J(p, q)$, is equal unity matrix except for the elements

$$\begin{aligned} J(p,q)_{pp} = J(p,q)_{qq} &= \cos\phi, \\ J(p,q)_{pq} = -J(p,q)_{qp} &= \sin\phi. \end{aligned} \tag{1}$$

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

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

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,

$$\cot(2\phi) = \frac{A_{qq} - A_{pp}}{2A_{pq}}\ . \tag{3}$$

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. (Refinement: if the element is "small enough" the rotation can be skipped.)

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 moduli 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, $\max |\Delta A_{ii}| < \epsilon$.

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

$$\begin{aligned} V_{ij} &\to V_{ij}\ ,\ j \neq p,q \\ V_{ip} &\to cV_{ip} - sV_{iq} \\ V_{iq} &\to sV_{ip} + cV_{iq} \end{aligned} \tag{4}$$

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 Power iteration methods

1.2.1 Power method

Power method is an iterative method to calculate an eigenvalue and the corresponding eigenvector using the iteration

$$\mathbf{x}_{i+1} = A\mathbf{x}_i\ . \tag{5}$$

The iteration converges to the eigenvector of the largest eigenvalue. The eigenvalue can be estimated using the *Rayleigh quotient*

$$\lambda[\mathbf{x}_i] = \frac{\mathbf{x}_i^T A\mathbf{x}_i}{\mathbf{x}_i^T\mathbf{x}_i} = \frac{\mathbf{x}_{i+1}^T\mathbf{x}_i}{\mathbf{x}_i^T\mathbf{x}_i}. \tag{6}$$

1.2.2 Inverse power method

The iteration with the inverse matrix

$$\mathbf{x}_{i+1} = A^{-1}\mathbf{x}_i \tag{7}$$

converges to the smallest eigenvalue of matrix A. Alternatively, the iteration

$$\mathbf{x}_{i+1} = (A - s)^{-1}\mathbf{x}_i \tag{8}$$

converges to an eigenvalue closest to the given number s.

1.2.3 Inverse iteration method

Inverse iteration method is the refinement of the inverse power method where the trick is not to invert the matrix in (8) but rather solve the linear system

$$(A - \lambda)\mathbf{x}_{i+1} = \mathbf{x}_i \tag{9}$$

using eg QR decomposition.

One can update the estimate for the eigenvalue using the Rayleigh quotient $\lambda[\mathbf{x}_i]$ after each iteration and get faster convergence for the price of $O(n^3)$ operations per QR-decomposition; or one can instead make more iterations (with $O(n^2)$ operations per iteration) using the same matrix $(A-\lambda)$. The optimal strategy is probably an update after several iterations.

1.3 JavaScript implementation

1.3.1 Jacobi diagonalization

```
function jacobi(A){
// Jacobi diagonalization.
// Upper triangle of A is destroyed.
// V accumulates eigenvectors.
  var V=[[(i==j?1:0) for (i in A)] for(j in
      A)]
  var tiny = 1e-12
  var max_rotations=5*n*n;
  var rotated, rotations=0;
  do{
    rotated=0;
    for(var row=0;row<n;row++)
    for(var col=row+1;col<n;col++){
      if( tiny*Math.abs(A[row][row])<Math.
          abs(A[col][row])
          || tiny*Math.abs(A[col][col])<
              Math.abs(A[col][row])){
        rotated=1;
        rotations++;
        if(rotations>max_rotations){
          print("max_rotations reached: ",
              rotations);
          return undefined;}
        rotate(row,col,A,V);
      }
    }
  }while(rotated==1);
  return [V,rotations];
}
```

1.3.2 Jacobi rotation

```
function rotate(p,q,A,V){
// Jacobi rotation eliminating A_pq.
// Only upper triangle of A is updated.
// The matrix of eigenvectors V is also
     updated.
  if(q<p) [p,q]=[q,p]
  var app = A[p][p];
  var aqq = A[q][q];
  var apq = A[q][p];
  var phi=0.5*Math.atan2(2*A[q][p],A[q][q]-
      A[p][p]);
  var c=Math.cos(phi), s=Math.sin(phi);
  A[p][p] = c * c * app + s * s * aqq - 2 *
       s * c * apq;
  A[q][q] = s * s * app + c * c * aqq + 2 *
       s * c * apq;
//  A[q][p] = ( c * c - s * s ) * apq + s *
     c * ( app - aqq );
  A[q][p]=0;

  for(var i=0;i<p;i++){
    var aip=A[p][i],aiq=A[q][i];
    A[p][i] = c*aip-s*aiq;
    A[q][i] = c*aiq+s*aip;
  }

  for(var i=p+1;i<q;i++){
    var api=A[i][p],aiq=A[q][i];
    A[i][p] = c*api-s*aiq;
    A[q][i] = c*aiq+s*api;
  }

  for(var i=q+1;i<n;i++){
    var api=A[i][p],aqi=A[i][q];
    A[i][p] = c*api-s*aqi;
    A[i][q] = c*aqi+s*api;
  }

  if(typeof(V)!=undefined) for(var i=0;i<n;
      i++){
    var vip=V[p][i],viq=V[q][i];
    V[p][i] = c*vip-s*viq;
    V[q][i] = c*viq+s*vip;
  }
  return 0;
}
```