EIGEN VALUES

AI24BTECH11024-Pappuri Prahladha

Eigen values of a matrix

Eigenvalues are scalars that provide insight into the properties of a matrix, particularly in relation to how it acts on vectors. When a matrix acts on an eigenvector, the output is simply a scaled version of the original eigenvector. The eigenvalue is the factor by which the eigenvector is scaled.

So a square matrix A, an eigenvalue λ and its corresponding eigenvector $\mathbf{v}$ satisfy the following equation:

$$A\mathbf{v} = \lambda\mathbf{v}$$

Where:

- A is an $n \times n$ square matrix,
- $\mathbf{v}$ is an eigenvector (a non-zero vector),
- λ is the eigenvalue (a scalar).

Finding eigen values(general process)

To find the eigenvalues of a matrix, we need to solve the characteristic equation:

$$\det(A - \lambda I) = 0$$

The equation $\det(A - \lambda I) = 0$ is called the characteristic equation, and solving it gives the eigenvalues $\lambda_1, \lambda_2, \ldots, \lambda_n$.

Geometric Interpretation

- Eigenvectors represent directions that are invariant under the matrix transformation.
- Eigenvalues represent the scaling factor along those directions.

For example:

- If A represents a transformation that rotates vectors, then the eigenvalues are typically complex numbers (if the rotation isn't just a scaling).
- If A represents a scaling transformation, the eigenvalues are real numbers that describe the scaling factor in the direction of the corresponding eigenvectors.

Types of Eigenvalues

- **Real Eigenvalues**: For symmetric matrices, the eigenvalues are always real numbers.
- **Complex Eigenvalues**: For non-symmetric matrices, the eigenvalues can be complex numbers. A matrix may have complex eigenvalues if it represents a rotation or some non-diagonalizable transformation.
- **Positive and Negative Eigenvalues**:
  - Positive eigenvalues indicate a scaling in the same direction as the eigenvector.
  - Negative eigenvalues indicate a reflection in addition to scaling.

- **Zero Eigenvalues**: If a matrix has a zero eigenvalue, it is **singular**, meaning it does not have an inverse. This indicates that the transformation reduces the dimension of the vector space (e.g., it squashes vectors into a lower-dimensional subspace).

Here we will see an algorithm for computing eigen values
Jacobi algorithm

- Gurantees for real symmetric matrices.
- Gurantees for non-real symmetric hermitian matrices.
- Not gurantees for non-real symmetric non hermitian matrices.

In this method we will apply some sort of similarity transformations on the given matrix such that after a sequence of a similarity transformations the matrix convert into a diagonal matrix and from the diagonal matrix we can see the eigenvalue directly as the diagonal element. Furthermore the sequence will contain the information about the eigenvectors of the matrix. This gives a guarantee for finding the eigenvalues of real symmetric matrices as well as the eigenvectors for the real symmetric matrix.

A 2×2 rotation matrix with angle θ is given by:

$$R(\theta) = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix}$$

An $n \times n$ rotation matrix $J(p, q, \theta)$ has the form:

$$J(p,q,\theta) = \begin{bmatrix} \ddots & & & & \\ & \cos\theta & \cdots & -\sin\theta & \\ & \vdots & \ddots & \vdots & \\ & \sin\theta & \cdots & \cos\theta & \\ & & & & \ddots \end{bmatrix}$$

Here, $\cos\theta$ and $\sin\theta$ terms appear in the pth and qth rows and columns, and all other elements are zero.

The matrix $J(p, q, \theta)$ is known as Jacobiâs rotation. The matrix $J(p, q, \theta)$ is applied to symmetric matrix A as a similarity transformation which rotates row and column p and q of A through an angle θ so that (p, q) and (q, p) entries become zero.Think of the Jacobi rotation as ”rotating” the axes corresponding to rows p and q in a way that aligns the matrix to progressively eliminate off diagonal elements while preserving its eigenvalues. So let us denote this similarity transformation as:

$$A' = J^T A J,$$

where A' is the transformed matrix.

The relation between the elements of matrix A and A' are given by the formulas:

$$\begin{aligned}
a' jp &= cajp - sa_{jq} \quad \text{when } j \neq p \text{ and } j \neq q \\
a' jq &= sajp + ca_{jq} \quad \text{when } j \neq p \text{ and } j \neq q \\
a' pp &= c^2 app + s^2 a_{qq} - 2csa_{pq} \\
a' qq &= s^2 app + c^2 a_{qq} + 2csa_{pq} \\
a' pq &= (c^2 - s^2)apq + cs(a_{pp} - a_{qq})
\end{aligned}$$

As we want to make the off-diagonal element of the new matrix A' zero, we can write that condition:

$$(c^2 - s^2)a_{pq} + cs(a_{pp} - a_{qq} = 0)$$

Here $c = \cos\theta$ and $s = \sin\theta$, thus if $a_{pq} \neq 0$, then the above condition gives us

$$\cot 2\theta = \frac{(c^2 - s^2)}{2cs} = \frac{(a_{qq} - a_{pp})}{2a_{pq}}$$

Hence the value of the θ from the above equation is given as:

$$\theta = \frac{1}{2} tan^{-1} (\frac{2cs}{c^2 - s^2})$$

and hence the angle for the similarity transformation can be derived. The value of c and s will be given by the formula as

$$c = \frac{1}{\sqrt{1 + t^2}} \quad \text{and} \quad s = c \cdot t$$

If we are doing with the hermitian matrix with the complex numbers then θ is calculated using real parts and $c = \cos\theta + i\sin\theta$ and $= -ic$

Following steps are adopted in the Jacobi method

- Find the p^{th} and q^{th} row and column which correspond to the off-diagonal element having the highest value.(Because These large elements have the most significant effect on the matrix's non-diagonal structure, meaning they contribute most to the ”non-diagonal-ness” of the matrix. By focusing on the largest off-diagonal element, we can maximize the ”correction” that each rotation makes, improving convergence speed.)
- Compute the Jacobi matrix after calculating the angle of similarity rotation.
- Apply the Jacobi matrix to the matrix as in the way mentioned above.
- Repeat the process until the matrix is converted completely into a diagonal matrix. The diagonal elements will be the eigenvalues.
- The eigenvectors will be the columns of the Jacobi matrix.

Writing C code

- This code gives the eigen values for real symmetric matries.
- This code uses Jacobi algorithm.

- we can make function for calculating or diagonalizing the matrix, hence diagnol elements are eigen values.
- we will write the above equation in this function.
- Also in this function we will update a matrix in each iteration for storing eigen vectors.
- also a function for printing eigen vectors.

<u>**The C code**</u>

```c
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

#define limit 1e-9
#define maxiter 100

void eigenvalues(double *A, double *V, int n) {
    int iterations = 0;
    double max;

    do {
        max = 0.0;
        int p, q;
        for (int i = 0; i < n - 1; i++) {
            for (int j = i + 1; j < n; j++) {
                if (fabs(A[i * n + j]) > max) {
                    max = fabs(A[i * n + j]);
                    p = i;
                    q = j;
                }
            }
        }

        if (max < limit) {
            break;
        }
        double theta = 0.5 * atan2(2 * A[p * n + q], A[q * n + q] -
            A[p * n + p]);
        double c = cos(theta);
        double s = sin(theta);
        double App = A[p * n + p];
        double Aqq = A[q * n + q];
        double Apq = A[p * n + q];

        A[p * n + p] = c * c * App + s * s * Aqq - 2 * s * c * Apq;
        A[q * n + q] = s * s * App + c * c * Aqq + 2 * s * c * Apq;
        A[p * n + q] = A[q * n + p] = 0.0;
        for (int i = 0; i < n; i++) {
            if (i != p && i != q) {
                double Aip = A[i * n + p];
                double Aiq = A[i * n + q];
                A[i * n + p] = A[p * n + i] = c * Aip - s * Aiq;
                A[i * n + q] = A[q * n + i] = s * Aip + c * Aiq;
            }
            double Vip = V[i * n + p];
            double Viq = V[i * n + q];
            V[i * n + p] = c * Vip - s * Viq;
            V[i * n + q] = s * Vip + c * Viq;
        }
        iterations++;
    } while (max > limit && iterations < maxiter);

    if (iterations >= maxiter) {
        printf("Maximum iterations reached without full
            convergence.\n");
    }
}

void printMatrix(double *matrix, int n) {
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            printf("%lf ", matrix[i * n + j]);
        }
        printf("\n");
    }
}

int main() {
    int n;
    printf("Enter the dimension of the matrix: ");
    scanf("%d", &n);
    double *A = (double *)malloc(n * n * sizeof(double));
    double *V = (double *)malloc(n * n * sizeof(double));
    printf("Enter the elements of the %dx%d symmetric matrix:\n", n,
        n);
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            scanf("%lf", &A[i * n + j]);
            if (i == j) {
                V[i * n + j] = 1.0;
            } else {
                V[i * n + j] = 0.0;
            }
        }
    }

    eigenvalues(A, V, n);

    printf("Eigenvalues:\n");
    for (int i = 0; i < n; i++) {
        printf("%lf ", A[i * n + i]);
    }
    printf("\n\nEigenvectors:\n");
    printMatrix(V, n);
    free(A);
    free(V);
    return 0;
}
```

The above code works for real real symmetric matrices.
If we want to do for hermitian matrices also then replace include complex.h library and use c and s as stated above and with slight modification in the code.

INPUT-Format
suppose you want give the 2 by 2 matrix then give input as 1 2 3 4 where 1,2,3,4 are elements of the matrix of corresponding rows.simply write the each row one after another.

TIME-COMPLEXITY

1. Finding the Largest Off-Diagonal Element
This part of the code iterates over the **upper triangle** of the matrix to find the largest off-diagonal element. For an $n \times n$ matrix, the number of off-diagonal elements is approximately $\frac{n(n-1)}{2}$. In each iteration of the algorithm, we scan all these elements to find the largest one. Therefore, the time complexity for finding the largest off-diagonal element is:

$$O(n^2) \quad \text{for each iteration.}$$

2. Rotation and Matrix Updates
After identifying the largest off-diagonal element, the algorithm updates the matrix elements (the A matrix and the V matrix). The updates for the matrix involve looping over the matrix to apply the Jacobi rotation formula. This involves two nested loops over the matrix, resulting in a time complexity of:

$$O(n^2) \quad \text{for each rotation.}$$

3. Number of Iterations; The Jacobi algorithm iterates until convergence (when the

largest off-diagonal element becomes smaller than a threshold, ϵ, or the maximum number of iterations is reached). In the worst case, the number of iterations required for convergence can be up to $O(n^2)$, although generally, the number of iterations is often much smaller. Hence, the total number of iterations is at most:

$$O(n^2).$$

Total Time Complexity;
Now, combining these factors:
- In each iteration of the Jacobi method, we perform $O(n^2)$ operations for both finding the largest off-diagonal element and updating the matrix.
- The number of iterations is $O(n^2)$ in the worst case.

Thus, the overall time complexity of the Jacobi eigenvalue algorithm in this code is:

$$O(n^4).$$

The algorithm's time complexity grows rapidly with the size of the matrix, making it less efficient for very large matrices compared to other eigenvalue algorithms like the QR decomposition or Divide-and-Conquer methods.
So if you want to make it more efficient use the QR decomposition or Divide-and-Conquer method.

Different Algorithms - Comparision

Characteristic Polynomial Method

This method can be applied to any matrix for eigenvalues, but it becomes computationally tough for large matrices. Additionally, it provides less accurate answers and is numerically unstable for larger matrices.

QR Algorithm

This is applicable to all types of matrices and is numerically stable. However, it is computationally intensive for large matrices and not suitable for sparse matrices.

Jacobi Method

This method gives accurate answers for real symmetric and Hermitian matrices. However, it is also computationally expensive, and its convergence rate is low.

Divide and Conquer Method

This method is efficient for symmetric matrices but not suitable for other types of matrices. Additionally, it is challenging to handle.

Lanczos Method

This method is efficient for large sparse matrices and is more memory-efficient compared to other methods. However, it is also restricted to symmetric matrices.

conclusion: We will prefer the method based on our matrix.