Computation of eigen values

EE24BTECH11060-Sruthi Bijili

Introduction

The main objective of this project is to write a code that should compute the Eigenvalues of the matrix. Eigenvalues and Eigenvectors are the fundamental concepts in linear algebra. An eigenvector is a non-zero vector x that remains in the same direction (though it may be stretched or shrunk) when the linear transformation represented by matrix A is applied to it. More formally, if λ is an eigenvalue, then the vector x is the corresponding eigenvector that satisfies the equation:

$$A\bar{x}=\lambda\bar{x}$$

The eigenvector remains in the same direction after the matrix transformation. The matrix only scales it by the factor λ. The equation can be written as:

$$A\bar{x}-\lambda I\bar{x}=0$$
$$(A-\lambda I)\bar{x}=0$$

Now, the possible solutions for the above equation are the eigenvector to be zero or $\det(A-\lambda I)=0$. As we need nonzero eigenvectors, we take $\det(A-\lambda I)=0$ as the solution. This is one of the simplest methods for computing eigenvalues of a matrix.

Algorithm Selection

There are several algorithms for computing eigenvalues, each with varying trade-offs in terms of time complexity, accuracy, and suitability for different types of matrices. Some of these algorithms are:

- QR Algorithm
- Power Iteration
- Lanczos Algorithm
- Jacobi Method

These are some of the algorithms that can be used to compute eigenvalues of the matrix. But each method has its own disadvantages. Here are some pros and cons for each algorithm:

QR Algorithm

Pros:

- Works for any square matrix (symmetric, non-symmetric).
- Guarantees convergence to eigenvalues and computes all of them.
- Simple and robust, with no need for preprocessing.

Cons:

- Time complexity of $O(n^3)$ per iteration, slow for large matrices.
- Requires $O(n^2)$ memory, inefficient for large matrices.
- Slow convergence for matrices with clustred or poorly separated eigenvalues.

Power Iteration

Pros:

- Simple and memory efficient, requires only the largest eigenvector.
- Fast for large, sparse matrices when only the largest eigenvalue is needed.
- Easy to implement with low computational overhead.

Cons:

- Only finds the largest eigenvalue, not suitable for computing all eigenvalues.
- Slow convergence if eigenvalues are not well-separated.
- Sensitive to the initial guess, can fail with non-dominant eigenvalues.

Lanczos Algorithm

Pros:

- Efficient for large, sparse matrices by exploiting sparsity.
- Fast convergence for symmetric matrices, especially for a few eigenvalues.
- Memory efficient, requiring only a subspace of the matrix.

Cons:

- Requires restarting to maintain orthogonality, adding complexity.
- Numerical instability due to loss of orthogonality with many iterations.
- Primarily suited for symmetric matrices, less effective for non-symmetric ones.

Jacobi Method

Pros:

- Exact eigenvalues for symmetric matrices, with guaranteed convergence.
- Simple to implement and understand, with straightforward iteration.
- Parallelizable, suitable for high-performance computing on symmetric matrices.

Cons:

- Slow convergence for large matrices, especially with many iterations.
- Inefficient for non-symmetric matrices, limited to symmetric cases.
- Time complexity of $O(n^3)$, impractical for large systems.

From all these algorithms, the best algorithm for general-purpose eigenvalue computation is the **QR algorithm**, as it can be applied to both symmetric and non-symmetric matrices and can handle complex eigenvalue cases.

QR Algorithm Description

Given a matrix A, decompose it into the product of an orthogonal matrix Q and an upper triangular matrix R, such that:

$$A = QR$$

Form a new matrix by updating A to the matrix $A' = RQ$. This new matrix is expected to have the same eigenvalues as A, but its structure improves with each iteration:

$$A' = RQ$$

Repeat the process by taking A' as the new A. As the iterations progress, the matrix converges to an upper triangular matrix, whose diagonal elements will be the eigenvalues of the original matrix $A.A'$ and all other matrices along the process have the same eigenvalues. So the main idea here is using repeated QR decomposition, we are trying to transform the matrix into an upper triangular matrix because the eigenvalues for a triangular matrix are its diagonal elements.
Most important part in this algorithm is the QR Decomposition. There are various methods to do it. I have looked into three methods, they are:

- Householder Reflections
- Givens Reflections
- Gram-schmidt Orthogonalization

Givens Rotations is numerically stable, is efficient but it is best used for only sparse matrices. Gram-Schmidt method even though is simple to implement, but only limited for small matrices, and it also numerically less stable.so,the best method is Householder Reflections,because of its high numerical stability,efficiency,and also applicable for dense matrices.

Householder Reflections

we have to compute A=QR, for that the elements below the diagonal of A is zero to make it as an upper triangular matrix. For this we use Householder matrices.
A Householder matrix $H_k$that zeros out elements below the diagonal in column k is given by:

$$H_k = I - 2vv^T$$

where $\vec{v}=\frac{\vec{u}}{||\vec{u}||}$ and $\vec{u}=\vec{r_k}-\begin{pmatrix} ||\vec{r_k}|| \\ 0 \\ . \\ . \\ 0 \end{pmatrix}$ where $\vec{r_k}$ is the column vector of the k^th column in R from the diagonal element.All the Householder matrices are orthogonal.So when we multiply all the holder matrices we get

$$H_n H_{n-1} \dots H_1 A = R$$

from this we get Q as $(H_n H_{n-1} \dots H_1)^T = Q$

Time Complexity

QR decomposition (Householder Reflections) requires $O(n^3)$ operations for an $n \times n$ matrix. While applying the algorithm itself, a matrix generally converges after $O(n)$ iterations. So, the overall time complexity of the QR algorithm for a general matrix is:

$$O(n^4)$$

In general, if the algorithm converges in k iterations, the overall time complexity is approximately:

$$O(kn^3)$$

I. C CODE

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
void qr_decompose(double **A, double **Q, double **R, int n)
{
for (int i = 0; i < n; i++){
for (int j = 0; j < n; j++)
Q[i][j] = A[i][j];
for (int j = 0; j < n; j++) {
   for (int k = 0; k < j; k++) {
R[i][j] = 0;
   }
}
double dot_product = 0;
for (int i = 0; i < n; i++) {
   dot_product += Q[i][k] * Q[i][j];
}
for (int i = 0; i < n; i++) {
    Q[i][j] -= dot_product * Q[i][k];
   }
}
double norm = 0;
for (int i = 0; i < n; i++) {
norm += Q[i][j] * Q[i][j];
}
norm = sqrt(norm);
for (int i = 0; i < n; i++) {
 Q[i][j] /= norm;
}
for (int k = 0; k < n; k++){
   R[k][j] = 0;
   for (int i = 0; i < n; i++) {
       R[k][j] += Q[i][k] * A[i][j];
    }
  }
 }
}
void qr_iteration(double **A, int n) {
 double *Q = (double *)malloc(n * sizeof(double *));
 double *R = (double *)malloc(n * sizeof(double *));
for (int i = 0; i < n; i++) {
  Q[i] = (double *)malloc(n * sizeof(double));
  R[i] = (double *)malloc(n * sizeof(double));
}

for (int iter = 0; iter < 1000; iter++) {
 qr_decompose(A, Q, R, n);

for (int i = 0; i < n; i++)
  for (int j = 0; j < n; j++) {
     A[i][j] = 0;}
     for (int k = 0; k < n; k++) {
      A[i][j] += R[i][k] * Q[k][j];
    }
  }
}

double off_diag_sum = 0;
for (int i = 0; i < n-1; i++)
 for (int j = i+1; j < n; j++)
    off_diag_sum += A[i][j] * A[i][j];
  }
 }
if (off_diag_sum < 1e-6) break;
}

for (int i = 0; i < n; i++) {
 printf("%f ",A[i][i]);
}
printf("\n");

for (int i = 0; i < n; i++) {
  free(Q[i]);
  free(R[i]);
 }
 free(Q);
 free(R);
}

int main() {
int n;
scanf("%d", &n);

double *A = (double *)malloc(n * sizeof(double *));
for (int i = 0; i < n; i++) {
A[i] = (double *)malloc(n * sizeof(double));
}
for (int i = 0; i < n; i++) {
  for (int j = 0; j < n; j++) {
    scanf("%lf", &A[i][j]);
  }
}

qr_iteration(A, n);

for (int i = 0; i < n; i++) {
  free(A[i]);
 }
free(A);
return 0;
}
```