EigenVales Calculation

AI24BTECH11033-Tanishq Rajiv Bhujbale

Contents

1 Introduction

Eigenvalues play a critical role in numerous scientific and engineering fields, including stability analysis, quantum mechanics, and principal component analysis. This report explores various algorithms for eigenvalue calculation, implements one, and analyzes its performance.

2 Problem Statement

The task is to implement a program to compute the eigenvalues of a given matrix without relying on external libraries for matrix computations. This report documents the algorithm selection, implementation, and performance analysis.

3 Algorithm Selection

3.1 Overview of Algorithms

1) Power Iteration:
   Complexity: $O(n^2)$ per iteration.
   - **Pros**: Simple to implement, works well for finding the largest eigenvalue.
   - **Cons**: Slow convergence; ineffective for finding all eigenvalues.
2) QR Algorithm:
   Complexity: $O(n^3)$.
   - **Pros**: Reliable, widely used, and capable of finding all eigenvalues.
   - **Cons**: High computational cost, particularly for dense matrices.
3) Jacobi Method:
   Complexity: $O(n^3)$.
   - **Pros**: Effective for symmetric matrices, high accuracy.
   - **Cons**: Limited to specific types of matrices; not suitable for large systems.
4) Inverse Iteration:
   Complexity: $O(n^3)$.
   - **Pros**: Efficient for computing eigenvalues near a specified point.
   - **Cons**: Requires a good initial estimate; sensitivity to singular matrices.

3.2 Chosen Algorithm

The **QR Algorithm** was chosen because:

- Capable of finding all eigenvalues without requiring prior knowledge or good initial estimates.
- Suitable for matrices that are not extremely sparse.
- Balances accuracy and general applicability.

4 Algorithm Description

4.1 QR Algorithm

The QR Algorithm iteratively factorizes the matrix A into Q and R such that $A = QR$ and updates $A \leftarrow RQ$. After sufficient iterations, A converges to an upper triangular matrix whose diagonal elements are the eigenvalues.

4.2 Time Complexity Analysis

- The time complexity of the QR Algorithm for dense matrices is $O(n^3)$ per iteration.
- The number of iterations required depends on convergence properties, often related to the spectral gap.

4.3 Additional Insights

:

- **Memory Usage**: Moderate, as it involves storing intermediate matrices Q and R.
- **Convergence**: Exponential convergence for matrices with distinct eigenvalues.
- **Suitability**: Works well for small to medium-sized dense matrices but is computationally expensive for very large systems.

4.4 Pseudocode

```
Input: Matrix A (n x n)
Output: Eigenvalues of A

1. for k = 1 to max_iter:
2.     Compute QR factorization: A = QR
3.     Update A = RQ
4.     Check convergence
5. return diagonal elements of A
```

5 Implementation

5.1 Programming Language and Libraries

The implementation was written in C using only standard libraries. Matrix operations such as multiplication and QR decomposition were implemented manually. Dynamic memory allocation was used to handle matrices of varying sizes.

5.2 Code Snippets

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

// Function to allocate memory for a 2D matrix
double** allocateMatrix(int n) {
    double** mat = (double**)malloc(n * sizeof(double*));
    for (int i = 0; i < n; i++) {
        mat[i] = (double*)malloc(n * sizeof(double));
    }
    return mat;
}

// Function to free a 2D matrix
void freeMatrix(double** mat, int n) {
    for (int i = 0; i < n; i++) {
```

```
        free(mat[i]);
    }
    free(mat);
}

// Function to multiply two matrices
void multiplyMatrices(double** A, double** B, double** result, int n) {
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            result[i][j] = 0.0;
            for (int k = 0; k < n; k++) {
                result[i][j] += A[i][k] * B[k][j];
            }
        }
    }
}

// Function to perform QR decomposition using Gram-Schmidt
void qrDecomposition(double** A, double** Q, double** R, int n) {
    for (int k = 0; k < n; k++) {
        for (int i = 0; i < n; i++) R[k][i] = 0.0;
        for (int i = 0; i < n; i++) Q[i][k] = A[i][k];

        for (int i = 0; i < k; i++) {
            for (int j = 0; j < n; j++) {
                R[i][k] += Q[j][i] * A[j][k];
            }
            for (int j = 0; j < n; j++) {
                Q[j][k] -= R[i][k] * Q[j][i];
            }
        }
        R[k][k] = 0.0;
        for (int i = 0; i < n; i++) R[k][k] += Q[i][k] * Q[i][k];
        R[k][k] = sqrt(R[k][k]);
        for (int i = 0; i < n; i++) Q[i][k] /= R[k][k];
    }
}

// Function to compute eigenvalues using the QR algorithm
void qrAlgorithm(double** A, int n, int maxIter) {
    double** Q = allocateMatrix(n);
    double** R = allocateMatrix(n);
    double** temp = allocateMatrix(n);

    for (int iter = 0; iter < maxIter; iter++) {
        qrDecomposition(A, Q, R, n);
        multiplyMatrices(R, Q, temp, n);

        // Copy temp back into A
        for (int i = 0; i < n; i++) {
            for (int j = 0; j < n; j++) {
                A[i][j] = temp[i][j];
            }
```

```
        }
    }

    // Print eigenvalues (diagonal of A)
    printf("Eigenvalues:\n");
    for (int i = 0; i < n; i++) {
        printf("%f\n", A[i][i]);
    }

    // Free allocated memory
    freeMatrix(Q, n);
    freeMatrix(R, n);
    freeMatrix(temp, n);
}

int main() {
    int n, maxIter;

    // Input matrix size
    printf("Enter the size of the matrix (n x n): ");
    scanf("%d", &n);

    // Allocate matrix
    double** A = allocateMatrix(n);

    // Input matrix elements
    printf("Enter the elements of the matrix row by row:\n");
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            scanf("%lf", &A[i][j]);
        }
    }

    // Input maximum iterations
    printf("Enter the maximum number of iterations: ");
    scanf("%d", &maxIter);

    // Perform QR algorithm
    qrAlgorithm(A, n, maxIter);

    // Free allocated memory
    freeMatrix(A, n);

    return 0;
}
```

Listing 1: C code

6 Results and Analysis

6.1 Test Cases

- Test Case 1: $A = \begin{pmatrix} 4 & 2 \\ 1 & 3 \end{pmatrix}$

- Computed Eigenvalues: 5 and 2
- Test Case 2: $B = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 2 & 0 & 0 & 0 \\ 0 & 0 & 3 & 0 & 0 \\ 0 & 0 & 0 & 4 & 0 \\ 0 & 0 & 0 & 0 & 5 \end{pmatrix}$
- Computed Eigenvalues : 1, 2, 3, 4 and 5

6.2 Performance Analysis

- Time Complexity: $O(n^3)$
- Convergence rate: Approximately ...
- Limitations: High computational cost for very large matrices.

7 Comparison of Algorithms

Algorithm	Time Complexity	Matrix Suitability	Notes
Power Iteration	$O(n^2)$ per iteration	Dominant eigenvalue	Simple but limited
QR Algorithm	$O(n^3)$	General-purpose	Chosen algorithm
Jacobi Method	$O(n^3)$	Symmetric matrices	Stable but slower
Inverse Iteration	$O(n^3)$	Eigenvalues near shifts	Precise but sensitive

8 Conclusion

This report demonstrated the implementation of the QR Algorithm for eigenvalue calculation and analyzed its performance. While efficient for dense matrices, future work could explore optimizations for sparse or large-scale matrices.