Eigen Values

EE24BTECH11027 - satwikagv

1

1.1 Chosen Algorithm

Among all the different algorithms I chose QR Algorithm as it is applicable for both symmetric and non-symmetric matrices. In QR Algorithm it can be performed by using the process:

- Hessenberg reduction
- QR decomposition by Gram-schmidt process

First the matrix is reduced into hessenberg form so it will be easy to simplify the computational complexity of matrix operations, especially for eigenvalue computations.

1.2 QR Algorithm

QR algorithm is iterative and involves decomposing the matrix into its QR factors where Q is a orthogonal matrix and R is a upper triangular matrix and recombining them in a way that progressively reveals the eigenvalues. For a $n \times n$ matrix the QR Algorithms performs

- QR factorization - Decompose A into Q(orthogonal matrix) and R(upper triangular matrix) such that $A = QR$
- Matrix update - Form a new matrix by multiplying R and Q in the order $A' = RQ$. This step creates a matrix similar to A preserving its eigenvalues.
- Repeat - Replace A with A' and repeat the steps until A converges to a form where its eigenvalues are evident (a triangular or nearly diagonal form).

After enough iterations, the algorithm yields a matrix where the diagonal elements are the eigen values of the matrix.

1.3 Hessenberg form

The Hessenberg form has the same eigen values as the original matrix.
We will use Householder reflectors to transform the matrix into Upper Hessenberg form.
The Householder reflector is defined as:

$$P = \frac{VV^T}{V^T V}$$

We apply the Householder reflector to the original matrix:

$$H_1 = 1 - 2P$$

We apply the Householder reflector to the matrix:

$$A_1 = H_1 A H_1$$

We repeat the process to transform the matric into Upper Hessenberg form.

1.4 Gram-Schmidt Process

A method to convert a set of non-orthonormal vectors into a set of orthonormal vectors.

- Start with a set of non-orthonormal vectors say $\{v_1, v_2, \ldots, v_3\}$
- Find the first orthonormal vector $e_1 = \frac{v_1}{||v_1||}$
- Find the second orthonormal vectoe $e_2 = \frac{v_2-(v_2.e_1)e_1}{||v_2-(v_2.e_1)e_1||}$
- Repeat the above for each remaining vector

1.5 QR Factorization

A method to decompose a matrix A into the product of an orthogonal matrix Q and an upper triangular matrix R.
To find the QR factorization of a matrix A,we can use the following steps:

- Take the columns of A as vectors $A_1, A_2, \ldots, A_n$
- Find the orthogonal vectors $U_1, U_2, \ldots, U_n$ using the Gram-schmidt process
- Normalize the vectors $U_1, U_2, \ldots, U_n$ to get the orthonormal vectors $E_1, E_2, \ldots, E_n$
- The matrix Q is formed by taking the dot products of the vectors $A_1, A_2, \ldots, A_n$ with the orthonormal vectors $E_1, E_2, \ldots, E_n$

1.6 Time Complexity Algorithm

- *QR* Gram-Schmidt has a time complexity of $\mathrm{O}\left(n^3\right)$ for decomposing an $n \times n$ matrix. This is due to the need for computing inner products and projections between each pair of vectors and performing orthogonalization across all columns of the matrix.
- *QR* Householder Reflection method is more efficient for *QR* decomposition, with a time complexity of $\mathrm{O}\left(n^2\right)$ per reflection, and since there are typically n reflections required (one for each column), the overall complexity remains $\mathrm{O}\left(n^3\right)$.

1.7 Convergence

The *QR* algorithm converges quickly for matrices that are already nearly triangular or Hessenberg (triangular except for one diagonal below the main diagonal). For general matrices, it can be transformed to Hessenberg form first, which requires $\mathrm{O}\left(n^3\right)$ work but accelerates convergence in the *QR* steps.

1.8 Comparison of Algorithms

QR Algorithm can find all eigen values when compared to other methods like power iteration,inverse iteration which are mainly used to find the dominant eigen value or a particular eigen value. *QR* Algorithm is used to find all the eigen values for both symmetric and non-symmetric and also for both dense and sparse matrices unlike many other methods. It is accurate and robust. It is numerically stable.

1.9 C code

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

#define MAX_ITER 1000
#define TOLERANCE 1e-6

// Function to allocate and free matrices
double** allocateMatrix(int n) {
    double** matrix = malloc(n * sizeof(double*));
    for (int i = 0; i < n; i++) {
        matrix[i] = malloc(n * sizeof(double));
    }
    return matrix;
}

void freeMatrix(double** matrix, int n) {
    for (int i = 0; i < n; i++) {
        free(matrix[i]);
    }
    free(matrix);
}

// Function to copy a matrix
void copyMatrix(double** src, double** dest, int n) {
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            dest[i][j] = src[i][j];
        }
    }
}

// Hessenberg Reduction using Householder reflections
void hessenbergReduction(double** A, int n) {
    for (int k = 0; k < n - 2; k++) {
        double norm = 0.0;
        for (int i = k + 1; i < n; i++) {
            norm += A[i][k] * A[i][k];
        }
        norm = sqrt(norm);

        if (fabs(norm) < TOLERANCE) continue;

        double v[n];
        for (int i = 0; i < n; i++) v[i] = 0.0;
        v[k + 1] = A[k + 1][k] - norm;
        for (int i = k + 2; i < n; i++) {
            v[i] = A[i][k];
        }

        double v_norm = 0.0;
        for (int i = k + 1; i < n; i++) {
            v_norm += v[i] * v[i];
        }
        v_norm = sqrt(v_norm);
        if (v_norm < TOLERANCE) continue;
        for (int i = k + 1; i < n; i++) {
```

```
                v[i] /= v_norm;
            }

            for (int i = k; i < n; i++) {
                double sum = 0.0;
                for (int j = k + 1; j < n; j++) {
                    sum += A[i][j] * v[j];
                }
                for (int j = k + 1; j < n; j++) {
                    A[i][j] -= 2.0 * sum * v[j];
                }
            }
            for (int j = k; j < n; j++) {
                double sum = 0.0;
                for (int i = k + 1; i < n; i++) {
                    sum += A[i][j] * v[i];
                }
                for (int i = k + 1; i < n; i++) {
                    A[i][j] -= 2.0 * sum * v[i];
                }
            }
        }
    }

// QR Decomposition using Gram-Schmidt process
void gramSchmidt(double** A, double** Q, double** R, int n) {
    for (int j = 0; j < n; j++) {
        R[j][j] = 0.0;
        for (int i = 0; i < n; i++) {
            R[j][j] += A[i][j] * A[i][j];
        }
        R[j][j] = sqrt(R[j][j]);

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

        for (int k = j + 1; k < n; k++) {
            R[j][k] = 0.0;
            for (int i = 0; i < n; i++) {
                R[j][k] += Q[i][j] * A[i][k];
            }
            for (int i = 0; i < n; i++) {
                A[i][k] -= R[j][k] * Q[i][j];
            }
        }
    }
}

// QR Algorithm for Eigenvalue Calculation
void qrAlgorithm(double** A, double* eigenvalues, int n) {
    double** Q = allocateMatrix(n);
    double** R = allocateMatrix(n);
    double** AQ = allocateMatrix(n);

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

        for (int i = 0; i < n; i++) {
```

```c
            for (int j = 0; j < n; j++) {
                AQ[i][j] = 0.0;
                for (int k = 0; k < n; k++) {
                    AQ[i][j] += R[i][k] * Q[k][j];
                }
            }
        }
        copyMatrix(AQ, A, n);

        int converged = 1;
        for (int i = 1; i < n; i++) {
            if (fabs(A[i][i - 1]) > TOLERANCE) {
                converged = 0;
                break;
            }
        }
        if (converged) break;
    }

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

    freeMatrix(Q, n);
    freeMatrix(R, n);
    freeMatrix(AQ, n);
}

int main() {
    int n;
    printf("Enter the size of the matrix (n): ");
    scanf("%d", &n);

    double** A = allocateMatrix(n);
    double* eigenvalues = malloc(n * sizeof(double));

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

    hessenbergReduction(A, n);
    qrAlgorithm(A, eigenvalues, n);

    printf("Eigenvalues: ");
    for (int i = 0; i < n; i++) {
        printf("%lf ", eigenvalues[i]);
    }
    printf("\n");

    freeMatrix(A, n);
    free(eigenvalues);

    return 0;
}
```