Eigenvalue Computation Using the QR Algorithm with Hessenberg Reduction

DHWANITH M DODDAHUNDI - EE24BTECH11016

November 18, 2024

Contents

1 Introduction

Eigenvalue computation is a fundamental operation in numerical linear algebra, with applications in engineering, physics, and data science. The QR algorithm, combined with Hessenberg reduction, is a robust method for finding eigenvalues of general square matrices. This report discusses the algorithm, its implementation, mathematical foundation, and advantages, and compares it to alternative methods.

2 Mathematical Background

2.1 Eigenvalues and Eigenvectors

Eigenvalues are solutions to the characteristic equation $\det(A - \lambda I) = 0$, where A is a square matrix and λ represents the eigenvalues. Eigenvectors satisfy $A\mathbf{x} = \lambda\mathbf{x}$.

2.2 The QR Algorithm

The QR algorithm iteratively decomposes A_k into $A_k = Q_k R_k$, where Q_k is orthogonal and R_k is upper triangular. The next iteration is computed as $A_{k+1} = R_k Q_k$. Convergence ensures that A_k becomes nearly upper triangular, with eigenvalues on the diagonal.

2.3 Hessenberg Reduction

Hessenberg reduction transforms A into a similar matrix H, which has zeros below the first subdiagonal. This simplifies the QR decomposition and reduces computational cost.

2.4 Complex Eigenvalues

For 2×2 blocks representing complex eigenvalues, the eigenvalues are roots of $\lambda^2 - (\text{trace})\lambda + (\text{determinant}) = 0$, calculated as:

$$\lambda_{1,2} = \frac{\text{trace}}{2} \pm \sqrt{\left(\frac{\text{trace}}{2}\right)^2 - \text{determinant}}.$$

3 Algorithm Implementation

3.1 Code Overview

The QR algorithm with Hessenberg reduction was implemented in C. Key highlights include: - Dynamic memory allocation for flexibility. - Modular design with functions for Hessenberg reduction, QR decomposition, and matrix multiplication.

3.2 Practical Considerations

Convergence tolerance was set to 10^{-6}, ensuring accurate results without excessive computation. The algorithm was capped at 1000 iterations.

4 Example and Results

4.1 Example 1: Non-Symmetric matrix

The input matrix is:

$$A = \begin{bmatrix} 1 & 2 & 5 \\ 4 & 5 & 8 \\ 2 & 4 & 6 \end{bmatrix}.$$

The intermediate Hessenberg form is:

$$H = \begin{bmatrix} 1.0000 & 4.0249 & 3.5777 \\ 4.4721 & 10.0000 & 6.0000 \\ 0.0000 & 2.0000 & 1.0000 \end{bmatrix}.$$

After convergence, the eigenvalues are:

- $\lambda_1 = 12.7778$
- $\lambda_2 = -0.3889 + 0.8876i$
- $\lambda_3 = -0.3889 - 0.8876i$

4.2 Example 2: Symmetric Matrix

The input matrix is

$$B = \begin{bmatrix} 4 & 1 & 2 \\ 1 & 3 & 1 \\ 2 & 1 & 5 \end{bmatrix}$$

For symmetric matrices, the Hessenberg form is tridiagonal:

$$H = \begin{bmatrix} 4.0000 & 2.2361 & 0.0000 \\ 2.2361 & 5.4000 & 0.2000 \\ 0.0000 & 0.2000 & 2.6000 \end{bmatrix}$$

After QR iterations, the eigenvalues are:

- $\lambda_1 = 7.048917$
- $\lambda_2 = 2.643104$
- $\lambda_3 = 2.307979$

5 Time Complexity Analysis

5.1 Hessenberg Reduction

Hessenberg reduction has a complexity of $O(N^3)$ for dense $N \times N$ matrices. It reduces computational overhead in subsequent QR steps.

5.2 QR Iterations

Each QR step has a complexity of $O(N^2)$, and the algorithm requires $O(N)$ iterations for convergence. Thus, the overall complexity is $O(N^3)$.

6 Other Insights

6.1 Memory Usage

The memory usage of the QR algorithm is $O(N^2)$, as it requires storing multiple matrices during each iteration: the matrix A_k, the orthogonal matrix Q_k, and the upper triangular matrix R_k. For large matrices, this can become a significant consideration, but for matrices with $N \leq 1000$, the QR algorithm remains practical.

6.2 Convergence Rate

The convergence rate of the QR algorithm is typically fast for matrices with distinct eigenvalues. The Hessenberg reduction step helps by simplifying the matrix structure, allowing for more efficient QR decompositions. In general, the convergence is linear, and the method is effective for a wide range of matrices.

6.3 Suitability for Different Types of Matrices

The QR algorithm with Hessenberg reduction is suitable for:

- Dense matrices: The algorithm performs well on dense matrices that do not exhibit any special structure.
- Symmetric matrices: The algorithm converges very quickly for symmetric matrices, and the eigenvalues are guaranteed to be real.
- Non-symmetric matrices: While the algorithm works for non-symmetric matrices, it may converge more slowly compared to symmetric matrices.

7 Comparison of Algorithms

Several other methods exist for computing eigenvalues. Below is a brief comparison between the QR algorithm and other methods:

7.1 QR Algorithm vs. Jacobi Method

The Jacobi method is another iterative method for computing eigenvalues of symmetric matrices. While both algorithms are iterative and converge to the eigenvalues, the Jacobi method has a slower convergence rate than the QR algorithm, especially for large matrices.

7.2 QR Algorithm vs. Divide and Conquer

The divide and conquer method is faster than the QR algorithm for large, symmetric matrices, and has a time complexity of $O(N^3)$. It is particularly effective for very large matrices and is used in specialized libraries for eigenvalue computation.

7.3 QR Algorithm vs. Lanczos Algorithm

The Lanczos algorithm is effective for sparse matrices, as it reduces the matrix to a much smaller size, making it computationally efficient. However, it does not guarantee all eigenvalues, and is not as general-purpose as the QR algorithm.

8 Error Analysis

8.1 Sources of Error

- Finite precision arithmetic can introduce rounding errors. - Ill-conditioned matrices may slow convergence or yield inaccurate results.

8.2 Mitigation Strategies

- Using double-precision arithmetic minimizes rounding errors. - Matrix scaling improves numerical stability.

9 Advantages of the QR Algorithm

9.1 General Benefits

The QR algorithm is robust and applicable to a wide range of matrices. It accurately computes both real and complex eigenvalues without requiring matrix symmetry.

9.2 Efficiency with Hessenberg Reduction

Hessenberg reduction decreases QR step complexity, preserving eigenvalues while simplifying computations.

9.3 Applications

- Engineering: Stability analysis and vibration modes.
- Machine Learning: Principal Component Analysis (PCA).
- Physics: Quantum mechanics and energy state computation.

9.4 Future Directions

- GPU-based parallelization can accelerate computations. - Adaptive shift strategies improve convergence for clustered eigenvalues.

10 Conclusion

The QR algorithm with Hessenberg reduction is a powerful tool for eigenvalue computation. Its robustness, efficiency, and accuracy make it suitable for various applications in science and engineering.

11 References

- Trefethen, Lloyd N., and David Bau III. *Numerical Linear Algebra.* SIAM, 1997.
- Golub, Gene H., and Charles F. Van Loan. *Matrix Computations.* Johns Hopkins University Press, 2013.
- Google.com
- Some AI generated models

A Code Listing

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

#define TOLERANCE 1e-6
#define MAX_ITERATIONS 10000

// Function declaration
void matrixPrint(double **matrix, int size);
void matrixMultiply(double **A, double **B, double **result, int size);
void hessenbergTransform(double **matrix, int size);
void qrFactorize(double **matrix, int size, double **Q, double **R);
int hasConverged(double **matrix, int size);
void computeComplexEigenvalues(double a, double b, double c, double d, double complex *eig1,

int main() {
    int size;

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

    // Dynamic memory allocation
    double **A = (double **)malloc(size * sizeof(double *));
    double **Q = (double **)malloc(size * sizeof(double *));
    double **R = (double **)malloc(size * sizeof(double *));
    double **currentMatrix = (double **)malloc(size * sizeof(double *));
    double **tempMatrix = (double **)malloc(size * sizeof(double *));
    for (int i = 0; i < size; i++) {
        A[i] = (double *)malloc(size * sizeof(double));
        Q[i] = (double *)malloc(size * sizeof(double));
        R[i] = (double *)malloc(size * sizeof(double));
        currentMatrix[i] = (double *)malloc(size * sizeof(double));
        tempMatrix[i] = (double *)malloc(size * sizeof(double));
    }

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


    hessenbergTransform(currentMatrix, size);
    printf("\nHessenberg Form:\n");
```

```
matrixPrint(currentMatrix, size);


int iterationCount = 0;
while (!hasConverged(currentMatrix, size) && iterationCount < MAX_ITERATIONS) {
    qrFactorize(currentMatrix, size, Q, R);


    matrixMultiply(R, Q, tempMatrix, size);


    for (int i = 0; i < size; i++) {
        for (int j = 0; j < size; j++) {
            currentMatrix[i][j] = tempMatrix[i][j];
        }
    }

    iterationCount++;
}

if (iterationCount >= MAX_ITERATIONS) {
    printf("\nWarning: Algorithm did not converge within %d iterations.\n", MAX_ITERATIO
}


printf("\nConverged Matrix (Approx. Upper Triangular):\n");
matrixPrint(currentMatrix, size);


printf("\nEigenvalues:\n");
for (int i = 0; i < size; i++) {
    if (i < size - 1 && fabs(currentMatrix[i + 1][i]) > TOLERANCE) {

        double complex eig1, eig2;
        computeComplexEigenvalues(
            currentMatrix[i][i], currentMatrix[i][i + 1],
            currentMatrix[i + 1][i], currentMatrix[i + 1][i + 1],
            &eig1, &eig2);
        printf("Eigenvalue %d: %.20f + %.20fi\n", i + 1, creal(eig1), cimag(eig1));
        printf("Eigenvalue %d: %.20f + %.20fi\n", i + 2, creal(eig2), cimag(eig2));
        i++;
    } else {

        printf("Eigenvalue %d: %.20f\n", i + 1, currentMatrix[i][i]);
    }
}

// Free memory
for (int i = 0; i < size; i++) {
    free(A[i]);
    free(Q[i]);
```

```
        free(R[i]);
        free(currentMatrix[i]);
        free(tempMatrix[i]);
    }
    free(A);
    free(Q);
    free(R);
    free(currentMatrix);
    free(tempMatrix);

    return 0;
}

// Print a matrix
void matrixPrint(double **matrix, int size) {
    for (int i = 0; i < size; i++) {
        for (int j = 0; j < size; j++) {
            printf("%8.4f ", matrix[i][j]);
        }
        printf("\n");
    }
}

// Matrix multiplication
void matrixMultiply(double **A, double **B, double **result, int size) {
    for (int i = 0; i < size; i++) {
        for (int j = 0; j < size; j++) {
            result[i][j] = 0.0;
            for (int k = 0; k < size; k++) {
                result[i][j] += A[i][k] * B[k][j];
            }
        }
    }
}

//  Hessenberg form
void hessenbergTransform(double **matrix, int size) {
    for (int k = 0; k < size - 2; k++) {
        for (int i = k + 2; i < size; i++) {
            double x = matrix[k + 1][k];
            double y = matrix[i][k];
            double r = hypot(x, y);
            double c = x / r;
            double s = -y / r;

            for (int j = k; j < size; j++) {
                double temp1 = c * matrix[k + 1][j] - s * matrix[i][j];
                double temp2 = s * matrix[k + 1][j] + c * matrix[i][j];
                matrix[k + 1][j] = temp1;
                matrix[i][j] = temp2;
            }
```

```
            for (int j = 0; j < size; j++) {
                double temp1 = c * matrix[j][k + 1] - s * matrix[j][i];
                double temp2 = s * matrix[j][k + 1] + c * matrix[j][i];
                matrix[j][k + 1] = temp1;
                matrix[j][i] = temp2;
            }
        }
    }
}

//  QR decomposition
void qrFactorize(double **matrix, int size, double **Q, double **R) {
    for (int i = 0; i < size; i++) {
        for (int j = 0; j < size; j++) {
            R[i][j] = 0.0;
        }
    }

    for (int i = 0; i < size; i++) {
        R[i][i] = 0.0;
        for (int k = 0; k < size; k++) {
            R[i][i] += matrix[k][i] * matrix[k][i];
        }
        R[i][i] = sqrt(R[i][i]);

        for (int k = 0; k < size; k++) {
            Q[k][i] = matrix[k][i] / R[i][i];
        }

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

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

// Check for convergence
int hasConverged(double **matrix, int size) {
    for (int i = 0; i < size - 1; i++) {
        if (fabs(matrix[i + 1][i]) > TOLERANCE) {
            return 0;
        }
```

```
    }
    return 1;
}

// Complex eigenvalues of a 2x2 matrix
void computeComplexEigenvalues(double a, double b, double c, double d, double complex *eig1,
    double trace = a + d;
    double determinant = a * d - b * c;
    double discriminant = (trace * trace) / 4.0 - determinant;

    if (discriminant >= 0) {
        *eig1 = trace / 2.0 + sqrt(discriminant);
        *eig2 = trace / 2.0 - sqrt(discriminant);
    } else {
        *eig1 = trace / 2.0 + I * sqrt(-discriminant);
        *eig2 = trace / 2.0 - I * sqrt(-discriminant);
    }
}
```