Software Assignment

Eigenvalue Determination in Complex Matrices through the QR Iterative Method

EE24BTECH11049
Patnam Shariq Faraz Muhammed
Department Of Electrical Engineering
IIT Hyderabad

Abstract

This report outlines the calculation of eigenvalues of a complex matrix using QR decomposition, implemented using the classical QR algorithm. It also compares the QR algorithm with other eigenvalue computation methods, focusing on time complexity, accuracy, and suitability for different types of matrices.

1 Introduction

Eigenvalues and eigenvectors are critical in many fields, including machine learning, quantum mechanics, and electrical engineering. The QR decomposition, along with the Gram-Schmidt process, is a powerful tool in computing eigenvalues of matrices. This report explains the QR decomposition methods and Gram-Schmidt variants used for eigenvalue determination, with a focus on the Classical, Modified, and Block Gram-Schmidt processes. Additionally, we explore other QR decomposition variants, such as Householder-based QR.

2 Eigenvalues and Eigenvectors

Eigenvalues and eigenvectors are central to understanding the behavior of linear transformations. The eigenvalue problem is defined as follows:

Given a square matrix $A \in \mathbb{C}^{n\times n}$, a scalar $\lambda \in \mathbb{C}$, and a non-zero vector $\mathbf{v} \in \mathbb{C}^n$, they satisfy the equation:

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

The scalar λ is called the **eigenvalue** and $\mathbf{v}$ is the corresponding **eigenvector**.

2.1 Characteristic Equation

Well, the most popular and well known method used to calculated eigen values for matrices with order 2 and 3 is **Characteristic Equation**.

equation:

$$\boxed{A\mathbf{v} - \lambda\mathbf{v} = 0}$$

This can be expressed as:

$$\boxed{(A - \lambda I)\mathbf{v} = 0}$$

For a non-trivial solution, the matrix $A - \lambda I$ must be singular, meaning:

$$\boxed{\det(A - \lambda I) = 0}$$

The roots of this polynomial give the eigenvalues $\lambda_1, \lambda_2, \ldots, \lambda_n$ of the matrix A.

However, we tend to assume how easy it would be to use this method instead of QR decomposition, for matrices with order 2 and 3. Now, grab a pen and solve for it 4×4 matrix. Right, understood? How complicated it gets with increasing order. There are several methods to calculating eigen values

2.2 Characteristic Polynomial Method

The characteristic polynomial method is a direct way of finding eigenvalues. The eigenvalues are obtained by solving the characteristic equation:

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

where A is the matrix, λ is the eigenvalue, and I is the identity matrix of the same dimension as A. The solutions to this polynomial equation give the eigenvalues of the matrix.

- **Pros:** This method works well for small matrices, especially for matrices of size $n \leq 4$.
- **Cons:** The method becomes computationally expensive and impractical for large matrices due to the difficulty in solving the resulting high-degree polynomial.

2.3 QR Algorithm

The QR algorithm is an iterative method that uses QR decomposition to compute the eigenvalues of a matrix. The basic steps of the algorithm are as follows:

1. Perform QR decomposition on the matrix A:

$$A = QR$$

where Q is an orthogonal matrix and R is an upper triangular matrix.

2. Compute the new matrix A_1 as:

$$A_1 = RQ$$

3. Repeat the above steps iteratively until the matrix A_1 converges to a triangular matrix, at which point the eigenvalues can be read directly from the diagonal of the matrix.

- **Pros:** Efficient for large matrices and works for both symmetric and nonsymmetric matrices.
- **Cons:** Slow convergence for ill-conditioned or very large matrices.

2.4 Power Method

The power method is an iterative technique used to find the dominant eigenvalue (the largest in magnitude) of a matrix. The method is based on repeatedly multiplying the matrix by a vector, normalizing it at each step.

The power method proceeds as follows:

1. Choose an initial guess for the vector b_0. 2. Update the vector using the matrix A:

$$b_{k+1} = Ab_k$$

3. Normalize the vector at each step:

$$\lambda_k = \frac{b_k^T A b_k}{b_k^T b_k}$$

where λ_k is the current estimate of the dominant eigenvalue.

- **Pros:** Simple and fast for finding the dominant eigenvalue.
- **Cons:** Only works for the largest eigenvalue. If the dominant eigenvalue is degenerate, convergence may be slow.

2.5 Inverse Power Method

The inverse power method is an iterative technique used to find the smallest eigenvalue (in magnitude) of a matrix. It involves applying the power method to the matrix A^{-1}, which converges to the smallest eigenvalue of A.

The inverse power method proceeds as follows:

1. Choose an initial guess for the vector b_0. 2. Solve the system $(A-\lambda I)b = x$ iteratively. 3. Apply the power method on the matrix A^{-1} and estimate the smallest eigenvalue λ.

- **Pros:** Efficient for finding the smallest eigenvalue.
- **Cons:** Requires the computation of the inverse of the matrix, which can be computationally expensive.

2.6 Jacobi Method

The Jacobi method is an iterative algorithm specifically designed for symmetric matrices. The idea is to use rotations to reduce the matrix to a diagonal form, where the eigenvalues are the diagonal elements.

The basic steps are as follows:

1. For each off-diagonal element of the matrix, calculate the rotation angle θ to zero it out.
2. Apply the rotation to the matrix, adjusting the off-diagonal elements.
3. Repeat the process until the matrix is sufficiently diagonalized.

- **Pros:** Converges quickly for symmetric matrices and guarantees finding all eigenvalues.
- **Cons:** Not efficient for large matrices.

2.7 Lanczos Algorithm

The Lanczos algorithm is a variation of the Arnoldi method used for large sparse matrices. It is designed to compute a few eigenvalues and eigenvectors of a large, sparse matrix without having to compute the entire eigenvalue spectrum.

The Lanczos algorithm builds an orthogonal basis for the Krylov subspace and approximates the eigenvalues of the matrix.

- **Pros:** Efficient for large sparse matrices.
- **Cons:** May not always converge to the desired eigenvalues.

3 QR Decomposition

3.1 Gram Schmidt

Uh?... Wondering why I wrote Gram Schmidt when the heading mentioned is about QR decomposition.
To fully understand how the QR decomposition is obtained, we need to explore the ocean of the Schmidt algorithm... don't be scared I won't yap about it, just a small briefing. (I am going yap about it in the later text thou)

In mathematics, particularly linear algebra and numerical analysis, the Gram-Schmidt process or Gram-Schmidt algorithm is a way of finding a set o two or more vectors that are perpendicular to each other.

The Gram-Schmidt process is a method for constructing an orthonormal basis from a set of vectors in an inner product space, most commonly in the Euclidean space $\mathbf{R}^n$ equipped with the standard inner product. The process takes a finite, linearly independent set of vectors $s = \{v_1, v_2, \ldots, v_k\}$ for $k \leq n$ and generates an orthogonal set $s' = \{u_1, u_2, \ldots, u_k\}$ that spans the same

The application of the Gram Schmidt process to the column vectors of a full column rank matrix yields the QR decomposition (it is decomposed into an orthogonal and a triangular matrix).

3.2 What is QR?

The basic goal of the QR decomposition is to factor a matrix as a product of two matrices (traditionally called Q and R, hence the name of this factorization). Each matrix has a simple structure that can be further exploited in dealing with, say, linear equations.

The QR decomposition is nothing else than the Gram-Schmidt procedure applied to the columns of the matrix, with the result expressed in matrix form. Consider an $m \times n$ matrix $A = (a_1, \ldots, a_n)$, with each $a_i \in \mathbb{R}^m$ a column of A.

3.3 How is it useful for my project?

The *QR algorithm* is an iterative method used to find the *eigenvalues* of a matrix. The reason why the QR algorithm is used for eigenvalue computation is due to its efficient and robust way of iterating on a matrix to reveal its eigenvalues, particularly for large matrices. Here's a step-by-step breakdown of why the QR method is useful for finding eigenvalues:

3.3.1 1. The QR Decomposition and its Key Property: Given a matrix A, the QR decomposition expresses it as a product:

$$\boxed{A = QR}$$

where:

- Q is an *orthogonal* (or unitary in the complex case) matrix, meaning $Q^T Q = I$,
- R is an *upper triangular* matrix.

This decomposition is significant because it preserves the eigenvalues of A in the sense that the eigenvalues of A are the same as the eigenvalues of $A^T A$. The QR algorithm leverages this fact to iteratively approximate the eigenvalues of a matrix.

3.3.2 2. Iterative Steps of the QR Algorithm: The basic idea behind the QR algorithm is to decompose a matrix A into its QR factors, and then form a new matrix A_1 by multiplying R and Q:

$$\boxed{A_1 = RQ}$$

This process is repeated iteratively:

1) Compute the QR decomposition of A.
2) Set $A_1 = RQ$.
3) Repeat the process with A_1.

At each iteration, the matrix A becomes closer to a form where it is nearly *diagonal*, and the diagonal elements are the *eigenvalues* of the matrix.

3.3.3 3. Why QR Converges to Eigenvalues:

- **Similarity Transformation**: The key property of the QR algorithm is that the repeated application of QR decompositions transforms the matrix A into a sequence of matrices that are "similar" to A. Matrices that are similar have the same eigenvalues. The QR algorithm effectively generates a sequence of matrices whose eigenvalues converge to those of the original matrix.
- **Triangularization**: After several iterations of QR decompositions, the matrix A approaches an upper triangular form, where the diagonal elements of the resulting matrix are the eigenvalues of A. This is because for any matrix A, if A is diagonalizable, after enough QR iterations, the matrix will approximate a diagonal matrix whose diagonal entries are the eigenvalues.

3.3.4 4. Eigenvalue Computation Process: At each step of the QR algorithm, the matrix A tends to become more triangular (upper triangular or near upper triangular). In the limit (after many iterations), the matrix becomes almost diagonal, and the eigenvalues can be read directly from the diagonal elements.

- **For a 2x2 matrix**: After applying the QR decomposition iteratively, the off-diagonal elements tend to zero, and the diagonal elements converge to the eigenvalues of the matrix.

3.3.5 5. Why the QR Method is Effective:

- **Convergence**: The QR algorithm converges relatively quickly for many types of matrices, especially if the matrix is diagonalizable. Even for non-diagonalizable matrices, the algorithm still provides an approximation to the eigenvalues.
- **Simplicity**: The QR algorithm does not require the explicit calculation of the determinant or solving the characteristic polynomial, which can be computationally expensive or unstable for large matrices. Instead, it focuses on iteratively transforming the matrix into a form where the eigenvalues are readily available.
- **Efficiency**: For large matrices, especially sparse or structured matrices, the QR algorithm can be more efficient than methods like direct diagonalization or finding roots of the characteristic polynomial.

3.3.6 6. QR Algorithm in Practice: In practice, the QR algorithm can be used in conjunction with other techniques (like shifts or deflation) to improve convergence speed and accuracy, especially for large matrices. Variants of the QR algorithm, such as the *Schur decomposition* or *QR with shifts*, are often used to speed up the process.

Hence, the QR algorithm is a powerful and efficient method for eigenvalue computation, particularly for large matrices where traditional methods may be slow or impractical.

4 Gram-Schmidt

Back to Gram-Schmidt again. There are several variants of the Gram-Schmidt process, each with different numerical properties. Below, we describe the main variants.

4.1 Variants

- Classical Gram-Schmidt (CGS)
- Modified Gram-Schmidt (MGS)
- Householder Transformations
- Givens Rotations(Alternative not a variant)

4.2 Classical Gram-Schmidt (CGS)

4.2.1 Process: The **Classical Gram-Schmidt** process is the original form of the Gram-Schmidt algorithm, where each vector is orthogonalized by subtracting projections onto all previous vectors.

Given a set of linearly independent vectors $\{v_1, v_2, \ldots, v_n\}$, the classical Gram-Schmidt process produces an orthogonal set $\{q_1, q_2, \ldots, q_n\}$ by:

1) Start with $q_1 = v_1$.
2) For $i = 2, 3, \ldots, n$, subtract the projection of v_i onto the previous q_j's:

$$q_i = v_i - \sum_{j=1}^{i-1} \langle v_i, q_j \rangle q_j$$

3) Normalize each q_i to get orthonormal vectors:

$$q_i = \frac{q_i}{\|q_i\|}$$

4.2.2 Result: For a matrix A, the classical Gram-Schmidt process computes the matrices Q and R such that:

$$A = QR$$

where Q is orthogonal and R is upper triangular.

4.2.3 Disadvantages: - Numerical Instability: Classical Gram-Schmidt is prone to numerical instability when the vectors are nearly linearly dependent, leading to the accumulation of rounding errors during the orthogonalization process.

4.3 Modified Gram-Schmidt (MGS)

4.3.1 Process: The **Modified Gram-Schmidt** process improves upon the classical version by performing orthogonalization in place, thereby reducing the accumulation of numerical errors.

1) Start with $q_1 = v_1$.
2) For $i = 2, 3, \ldots, n$, subtract the projection of v_i onto each previous q_j:

$$v_i = v_i - \langle v_i, q_j \rangle q_j \quad \text{for} \quad j = 1, 2, \ldots, i-1$$

3) After all projections have been subtracted, normalize v_i to obtain q_i:

$$q_i = \frac{v_i}{\|v_i\|}$$

4.3.2 Result: For a matrix A, the modified Gram-Schmidt process also computes the matrices Q and R such that:

$$A = QR$$

where Q is orthogonal and R is upper triangular.

4.3.3 Advantages: - Numerical Stability: The modified version is more stable than the classical version, as it avoids the accumulation of errors during the orthogonalization process.

4.4 Householder Transformations

4.4.1 Process: While not strictly a variant of Gram-Schmidt, **Householder transformations** are an alternative method for QR factorization that is more numerically stable. They are often used when working with large matrices.

1) For each column A_k, create a Householder vector v such that:

$$v = A_k - \|A_k\| e_1$$

   where e_1 is the first basis vector.
2) Apply the Householder transformation H_k to zero out all entries below the diagonal in the k-th column.
3) Repeat for all columns.

4.4.2 Result: Householder transformations also lead to a QR factorization of A:

$$A = QR$$

where Q is orthogonal and R is upper triangular.

4.5 Givens Rotations

4.5.1 Process: Another approach to QR factorization is using **Givens rotations**. This method applies a sequence of rotations to eliminate the off-diagonal entries, one at a time.

1) For each pair of elements (a_{ij}, a_{jj}), apply a Givens rotation to zero out the element a_{ij}:

$$G_{ij} = \begin{pmatrix} c & s \\ -s & c \end{pmatrix}$$

where $c = \frac{a_{ii}}{\sqrt{a_{ii}^2 + a_{ij}^2}}$ and $s = \frac{a_{ij}}{\sqrt{a_{ii}^2 + a_{ij}^2}}$.

2) Multiply the matrix A by the Givens rotation to eliminate the element a_{ij}.

4.5.2 Result: Givens rotations also lead to a QR factorization:

$$A = QR$$

where Q is orthogonal and R is upper triangular.

Variant	Description
Classical Gram-Schmidt (CGS)	The classical Gram-Schmidt process orthogonalizes a set of vectors one by one by subtracting the projections onto all previously orthogonalized vectors. It is prone to numerical instability when the vectors are nearly linearly dependent.
Modified Gram-Schmidt (MGS)	An improved version of the classical method, where orthogonalization is performed in place. It reduces the numerical instability issues that occur in the classical version by subtracting projections onto each previously computed orthogonal vector immediately.
Householder Transformations	A more stable alternative to Gram-Schmidt that uses reflections to zero out subdiagonal elements. Householder transformations are often used for large matrices and are preferred in practical numerical linear algebra due to their stability.
Givens Rotations	Uses a series of rotations to eliminate individual off-diagonal elements of the matrix. It is numerically stable and can be used to zero elements one at a time, typically for sparse or small systems.

TABLE 2: Variants of the Gram-Schmidt Process

5 Project - Method and algorithm:

QR Algorithm

- Let $A = A_1$ be a square matrix and also let its **QR decomposition** be Q_1R_1. Now let us define another matrix $A_2 = R_1Q_1$. Next the QR factorization of A_2 be Q_2R_2. Similarly we can define $A_3, A_4, A_5, \ldots$
  where $A_k = Q_kR_k$ (QR factorization of A_k), and $A_{k+1} = R_kQ_k$, i.e.,

$$A_{k+1} = R_kQ_k = Q_k^*Q_kR_kQ_k = Q_k^*A_kQ_k = Q_k^{-1}A_kQ_k$$

$$A_{k+1} = (Q_1Q_2 \ldots Q_k)^* A (Q_1Q_2 \ldots Q_k)$$

  If the process is continued for a long time then the matrices A_k becomes upper triangular (not always, though). In other words, $\lim_{k\to\infty} (A_k)_{ij} = 0$ for $j < i$, while the diagonal elements of the matrices A_k converge to the eigenvalues of the matrix A.
- **Theorem** :
  Suppose A be a square and also suppose that A is invertible and all its eigenvalues are distinct in modulus i.e., the algorith gives us a **Schur factorization** of A. Then, there exists (at least) one invertible matrix P such that

$$A = P\Lambda P^{-1}$$

  with $\Lambda = diag(\lambda_1, \lambda_2, \ldots, \lambda_n)$ and $|\lambda_1| > |\lambda_2| > \cdots > |\lambda_n| > 0$. Suppose that the matrix P^{-1} has a LU factorization, then the sequence of matrices (A_k) is such that

$$\lim_{k\to\infty} (A_k)_{ii} = \lambda_i, 1 \leq i \leq n$$

$$\lim_{k\to\infty} (A_k)_{ij} = 0, 1 \leq j < i \leq n$$

  These conditions are sufficient for the matrix to converge.

5.1 QR decomposition using Householder reflections

- The QR decomposition factors a matrix A (of size $m \times n$) into an orthogonal (or unitary) matrix Q and an upper triangular matrix R such that :

$$A = QR$$

- Q is orthogonal if A is real $\left(Q^TQ = I\right)$ or unitary if A is complex $(Q^*Q = I)$, where Q^* denotes the conjugate transpose of Q.
- R is an upper-triangular matrix.
- Let u and v be orthonormal vectors and let x be in the span of $\{u, v\}$ so that $x = c_1u + c_2v$. Consider the matrix $H = I - 2uu^T$. We can show that $Hx = x$ and hence reflection about v in the direction u can be represented as a matrix multiplication with H.
  For orthonormal u and v, the matrix $H = I - 2uu^T$ is a **Reflection** or **Householder matrix** .

- OBSERVATIONS :

$$Q_r R_r = Q_{r-1}(Q_r R_r)R_{r-1} = Q_{r-1}A_r R_{r-1}$$
$$Q_{r-1}^T A Q_{r-1} = A_r$$
$$Q_r R_r = A^r$$

- It can be shown that QR iteration converges. The rate of convergence depends on ratios $\left(\frac{\lambda_j}{\lambda_i}\right)^r$ for $j \neq i$ where r is the iteration number and λ_j and λ_i are the j^{th} and i^{th} eigenvalues of A. For complex eigenvalues, we observe a slow convergence, since the eigenvalues appear as cojugate pairs of equal magnitude.
  If the magnitudes of the largest eigenvalues are not well-separated one can apply a **shifted QR** to accelerate convergence.
  The shifted QR :
$$(A_r - k_r I) = Q_r R_r$$
  where, $A_{r+1} = R_r Q_r + k_r I$
- **Algorithm**
  Let $A_0 = A$, we iterate $i = 0$ repeat
  Choose a shift s_1
$$A_i - s_i I = Q_i R_i \text{ (QR decomposition)}$$
$$A_{i+1} = R_i Q_i + s_i I$$
$$i = i + 1$$
  until convergence.
- **Time complexity analysis**
  $O(n^3)$ or $O(k \cdot n^2)$ for square matrix of $n \times n$ and rectangular matrix of $k \times n (k > n)$.
- **Advantages**

  1) Numerical stability
  2) All eigenvalues (both real and complex) can be calculated.
  3) Versatility (both symmetric and non-symmetric matrices)
- This method is best used for Symmetric/Hermitian matrices, dense matrices and also for Numerical Software Libraries (LAPACK).

5.2 QR decomposition using Gram-Schmidt algorithm

- Gram-Schmidt algorithm starts with n independent vectors (usually the columns of the matrix A). It produces n orthonormal vectors (columns of Q).
- For practical reasons, having an orthonormal basis simplifies life partly because of the presence of many $\mathbf{w}_i \cdot \mathbf{w}_j$ terms that becomes zero.
- **Algorithm** :

  1) Let $\mathbf{v}_1, \mathbf{v}_2, \ldots, \mathbf{v}_n$ be a basis.

2) The Gram-Schmidt process iteratively constructs from the already constructed orthonormal set $\mathbf{w}_1, \mathbf{w}_2, \ldots, \mathbf{w}_n$ which spans a linear sub-space $\mathbf{V}_{i-1}$.
3) The new vector $\mathbf{u}_i$ is orthogonal to the linear space $\mathbf{V}_{i-1}$. The vector is then normalized.

- **Time complexity Analysis** :
  $O(n^3)$ for a $n \times n$ matrix.
  Quadratic convergence for well-separated eigenvalues. Convergence is faster if the matrix is already nearly upper-triangular.
- Best used for scenarios where the algorithm needs to be simple.
- **Advantages** :
  - Simplicity
  - Flexibility
  - Less memory usage

CODE

Matrix Structure and Memory Allocation

The program uses the following structure to store a matrix and its size:

Listing 1: Matrix Structure Definition

```
typedef struct {
    int size;
    complex double **matrix;
} Matrix;
```

The ‘Matrix‘ structure contains an integer ‘size‘ representing the matrix size and a pointer ‘matrix‘ to a dynamically allocated 2D array of complex numbers.

The function ‘CreateMatrix‘ dynamically allocates memory for a square matrix of complex numbers:

Listing 2: Matrix Creation Function

```
Matrix* CreateMatrix(int size) {
    Matrix *mat = (Matrix*)malloc(sizeof(Matrix));
    mat->size = size;
    mat->matrix = (complex double**)malloc(size * sizeof(complex double*));
    for (int i = 0; i < size; i++) {
        mat->matrix[i] = (complex double*)malloc(size * sizeof(complex double)
            );
    }
    return mat;
}
```

The ‘FreeMatrix‘ function is used to free the memory allocated for the matrix:

Listing 3: Matrix Free Function

```
void FreeMatrix(Matrix *mat) {
    for (int i = 0; i < mat->size; i++) {
```

```
        free(mat->matrix[i]);
    }
    free(mat->matrix);
    free(mat);
}
```

Matrix Multiplication

The ‘MatrixMultiply‘ function multiplies two matrices A and B, storing the result in matrix C. The algorithm iterates through each element of the result matrix and computes the sum of the products of the corresponding elements from rows of A and columns of B.

Listing 4: Matrix Multiplication Function

```
void MatrixMultiply(Matrix *A, Matrix *B, Matrix *C) {
    int size = A->size;
    for (int i = 0; i < size; i++) {
        for (int j = 0; j < size; j++) {
            C->matrix[i][j] = 0.0 + 0.0 * I;
            for (int k = 0; k < size; k++) {
                C->matrix[i][j] += A->matrix[i][k] * B->matrix[k][j];
            }
        }
    }
}
```

Gram-Schmidt Process

The ‘GramSchmidtProcess‘ function is responsible for performing the Gram-Schmidt orthogonalization to compute the Q and R matrices from the input matrix A. The function iterates over each column of A and orthogonalizes it with respect to the previously computed columns of Q.

Listing 5: Gram-Schmidt Process Function

```
void GramSchmidtProcess(Matrix *A, Matrix *Q, Matrix *R) {
    int size = A->size;
    for (int j = 0; j < size; j++) {
        for (int i = 0; i < size; i++) {
            Q->matrix[i][j] = A->matrix[i][j];
        }
        for (int m = 0; m < j; m++) {
            complex double dot_product = 0.0 + 0.0 * I;
            for (int i = 0; i < size; i++) {
                dot_product += conj(Q->matrix[i][m]) * Q->matrix[i][j];
            }
            R->matrix[m][j] = dot_product;
            for (int i = 0; i < size; i++) {
                Q->matrix[i][j] -= R->matrix[m][j] * Q->matrix[i][m];
            }
```

```
        }
        R->matrix[j][j] = 0.0;
        for (int i = 0; i < size; i++) {
            R->matrix[j][j] += creal(Q->matrix[i][j]) * creal(Q->matrix[i][j])
                 + cimag(Q->matrix[i][j]) * cimag(Q->matrix[i][j]);
        }
        R->matrix[j][j] = sqrt(R->matrix[j][j]);
        for (int i = 0; i < size; i++) {
            Q->matrix[i][j] /= R->matrix[j][j];
        }
    }
}
```

QR Decomposition

The ‘QRDecompose‘ function performs QR decomposition iteratively. It applies the Gram-Schmidt process repeatedly to find the Q and R matrices, then multiplies R and Q to update the matrix A. The process is repeated for up to 1000 iterations or until convergence.

The function also computes the eigenvalues from the diagonal elements of the matrix, handling the case of complex conjugate eigenvalues when necessary.

Listing 6: QR Decomposition Function

```
void QRDecompose(Matrix *A, complex double *eigenvalues) {
    int size = A->size;
    Matrix *Q = CreateMatrix(size);
    Matrix *R = CreateMatrix(size);
    Matrix *temp = CreateMatrix(size);
    for (int n = 0; n < 1000; n++) {
        GramSchmidtProcess(A, Q, R);
        MatrixMultiply(R, Q, temp);
        for (int i = 0; i < size; i++) {
            for (int j = 0; j < size; j++) {
                A->matrix[i][j] = temp->matrix[i][j];
            }
        }
    }
    int idx = 0;
    while (idx < size) {
        if ((idx < size - 1) && (cabs(A->matrix[idx + 1][idx]) > 1e-10)) {
            complex double a = A->matrix[idx][idx];
            complex double b = A->matrix[idx + 1][idx];
            complex double c = A->matrix[idx][idx + 1];
            complex double d = A->matrix[idx + 1][idx + 1];
            complex double trace = -(a + d);
            complex double determinant = (a * d - b * c);
            eigenvalues[idx] = (-trace + csqrt(trace * trace - 4.0 *
                determinant)) / 2.0;
            eigenvalues[idx + 1] = (-trace - csqrt(trace * trace - 4.0 *
                determinant)) / 2.0;
            A->matrix[idx + 1][idx] = 0;
```

```
            idx += 2;
        } else {
            eigenvalues[idx] = A->matrix[idx][idx];
            idx++;
        }
    }
    FreeMatrix(Q);
    FreeMatrix(R);
    FreeMatrix(temp);
}
```

Main Function

The main function performs the following tasks: 1. Reads the size and elements of the matrix from the user. 2. Allocates memory for the matrix and eigenvalues. 3. Calls the 'QRDecompose' function to compute the eigenvalues. 4. Prints the eigenvalues and the time taken for execution.

Listing 7: Main Function

```
int main() {
    int size;
    printf("Enter the size of matrix: ");
    scanf("%d", &size);
    if (size <= 0) {
        printf("Matrix size must be positive.\n");
        return 0;
    }
    Matrix *matrix = CreateMatrix(size);
    printf("Enter the elements row-wise (real imag): \n");
    for (int i = 0; i < size; i++) {
        for (int j = 0; j < size; j++) {
            double real, imag;
            scanf("%lf %lf", &real, &imag);
            matrix->matrix[i][j] = real + imag * I;
        }
    }
    complex double *eigenvalues = (complex double*)malloc(size * sizeof(
        complex double));
    clock_t start_time = clock();
    QRDecompose(matrix, eigenvalues);
    clock_t end_time = clock();
    printf("Eigenvalues:\n");
    for (int i = 0; i < size; i++) {
        printf("%.10lf + %.10lfi\n", creal(eigenvalues[i]), cimag(eigenvalues[
            i]));
    }
    printf("Duration of code run: %.10f seconds\n", (double)(end_time -
        start_time) / CLOCKS_PER_SEC);
    FreeMatrix(matrix);
    free(eigenvalues);
    return 0;
}
```

Conclusion

This C program implements QR decomposition using the Gram-Schmidt process to compute the eigenvalues of a square matrix. It utilizes dynamic memory allocation for handling matrices of arbitrary size and supports complex numbers for general applicability. The program's performance is measured by tracking execution time.

Explanation: The `main` function serves as the entry point of the program. It handles user input, initializes the matrix, and computes its eigenvalues using the QR algorithm. It then prints the results along with the execution time.

Time Complexity Analysis

The provided C code computes the eigenvalues of a matrix using the Gram-Schmidt process and the QR algorithm. To determine the overall time complexity, we will analyze the key functions in the code:

1. Matrix Creation (`CreateMatrix`)

The function `CreateMatrix` allocates memory for an $n \times n$ matrix. This involves:

- Allocating memory for the matrix structure, which requires $O(1)$ operations.
- Allocating memory for the array of pointers (rows) for an $n \times n$ matrix, which requires $O(n)$ operations.
- Allocating memory for each row (complex number array) of size n, which requires $O(n^2)$ operations in total.

Thus, the time complexity of `CreateMatrix` is:

$$O(n^2)$$

2. Matrix Multiplication (`MatrixMultiply`)

The function `MatrixMultiply` computes the product of two $n \times n$ matrices. This operation involves iterating over all rows of the first matrix and all columns of the second matrix, performing n multiplications and additions for each pair of row and column.

The time complexity of matrix multiplication is:

$$O(n^3)$$

3. Gram-Schmidt Process (`GramSchmidtProcess`)

The `GramSchmidtProcess` function orthogonalizes the columns of the input matrix A. For each column j, the following steps are performed:

- Calculating inner products with all previous columns: This requires $O(j \cdot n)$ operations for the j-th column.
- Subtracting projections from the j-th column: This step also requires $O(n)$ operations.
- Normalizing the column: This requires $O(n)$ operations.

Since the algorithm iterates over n columns, the total time complexity for Gram-Schmidt is the sum of the work done for each column, leading to a total complexity of:

$$O(n^3)$$

4. Eigenvalue Computation (`QRDecompose`)

The function `QRDecompose` uses the Gram-Schmidt process to iteratively compute the eigenvalues of the matrix. The steps for each iteration are:

- `Gram-Schmidt Process`: Each Gram-Schmidt step requires $O(n^3)$ as calculated earlier.
- `Matrix Multiplication`: After computing the orthogonal matrix Q and upper triangular matrix R, matrix multiplication $R \times Q$ is performed to update the matrix A. This requires $O(n^3)$ operations.

The `QRDecompose` function repeats the Gram-Schmidt process and matrix multiplication for a maximum of 1000 iterations (as specified in the code). Hence, for each iteration, the total complexity is $O(n^3)$, and with up to 1000 iterations, the overall complexity for the eigenvalue computation is:

$$O(1000 \times n^3) = O(n^3)$$

5. Overall Time Complexity

- The matrix creation function `CreateMatrix` has a time complexity of $O(n^2)$. - The Gram-Schmidt process has a time complexity of $O(n^3)$ for each iteration. - The matrix multiplication (used in `MatrixMultiply`) has a time complexity of $O(n^3)$ for each iteration. - The QR decomposition process runs for a maximum of 1000 iterations, with each iteration having a time complexity of $O(n^3)$.

Hence, the dominant factor in the time complexity is the QR decomposition and matrix multiplication, which gives a time complexity of:

$$O(n^3)$$

Space Complexity

- **Matrix storage:** The matrices A, Q, R, and *temp* are all $n \times n$ matrices, so they require $O(n^2)$ space each.
- **Eigenvalue storage:** The array of eigenvalues requires $O(n)$ space.

Thus, the overall space complexity is:

$$O(n^2)$$

Summary

- **Time Complexity**: $O(n^3)$ due to the repeated Gram-Schmidt process and matrix multiplication.
- **Space Complexity**: $O(n^2)$ due to the storage of matrices.

6 Conclusion

Therefore the Code is a decent implementation of the QR Algorithm for calculating accurate eigenvalues within a minimal time owing to its moderate time complexity. It can compute moderate to large matrices with not much difficulty.

7 References

Books, EE1010 lecture notes, Matrix Mathematics and research articles.