Finding Eigenvalues Using Hessenberg and QR Decomposition

EE24Btech11022 - Eshan Sharma

I. Introduction

In this project, we focus on finding the eigenvalues and eigenvectors of a matrix using two major methods: the Hessenberg transformation and the QR algorithm. These methods are widely used in numerical linear algebra for solving eigenvalue problems efficiently.

Eigenvalues and eigenvectors are fundamental in various applications such as stability analysis, vibration analysis, and principal component analysis.

II. Theory

A. Hessenberg Transformation

The Hessenberg transformation is a crucial technique for simplifying the eigenvalue computation of a matrix. The transformation converts a square matrix A into its Hessenberg form H, which is an upper triangular matrix with non-zero elements only on the first sub-diagonal. This form is useful for algorithms like QR iteration, which compute eigenvalues more efficiently.

To perform the transformation, we use Householder reflections, which are applied iteratively to zero out the entries below the first sub-diagonal of the matrix. The process involves constructing Householder vectors w for each iteration and then applying the Householder matrix P to transform A into H.

Steps for the Hessenberg Transformation: For an $n \times n$ matrix A, the transformation is achieved through $n-2$ iterations, with each iteration focusing on eliminating the elements below the diagonal in the current column. Here are the steps in detail:

1) **Compute the Householder Vector** w: For each iteration i, we compute a Householder vector w that will be used to eliminate the elements below the i-th column of A.
2) **Form the Householder Matrix** P: Using the vector w, the Householder matrix P is constructed as $P = I - 2ww^H$, where w^H is the conjugate transpose of w.
3) **Update the Matrix** A: The matrix A is updated as $A' = PAP^H$, where P^H is the conjugate transpose of P. This transformation zeros out the elements below the sub-diagonal in the i-th column of A.

Mathematical Derivation for w and P: Let's go through the mathematical process involved in calculating w and constructing the Householder matrix P.

a) Step 1: Finding the Householder Vector w: For a given column i, the Householder vector w is designed to zero out all the elements below the i-th diagonal element. Here's the process:

- First, extract the subvector x from the i-th column, starting from row $i+1$ to row n:

$$x = \begin{bmatrix} a_{i+1,i} \\ a_{i+2,i} \\ \vdots \\ a_{n,i} \end{bmatrix}$$

- Compute the norm $\|x\|$ of this vector:

$$\|x\| = \sqrt{\sum_{k=i+1}^{n} |a_{k,i}|^2}$$

- Define α as:
$$\alpha = -\text{sign}(a_{i+1,i}) \cdot \|x\|$$
- Define r as:
$$r = \sqrt{\frac{\alpha^2}{2} - \frac{\alpha \cdot a_{i+1,i}}{2}}$$
- The Householder vector w for the $i-th$ iteration is then calculated as:
$$w = \begin{bmatrix} w_i \\ w_{i+1} \\ w_{i+2} \\ . \\ . \\ w_n \end{bmatrix}$$
where w_i is set to zero (since it's the starting point of the transformation) and
$$w_{i+1} = \frac{a_{i+1,i} - \alpha}{2r}$$
Also, for $j = i+2, \ldots, n$, the elements of w are given by:
$$w_j = \frac{a_{ji}}{2r}$$
where r is a normalization constant used to scale the elements.

b) Step 2: Construct the Householder Matrix P: Once w is computed, the Householder matrix P is formed using the formula:
$$P = I - 2ww^H$$
where I is the identity matrix of size $n \times n$, and w^H is the conjugate transpose of w. The matrix P is symmetric, and it is used to perform the similarity transformation on A to produce the Hessenberg form.

c) Step 3: Apply the Transformation: After constructing the Householder matrix P, we update the matrix A as follows:
$$A' = PAP^H$$
This process is repeated for each column until all the elements below the diagonal are zero, except for the first sub-diagonal.

C Code Implementation for Hessenberg Transformation: The following C code implements the procedure described above:

```
comp** hess(comp** mat, int a) {
        comp** A = mat;
        for (int i = 0; i < a - 2; i++) {
                comp** w = mzeroes(a, 1);
                double norm = 0.0;

                // Calculate the norm of the column below diagonal
                for (int j = i + 1; j < a; j++) {
                        norm += creal(A[j][i]) * creal(A[j][i]);
                }
                norm = sqrt(norm);

                // Calculate alpha and Householder vector w
                double alpha = -sign(creal(A[i + 1][i])) * norm;
                double r = sqrt(0.5 * alpha * alpha - 0.5 * alpha * creal(A[i + 1][i]));
                w[i + 1][0] = (A[i + 1][i] - alpha) / (2 * r);
                for (int j = i + 2; j < a; j++) {
                        w[j][0] = A[j][i] / (2 * r);
                }

                // Construct the Householder transformation matrix
                comp** P = miden(a);
                comp** wwT = mmul2(w, mT(w, a, 1), a, 1, a);
                comp** H = madd(P, mmul1(wwT, a, a, -2.0 + 0 * I), a, a);

                // Update the matrix A
                A = mmul2(H, A, a, a, a);
                A = mmul2(A, mT(H, a, a), a, a, a);

                // Free allocated memory for intermediate matrices
                for (int j = 0; j < a; j++) {
                        free(H[j]);
                        free(P[j]);
                        free(wwT[j]);
                }
                free(H);
                free(P);
                free(wwT);
                free(w);
                }
        return A; // Return the transformed Hessenberg matrix
}
```

B. QR Algorithm

The QR algorithm is an iterative method used to compute the eigenvalues of a matrix. It decomposes the matrix A into the product of an orthogonal matrix Q and an upper triangular matrix R, such that:

$$A_0 = QR$$

The next iteration involves computing:

$$A_1 = RQ \quad \text{or equivalently} \quad A_1 = Q_1 R_1$$

This process is repeated iteratively as:

$$A_2 = R_1 Q_1, \quad A_3 = Q_2 R_2, \quad \ldots, \quad A_{n+1} = R_n Q_n$$

Through these iterations, the matrices $A_0, A_1, A_2, \ldots, A_n$ are all unitarily similar, which implies they share the same eigenvalues.

Over successive iterations, A_n converges to a triangular matrix whose diagonal elements are the eigenvalues of A. The eigenvectors can be computed by accumulating the transformations that generate Q.

The iterative process can be summarized as:

$$A_0 = Q_0 R_0, \quad A_1 = R_0 Q_0, \quad A_1 = Q_1 R_1, \quad A_2 = R_1 Q_1, \quad \ldots$$

The key property of this sequence is that all intermediate matrices $A_0, A_1, \ldots, A_n$ will have the same eigenvalues as the original matrix A.

As the iterations progress, the matrix A_n takes on a form where the diagonal elements approximate the eigenvalues of A, as shown below:

$$A_n = \begin{bmatrix} \lambda_1 & a_{12} & \cdots & a_{1n} \\ \epsilon & \lambda_2 & \cdots & a_{2n} \\ 0 & \epsilon & \ddots & \vdots \\ 0 & 0 & \cdots & \lambda_n \end{bmatrix}$$

Here, $\lambda_1, \lambda_2, \ldots, \lambda_n$ are the eigenvalues of the original matrix, and ϵ represents small off-diagonal values that approach zero as $n \to \infty$.

C. Gram-Schmidt Process

The QR decomposition is commonly achieved using the Gram-Schmidt process, which orthogonalizes the columns of a matrix A to generate the orthogonal matrix Q. The process proceeds as follows:

Given a matrix $A = [a_1, a_2, \ldots, a_n]$, where a_i are the column vectors:

1) Start with $q_1 = \frac{a_1}{\|a_1\|}$, where $\|a_1\| = \sqrt{a_1^T a_1}$.
2) For each subsequent column a_k, subtract the projection of a_k onto the already computed $q_1, q_2, \ldots, q_{k-1}$:

$$u_k = a_k - \sum_{i=1}^{k-1} \text{proj}_{q_i}(a_k)$$

where the projection is given by:

$$\text{proj}_{q_i}(a_k) = \frac{q_i^T a_k}{q_i^T q_i} q_i$$

3) Normalize u_k to obtain q_k:

$$q_k = \frac{u_k}{\|u_k\|}$$

At the end of this process, the matrix $Q = [q_1, q_2, \ldots, q_n]$ is orthogonal, and the matrix R is obtained as:

$$R = Q^T A$$

1) Implementation in Code: Below is a code snippet demonstrating the QR decomposition with Gram-Schmidt orthogonalization:

```
comp*** QR(comp** A, int m, int n) {
        comp** Q = mzeroes(m, n); // Matrix Q
        comp** R = mzeroes(n, n); // Matrix R

        for (int i = 0; i < n; i++) {
                comp** a = mcol(A, m, n, i); // Get the i-th column of A
                comp** q = mcopy(a, m, 1); // Start q as a copy of the i-th column of A

                // Orthogonalize q against the previous q's in Q
                for (int j = 0; j < i; j++) {
                        comp x = mmul2(mT(a, m, 1), mcol(Q, m, n, j), 1, m, 1)[0][0];
                        q = madd(q, mmul1(mcol(Q, m, n, j), m, 1, -creal(x)), m, 1);
                        R[j][i] = x;
                }

                // Normalize q to get the i-th column of Q
                R[i][i] = vnorm(q, m);
                q = mmul1(q, m, 1, 1.0 / creal(R[i][i])); // Normalize q

                // Store the normalized q in the i-th column of Q
                for (int j = 0; j < m; j++) {
                        Q[j][i] = q[j][0];
                }
        }

        // Allocate memory for returning Q and R matrices
        comp*** ret = (comp***) malloc(sizeof(comp**) * 2);
        ret[0] = Q; // First element is Q matrix
        ret[1] = R; // Second element is R matrix
        return ret;

}
```

The QR algorithm leverages this decomposition iteratively to converge to the eigenvalues of the matrix. Further optimizations, such as the use of the Hessenberg form for A, improve computational efficiency.

2) Implementation in Code: Below is a code snippet demonstrating the iterations being carried out for the eigen values:

```
void eig(comp** A, int n) {
        comp** B = mzeroes(n, n);
        comp** Q;
        comp** R;
        int iterations = 0;
        for (iterations = 0; iterations < 1000; iterations++) {
                // Perform QR decomposition
                comp*** qr = QR(A, n, n);
                Q = qr[0]; // Extract Q matrix
                R = qr[1]; // Extract R matrix

                // Multiply R * Q to get the new matrix B
                B = mmul2(R, Q, n, n, n);

                // Check if the off-diagonal elements are close to 0 (convergence)
                double diff = 0.0;
                for (int i = 0; i < n - 1; i++) {
                        diff += cabs(A[i][i+1]);
                }

                if (diff < 1e-10) { // Convergence threshold
                        printf("\nConverged after %d iterations.\n", iterations);
                        break;
                }

                // Update A to B
                for (int i = 0; i < n; i++) {
                        for (int j = 0; j < n; j++) {
                                A[i][j] = B[i][j];
                        }
                }
}
```

This code snippet implements the QR algorithm to compute the eigenvalues of a given matrix A. The process involves performing the QR decomposition in each iteration, followed by updating A as RQ, where R and Q are the upper triangular and orthogonal matrices, respectively. The algorithm monitors convergence by checking if the off-diagonal elements approach zero, indicating that A is nearing an upper triangular form. This iterative approach ensures the eigenvalues appear as the diagonal elements of the final matrix.

III. Methodology

We implemented the Hessenberg and QR algorithms using C. The following steps were taken:

1) Define the matrix A to find its eigenvalues.
2) Use the Householder transformation to compute the Hessenberg form of the matrix.
3) Apply the QR algorithm iteratively to the Hessenberg matrix to find the eigenvalues.
4) Extract the eigenvectors using the final orthogonal matrices from the QR decomposition.

IV. Results

The eigenvalues and eigenvectors were computed for a sample matrix using the Hessenberg and QR algorithms. The following output was obtained:

1) **Original Matrix A:**

$$A = \begin{pmatrix} 2.00 + 0.00i & 4.00 + 0.00i & 1.00 + 0.00i & 3.00 + 0.00i & 5.00 + 0.00i & 7.00 + 0.00i \\ 6.00 + 0.00i & 1.00 + 0.00i & 2.00 + 0.00i & 4.00 + 0.00i & 3.00 + 0.00i & 8.00 + 0.00i \\ 5.00 + 0.00i & 2.00 + 0.00i & 3.00 + 0.00i & 6.00 + 0.00i & 1.00 + 0.00i & 9.00 + 0.00i \\ 7.00 + 0.00i & 8.00 + 0.00i & 4.00 + 0.00i & 2.00 + 0.00i & 6.00 + 0.00i & 3.00 + 0.00i \\ 1.00 + 0.00i & 7.00 + 0.00i & 9.00 + 0.00i & 5.00 + 0.00i & 8.00 + 0.00i & 4.00 + 0.00i \\ 3.00 + 0.00i & 4.00 + 0.00i & 2.00 + 0.00i & 7.00 + 0.00i & 5.00 + 0.00i & 6.00 + 0.00i \end{pmatrix}$$

2) **Hessenberg Matrix H:**

$$H = \begin{pmatrix} 2.00 + 0.00i & -6.94 + 0.00i & -5.80 + 0.00i & 2.58 + 0.00i & -1.40 + 0.00i & 3.10 + 0.00i \\ -10.95 + 0.00i & 17.08 + 0.00i & 9.15 + 0.00i & -1.67 + 0.00i & 1.61 + 0.00i & -4.60 + 0.00i \\ -0.00 + 0.00i & 12.99 + 0.00i & 6.39 + 0.00i & 0.24 + 0.00i & -2.21 + 0.00i & 1.25 + 0.00i \\ -0.00 + 0.00i & -0.00 + 0.00i & -2.14 + 0.00i & 0.08 + 0.00i & 1.49 + 0.00i & -1.14 + 0.00i \\ 0.00 + 0.00i & 0.00 + 0.00i & -0.00 + 0.00i & -3.95 + 0.00i & -4.87 + 0.00i & 3.89 + 0.00i \\ 0.00 + 0.00i & -0.00 + 0.00i & -0.00 + 0.00i & 0.00 + 0.00i & -6.05 + 0.00i & 1.32 + 0.00i \end{pmatrix}$$

3) **Eigenvalues:**

$$\lambda_1 = 27.41{+}0.00i, \lambda_2 = 0.60{+}0.00i, \lambda_3 = 1.03{+}0.00i, \lambda_4 = -3.04{+}0.00i, \lambda_5 = -1.45{+}0.00i, \lambda_6 = -2.55{+}0.00i$$

V. Conclusion

In this project, we successfully applied the Hessenberg transformation and the QR algorithm to compute the eigenvalues and eigenvectors of a given matrix. These methods provided accurate results, and their implementation in C demonstrated the power of numerical linear algebra techniques in solving eigenvalue problems.

VI. Matrix Functions

Below are the matrix functions:

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

typedef double complex comp;

void mprint(comp** mat, int a, int b){
        printf("[\n");
        for(int i = 0; i < a; i++) {
                printf("[ ");
                for(int j = 0; j < b; j++) {
                        printf("%lf + %lfi ", creal(mat[i][j]), cimag(mat[i][j]));
                }
                printf("]\n");
        }
        printf("]\n");
}

comp** mzeroes(int a, int b){
        comp** mat = (comp**) malloc(sizeof(comp*) * a); // Corrected to allocate pointer array
        for(int i = 0; i < a; i++) {
                mat[i] = (comp*) malloc(sizeof(comp) * b);
                for(int j = 0; j < b; j++) {
                        mat[i][j] = 0 + 0*I;
                }
        }
        return mat;
}

comp** miden(int a){
        comp** mat = mzeroes(a, a);
        for(int i = 0; i < a; i++) {
                mat[i][i] = 1 + 0*I;
        }
        return mat;
}

comp** madd(comp** mat1, comp** mat2, int a, int b){
        comp** mat = (comp**) malloc(sizeof(comp*) * a); // Corrected to allocate pointer array
        for(int i = 0; i < a; i++) {
                mat[i] = (comp*) malloc(sizeof(comp) * b);
                for(int j = 0; j < b; j++) {
                        mat[i][j] = mat1[i][j] + mat2[i][j];
                }
        }
```

```
        return mat;
}

comp** mmul1(comp** mat, int a, int b, comp k){
        comp** nmat = (comp**) malloc(sizeof(comp*) * a);
        for(int i = 0; i < a; i++) {
                nmat[i] = (comp*) malloc(sizeof(comp) * b);
                for(int j = 0; j < b; j++) {
                        nmat[i][j] = k * mat[i][j];
                }
        }
        return nmat;
}

comp** mmul2(comp** mat1, comp** mat2, int a, int b, int c){
        comp** mat = mzeroes(a, c);
        for(int i = 0; i < a; i++) {
                for(int j = 0; j < c; j++) {
                        for(int k = 0; k < b; k++) {
                                mat[i][j] += mat1[i][k] * mat2[k][j];
                        }
                }
        }
        return mat;
}

comp** mT(comp** mat, int a, int b){
        comp** nmat = (comp**) malloc(sizeof(comp*) * b);
        for(int i = 0; i < b; i++) {
                nmat[i] = (comp*) malloc(sizeof(comp) * a);
                for(int j = 0; j < a; j++) {
                        nmat[i][j] = conj(mat[j][i]);
                }
        }
        return nmat;
}

comp** mcol(comp** mat, int a, int b, int k){
        comp** nmat = (comp**) malloc(sizeof(comp*) * a);
        for(int i = 0; i < a; i++) {
                nmat[i] = (comp*) malloc(sizeof(comp)); // Allocating only for one element per row
                nmat[i][0] = mat[i][k]; // Only store the k-th column in each row
        }
        return nmat;
}

comp** mcopy(comp** mat, int a, int b){
        comp** newmat = (comp**) malloc(sizeof(comp)*a);
        for(int i = 0; i < a; i++){
                newmat[i] = (comp*) malloc(sizeof(comp)*b);
```

```
                for(int j = 0; j < b; j++){
                        newmat[i][j] = mat[i][j];
                }
        }
        return newmat;
}

double vnorm(comp** vec, int m) {
        double sum = 0;
        for(int i = 0; i < m; i++) {
                sum += creal(vec[i][0]) * creal(vec[i][0]) + cimag(vec[i][0]) * cimag(vec[i][0]);
        }
        return sqrt(sum);
        }
```