Software Assignment: Eigenvalue Calculation

Shreedhanvi Yadlapally
AI24BTECH11036

Abstract

This report presents an implementation of the QR algorithm with optimizations for calculating the eigenvalues of matrices. The QR algorithm was chosen for its effectiveness and balance between accuracy and computational efficiency. The report also the QR algorithm with other methods, highlighting its advantages in handling different types of matrices. These optimizations make the QR algorithm an efficient choice for practical eigenvalue computations.

1 QR-Algorithm

The QR algorithm is a popular iterative method for finding the eigenvalues of a matrix. It involves decomposing the matrix into a product of an orthogonal matrix **Q** and an upper triangular matrix **R**, then reassembling these factors in a specific way to iteratively converge towards a diagonal or nearly upper-triangular matrix, where the eigenvalues appear as entries on the main diagonal.

For a general $n \times n$ matrix **A**

Step-by-step breakdown

Checking for symmetry

$$A \text{ is symmetric is and only if } A[i, j] = A[j, i] \forall i, j$$

Significance: Checking symmetry helps optimize calculations. For symmetric matrices, we can use specific methods like tridiagonalization that are more efficient than general algorithms.

Householder Transformation (Tridiagonalization)

Mathematical Steps: Calculate Norm:

$$\|u\| = \sqrt{\sum_{i=k+1}^{n} |A[i,k]|^2}$$

Construct Householder Vector:

$$\alpha = -\operatorname{sign}(A[k+1,k]) \times \|u\|$$

$$r = \sqrt{\frac{1}{2}(\alpha^2 - \alpha \cdot A[k+1,k])}$$

$$v[k+1] = \frac{A[k+1,k] - \alpha}{2r}$$

$$v[i] = \frac{A[i,k]}{2r} \text{ for } i > k+1$$

Apply Transformation:

$$A[i, j] = A[i, j] - 2v[i] \sum_{l=k+1}^{n} v[l]A[l, j]$$

Significance: Transforming the matrix to tridiagonal form simplifies the subsequent QR iterations by reducing computational complexity and improving numerical stability.

QR Iteration With Wilkinson Shifts

Mathematical Steps:

- Wilkinson Shift calculation:

$$d = \frac{a_{n-2,n-2} - a_{n-1,n-1}}{2}$$

$$\mu = a_{n-1,n-1} - \frac{{a_{n-2,n-1}}^2}{d + \sqrt{d^2 + {a_{n-2,n-1}}^2}}$$

- Apply the shift:

$$A = A - \mu I$$

QR decomposition using Givens Rotations:

- For each element:

$$a = a_{k,k} - \mu, b = a_{k+1,k}$$

$$r = \sqrt{a^2 + b^2}$$

$$c = \frac{a}{r}, s = \frac{-b}{r}$$

- Update rows and columns:

- Reverse the shift

$$A = A + \mu I$$

Convergence check:

$$\text{off-diagonal norm} = \sum_{i=1}^{n-1} \left| a_{i,i-1} \right|$$

If off-diagonal norm < TOL, stop the iterations.

Significance: Wilkinson shifts improve convergence by focusing the QR iterations, making them more stable and efficient.

Givens rotations zero out sub-diagonal elements, transforming the matrix into an upper triangular form, which is easier to analyze.

2 Time Complexity Analysis

1. Matrix Storage and initialization: Reading $n \times n$ matrix elements requires $O(n^2)$ operations.
2. Checking for symmetry:
   - We need to check each pair of elements $(A, [i, j])$ and $(A[j, i])$ for $i \leq j$
   - This requires examining $\frac{n(n-1)}{2}$ elements.
   - Time complexity: $O(n^2)$
3. Tridiagonalization
   (a) Norm Calculation: For each column k, calculate the norm of the sub-column from $k + 1$ to n.
      - Time Complexity: $O(n - K)$ for each column k

- Total for all columns:

$$O(n) + O(n-1) + \ldots O(1) = O(n^2)$$

(b) Vector Updates: Normalizing and updating the Householder vector involves operations proportional to the size of the sub-matrix.
- Time Complexity: $O(n-K)$ for each column k
- Total for all columns: $O(n^2)$

(c) Matrix Updates: Each update involves a rank-1 update of the matrix A requiring $O(n^2)$ operations.
- Outer loop: $O(n-k)rows$
- Inner loop: $O(n-k)columns$
- Each iteration: $O((n-k)^2)$ multiplications.

$$O((n-1)^2) + O((n-2)^2) + \ldots + O(1^2) = O(n^3)$$

Since we do it for n columns, the time complexity would be $O(n^3)$
Overall time-complexity for tridiagonalization $O(n^3)$

4. QR Iterations with Givens Rotations:
   (a) Givens Rotations: Each rotation involves $O(n)$ operations
   - We perform these rotations for each off-diagonal element, there are $n-1$ off-diagonal elements per iteration to eliminate.
   - Time Complexity per iteration is $O(n^2)$

   (b) Convergence: Typically, the number of iterations required for convergence is proportional to the matrix size.
   - Total iterations: $O(n^2)$
5. Eigenvalues for small complex matrices,
   (a) for $n = 1$: $O(1)$ (returning single element)
   (b) for $n = 2$:
   - Calculating the trace and determinant: $O(1)$
   - Computing the discriminant using the square root: $O(1)$

   Total: $O(1)$.

Based on this, the overall time complexity of this algorithm is $O(n^3)$.

3 Memory Usage Analysis

1. Memory for Matrix Storage
   (a) Real Symmetric Matrices: A real symmetric matrix pf size $n \times n$ requires n^2 elements each of size `sizeof(double)` (typically 8 bytes).
   (b) Small Complex Matrices: For $n = 1$ or $n = 2$, $n \times n$ elements, each of size `sizeof(double complex)` (typically 16 bytes).
2. Temporary Arrays and Variables
   (a) Tridiagonalization: A temporary vector v of size n is allocated on the stack. This uses n `double` elements. ($n \times 8$ bytes)
   (b) QR Algorithm: A copy of the matrix A is created for transformations, using $n^2 \times 8$ bytes of memory.
   (c) Eigenvalues array: An array of size n is allocated, requiring $n \times 8$ bytes.
3. Function Overheads such as `apply_givens_rotation` and `tridiagonalize` use local variables and temporary storage, but these are small in comparison to the matrix.

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

```
#define EPSILON 1e−8
#define MAX_ITER 1000

#define GET(A, n, i, j) (A[(i) ∗ (n) + (j)])
#define SET(A, n, i, j, value) (A[(i) ∗ (n) + (j)] = (value))

// Helper functions
double sign(double x) {
    return (x > 0) − (x < 0);
}

// Access elements in a 1D representation of the NxN matrix
double get_element(double ∗matrix, int N, int row, int col) {
    return matrix[row ∗ N + col];
}

void set_element(double ∗matrix, int N, int row, int col, double value) {
    matrix[row ∗ N + col] = value;
}

// Check if a matrix is symmetric
int is_symmetric(double ∗matrix, int N) {
    for (int i = 0; i < N; i++) {
        for (int j = i + 1; j < N; j++) {
            if (fabs(get_element(matrix, N, i, j) − get_element(matrix, N, j, i)) > EPSILON) {
                return 0;
            }
        }
    }
    return 1;
}

// Givens rotation for QR decomposition (for symmetric matrices)
void apply_givens_rotation(double ∗matrix, int N, int p, int q, double c, double s) {
    for (int i = 0; i < N; i++) {
        double tmp1 = c ∗ get_element(matrix, N, i, p) − s ∗ get_element(matrix, N, i, q);
        double tmp2 = s ∗ get_element(matrix, N, i, p) + c ∗ get_element(matrix, N, i, q);
        set_element(matrix, N, i, p, tmp1);
        set_element(matrix, N, i, q, tmp2);
    }
    for (int i = 0; i < N; i++) {
        double tmp1 = c ∗ get_element(matrix, N, p, i) − s ∗ get_element(matrix, N, q, i);
        double tmp2 = s ∗ get_element(matrix, N, p, i) + c ∗ get_element(matrix, N, q, i);
        set_element(matrix, N, p, i, tmp1);
        set_element(matrix, N, q, i, tmp2);
    }
}

// Perform tridiagonalization for symmetric matrices
void tridiagonalize(double ∗matrix, int N) {
    for (int k = 0; k < N − 2; k++) {
        double norm = 0;
        for (int i = k + 1; i < N; i++) norm += get_element(matrix, N, i, k) ∗ get_element(matrix,
            N, i, k);
        norm = sqrt(norm);

        double alpha = −sign(get_element(matrix, N, k + 1, k)) ∗ norm;
        double r = sqrt(0.5 ∗ (alpha ∗ alpha − get_element(matrix, N, k + 1, k) ∗ alpha));
```

```
            double v[N];
            for (int i = 0; i < N; i++) v[i] = 0;
            v[k + 1] = (get_element(matrix, N, k + 1, k) − alpha) / (2 ∗ r);
            for (int i = k + 2; i < N; i++) v[i] = get_element(matrix, N, i, k) / (2 ∗ r);

            // Apply the Householder transformation
            for (int i = k; i < N; i++) {
                double sum = 0;
                for (int j = k; j < N; j++) sum += get_element(matrix, N, i, j) ∗ v[j];
                for (int j = k; j < N; j++) set_element(matrix, N, i, j, get_element(matrix, N, i, j) − 2
                    ∗ sum ∗ v[j]);
            }
            for (int i = 0; i < N; i++) {
                double sum = 0;
                for (int j = k; j < N; j++) sum += v[j] ∗ get_element(matrix, N, j, i);
                for (int j = k; j < N; j++) set_element(matrix, N, j, i, get_element(matrix, N, j, i) − 2
                    ∗ sum ∗ v[j]);
            }
        }
}

// Perform QR iteration for eigenvalues
void qr_algorithm(double ∗matrix, int N, double eigenvalues[]) {
        double A[N ∗ N];
        for (int i = 0; i < N; i++)
            for (int j = 0; j < N; j++)
                set_element(A, N, i, j, get_element(matrix, N, i, j));

        for (int iter = 0; iter < MAX_ITER; iter++) {
            // Apply Givens rotations to create a tridiagonal form
            for (int i = 0; i < N − 1; i++) {
                double a = get_element(A, N, i, i);
                double b = get_element(A, N, i + 1, i);
                double r = sqrt(a ∗ a + b ∗ b);
                double c = a / r;
                double s = −b / r;

                apply_givens_rotation(A, N, i, i + 1, c, s);
            }

            // Check for convergence
            double off_diagonal_sum = 0;
            for (int i = 0; i < N − 1; i++) {
                off_diagonal_sum += fabs(get_element(A, N, i, i + 1));
            }
            if (off_diagonal_sum < EPSILON) break;
        }

        // Eigenvalues are the diagonal elements after convergence
        for (int i = 0; i < N; i++) {
            eigenvalues[i] = get_element(A, N, i, i);
        }
}

// Calculate eigenvalues for 1x1 or 2x2 complex matrices
void eigenvalues_small(double complex ∗A, int n) {
        if (n == 1) {
            printf("Eigenvalue:␣%.10f␣+␣%.10fi\n", creal(GET(A, n, 0, 0)), cimag(GET(A, n, 0, 0)))
                ;
```

```
    } else if (n == 2) {
        double complex trace = GET(A, n, 0, 0) + GET(A, n, 1, 1);
        double complex determinant = GET(A, n, 0, 0) * GET(A, n, 1, 1) - GET(A, n, 0, 1) *
            GET(A, n, 1, 0);
        double complex discriminant = csqrt(trace * trace - 4.0 * determinant);

        double complex lambda1 = (trace + discriminant) / 2.0;
        double complex lambda2 = (trace - discriminant) / 2.0;

        printf("Eigenvalues: %.10f + %.10fi, %.10f + %.10fi\n",
               creal(lambda1), cimag(lambda1), creal(lambda2), cimag(lambda2));
    }
}

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

    if (N <= 0) {
        printf("Invalid matrix size.\n");
        return 1;
    }

    if (N == 1 || N == 2) {
        double complex *A = (double complex *)malloc(N * N * sizeof(double complex));

        if (A == NULL) {
            printf("Memory allocation failed.\n");
            return 1;
        }

        printf("Enter the elements of the matrix (real and imaginary parts):\n");
        for (int i = 0; i < N; i++) {
            for (int j = 0; j < N; j++) {
                double real, imag;
                printf("A[%d][%d] = ", i, j);
                scanf("%lf %lf", &real, &imag);
                SET(A, N, i, j, real + imag * I);
            }
        }

        eigenvalues_small(A, N);
        free(A);
        return 0;
    }

    double *matrix = (double *)malloc(N * N * sizeof(double));
    if (matrix == NULL) {
        printf("Memory allocation failed.\n");
        return 1;
    }

    printf("Enter the elements of the real symmetric matrix:\n");
    for (int i = 0; i < N; i++) {
        for (int j = 0; j < N; j++) {
            double value;
            printf("A[%d][%d] = ", i, j);
            scanf("%lf", &value);
```

```
                set_element(matrix, N, i, j, value);
            }
        }

        double eigenvalues[N];

        // Check if the matrix is symmetric and tridiagonalize if it is
        if (is_symmetric(matrix, N)) {
            printf("Matrix is symmetric. Optimizing for symmetric matrix.\n");
            tridiagonalize(matrix, N);
        } else {
            printf("Matrix is not symmetric. Using general QR algorithm.\n");
        }

        // Apply QR algorithm to find eigenvalues
        qr_algorithm(matrix, N, eigenvalues);

        printf("Eigenvalues:\n");
        for (int i = 0; i < N; i++) {
            printf("%.6f\n", eigenvalues[i]);
        }

        free(matrix);
        return 0;
}
```

4 Suitability for Different Types of Matrices

1. Symmetric or Hermitian Matrices:
   - The QR algorithm with shifts is particularly well-suited for symmetric (or Hermitian) matrices.
   - The tridiagonalization step simplifies the matrix structure, making the QR iterations more efficient.
2. General Matrices:
   - For general non-symmetric matrices, the QR algorithm still works but may require more iterations for convergence.
   - Hessenberg reduction is used instead of tridiagonalization, transforming the matrix into an upper Hessenberg form, which is a more general approach but less efficient than the symmetric case.
3. Sparsity:
   - The QR algorithm is less efficient for large sparse matrices because it tends to fill in zero entries, leading to a denser matrix and increased computational cost.
   - Alternative methods like the Lanczos algorithm may be more suitable for sparse matrices.

5 Other Algorithms to find Eigenvalues

1. **Power Iteration**: Iteratively multiplies a random vector by the matrix to converge to the largest eigenvalue. Eigenvalue λ_1 is computed as

   $$\lambda_1 = \frac{v^T A v}{v^T v}$$

   where v is the vector that converges to the dominant eigenvector.

   Pros:

   - Simple to implement.

- Efficient for sparse matrices and finding the largest eigenvalue.
- Requires $O(N^2)$ operations per iteration for dense matrices, and $O(N)$ for sparse.

Cons:

- Only computes the largest eigenvalue (or the one with the largest magnitude).
- Slow convergence if eigenvalues are close in magnitude.
- Does not compute all eigenvalues.

2. **Inverse Iteration (Shifted Power Iteration)**: A modification of power iteration that finds eigenvalues near a given shift μ by solving:

$$(A - \mu I)^{-1} v = \lambda v$$

Useful when a few specific eigenvalues are needed.

Pros:

- Fast convergence
- Capable of finding eigenvalues close to a specified shift, making it suitable for locating specific eigenvalues within a spectrum

Cons:

- The success and speed of convergence depend heavily on the choice of the initial shift. A poor choice can lead to slow convergence or convergence to a wrong eigenvalue.
- Each iteration involves solving a linear system, which can introduce numerical instability, especially for poorly conditioned matrices.

3. **Jacobi Method**: The Jacobi method is an iterative algorithm used primarily for finding the eigenvalues and eigenvectors of a symmetric matrix. Iterative rotation is done until it converges.

Pros:

- Easy to implement and understand.
- Guaranteed to converge for symmetric matrices.
- Each rotation can be done independently, making it suitable for parallel processing.

Cons:

- Converges very slowly for large matrices, slower than QR.
- Primarily designed for symmetric matrices, does not generalize well to non-symmetric matrices.

4. **Divide-and-Conquer Method**: Splits a matrix into smaller submatrices, computes eigenvalues for submatrices recursively, and combines results.

Pros:

- The method is well-suited for large matrices because it breaks the problem into smaller, more manageable pieces.
- Parallel processing, as subproblems can be solved independently.
- smaller subproblems so scales well, more numerical stability, really good for large dense matrices.

Cons:

- The method can be complex to implement due to the need for efficient partitioning and merging techniques.
- Requires additional storage for intermediate submatrices and results, which can increase memory usage.
- overhead for small matrices.

5. **Lanczos Algorithm**: The Lanczos algorithm is a powerful method for finding eigenvalues and eigenvectors of large sparse symmetric matrices.

Pros:

- Particularly efficient for large, sparse matrices.

- Converges quickly for well-conditioned problems, especially for finding the largest or smallest eigenvalues.
- Conceptually simple steps involving matrix-vector multiplications and orthogonalization.

Cons:

- Initial Vector Sensitivity: The choice of the initial vector can affect the convergence and accuracy of the results.
- Prone to numerical instability and loss of orthogonality over many iterations.
- Computing accurate eigenvectors can be more challenging and often requires re-orthogonalization techniques.

6. **Arnoldi Iteration**: It is used for finding eigenvalues and eigenvectors of large, sparse non-symmetric matrices. It is an extension of the Lanczos algorithm and can handle both symmetric and non-symmetric matrices.

   Pros:

   - Works for both symmetric and non-symmetric matrices
   - Particularly efficient for large, sparse matrices

   Cons:

   - Prone to numerical instability and loss of orthogonality over many iterations
   - Computing accurate eigenvectors can be more challenging and often requires re-orthogonalization techniques
   - Requires storing multiple vectors, which can increase memory usage for large problems.

7. **Singular Value Decomposition**: It decomposes a matrix into three simpler matrices, revealing intrinsic properties such as the range, rank, and null space of the original matrix. Computes eigen values of A^TA or AA^T by factoring $A = U\Sigma V^T$

   Pros:

   - is not restricted to square matrices
   - Provides additional insights into the matrix structure, such as rank and range.
   - Numerically stable and robust to rounding errors.

   Cons:

   - Indirect Method: For finding eigenvalues, direct methods such as the QR algorithm are typically faster and more efficient.
   - Requires significant memory to store the three matrices U, Σ, V.

8. **Davidson Algorithm**: The Davidson algorithm is an iterative method used to compute a few of the smallest or largest eigenvalues of a large, sparse, real symmetric matrix.

   Pros:

   - The algorithm converges quickly for matrices that are nearly diagonal or where the eigenvectors are close to the identity matrix.
   - It excels at finding a few of the smallest or largest eigenvalues, which is often what is needed in practical applications.
   - memory efficient compared to methods that operate on the entire matrix.
   - It can handle very large matrices efficiently.

   Cons:

   - To maintain stability, re-orthogonalization steps may be required, adding complexity and computational cost, can suffer from loss of orthogonality.
   - not suitable for non symmetric
   - Implementing the Davidson algorithm can be complex