SOFTWARE ASSIGNMENT

EE24BTECH11004 - ANKIT

November 18, 2024

1 What Are Eigenvalues?

Eigenvalues are a fundamental concept in linear algebra, widely used in mathematics, physics, engineering, and computer science. They describe important properties of matrices and linear transformations.

1.1 Formal Definition

Given a square matrix A of size $n \times n$, a scalar λ is called an **eigenvalue** of A if there exists a non-zero vector $\mathbf{v}$ (called an **eigenvector**) such that: $A\mathbf{v} = \lambda\mathbf{v}$.

Here:

- A: The matrix representing the linear transformation.
- $\mathbf{v}$: The eigenvector associated with λ.
- λ: The eigenvalue.

1.2 Key Points

- **Geometric Interpretation:** Eigenvalues represent *scaling factors* by which eigenvectors are stretched or compressed during the linear transformation defined by A. Eigenvectors remain in the same or opposite direction after the transformation.
- **Algebraic Interpretation:** To find eigenvalues, we solve the *characteristic equation*: $\det(A - \lambda I) = 0$, where I is the identity matrix of the same size as A.

1.3 Examples

1. **Diagonal Matrix:**
   If $A = \begin{bmatrix} 3 & 0 \\ 0 & 5 \end{bmatrix}$, the eigenvalues are the diagonal elements: $\lambda_1 = 3, \quad \lambda_2 = 5$.
2. **Non-Diagonal Matrix:**
   For $A = \begin{bmatrix} 1 & 2 \\ 2 & 1 \end{bmatrix}$, the eigenvalues are obtained by solving: $\det\left(\begin{bmatrix} 1 & 2 \\ 2 & 1 \end{bmatrix} - \lambda \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}\right) = 0$. The eigenvalues are: $\lambda_1 = 3, \quad \lambda_2 = -1$.

1.4 Properties of Eigenvalues

- **Trace and Determinant:**
  - The sum of the eigenvalues equals the **trace** of the matrix: $\mathrm{tr}(A) = \sum \lambda_i$.
  - The product of the eigenvalues equals the **determinant** of the matrix: $\det(A) = \prod \lambda_i$.
- **Multiplicity:**
  - *Algebraic Multiplicity:* The number of times an eigenvalue λ appears as a root of the characteristic polynomial.
  - *Geometric Multiplicity:* The dimension of the eigenspace corresponding to λ (the set of all eigenvectors associated with λ).
- **Symmetric Matrices:**
  - All eigenvalues of symmetric matrices are real.
  - Eigenvectors corresponding to different eigenvalues are orthogonal.
- **Stability Analysis:** Eigenvalues determine the stability of dynamical systems:
  - If all eigenvalues have negative real parts, the system is stable.
  - Positive real parts indicate instability.

1.5 Conclusion

Eigenvalues provide critical insight into the behavior of a matrix and the systems it represents. They are indispensable in solving practical problems across scientific and engineering domains.

2 Introduction

Eigenvalue computation plays a pivotal role in numerous applications across mathematics, physics, engineering, and computer science. Typical uses include solving systems of differential equations, analyzing the stability of systems, and performing matrix decompositions in numerical methods. This report focuses on the QR algorithm, a widely used iterative method for computing the eigenvalues of general matrices. The QR algorithm, which relies on orthogonal transformations, is particularly effective for large matrices and is commonly employed in numerical linear algebra due to its efficiency and robustness.

3 Chosen Algorithm: QR Algorithm

The QR algorithm is an iterative method used for computing the eigenvalues of a square matrix. By repeatedly decomposing the matrix into orthogonal (Q) and upper triangular (R) matrices and recomposing it, the algorithm drives the matrix closer to an upper triangular form, where the diagonal elements approximate the eigenvalues.

3.1 Description of the QR Algorithm

Given a square matrix A, the QR algorithm applies the following iterative process:

1. Perform a **QR decomposition**: Decompose the matrix A_k into Q_k (an orthogonal matrix) and R_k (an upper triangular matrix) such that: $A_k = Q_k R_k$.
2. Recompose the matrix: Compute the next iteration matrix as: $A_{k+1} = R_k Q_k$.
3. **Apply a shift** to accelerate convergence: $A_k - \mu_k I = Q_k R_k$, $A_{k+1} = R_k Q_k + \mu_k I$, where μ_k is a scalar shift (e.g., the bottom-right element of A_k).

3.2 Algorithm Steps

1. **Initialization**: Start with the matrix $A_0 = A$.
2. **Iterative Process**:
   - Compute Q_k and R_k from the QR decomposition of A_k.
   - Update $A_{k+1} = R_k Q_k$ (with or without shifts).
3. **Convergence Check**: Terminate when the off-diagonal elements of A_k are below a predefined tolerance, indicating that A_k is close to diagonal.
4. **Output**: Extract eigenvalues from the diagonal elements of A_k.

4 Time Complexity Analysis

- **Per Iteration**: The QR decomposition has a complexity of $O(n^3)$ for an $n \times n$ matrix.
- **Overall Complexity**: The total number of iterations depends on the matrix properties and the use of shifts. Typically, the QR algorithm converges faster with shifts, especially for well-conditioned matrices.

5 Other Insights

5.1 Memory Usage

The QR algorithm requires $O(n^2)$ space to store the matrix and intermediate results, making it suitable for dense matrices.

5.2 Convergence Rate

- **Without shifts**: The algorithm exhibits linear convergence and may require many iterations.
- **With shifts**: Convergence is significantly faster, often quadratic, for symmetric matrices.

5.3 Suitability

The QR algorithm is highly versatile, handling both real symmetric and non-symmetric matrices effectively. It is preferred for dense matrices and general eigenvalue problems.

6 Comparison of Algorithms

- **QR Algorithm**: Handles both symmetric and non-symmetric matrices; robust for large problems; complexity of $O(n^3)$ per iteration.
- **Jacobi Method**: Slower convergence; tailored for symmetric matrices; best suited for small to medium-sized problems.
- **Power Iteration**: Efficient for computing a single dominant eigenvalue; $O(n^2)$ per iteration but limited in scope.
- **Divide-and-Conquer**: Faster for large symmetric matrices; more complex to implement; suitable for advanced numerical libraries.

7 Conclusion

The QR algorithm is a robust and versatile method for eigenvalue computation, particularly for dense matrices. While computationally expensive for very large matrices, its ability to handle general matrix types and its accelerated convergence with shifts make it a standard choice in numerical linear algebra.

8 Implementation in C

Below is the library used for implementing code

```c
#include <stdio.h>
#include <math.h>
#include <stdlib.h>
#define MAX_ITER 1000
#define TOL 1e-6
```

Below is a detailed explanation of the QR algorithm implementation in C.

8.0.1 QR Decomposition Function

The `qr_decomposition()` function performs the QR decomposition of matrix A into matrices Q and R using the Modified Gram-Schmidt process. It ensures Q is orthogonal, and R is upper triangular.

```c
void qr_decomposition(int n, double A[n][n], double Q[n][n],
   double R[n][n]) {
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            Q[i][j] = 0.0;
            R[i][j] = 0.0;
        }
    }

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

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

        double norm = 0.0;
        for (int i = 0; i < n; i++) {
            norm += Q[i][j] * Q[i][j];
        }
        norm = sqrt(norm);

        if (fabs(norm) < TOL) {
            norm = TOL;
        }
        R[j][j] = norm;

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

- **Orthogonalization:** Each column of A is iteratively orthogonalized with respect to the previous columns. $R[i][j] = Q[:, i]^T \cdot A[:, j]$ $Q[:, j] = A[:, j] - \sum_{i=0}^{j-1} R[i][j] \cdot Q[:, i]$
- **Normalization:** After orthogonalization, each column vector of Q is normalized: $Q[:, j] = \frac{Q[:,j]}{\|Q[:,j]\|}$

8.0.2 Matrix Multiplication Function

The `multiply_matrices()` function computes the product $R \cdot Q$, updating the matrix A in each iteration.

```c
void multiply_matrices(int n, double A[n][n], double B[n][n],
   double C[n][n]) {
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            C[i][j] = 0.0;
            for (int k = 0; k < n; k++) {
                C[i][j] += A[i][k] * B[k][j];
            }
        }
    }
}
```

- This step recomposes the matrix A using: $A' = R \cdot Q$ where R is upper triangular, and Q is orthogonal from the previous decomposition.
- This operation ensures that the matrix evolves toward a diagonal form as the iterations proceed.

8.0.3 Diagonal Check Function

The `is_diagonal()` function verifies if A is sufficiently close to a diagonal matrix by comparing the magnitude of off-diagonal elements to a tolerance value (`TOL`).

```c
int is_diagonal(int n, double matrix[n][n]) {
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            if (i != j && fabs(matrix[i][j]) > TOL) {
                return 0;
            }
        }
    }
    return 1;
}
```

- **Tolerance Comparison:** If all off-diagonal elements satisfy: $|A[i][j]| < \text{TOL}, \quad \forall i \neq j$ the matrix is considered diagonal.
- **Purpose:** Ensures convergence by monitoring the evolution of A during iterations.

8.0.4 Main Iterative Loop

The iterative loop orchestrates the steps of the QR algorithm until A converges to a diagonal matrix or the maximum number of iterations (`MAX_ITER`) is reached.

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

    double A[n][n];
    double eigenvalues[n];

    printf("Enter the elements of the %d x %d matrix row by row:\
       n", n, n);
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            printf("A[%d][%d]: ", i + 1, j + 1);
            scanf("%lf", &A[i][j]);
        }
    }

    printf("\nOriginal matrix A:\n");
    print_matrix(n, A);

    qr_algorithm(n, A, eigenvalues);

    printf("Eigenvalues:\n");
    for (int i = 0; i < n; i++) {
        printf("%8.4f\n", eigenvalues[i]);
    }

    return 0;
}
```

- **Step 1:** Perform QR decomposition of A into Q and R.
- **Step 2:** Update A using the matrix product $R \cdot Q$.
- **Step 3:** Check convergence using `is_diagonal()`.
- **Step 4:** Extract eigenvalues from the diagonal elements of A if convergence is achieved: Eigenvalues: $\lambda_i = A[i][i], \quad i = 1, 2, \ldots, n$

Time Complexity Analysis

The given program implements the QR algorithm to compute the eigenvalues of a square matrix of size $n \times n$. The time complexity can be summarized as follows:

- **QR Decomposition:** $O(n^3)$ This step involves orthogonalization and normalization, both contributing to $O(n^3)$.
- **Matrix Multiplication:** $O(n^3)$ Multiplying two $n \times n$ matrices takes $O(n^3)$.
- **Diagonal Check:** $O(n^2)$ Checking if the matrix is diagonal requires examining n^2 elements.
- **Overall Per Iteration:** $O(n^3)$ The dominant operations in each iteration are QR decomposition and matrix multiplication.
- **Total Complexity:** $O(k \cdot n^3)$ where k is the number of iterations required for convergence. In the worst case, $k = \text{MAX\_ITER}$, making the complexity: $O(\text{MAX\_ITER} \cdot n^3)$ If `MAX_ITER` is treated as a constant, the complexity simplifies to $O(n^3)$ for practical purposes.

Efficiency Analysis of the QR Algorithm

Strengths

- **Accuracy:** The QR algorithm is numerically stable and provides precise eigenvalues, particularly for symmetric or Hermitian matrices.
- **Versatility:** It can handle both real and complex eigenvalues effectively.
- **Reliability:** The algorithm is robust and well-studied for most types of square matrices.

Weaknesses

- **Computational Cost:** The time complexity is $O(k \cdot n^3)$, where k is the number of iterations required for convergence. This can be expensive for large matrices.
- **Scalability:** The algorithm is inefficient for very large matrices, where iterative methods such as Lanczos or Arnoldi are more suitable.

Practical Use

- **Small Matrices:** Efficient and accurate for matrices with $n < 1000$.
- **Large Matrices:** Computationally expensive, making iterative methods a better choice for high-dimensional problems.

8.1 Output of the Program

When the above code is run, the following results are printed:

- The original matrix A.
- The computed eigenvalues.

```
Enter the size of the matrix (n x n): 3
Enter the elements of the 3 x 3 matrix row by row:
A[1][1]: 4
A[1][2]: 1
A[1][3]: 2
A[2][1]: 1
A[2][2]: 5
A[2][3]: 3
A[3][1]: 2
A[3][2]: 3
A[3][3]: 6
```

Listing 1: input

```
Original matrix A:
   4.0000    1.0000    2.0000
   1.0000    5.0000    3.0000
   2.0000    3.0000    6.0000

Eigenvalues:
 7.0000
 5.0000
 3.0000
```

Listing 2: output