Software assignment

ai24btech11035 - V.Preethika

1 Eigenvalue Determination Algorithms

1) **Power Iteration**

The power iteration method is designed to compute the **dominant eigenvalue** (the eigenvalue with the largest magnitude) of a matrix A. Given an initial random vector v_0 , the algorithm iteratively multiplies the matrix A by this vector and normalizes the result at each step.

- **Iteration** Multiply the matrix A by the current vector v_k to get a new vector:

$$v_{k+1} = Av_k$$

- **Normalization** Normalize the resulting vector to ensure that its magnitude does not grow indefinitely:

$$v_k = \frac{v_k}{\|v_k\|}$$

Where:

- v_k is the vector at iteration k ,
- A is the matrix whose eigenvalue and eigenvector are being computed,
- $\|v_k\|$ is the norm of vector v_k, typically the Euclidean norm (i.e.,)

$$\|v_k\| = \sqrt{v_k^T v_k}.$$

This process is repeated until the vector v_k converges to the eigenvector corresponding to the largest eigenvalue.The vector v_k converges to the eigenvector corresponding to the dominant eigenvalue, λ_1 , as k increases. The eigenvalue is approximated by

$$\lambda_1 \approx \frac{v_k^T A v_k}{v_k^T v_k}.$$

Mathematical Insight behind Power Iteration

The Power Iteration method relies on the principle that repeated multiplication by a matrix A amplifies the influence of its dominant eigenvalue and eigenvector. Any vector v_0 can be written as:

$$v_0 = c_1 u_1 + c_2 u_2 + \cdots + c_n u_n,$$

- $u_1, u_2, \ldots, u_n$ are the eigenvectors of A
- $c_1, c_2, \ldots, c_n$ are the coefficients representing the contribution of each eigenvector v_0.

Thus after k iterations,

$$A^k v_0 \approx c_1 \lambda_1^k u_1.$$

As $k \to \infty$, v_k aligns with u_1, enabling the calculation of the dominant eigenvalue and eigen vector.

Pros:

- **Simple to implement:** The process involves basic matrix-vector multiplications and normalization.
- **Low memory requirements:** Only the current vector needs to be stored at each iteration.

Cons:

- **Only finds the dominant eigenvalue:** The method does not work for other eigenvalues, especially when the matrix has eigenvalues with similar magnitudes.
- **Slow convergence:** If the dominant eigenvalue is not well-separated from the others, convergence can be slow.

2) **QR Algorithm**

The **QR Algorithm** is an iterative method used to compute the eigenvalues of a matrix. It is a widely used technique due to its efficiency and versatility in handling both diagonalizable and non-diagonalizable matrices.

Let $A_0 = A$, the matrix whose eigenvalues are to be computed.

QR Decomposition Decompose the matrix A_k at each iteration into an orthogonal matrix Q_k and an upper triangular matrix R_k:

$$A_k = Q_k R_k$$

Update the matrix for the next iteration:

$$A_{k+1} = R_k Q_k$$

After sufficient iterations,A_k converges to an upper triangular matrix. The eigenvalues are then the diagonal elements of this matrix.

Eigen Value Calculation The QR algorithm computes all eigenvalues of a matrix, not just the dominant one, making it suitable for full eigenvalue decomposition.

Efficiency The algorithm typically converges relatively quickly, especially for well-conditioned matrices. The convergence rate is logarithmic in terms of the number of iterations.

Pros:

- **Complete Eigenvalue Calculation:** The algorithm provides all eigenvalues, making it comprehensive compared to methods that compute only one eigenvalue.
- **Numerical Stability:** The use of orthogonal matrices in the QR decomposition ensures stability and reduces errors in numerical computations.
- **Handles General Matrices:** The QR algorithm can handle matrices of any form (not just symmetric matrices), unlike some specialized eigenvalue algorithms.
- **Fast Convergence:** The algorithm converges quickly for many matrices, especially those that are well-conditioned, making it efficient in practice.

Cons:

- **Computationally Expensive:** Each iteration requires the computation of the QR decomposition, which is computationally expensive for large matrices. This can be a bottleneck for very large-scale problems.
- **Not Optimal for Sparse Matrices:** For sparse matrices, the QR algorithm can be less efficient compared to specialized methods, such as Lanczos or Arnoldi iteration.
- **Eigenvectors Not Directly Computed:** While the algorithm gives eigenvalues, computing eigenvectors requires additional work (i.e., accumulating the matrices), which adds to the computational cost.
- **Memory Intensive:** Storing the matrices and during iterations can be memory-intensive, particularly for large matrices.
- **Convergence Slows for Some Matrices:** The algorithm can converge more slowly for matrices with closely spaced eigenvalues or poorly conditioned matrices.

3) **Jacobi Algorithm:**

The **Jacobi Algorithm** is an iterative numerical method used to compute the eigenvalues and eigenvectors of a symmetric matrix. It is particularly effective for small to medium-sized symmetric matrices and is based on the principle of reducing off-diagonal elements to zero through successive rotations.

The matrix A must be symmetric ($A^T = A$) to guarantee convergence.

Find the largest off-diagonal element a_{ij} in the matrix. This element is the target for reduction in the current iteration.

Construct a rotation matrix P that zeroes out the largest off-diagonal element a_{ij} while preserving the symmetry of the matrix.

Update matrix as:

$$A' = P^T A P$$

Iterate: Repeat steps , reducing the largest off-diagonal element at each iteration. With each iteration, the matrix A becomes increasingly diagonal.

Convergence: The algorithm stops when all off-diagonal elements are below a predefined tolerance level. At this point, the matrix is approximately diagonal, and the diagonal elements are the eigenvalues.

Pros:

- **Simplicity:** The algorithm is conceptually simple and easy to implement for symmetric matrices.
- **Numerical Stability:** Since it uses orthogonal transformations, the Jacobi algorithm is numerically stable and robust against rounding errors.
- **Accurate for Small Matrices:** The method is highly accurate for small to medium-sized symmetric matrices.
- **Computes Eigenvectors:** Unlike some other methods, the Jacobi algorithm naturally provides eigenvectors as a byproduct of the accumulated rotation matrices.

Cons:

- **Computationally Expensive:** The algorithm has a time complexity of $O\left(n^3\right)$, making it inefficient for large matrices compared to other algorithms like the QR algorithm or Lanczos method.

- **Slow Convergence:** While the method is accurate, it often requires a large number of iterations to converge, especially for matrices with closely spaced eigenvalues.
- **Only for Symmetric Matrices:** The algorithm cannot handle non-symmetric matrices, which limits its applicability.

4) **Divide and Conquer Algorithm**

The **Divide and Conquer algorithm** is a problem-solving strategy that involves breaking a larger problem into smaller subproblems, solving them independently, and then combining their solutions to solve the original problem. This approach is recursive in nature and is widely used in algorithm design to optimize computational efficiency.

Divide Break the main problem into smaller, simpler subproblems. These subproblems should ideally be of the same type as the original problem.

Conquer Solve each subproblem recursively. If the subproblems become small enough, solve them directly (base case).

Combine Merge the solutions of the subproblems to construct the solution for the original problem.

Example

Strassen's Matrix Multiplication:

- **Divide** Split matrices into smaller submatrices.
- **Conquer** Narrow the search to the left or right half.
- **Combine** No combination required as the result is found during the recursive steps.
- **Time Complexity:**

$$O\left(n^{\log_2(7)}\right) \approx O\left(n^{2.81}\right)$$

Pros:

- **Eficiency:** Often reduces the time complexity of problems compared to brute-force approaches (e.g., $O(nlogn)$ for sorting).
- **Parallelism:** Subproblems are independent and can be solved concurrently, making the method suitable for parallel computing.
- **Simplicity:** Provides a structured way to approach complex problems by breaking them into manageable subproblems.

Cons:

- **Overhead:** Recursive function calls and additional memory usage for storing intermediate results can introduce overhead.
- **Not Always Optimal:** For certain problems, iterative methods may be more efficient (e.g., Fibonacci sequence).
- **Combination Complexity** In some cases, combining solutions from subproblems can be challenging and computationally expensive.

2 Chosen Algorithm: QR Algorithm

After reviewing the options, I will implement the QR Algorithm, as it works for both symmetric and non-symmetric matrices and is generally reliable for finding all eigenvalues of a matrix.

Justification

- **Versatile:** Works for any square matrix, not just symmetric ones, and computes all eigenvalues.
- **Efficient Convergence:** Converges quickly for most matrices, especially with shifted versions.
- **Numerically Stable:** More accurate and stable than methods like characteristic polynomials or power iteration.
- **Widely Used:** Supported by libraries like LAPACK, making it easy to implement and use in practice.

Time Complexity

- **QR Decomposition:** Takes $O\left(n^3\right)$ operations for a matrix of size $n \times n$.
- **Iterations:** Typically, the number of iterations is proportional to n, making the total complexity $O\left(n^3\right)$.
- **With Shifts:** Convergence is faster, but the overall complexity remains $O\left(n^3\right)$.

Gram-Schmidt Process for QR Decomposition

The Gram-Schmidt process is used to orthogonalize a set of vectors and is a key method for performing QR decomposition of a matrix.

Input Let A be an $m \times n$ matrix with linearly independent columns $v_1, v_2, \ldots, v_n$. The goal is to find an orthogonal matrix Q and an upper triangular matrix R such that:

$$A = QR,$$

where:

- Q is an $m \times n$ orthogonal matrix ($Q^\top Q = I$),
- R is an $n \times n$ upper triangular matrix.

Process

1. Initialize Start with the columns of matrix A: $v_1, v_2, \ldots, v_n$.

- Q will store the orthonormalized vectors.
- R will store the coefficients of the projections.

2. Orthogonalize each column v_k (for $k = 1, 2, \ldots, n$)

For each column v_k, subtract the projection onto all previously computed orthonormal vectors $q_1, q_2, \ldots, q_{k-1}$:

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

where the projection of v_k onto q_i is given by:

$$\text{proj}_{q_i}(v_k) = (q_i^\top v_k) q_i.$$

3. Normalize the resulting vector u_k to obtain q_k:

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

where $\|u_k\|$ is the Euclidean norm (length) of u_k.

4. Construct the upper triangular matrix R: The elements of the upper triangular

matrix R are the coefficients of the projections:

$$r_{ij} = q_i^\top v_j \quad \text{for } 1 \le i \le j \le n.$$

5. Repeat the process for all columns Continue the process for each column of A until all n columns are orthonormalized.

Final Output

- The orthogonal matrix Q, whose columns are the orthonormal vectors $q_1, q_2, \ldots, q_n$.
- The upper triangular matrix R, whose elements are the coefficients of the projections of $v_1, v_2, \ldots, v_n$ onto the previously computed $q_1, q_2, \ldots, q_n$.

Thus, the QR decomposition of A is:

$$A = QR,$$

where:

- Q is an $m \times n$ matrix with orthonormal columns.
- R is an $n \times n$ upper triangular matrix.

5) **Chosen language:C C code to find eigenvalues**

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

// Function to perform matrix multiplication: C = A * B
void matrixMultiply(double **A, double **B, double **result, int n) {
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            result[i][j] = 0;
            for (int k = 0; k < n; k++) {
                result[i][j] += A[i][k] * B[k][j];
            }
        }
    }
}

// Function to perform QR decomposition (simplified for any square matrix)
void qrDecomposition(double **A, double **Q, double **R, int n) {
    // Initialize Q and R matrices
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            Q[i][j] = A[i][j];
            R[i][j] = 0;
        }
    }
```

```
    // QR Decomposition using Gram-Schmidt process
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < i; j++) {
            double dotProduct = 0;
            for (int k = 0; k < n; k++) {
                dotProduct += Q[k][i] * Q[k][j];
            }
            R[j][i] = dotProduct;
            for (int k = 0; k < n; k++) {
                Q[k][i] -= R[j][i] * Q[k][j];
            }
        }
        double norm = 0;
        for (int k = 0; k < n; k++) {
            norm += Q[k][i] * Q[k][i];
        }
        norm = sqrt(norm);
        R[i][i] = norm;
        for (int k = 0; k < n; k++) {
            Q[k][i] /= norm;
        }
    }
}

// Function to compute eigenvalues using the QR algorithm
void qrAlgorithm(double **A, double *eigenvalues, int n, int maxIterations) {
    double **Q = (double **)malloc(n * sizeof(double *));
    double **R = (double **)malloc(n * sizeof(double *));
    double **A_new = (double **)malloc(n * sizeof(double *));
    for (int i = 0; i < n; i++) {
        Q[i] = (double *)malloc(n * sizeof(double));
        R[i] = (double *)malloc(n * sizeof(double));
        A_new[i] = (double *)malloc(n * sizeof(double));
    }

    // Iterate QR decomposition process to approximate eigenvalues
    for (int iter = 0; iter < maxIterations; iter++) {
        qrDecomposition(A, Q, R, n);

        // A_new = R * Q
        matrixMultiply(R, Q, A_new, n);

        // Copy A_new to A
        for (int i = 0; i < n; i++) {
```

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

    // Extract eigenvalues from the diagonal of A (as it becomes upper triangular)
    for (int i = 0; i < n; i++) {
        eigenvalues[i] = A[i][i];
    }

    // Free dynamically allocated memory
    for (int i = 0; i < n; i++) {
        free(Q[i]);
        free(R[i]);
        free(A_new[i]);
    }
    free(Q);
    free(R);
    free(A_new);
}

int main() {
    int n;

    // Get matrix size
    printf("Enter the size of the matrix (n x n): ");
    scanf("%d", &n);

    // Dynamically allocate memory for the matrix
    double **A = (double **)malloc(n * sizeof(double *));
    for (int i = 0; i < n; i++) {
        A[i] = (double *)malloc(n * sizeof(double));
    }

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

    // Array to store the eigenvalues
    double *eigenvalues = (double *)malloc(n * sizeof(double));
```

```
    // Run QR algorithm
    int maxIterations = 100;
    qrAlgorithm(A, eigenvalues, n, maxIterations);

    // Output the eigenvalues
    printf("Eigenvalues of the matrix are:\n");
    for (int i = 0; i < n; i++) {
        printf("%lf\n", eigenvalues[i]);
    }

    // Free dynamically allocated memory
    for (int i = 0; i < n; i++) {
        free(A[i]);
    }
    free(A);
    free(eigenvalues);

    return 0;
}
```

Example
Input:
Enter the size of the matrix ($n \times n$): 2
Enter the elements of the matrix row by row:
4 5
2 1
Output:
Eigenvalues of the matrix are:
6.000000
-1.000000