Software Assignment

AI24BTECH11018 - Sreya

1 Different algorithms to find eigen values

1) **Power Iteration:**
Power Iteration is a simple and efficient algorithm for finding the dominant eigenvalue (the eigenvalue with the largest absolute value) and its corresponding eigenvector of a matrix A.
How It Works:
Initialize: Start with a random vector b_0 (typically a random non-zero vector).
Iteration: For each iteration k, the algorithm computes the next vector b_{k+1} by multiplying the matrix A with the current vector b_k:

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

Normalization: To prevent the vector from growing too large, it is normalized after each iteration:

$$b_{k+1} = \frac{b_{k+1}}{\|b_{k+1}\|}$$

Eigenvalue Approximation: After several iterations, the vector b_k will converge to the eigenvector corresponding to the dominant eigenvalue λ_1, and the eigenvalue can be approximated by the Rayleigh quotient:

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

Convergence: The process continues until the vector b_k stops changing significantly, or until the desired tolerance level is reached.

Time Complexity:

Each iteration requires matrix-vector multiplication, which takes $O(n^2)$ operations for an $n \times n$ matrix. Therefore, the time complexity per iteration is $O(n^2)$.

Pros and Cons:

Pros:

- Simple to implement.
- Very efficient for finding the dominant eigenvalue of large, sparse matrices.
- Does not require the full matrix decomposition.

Cons:

- Only finds the dominant eigenvalue (not all eigenvalues).

- Requires that the matrix has a dominant eigenvalue (the largest in absolute magnitude).
- The convergence rate can be slow if the gap between the dominant and other eigenvalues is small.

Suitability:

Power iteration is particularly suitable for large, sparse matrices where only the largest eigenvalue is needed, such as in certain applications of principal component analysis (PCA). The

2) QR Algorithm

The **QR Algorithm** is a classical iterative method used to compute the eigenvalues of a matrix. The algorithm is based on two fundamental concepts in linear algebra: **QR decomposition** and matrix iteration. It works by decomposing the matrix repeatedly using the **QR decomposition** (which factors a matrix into an orthogonal matrix Q and an upper triangular matrix R), and then iteratively transforming the matrix such that its eigenvalues gradually emerge on the diagonal.
The QR algorithm can be described as follows:

Steps of the QR Algorithm

a) **Initialization:** Start with the original matrix A. Set the initial iteration matrix to be $A_0 = A$.
b) **QR Decomposition:** At each iteration k, perform the QR decomposition of A_k:

$$A_k = Q_k R_k$$

where Q_k is an orthogonal matrix and R_k is an upper triangular matrix. This decomposition is performed using methods like Gram-Schmidt or Householder reflections.
c) **Form the Next Matrix:** After the QR decomposition, update the matrix A_{k+1} by multiplying R_k and Q_k in the reverse order:

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

The reason for this is that, at each step, the matrix A_k is transformed towards a triangular form, and this transformation progressively reveals the eigenvalues on the diagonal.
d) **Repeat:** Continue iterating until the matrix A_k converges to an upper triangular form. The diagonal elements of the resulting matrix A_k will be the eigenvalues of the original matrix A.

Convergence and Behavior

- **Convergence to Triangular Form:** With each iteration, the matrix A_k approaches an upper triangular matrix, and the diagonal elements converge to the eigenvalues

of A. The algorithm is guaranteed to converge for most matrices, but the rate of convergence can vary depending on the properties of the matrix.

- **Diagonalization:** As the process continues, the off-diagonal elements become smaller, and the matrix becomes closer to an upper triangular matrix. In the limit, the matrix becomes a diagonal matrix whose diagonal elements are the eigenvalues of the original matrix.
- **Convergence Speed:** The speed of convergence depends on the spectral gap between the largest and smallest eigenvalues of the matrix. If the gap is large, convergence is fast. However, if the matrix has eigenvalues that are very close together, convergence may be slower, requiring more iterations.

Numerical Considerations

- **Stability:** The QR algorithm is numerically stable when using orthogonal decompositions like Householder reflections, but it may encounter issues with matrices that are nearly defective or poorly conditioned. Small numerical errors can accumulate, especially if the matrix has a large condition number or nearly repeated eigenvalues.
- **Eigenvalue Extraction:** After sufficient iterations, the diagonal elements of the matrix A_k provide increasingly accurate approximations to the eigenvalues of the original matrix A. In practice, convergence is typically achieved when the off-diagonal elements of A_k are sufficiently close to zero.
- **Shifting for Faster Convergence:** To accelerate convergence, a "shifted" QR algorithm is often used. In this version, the matrix is shifted by subtracting a scalar multiple of the identity matrix before performing the QR decomposition. The shift helps improve convergence, particularly for matrices with clustered eigenvalues.

Time Complexity

- **Per Iteration Complexity:** Each iteration of the QR algorithm involves performing a QR decomposition, which is an $O(n^3)$ operation for an $n \times n$ matrix. The complexity arises from the matrix-matrix multiplication involved in the decomposition process.
- **Number of Iterations:** In general, the QR algorithm requires $O(n)$ iterations to converge, where n is the size of the matrix. Thus, the total time complexity of the algorithm is $O(n^3)$, since each iteration is $O(n^3)$, and the algorithm usually requires $O(n)$ iterations.
- **Efficiency:** The QR algorithm is efficient for dense matrices, but for very large matrices, specialized algorithms such as Lanczos, Arnoldi, or Power Iteration may be more appropriate, especially if only a few eigenvalues are needed.

Advantages of the QR Algorithm

- **General Purpose:** The QR algorithm can compute all eigenvalues of a matrix (not just the dominant ones), making it a versatile tool for general eigenvalue problems.

- **Numerical Stability:** When implemented with numerically stable methods such as Householder reflections, the QR algorithm is highly reliable for computing eigenvalues, even for ill-conditioned matrices.
- **Convergence:** The QR algorithm is guaranteed to converge for most matrices. It is suitable for a wide range of matrix types, including dense, symmetric, and nonsymmetric matrices.

Disadvantages of the QR Algorithm

- **Computational Cost:** The $O(n^3)$ time complexity per iteration can be prohibitive for very large matrices, particularly when many eigenvalues are needed.
- **Slow Convergence:** For matrices with closely spaced eigenvalues, convergence can be slow, requiring many iterations to reach the desired accuracy.
- **Storage Requirements:** The QR algorithm requires storing both the orthogonal matrix Q and the upper triangular matrix R, which can be memory-intensive for large matrices.

Jacobi Method: Overview

The **Jacobi method** is an iterative algorithm used for solving a system of linear equations:

$$\mathbf{Ax} = \mathbf{b}$$

where $\mathbf{A}$ is a square matrix, $\mathbf{x}$ is the vector of unknowns, and $\mathbf{b}$ is the constant vector. This method is especially useful for solving large, sparse linear systems, and is typically used when the system is diagonally dominant or symmetric.
Given a system of linear equations:

$$\mathbf{A} = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{bmatrix}$$

The Jacobi method is an iterative process that decomposes the matrix $\mathbf{A}$ into its diagonal, lower triangular, and upper triangular parts:

$$\mathbf{A} = \mathbf{D} + \mathbf{L} + \mathbf{U}$$

where:

- $\mathbf{D}$ is the diagonal matrix, containing the elements a_{ii} of $\mathbf{A}$,
- $\mathbf{L}$ is the strictly lower triangular matrix,
- $\mathbf{U}$ is the strictly upper triangular matrix.

Steps of the Jacobi Method

1. Initialization

Start with an initial guess for the solution $\mathbf{x}^{(0)} = [x_1^{(0)}, x_2^{(0)}, \ldots, x_n^{(0)}]^T$.

2. Iteration

Update each component of $\mathbf{x}^{(k)}$ using the formula:

$$x_i^{(k+1)} = \frac{1}{a_{ii}} \left(b_i - \sum_{j \neq i} a_{ij} x_j^{(k)} \right)$$

where $x_i^{(k+1)}$ is the new value of the i-th unknown, and the summation runs over all the other variables except i.

3. Repeat

Repeat the process until the solution converges to a sufficiently accurate value. The convergence is typically determined by checking if the difference between successive iterations is smaller than a predefined tolerance, i.e., if:

$$\|\mathbf{x}^{(k+1)} - \mathbf{x}^{(k)}\| < \epsilon$$

where ϵ is a small tolerance value.

Convergence Criteria

The Jacobi method converges if the matrix $\mathbf{A}$ is **diagonally dominant**, meaning that for each row of the matrix, the magnitude of the diagonal entry is larger than the sum of the magnitudes of the other entries in that row:

$$|a_{ii}| > \sum_{j \neq i} |a_{ij}|$$

Alternatively, the method can also converge for symmetric positive definite matrices, but it may not converge for other types of matrices.

Advantages and Disadvantages

Advantages

- **Simplicity:** The method is conceptually simple and easy to implement.
- **Parallelism:** The Jacobi method is easily parallelizable because each equation is updated independently of others.
- **Memory Efficiency:** For large, sparse systems, it requires relatively low memory overhead.

Disadvantages

- **Slow Convergence:** The Jacobi method can converge slowly, especially for large systems or systems that are not diagonally dominant.
- **Limited Applicability:** It is not always the fastest method, and its convergence can be problematic for non-diagonally dominant matrices.

2 I choose QR Algorithm

The QR algorithm is often preferred over the Jacobi method and power iteration because it can compute all eigenvalues of a matrix efficiently and is applicable to both symmetric and non-symmetric matrices. Unlike the Jacobi method, which is specifically designed for symmetric matrices, and power iteration, which only computes the dominant eigenvalue, the QR algorithm iteratively transforms the matrix into a triangular form, extracting all eigenvalues from the diagonal. The QR algorithm generally converges faster and is more numerically stable for dense matrices, making it a versatile and robust choice. While the Jacobi method is suitable for symmetric matrices with high precision, and power iteration is ideal for large sparse matrices when only the largest eigenvalue is needed, the QR algorithm is a more efficient option for general eigenvalue problems. The **QR Algorithm** is a method used to compute the eigenvalues of a matrix, and it involves the QR decomposition of the matrix at each iteration. The **Gram-Schmidt Process** is a method for computing the QR decomposition of a matrix, making it an important part of the QR algorithm. Below, we will clearly explain both methods and how they work together.

3 1. Gram-Schmidt Process: Orthogonalization of Vectors

The Gram-Schmidt Process is a procedure for orthogonalizing a set of vectors, meaning transforming them into a set of orthogonal (or orthonormal) vectors. Given a matrix A, the goal is to decompose it into an orthogonal matrix Q and an upper triangular matrix R such that:

$$A = QR$$

3.1 Step-by-Step Process:

Given a matrix $A = [a_1, a_2, \ldots, a_n]$, where a_i are the columns of A, the Gram-Schmidt process proceeds as:

a) **Start with the first column:** The first orthonormal vector q_1 is simply the normalized version of the first column of A:

$$q_1 = \frac{a_1}{||a_1||}$$

b) **For each subsequent column** a_k**,** subtract the projections of the column vector onto the previous orthonormal vectors $q_1, q_2, \ldots, q_{k-1}$:

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

where the projection of a_k onto q_i is given by:

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

c) **Normalize the result** to get the orthonormal vector q_k:

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

d) **Form the matrix Q:** Once all the vectors $q_1, q_2, \ldots, q_n$ are computed, the matrix Q is formed by placing these vectors as columns.

e) **Compute matrix R:** The matrix R is obtained by:

$$R = Q^T A$$

R will be an upper triangular matrix.

3.2 Result:

After applying the Gram-Schmidt process, we have the QR decomposition of matrix A:

$$A = QR$$

where Q is orthogonal and R is upper triangular.

3) **C code on computing eigenvalues using QR algorithm:**

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

#define MAX_SIZE 10
#define EPSILON 1e-6

// Function to compute the dot product of two vectors
double dot_product(double a[], double b[], int n) {
    double result = 0.0;
    for (int i = 0; i < n; i++) {
        result += a[i] * b[i];
    }
    return result;
}

// Function to compute the norm (magnitude) of a vector
double norm(double a[], int n) {
    return sqrt(dot_product(a, a, n));
}

// Function to subtract two vectors
void subtract(double a[], double b[], double result[], int n) {
    for (int i = 0; i < n; i++) {
        result[i] = a[i] - b[i];
    }
}

// Function to multiply a matrix and a vector
void mat_vec_mult(double mat[MAX_SIZE][MAX_SIZE], double vec[
    MAX_SIZE], double result[MAX_SIZE], int n) {
```

```
    for (int i = 0; i < n; i++) {
        result[i] = 0;
        for (int j = 0; j < n; j++) {
            result[i] += mat[i][j] * vec[j];
        }
    }
}

// Function to perform QR decomposition of matrix A and return Q and R
    matrices
void qr_decomposition(double A[MAX_SIZE][MAX_SIZE], double Q[
    MAX_SIZE][MAX_SIZE], double R[MAX_SIZE][MAX_SIZE], int n) {
    double A_copy[MAX_SIZE][MAX_SIZE];

    // Copy matrix A into A_copy for manipulation
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            A_copy[i][j] = A[i][j];
        }
    }

    // Perform Gram-Schmidt to get Q and R
    for (int i = 0; i < n; i++) {
        // Column of A_copy[i] (A_i)
        double a_i[MAX_SIZE];
        for (int j = 0; j < n; j++) {
            a_i[j] = A_copy[j][i];
        }

        // For each j < i, subtract the projection of a_i on Q_j
        for (int j = 0; j < i; j++) {
            double proj = dot_product(a_i, Q[j], n);
            for (int k = 0; k < n; k++) {
                a_i[k] -= proj * Q[j][k];
            }
        }

        // Normalize a_i to create Q_i
        double norm_a_i = norm(a_i, n);
        for (int j = 0; j < n; j++) {
            Q[j][i] = a_i[j] / norm_a_i;
        }

        // Fill R with the dot products of columns of A_copy with columns of Q
        for (int j = 0; j < n; j++) {
```

```
            R[j][i] = dot_product(A_copy[j], Q[i], n);
        }
    }
}

// Function to multiply matrices A and B
void mat_mult(double A[MAX_SIZE][MAX_SIZE], double B[MAX_SIZE][
    MAX_SIZE], double result[MAX_SIZE][MAX_SIZE], 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 check if the matrix is diagonal (to check convergence)
int is_diagonal(double A[MAX_SIZE][MAX_SIZE], int n) {
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            if (i != j && fabs(A[i][j]) > EPSILON) {
                return 0; // Matrix is not diagonal
            }
        }
    }
    return 1; // Matrix is diagonal
}

// Function to compute eigenvalues using QR algorithm
void qr_algorithm(double A[MAX_SIZE][MAX_SIZE], double eigenvalues[
    MAX_SIZE], int n) {
    double Q[MAX_SIZE][MAX_SIZE], R[MAX_SIZE][MAX_SIZE],
        A_copy[MAX_SIZE][MAX_SIZE], A_next[MAX_SIZE][MAX_SIZE
        ];

    // Initialize A_copy to be the same as A
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            A_copy[i][j] = A[i][j];
        }
    }

    int iteration = 0;
```

```
    while (!is_diagonal(A_copy, n) && iteration < 1000) {
        // Perform QR decomposition
        qr_decomposition(A_copy, Q, R, n);

        // Compute A_next = R * Q
        mat_mult(R, Q, A_next, n);

        // Copy A_next to A_copy
        for (int i = 0; i < n; i++) {
            for (int j = 0; j < n; j++) {
                A_copy[i][j] = A_next[i][j];
            }
        }

        iteration++;
    }

    // Extract the eigenvalues from the diagonal of the matrix
    for (int i = 0; i < n; i++) {
        eigenvalues[i] = A_copy[i][i];
    }
}

// Function to print a matrix
void print_matrix(double mat[MAX_SIZE][MAX_SIZE], int n) {
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            printf("%f ", mat[i][j]);
        }
        printf("\n");
    }
}

int main() {
    int n;

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

    double A[MAX_SIZE][MAX_SIZE], eigenvalues[MAX_SIZE];

    // Input the matrix A
    printf("Enter the elements of the matrix A (%d x %d):\n", n, n);
    for (int i = 0; i < n; i++) {
```

```
            for (int j = 0; j < n; j++) {
                scanf("%lf", &A[i][j]);
            }
        }

        // Compute the eigenvalues using the QR algorithm
        qr_algorithm(A, eigenvalues, n);

        // Output the eigenvalues
        printf("\nThe eigenvalues are:\n");
        for (int i = 0; i < n; i++) {
            printf("Eigenvalue %d: %f\n", i + 1, eigenvalues[i]);
        }

        return 0;
}
```