COLLEGE OF ENGINEERING GUINDY
ANNA UNIVERSITY

CHENNAI 600025

LINEAR ALGEBRA AND NUMERICAL METHODS
MA23C03

SEMESTER III

AUGUST - DECEMBER (2024)

RECORD SUBMITTED BY

NAME:SHIBANI SELVAKUMAR

REG NO. : 2023103005

In partial fulfilment for the award of the degree of

BACHELOR OF ENGINEERING

IN

COMPUTER SCIENCE AND ENGINEERING

INDEX

OUTPUT:

Case 1: Linearly Dependent Vectors
For A = [1 2 3; 4 5 6; 7 8 9]:

Case 2: Linearly Independent Vectors
For A = [1 0 0; 0 1 0; 0 0 1]:

EXPT NO: 1	LINEAR DEPENDENCE AND INDEPENDENCE OF VECTORS

AIM:

To determine the linear dependence or independence of a set of vectors using MATLAB.

THEORY:

A set of vectors is linearly dependent if at least one vector can be expressed as a linear combination of others. Otherwise, the vectors are linearly independent. In MATLAB, this can be tested using the rank of the matrix formed by the vectors. If the rank equals the number of vectors, they are linearly independent; otherwise, they are dependent.

CODE:

```
% Define the matrix with column vectors
A = [1 2 3; 4 5 6; 7 8 9];

% Calculate the rank of the matrix
r = rank(A);

% Display the result
if r == size(A, 2)
    disp('The vectors are linearly independent.');
else
    disp('The vectors are linearly dependent.');
end
```

RESULT:

Thus, the linear dependence and independence of the given vectors were verified. If the rank of the matrix formed by the vectors equals the number of vectors, the vectors are linearly independent. Otherwise, they are linearly dependent. This method was used to validate the relationship between t

OUTPUT:

```
Matrix A:
 4   2
 1   3

Eigenvalues:
[5. 2.]

Eigenvectors:
[[ 0.89442719 -0.70710678]
 [ 0.4472136 0.70710678]]
```

EXPT NO: 2	COMPUTATION OF EIGENVALUES AND EIGENVECTORS

AIM:

To compute the eigenvalues and eigenvectors of a given square matrix using computational tools.

THEORY:

Eigenvalues and eigenvectors are fundamental concepts in linear algebra. For a square matrix A, an eigenvalue λ and corresponding eigenvector v satisfy A·v=λ·v. These quantities have applications in stability analysis, quantum mechanics, and principal component analysis (PCA).

CODE:

```
import numpy as np

# Define the matrix
A = np.array([[4, 2],
              [1, 3]])

# Compute eigenvalues and eigenvectors
eigenvalues, eigenvectors = np.linalg.eig(A)

# Display the results
print("Matrix A:")
print(A)
print("\nEigenvalues:")
print(eigenvalues)
print("\nEigenvectors:")
print(eigenvectors)
```

RESULT:

For the matrix

A = 4 3

1 2 the eigenvalues and eigenvectors are calculated.

OUTPUT:

```
Original Matrix A:
 4   1
 2   3
Eigenvalues (Diagonal Matrix
D):Diagonal Matrix
 5  0
 0  2
Eigenvectors (Matrix
 P):0.7071 -0.4472
 0.7071     0.8944
Reconstructed matrix A as Matrix
 B:4.0000  1.0000
 2.0000     3.0000
```

EXPT NO: 3	DIAGONALIZATION OF LINEAR TRANSFORMATION

AIM:

To diagonalize a square matrix representing a linear transformation by finding its eigenvalues and eigenvectors and verify if the matrix is diagonalizable.

THEORY:

Diagonalization is the process of converting a square matrix A into a diagonal matrix D such that A = PDP^-1, where P is a matrix of eigenvectors, and D is a diagonal matrix with eigenvalues of A as its entries. A matrix is diagonalizable if it has n linearly independent eigenvectors (for an n x n matrix).

CODE:

```
A = [4, 1; 2, 3];

[P, D] = eig(A);

B = P * D * inv(P);

disp('Original Matrix

A:');disp(A);

disp('Eigenvalues (Diagonal Matrix D):');

disp(D);

disp('Eigenvectors (Matrix

P):');disp(P);

disp('Reconstructed matrix A as Matrix B:');

disp(B);
```

RESULT:

The matrix A was successfully diagonalized into D with the corresponding eigenvectors in P. The reconstruction PDP^−1 verifies the diagonalization.

OUTPUT:

```
2     1     0
1     0    -2
0     2     1
-1   -1     0
```

```
  the orthonormal basis is

  0.8165    0.0000    0.4364

  0.4082   -0.2357   -0.8729

     0         0.9428    -0.2182

  -0.4082    -0.2357       0
```

EXPT NO: 4	GRAM-SCHMIDT ORTHOGONALIZATION PROCESS

AIM:

To construct an orthonormal basis from a given set of linearly independent vectors

THEORY:

It is a method of constructing an orthonormal basis from a set of vectors in an inner product space, most commonly the Euclidean space equipped with the standard inner product The Gram–Schmidt process takes a finite, linearly independent set of vectors and generates an orthogonal set.

CODE:

```
A = [2,1,0;1,0,-2;0,2,1;-1,-1,0];
[m, n] = size(A);
Q = zeros(m, n);
for j = 1:n
   v = A(:, j);
   for i = 1:j-1
      v = v - (Q(:, i)' * v) * Q(:, i);
   end
   Q (: , j) = v / norm(v);
end
disp(A);
disp("the orthonormal basis is ");
disp(Q);
```

RESULT:

Thus the columns of A vectors are successfully converted into orthonormal basis by Gram Schmidt process.

OUTPUT:

```
Newton-Raphson  Method:
Iter   x_n          f(x_n)
1      1.521739     -0.125000
2      1.521380     0.002137
3      1.521380     0.000001
```

EXPT NO: 5	SOLUTION OF ALGEBRAIC AND TRANSCENDENTAL EQUATIONS

AIM:

To find the solutions of algebraic and transcendental equations using numerical methods such as the Newton-Raphson Method.

THEORY:

Algebraic equations are equations involving polynomial expressions, while transcendental equations involve non-polynomial expressions like exponential, logarithmic, or trigonometric functions. Finding exact solutions to these equations is not always possible, so numerical methods are employed to approximate solutions within a given tolerance.

Newton-Raphson Method:

- An iterative method that uses the derivative of the function for fast convergence.
- The formula is: $x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$
- Converges quadratically when the initial guess is close to the root.

CODE:

```
% Define the function and its derivative
f = @(x) x^3 - x - 2;        % The given equation
f_prime = @(x) 3*x^2 - 1;     % Derivative of the equation

% Initial guess and parameters
x0 = 1.5;  % Initial guess
tol = 1e-6; % Tolerance for stopping criteria
max_iter = 100; % Maximum number of iterations

% Newton-Raphson Iteration
disp('Newton-Raphson Method:');
disp('Iter  x_n        f(x_n)');
```

```
  for i = 1:max_iter
      fx = f(x0);
      fpx = f_prime(x0);
      x1 = x0 - fx / fpx; % Newton-Raphson formula
      fprintf('%d    %.6f    %.6f\n', i, x1, fx);
      if abs(x1 - x0) < tol
          break;
      end
      x0 = x1;
  end

  disp('Root:');
disp(x1);
```

RESULT:

Using the Newton-Raphson Method, the root of the equation $\boldsymbol{x^3 - x - 2 = 0}$ was approximated as $\boldsymbol{x \approx 1.5214}$. The method converged within the given tolerance of answer.

OUTPUT:

The LU decomposition output will be:

```
Matrix A:

    4   3

    6   3

Lower triangular matrix L:

      0.6667  1.0000

      1.0000      0

Upper triangular matrix U:

        6   3

       0      1

Solution vector Y:

      12

       2
Solution vector X:

       1

       2
```

EXPT NO:06	MATRIX DECOMPOSITION: LU/CHOLESKY DECOMPOSITION

AIM:

To decompose a given matrix into simpler constituent matrices (such as triangular, diagonal,or orthogonal matrices) to simplify matrix operations, such as solving linear systems, computing determinants, performing matrix inversion, and eigenvalue problems, making computations more efficient

THEORY:

LU DECOMPOSITION:

LU decomposition is a matrix factorization technique where a given matrix A is decomposed into two triangular matrices: a lower triangular matrix L and an upper triangular matrix U, such that $A=LU$. LU decomposition simplifies complex matrix operations by breaking them into more manageable triangular systems.

CODE:

```
A = [4, 3; 6, 3];

 B = [10; 12];

[L, U] = lu(A);

Y =LB;
 X = U \ Y;
 disp('Matrix A:');disp(A);

 disp('Lower triangular matrix L:');

 disp(L);

 disp('Upper triangular matrix U:');

 disp(U);

 disp('Solution vector Y:');

 disp(Y);

 disp('Solution vector X:');

 disp(X);
```

OUTPUT:

```
 Cholesky
    Decomposition (L):2
           0    0

    6    1    0

   -8    5    3
 holesky Decomposition
    (L^T):2      6        -8

    0    1    5

      0 0    3
  Solution
     (y):
  0.5000

  -1.0000

   4.0000
  Solution
     (x):
 28.5833

  -7.6667

   1.3333
```

THEORY:

CHOLESKY DECOMPOSITION:

Cholesky decomposition is a numerical method for factorizing a symmetric positive definite matrix into the product of a lower triangular matrix and its transpose . The decomposition is particularly useful in solving systems of linear equations, numerical optimization, and simulations.

CODE:

```
A = [4, 12, -16;
     12, 37, -43;
        -16, -43, 98];
     b = [1; 2; 3];
     L = chol(A, 'lower'); y =
     L \ b;x = L' \ y;
     disp('Cholesky Decomposition
     (L):');disp(L);
     disp('Cholesky Decomposition
     (L^T):');disp(L’);
     disp('Solution
     (y):');disp(y);
     disp('Solution
     (x):');disp(x);
```

RESULT:

The matrix A was successfully decomposed using the LU and Cholesky method and system of equations also solved.

OUTPUT:

```
Gauss-Jacobi Solution:
    0.6667
    1.0000
    0.3333



Gauss-Seidel Solution:
    0.6667
    1.0000
    0.3333
```

EXPTNO: 7	**ITERATIVE METHODS OF GAUSS JACOBI AND GAUSS SEIDEL**

AIM:

To solve a system of linear equations using two iterative methods: Gauss-Jacobi and Gauss-Seidel. These methods are widely used when direct methods (like Gaussian elimination) are not feasible due to the large size of the system.

THEORY:

❒ **Gauss-Jacobi Method:**

- The Gauss-Jacobi method is an iterative technique for solving a system of linear equations of the form $Ax = B$
- In this method, the equations are solved independently in a successive manner, updating all variables at the same time in each iteration.
- The general iterative formula for the i-th variable is:

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

where $x_i^{(k)}$ is the value of x_i at the k-th iteration, and $x_i^{(k+1)}$ is the updated value.

❒ **Gauss-Seidel Method:**

- The Gauss-Seidel method is similar to Gauss-Jacobi, but instead of updating all variables simultaneously, it updates the values of variables as soon as they are computed.
- The iterative formula is:

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

Here, $x_j^{(k+1)}$ is updated immediately after computing each value, leading to a more efficient method.

CODE:

```
% System of Linear Equations
A = [5 1 -1; 1 4 1; 2 3 5];
b = [4; 5; 6];
x = zeros(3,1);  % Initial guess
tol = 1e-6;  % Convergence tolerance
maxIter = 100;  % Maximum iterations
```

```
% Gauss-Jacobi Method
  x_jacobi = x;
  for k = 1:maxIter
      x_new = x_jacobi;
      for i = 1:3
          sum = b(i) - A(i,[1:i-1,i+1:end])*x_jacobi([1:i-1,i+1:end]);
          x_new(i) = sum / A(i,i);
      end
      if norm(x_new - x_jacobi, inf) < tol, break; end
      x_jacobi = x_new;
  end
  % Gauss-Seidel Method
  x_seidel = x;
  for k = 1:maxIter
      for i = 1:3
          sum = b(i) - A(i,[1:i-1,i+1:end])*x_seidel([1:i-1,i+1:end]);
          x_seidel(i) = sum / A(i,i);
      end
      if norm(x_seidel - x_jacobi, inf) < tol, break; end
  end
  % Display Results
  disp('Gauss-Jacobi Solution:');
  disp(x_jacobi);
  disp('Gauss-Seidel Solution:');
disp(x_seidel);
```

RESULT:

- Both Gauss-Jacobi and Gauss-Seidel methods converged to the same solution for the system of equations, but the Gauss-Seidel method converged in fewer iterations.
- Gauss-Seidel is generally faster due to its nature of updating values immediately after they are computed, making it more efficient than Gauss-Jacobi.

OUTPUT:

```
Original Matrix:

[[2. 1.]

 [7. 4.]]


Inverse Matrix:

[[ 4. -1.]

 [-7.  2.]]
```

EXPT NO: 8	MATRIX INVERSION BY GAUSS JORDAN METHOD

AIM:

To experimentally implement matrix inversion using the Gauss-Jordan method.

THEORY:

The Gauss-Jordan method is an algorithm used to compute the inverse of a square matrix. It involves forming an augmented matrix by appending the identity matrix to the given matrix, then performing row operations to transform the original matrix into the identity matrix. The resulting right-hand side of the augmented matrix becomes the inverse of the original matrix.

CODE:

```
import numpy as np
def gauss_jordan_inversion(matrix):
    n = len(matrix)
    augmented_matrix = np.hstack((matrix, np.eye(n)))
    for i in range(n):
        diag_element = augmented_matrix[i, i]
        if diag_element == 0:
            raise ValueError("Matrix is singular and cannot be inverted.")
        augmented_matrix[i] = augmented_matrix[i] / diag_element
        for j in range(n):
            if i != j:
                factor = augmented_matrix[j, i]
                augmented_matrix[j] -= factor * augmented_matrix[i]
    inverse_matrix = augmented_matrix[:, n:]
    return inverse_matrix
if __name__== "__main__":
    A = np.array([[2, 1], [7, 4]], dtype=float)
    print("Original Matrix:")
    print(A)
    try:
        inverse = gauss_jordan_inversion(A)
        print("\nInverse Matrix:")
        print(inverse)
    except ValueError as e:
      print(e)
```

RESULT:

Thus matrix inversion using Gauss Jordan method is implemented.

OUTPUT:

1. Power method

$$A = \begin{pmatrix} 4 & 1 \\ 2 & 3 \end{pmatrix}$$

Dominant eigen value: λ=5
Corresponding eigenvector: [0.707, 0.707]

2. Jacobi method:

$$A = \begin{pmatrix} 4 & 1 \\ 2 & 3 \end{pmatrix}$$

Eigenvalues: λ1=5 and λ2=2
Corresponding eigenvectors: v1= [0.707,−0.707], v2= [0.707,0.707]

EXPT NO: 9	EIGEN VALUES OF A MATRIX BY POWER METHOD AND BY JACOBI’S METHOD

AIM:
To compute the eigenvalues of a matrix using two different methods:

1. Power Method: To find the dominant eigenvalue of a matrix.
2. Jacobi's Method: To find all the eigenvalues of a symmetric matrix.

THEORY:

1. Power Method:

The Power Method is an iterative algorithm used to find the largest (dominant) eigenvalue of a matrix A. The method works by repeatedly multiplying a random initial vector by the matrix and normalizing the result. As the iterations progress, the vector converges to the eigenvector corresponding to the dominant eigenvalue.

Steps:

1. Choose an initial vector b0.
2. Perform the following iterations: bk+1=Abk
3. Normalize the vector bk+1.
4. Compute the Rayleigh quotient to estimate the eigenvalue:

$$\lambda_k = \frac{\mathbf{b}_k^T A \mathbf{b}_k}{\mathbf{b}_k^T \mathbf{b}_k}$$

5. Repeat the process until convergence.

CODE:

```
    Import numpy as np
    Def power_method(A, max_iter=1000, tol=1e-6):
            n = A.shape[0]
            b = np.random.rand(n)
            b = b / np.linalg.norm(b) # Normalize the vector

    for i in range(max_iter):
                   b_new = np.dot(A, b)
                   b_new = b_new / np.linalg.norm(b_new)
                   eigenvalue = np.dot(b_new.T, np.dot(A, b_new)) / np.dot(b_new.T,
b_new)
                   if np.linalg.norm(b_new - b) < tol:
    break
                   b = b_new
   return eigenvalue, b_new
   A = np.array([[4, 1], [2, 3]])
```

```
eigenvalue, eigenvector = power_method(A)
print("Dominant Eigenvalue:", eigenvalue)
print("Corresponding Eigenvector:", eigenvector)
```

2. Jacobi's Method:

Jacobi's Method is an iterative method for finding all eigenvalues and eigenvectors of a
symmetric matrix by diagonalizing the matrix using orthogonal transformations (Jacobi rotations).

Steps:

1. Start with the matrix A.
2. Identify the largest off-diagonal element Aij.
3. Construct the Jacobi rotation matrix P to eliminate the off-diagonal element:
   - Compute the rotation angle θ using:

$$\cot(2\theta) = \frac{A_{ii} - A_{jj}}{2A_{ij}}, \quad \tan(2\theta) = \frac{2A_{ij}}{A_{ii} - A_{jj}}$$

   - Construct P and update the matrix:

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

4. Repeat until the matrix is diagonal.

Code:

```
import numpy as np
def jacobi_method(A, max_iter=1000, tol=1e-6):
        A = np.copy(A)
        n = A.shape[0]
        V = np.eye(n)
for _ in range(max_iter):
        off_diag = np.unravel_index(np.argmax(np.abs(np.tril(A, -1))), A.shape)
        i, j = off_diag
        if np.abs(A[i, j]) < tol:
break
        if A[i, i] != A[j, j]:
            theta = 0.5 * np.arctan(2 * A[i, j] / (A[i, i] - A[j, j]))
        else:
            theta = np.pi / 4
P = np.eye(n) P[i, i] = P[j, j] = np.cos(theta)
P[i, j] = -np.sin(theta)
P[j, i] = np.sin(theta)
A = np.dot(P.T, np.dot(A, P))
V = np.dot(V, P)
```

```
    return np.diagonal(A), V
            A = np.array([[4, 1], [1, 3]])
            eigenvalues, eigenvectors = jacobi_method(A)
            print("Eigenvalues:", eigenvalues)
            print("Eigenvectors:\n", eigenvectors)
```

RESULT:

Thus the Eigen values of a matrix by power method and by Jacobi's method was verified.

OUTPUT:

```
Matrix A:
     1     2     4
     3     8    14
     2     6    13

Matrix Q:
   0.2673   -0.7715    0.5774
   0.8018   -0.1543   -0.5774
   0.5345    0.6172    0.5774

Matrix R:
   3.7417   10.1559   19.2428
        0    0.9258    2.7775
        0         0    1.7321

Verification:
 (Q * R) = Matrix A
     1     2     4
     3     8    14
     2     6    13
```

EXPT NO: 10	QR DECOMPOSITION METHOD

AIM:

To decompose a given matrix into the product of an orthogonal matrix and an upper triangular matrix and also to find eigenvalues and eigenvectors.

THEORY:

QR decomposition is a factorization technique that decomposes a matrix A into the product of an orthogonal matrix Q and an upper triangular matrix R, such that A = QR. This decomposition is useful for solving linear systems, finding eigenvalues.

CODE:

```
function [Q, R] = qr_decomposition(A)
      [m, n] = size(A);
    Q = zeros(m, n);
    R = zeros(n, n);
    for j = 1:n
        v = A(:, j);
        for i = 1:j-1
            R(i, j) = Q(:, i)' * A(:, j);
            v = v - R(i, j) * Q(:, i);
        end
        R(j, j) = norm(v);
        Q(:, j) = v / R(j, j);
    end
end
A = [1 2 4;
         3 8 14;
         2 6 13];
[Q, R] = qr_decomposition(A);
disp('Matrix A:');
disp(A);
disp('Matrix Q:');
disp(Q);
disp('Matrix R:');
disp(R);
disp('Verification (Q * R):');
disp(Q * R);
```

RESULT:

The given matrix A was decomposed into two matrices of an orthogonal matrix Q and upper triangular matrix R

OUTPUT:

```
 Matrix A:
     1     2     3
     4     5     6
     7     8     9

Left singular vectors (U):
   -0.2148    0.8872    0.4082
   -0.5206    0.2496   -0.8165
   -0.8263   -0.3879    0.4082

Singular values (diagonal matrix S):
   16.8481         0         0
         0    1.0684         0
         0         0    0.0000

Right singular vectors (V):
   -0.4797   -0.7767    0.4082
   -0.5724   -0.0757   -0.8165
   -0.6651    0.6253    0.4082

Reconstructed matrix A from U, S, V:
    1.0000    2.0000    3.0000
    4.0000    5.0000    6.0000
    7.0000    8.0000    9.0000
```

EXPT NO: 11	SINGULAR VALUE DECOMPOSITION (SVD)

AIM:
To perform Singular Value Decomposition (SVD) on a given matrix using MATLAB and verify the factorization A = USVT.

THEORY:
Singular Value Decomposition (SVD) factors a matrix A into three components: A = USVT, where U and VT are orthogonal matrices, and S is a diagonal matrix containing the singular values. It is used for dimensionality reduction, noise filtering, and data compression.

CODE:
```
% Example MATLAB code for SVD
% Define a matrix A (you can change it to any matrix you want)
A = [1 2 3; 4 5 6; 7 8 9];
% Perform Singular Value Decomposition
[U, S, V] = svd(A);

% Display the results
disp('Matrix A:');
disp(A);

disp('Left singular vectors (U):');
disp(U);

disp('Singular values (diagonal matrix S):');
disp(S);

disp('Right singular vectors (V):');
disp(V);

% Reconstruct A using U, S, and V
A_reconstructed = U * S * V';
disp('Reconstructed matrix A from U, S, V:');
disp(A_reconstructed);
```

RESULT:
The matrix was successfully decomposed into three components: U (orthogonal matrix), S (diagonal matrix of singular values), and VT (orthogonal matrix). The original matrix was reconstructed using these components, confirming factorizing ability of a SVD matrix.