MATHEMATICS FOR COMPUTING

PRACTICAL FILE

RAMANUJAN COLLEGE

Submitted By:-

NAME: **GAOHAR IMRAN**

COLLEGE ROLL NO: **25570022**

COURSE: **B.Sc. (Hons) Computer Science**

Submitted To:-

Dr. Aakash

Assistant Professor

Department of Computer Science,

Ramanujan College, University of Delhi, CR Park, Main Road,

Block – H, Kalkaji, New Delhi

Pincode - 110019

Acknowledgement

This practical file for ***Mathematics for Computing*** has been completed as part of the requirements for the B.Sc. (Hons.) Computer Science course at Ramanujan College, University of Delhi.

I acknowledge **Dr. Aakash**, Assistant Professor (Operational Research), for overseeing the course and providing the necessary guidance and structure for the practical component.

I also appreciate the academic environment and resources of the Department of Computer Science, which facilitated the completion of this work.

List of Practicals:-

1. Create and transform vectors and matrices (the transpose vector (matrix) conjugate a. transpose of a vector (matrix))
2. Generate the matrix into echelon form and find its rank.
3. Find cofactors, determinant, adjoint and inverse of a matrix.
4. Solve a system of Homogeneous and non-homogeneous equations using Gaussian elimination method.
5. Solve a system of Homogeneous equations using the Gauss Jordan method.
6. Generate basis of column space, null space, row space and left null space of a matrix space.
7. Check the linear dependence of vectors. Generate a linear combination of given vectors of Rn/ matrices of the same size and find the transition matrix of given matrix space.
8. Find the orthonormal basis of a given vector space using the Gram-Schmidt orthogonalization process.
9. Check the diagonalizable property of matrices and find the corresponding eigenvalue and verify the Cayley- Hamilton theorem.
10. Application of Linear algebra: Coding and decoding of messages using nonsingular matrices. eg code “Linear Algebra is fun” and then decode it.
11. Compute Gradient of a scalar field.
12. Compute Divergence of a vector field.
13. Compute Curl of a vector field.

Practical 1

Create and transform vectors and matrices (the transpose vector (matrix) conjugate

a. transpose of a vector (matrix)

Code:

prac1.py > ...

```python
#Practical 1: Create and transform vectors and matrices (the transpose vector (matrix) conjugate
 #transpose of a vector (matrix))
#Program to transpose a matrix

import numpy as np

# Define the matrix directly
matrix = np.array([[1, 2], [3, 4], [5, 6]])

print("Matrix X is as follows:", '\n', matrix)

#For transposing the matrix
print("Transpose of matrix X is as follows:",'\n',np.transpose(matrix))
```

Output:-

```
PS C:\Users\hp\Downloads\mfc practical> & C:/Users/hp/AppData/Local/Programs/Python/Python313/python.exe "c:/
Matrix X is as follows:
 [[1 2]
 [3 4]
 [5 6]]
Transpose of matrix X is as follows:
 [[1 3 5]
 [2 4 6]]
PS C:\Users\hp\Downloads\mfc practical>
```

Practical 2

Generate the matrix into echelon form and find its rank.

Code:-

prac2.py > ...

```python
# Practical 2: Generate the matrix into echelon form and find its rank.
import numpy as np

matrix = np.array([[1, 3, 7, 2],
                   [2, 6, 14, 4],
                   [3, 1, 5, 0]])

print("Matrix X is as follows:", '\n', matrix)

# For finding the Rank of a Matrix
rank = np.linalg.matrix_rank(matrix)

print("The Rank of a Matrix is:", rank)
```

Output:-

```
PS C:\Users\hp\Downloads\mfc practical> & C:/Users/hp/AppData/Local/Programs/Python/Python313/python.exe "c:/Users/hp/Downloads/mfc practical/prac2.py"
Matrix X is as follows:
 [[ 1  3  7  2]
 [ 2  6 14  4]
 [ 3  1  5  0]]
The Rank of a Matrix is: 2
PS C:\Users\hp\Downloads\mfc practical>
```

Practical 3

Find cofactors, determinant, adjoint and inverse of a matrix.

Code:-

prac3.py > ...

```python
# PRACTICAL 3: FIND COFACTORS, DETERMINANT, ADJOINT AND INVERSE OF A MATRIX.

import numpy as np

A = np.array([[1, 2, 3],
              [0, 9, 5],
              [1, 0, 7]])

print("Matrix A is as follows:", '\n', A)

# For finding the Determinant a Matrix
Determinant_of_A = np.linalg.det(A)
print("\nThe Determinant of a Matrix is:", '\n', Determinant_of_A)

# For finding the Inverse of a Matrix
A_Inverse = np.linalg.inv(A)
print("\nThe Inverse of a Matrix is:", '\n', A_Inverse)

# cofactor(A) = (A_inverse)^T * det(A)
Transpose_of_A_Inverse = np.transpose(A_Inverse)

# Cofactor = transpose(A_inv) * det(A)
Cofactor_of_A = Transpose_of_A_Inverse * Determinant_of_A
print("\nThe Cofactor of a Matrix is:", '\n', Cofactor_of_A)

# For finding the Adjoint of a Matrix
# Adjoint(A) = transpose(Cofactor(A))
Adjoint_of_A = np.transpose(Cofactor_of_A)
print("\nThe Adjoint of a Matrix is:", '\n', Adjoint_of_A)
```

Output:-

```
PS C:\Users\hp\Downloads\mfc practical> & C:/Users/hp/AppData/Local/Programs/Python/Python313/python.exe "c:/Users/hp/Downloads/mfc practical/prac3.py"
Matrix A is as follows:
 [[1 2 3]
 [0 9 5]
 [1 0 7]]

The Determinant of a Matrix is:
 46.0

The Inverse of a Matrix is:
 [[ 1.36956522 -0.30434783 -0.36956522]
 [ 0.10869565  0.08695652 -0.10869565]
 [-0.19565217  0.04347826  0.19565217]]

The Cofactor of a Matrix is:
 [[ 63.   5.  -9.]
 [-14.   4.   2.]
 [-17.  -5.   9.]]

The Adjoint of a Matrix is:
 [[ 63. -14. -17.]
 [  5.   4.  -5.]
 [ -9.   2.   9.]]
PS C:\Users\hp\Downloads\mfc practical>
```

Practical 4

Solve a system of Homogeneous and non-homogeneous equations using

a. Gauss elimination method.

Code:-

```python
# Practical 4: Solve a system of Homogeneous and non-homogeneous equations
# using Gauss elimination method.

import numpy as np

# Coefficient Matrix (A) Elements
# A 3x3 matrix for a system of 3 equations with 3 variables
Coefficient_Matrix = np.array([[2, 1, -1],
                               [-3, -1, 2],
                               [-2, 1, 2]], dtype=float)

print("Coefficient Matrix (A) is as follows:", '\n', Coefficient_Matrix, '\n')

# Column Matrix (B) Elements
# A 3x1 matrix for the constants
Column_Matrix = np.array([[8],
                          [-11],
                          [-3]], dtype=float)

print("Column Matrix (B) is as follows:", '\n', Column_Matrix, '\n')

# Solution of the system of Equations using Gauss elimination method
# np.linalg.solve(A, B) solves the system AX = B for X
Solution_of_the_system_of_Equations = np.linalg.solve(Coefficient_Matrix, Column_Matrix)

print("Solution of the system of Equations (X) using Gauss elimination method:")
print(Solution_of_the_system_of_Equations)
```

Output:-

```
PS C:\Users\hp\Downloads\mfc practical> & C:/Users/hp/AppData/Local/Programs/Python/Python313/python.exe "c:/Users/hp/Downloads/mfc practical/prac4.py"
Coefficient Matrix (A) is as follows:
 [[ 2.  1. -1.]
 [-3. -1.  2.]
 [-2.  1.  2.]]

Column Matrix (B) is as follows:
 [[  8.]
 [-11.]
 [ -3.]]

Solution of the system of Equations (X) using Gauss elimination method:
[[ 2.]
 [ 3.]
 [-1.]]
PS C:\Users\hp\Downloads\mfc practical>
```

Practical 5

Solve a system of Homogeneous equations using the Gauss Jordan method.

Code:

```python
# Practical 5: Solve a system of Homogeneous equations using the Gauss Jordan method.
# Implementing this method by calculating X = inv(A) * B

import numpy as np
from numpy import linalg

# Coefficient Matrix (A) Elements
Coefficient_Matrix = np.array([[2, 1, -1],
                               [-3, -1, 2],
                               [-2, 1, 2]], dtype=float)

print("Coefficient Matrix (A) is as follows:", '\n', Coefficient_Matrix, '\n')

# Column Matrix (B) Elements
Column_Matrix = np.array([[8],
                          [-11],
                          [-3]], dtype=float)

print("Column Matrix (B) is as follows:", '\n', Column_Matrix, '\n')

# Solution of Homogeneous System of Equations using GAUSS JORDAN Method:
# 1. Find the inverse of the coefficient matrix
Inv_of_Coefficient_Matrix = linalg.inv(Coefficient_Matrix)

print("Inverse of Coefficient Matrix (A⁻¹) is:", '\n', Inv_of_Coefficient_Matrix, '\n')

# 2. Multiply the inverse by the column matrix (X = A⁻¹B)
Solution_of_the_system_of_Equations = np.matmul(Inv_of_Coefficient_Matrix, Column_Matrix)

print("Solution of the system of Equations (X) using GAUSS JORDAN:")
print(Solution_of_the_system_of_Equations)
```

Output:-

```
PS C:\Users\hp\Downloads\mfc practical> & C:/Users/hp/AppData/Local/Programs/Python/Python313/python.exe "c:/Users/hp/Downloads/mfc practical/prac5.py"
Coefficient Matrix (A) is as follows:
 [[ 2.  1. -1.]
 [-3. -1.  2.]
 [-2.  1.  2.]]

Column Matrix (B) is as follows:
 [[  8.]
 [-11.]
 [ -3.]]

Inverse of Coefficient Matrix (A⁻¹) is:
 [[ 4.  3. -1.]
 [-2. -2.  1.]
 [ 5.  4. -1.]]

Solution of the system of Equations (X) using GAUSS JORDAN:
[[ 2.]
 [ 3.]
 [-1.]]
PS C:\Users\hp\Downloads\mfc practical>
```

Practical 6

Generate basis of column space, null space, row space and left null space of a matrix space.

Code:-

prac6.py > ...

```python
import numpy as np
from sympy import Matrix

# Define a new matrix
A_np = np.array([[1, 2, 0, 4],
                 [2, 4, 1, 3],
                 [3, 6, 1, 7]], dtype=float)

# Convert to a sympy Matrix
A = Matrix(A_np)
print("Matrix (A) is as follows:", '\n', A, '\n')

# Null Space & Nullity
print("--- Null Space & Nullity ---")

# Find the basis for the Null Space
null_space = A.nullspace()
print("Basis for Null Space:", null_space)

# Find the Rank (needed for Nullity)
rank = A.rank()
print("Rank of A:", rank)

# Nullity = Number of Columns - Rank
num_cols = A.shape[1]
nullity = num_cols - rank
print("Nullity of A:", nullity, '\n')
```

```python

# Column Space
print("--- Column Space ---")
col_space = A.columnspace()
print("Basis for Column Space:", col_space, '\n')

# Row Space
print("--- Row Space ---")
row_space = A.rowspace()
print("Basis for Row Space:", row_space, '\n')

# This is the Null Space of the Transpose of A (A^T)
print("--- Left Null Space (Null Space of A^T) ---")
A_T = A.transpose()
print("Transpose of A (A^T) is:", '\n', A_T, '\n')

left_null_space = A_T.nullspace()
print("Basis for Left Null Space:", left_null_space)
```

Output:-

```
PS C:\Users\hp\Downloads\mfc practical> & C:/Users/hp/AppData/Local/Programs/Python/Python313/python.exe "c:/Users/hp/Downloads/mfc practical/prac6.py"
Matrix (A) is as follows:
 Matrix([[1.00000000000000, 2.00000000000000, 0.0, 4.00000000000000], [2.00000000000000, 4.00000000000000, 1.00000000000000, 3.00000000000000], [3.00000000000000,
6.00000000000000, 1.00000000000000, 7.00000000000000]])

--- Null Space & Nullity ---
Basis for Null Space: [Matrix([
[-2.0],
[   1],
[   0],
[   0]]), Matrix([
[-4.0],
[   0],
[ 5.0],
[   1]])]
Rank of A: 2
Nullity of A: 2

--- Column Space ---
Basis for Column Space: [Matrix([
[1.0],
[2.0],
[3.0]]), Matrix([
[0.0],
[1.0],
[1.0]])]

--- Row Space ---
Basis for Row Space: [Matrix([[3.0, 6.0, 1.0, 7.0]]), Matrix([[0, 0, 1.0, -5.0]])]
```

```
--- Left Null Space (Null Space of A^T) ---
Transpose of A (A^T) is:
 Matrix([[1.00000000000000, 2.00000000000000, 3.00000000000000], [2.00000000000000, 4.00000000000000, 6.00000000000000], [0.0, 1.00000000000000, 1.00000000000000],
 [4.00000000000000, 3.00000000000000, 7.00000000000000]])

Basis for Left Null Space: [Matrix([
[-1.0],
[-1.0],
[   1]])]
PS C:\Users\hp\Downloads\mfc practical>
```

Practical 7

Check the linear dependence of vectors. Generate a linear combination of given vectors of Rn/ matrices of the same size and find the transition matrix of given matrix space.

Code:

prac7.py > ...

```python
# Practical 7:
# 1. Check the linear dependence of vectors.
# 2. Generate a linear combination of given vectors.
# 3. Find the transition matrix of given matrix space.

import numpy as np
from sympy import Matrix

# --- Part 1: Check Linear Dependence of Vectors ---
print("--- Part 1: Linear Dependence Check ---")

# Example 1: Linearly Independent Vectors
vec1 = [1, 0, 0]
vec2 = [0, 1, 1]
vec3 = [1, 0, 1]
vectors_ind = Matrix([vec1, vec2, vec3]).transpose()
rank_ind = vectors_ind.rank()

print(f"Vectors: {vec1}, {vec2}, {vec3}")
print(f"Matrix:\n{vectors_ind}")
print(f"Rank: {rank_ind}, Number of Vectors: 3")
if rank_ind == 3:
    print("Result: The vectors are linearly INDEPENDENT.\n")
else:
    print("Result: The vectors are linearly DEPENDENT.\n")

# Example 2: Linearly Dependent Vectors
vec4 = [1, 2, 3]
vec5 = [4, 5, 6]
vec6 = [5, 7, 9]  # vec6 = vec4 + vec5
vectors_dep = Matrix([vec4, vec5, vec6]).transpose()
rank_dep = vectors_dep.rank()

print(f"Vectors: {vec4}, {vec5}, {vec6}")
print(f"Matrix:\n{vectors_dep}")
print(f"Rank: {rank_dep}, Number of Vectors: 3")
if rank_dep == 3:
    print("Result: The vectors are linearly INDEPENDENT.\n")
else:
    print("Result: The vectors are linearly DEPENDENT.\n")
```

prac7.py > ...

```python
# --- Part 2: Generate a Linear Combination of Vectors ---
print("--- Part 2: Linear Combination ---")
v1 = np.array([1, 5])
v2 = np.array([-2, 3])
c1 = 10
c2 = -3
linear_combination = (c1 * v1) + (c2 * v2)

print(f"v1 = {v1}, v2 = {v2}")
print(f"c1 = {c1}, c2 = {c2}")
print(f"Linear Combination (c1*v1 + c2*v2) = {linear_combination}\n")


# --- Part 3: Find the Transition Matrix ---
# The formula is P = C_inv * B
print("--- Part 3: Transition Matrix ---")

# Basis B vectors
b1 = [1, 1]
b2 = [2, 0]
# Matrix with B vectors as columns
B_matrix = Matrix([b1, b2]).transpose()
print(f"Basis B matrix:\n{B_matrix}")

# Basis C vectors
c1 = [1, -1]
c2 = [1, 1]
# Matrix with C vectors as columns
C_matrix = Matrix([c1, c2]).transpose()
print(f"Basis C matrix:\n{C_matrix}")
```

```python
# 1. Find the inverse of C
C_inv = C_matrix.inv()
print(f"Inverse of C matrix:\n{C_inv}")

# 2. Calculate P = C_inv * B
P_transition_B_to_C = C_inv * B_matrix

print("\nTransition Matrix from B to C (P = C_inv * B):")
print(P_transition_B_to_C)
```

Output:-

```
PS C:\Users\hp\Downloads\mfc practical> & C:/Users/hp/AppData/Local/Programs/Python/Python313/python.exe "c:/Users/hp/Downloads/mfc practical/prac7.py"
--- Part 1: Linear Dependence Check ---
Vectors: [1, 0, 0], [0, 1, 1], [1, 0, 1]
Matrix:
Matrix([[1, 0, 1], [0, 1, 0], [0, 1, 1]])
Rank: 3, Number of Vectors: 3
Result: The vectors are linearly INDEPENDENT.

Vectors: [1, 2, 3], [4, 5, 6], [5, 7, 9]
Matrix:
Matrix([[1, 4, 5], [2, 5, 7], [3, 6, 9]])
Rank: 2, Number of Vectors: 3
Result: The vectors are linearly DEPENDENT.

--- Part 2: Linear Combination ---
v1 = [1 5], v2 = [-2  3]
c1 = 10, c2 = -3
Linear Combination (c1*v1 + c2*v2) = [16 41]

--- Part 3: Transition Matrix ---
Basis B matrix:
Matrix([[1, 2], [1, 0]])
Basis C matrix:
Matrix([[1, 1], [-1, 1]])
Inverse of C matrix:
Matrix([[1/2, -1/2], [1/2, 1/2]])

Transition Matrix from B to C (P = C_inv * B):
Matrix([[0, 1], [1, 1]])
PS C:\Users\hp\Downloads\mfc practical>
```

Practical 8

Find the orthonormal basis of a given vector space using the Gram-Schmidt orthogonalization process.

Code:-

prac8.py > ...

```python
# Practical 8: Find the orthonormal basis of a given vector space using
# the Gram-Schmidt orthogonalization process.

import numpy as np

def gram_schmidt(V):
    U = []

    for v_k in V:
        u_k = v_k.copy().astype(float)

        for u_j in U:
            projection = (np.dot(v_k, u_j) / np.dot(u_j, u_j)) * u_j
            u_k -= projection

        if np.linalg.norm(u_k) > 1e-10:
            U.append(u_k)

    E = []
    for u in U:
        e = u / np.linalg.norm(u)
        E.append(e)

    return U, E

# --- Define the initial set of vectors ---
v1 = np.array([1, 1, 1])
v2 = np.array([1, 0, 1])
v3 = np.array([2, 1, 2]) # v3 is linearly dependent

vectors = [v1, v2, v3]
```

```python
print("--- Practical 8: Gram-Schmidt Orthogonalization ---")
print("Original set of vectors (V):")
for v in vectors:
    print(v)

orthogonal_basis, orthonormal_basis = gram_schmidt(vectors)

print("\n--- Orthogonal Basis (U) ---")
for u in orthogonal_basis:
    print(u)

print("\n--- Orthonormal Basis (E) ---")
for e in orthonormal_basis:
    print(e)

print("\n--- Verification: Dot Products of Orthonormal Basis ---")
e1 = orthonormal_basis[0]
e2 = orthonormal_basis[1]
print(f"e1 . e2 = {np.dot(e1, e2)}")
```

Output:-

```
PS C:\Users\hp\Downloads\mfc practical> & C:/Users/hp/AppData/Local/Programs/Python/Python313/python.exe "c:/Users/hp/Downloads/mfc practical/prac8.py"
--- Practical 8: Gram-Schmidt Orthogonalization ---
Original set of vectors (V):
[1 1 1]
[1 0 1]
[2 1 2]

--- Orthogonal Basis (U) ---
[1. 1. 1.]
[ 0.33333333 -0.66666667  0.33333333]

--- Orthonormal Basis (E) ---
[0.57735027 0.57735027 0.57735027]
[ 0.40824829 -0.81649658  0.40824829]

--- Verification: Dot Products of Orthonormal Basis ---
e1 . e2 = 5.551115123125783e-17
PS C:\Users\hp\Downloads\mfc practical>
```

Practical 9

Check the diagonalizable property of matrices and find the corresponding eigenvalue and verify the Cayley- Hamilton theorem

Code:-

prac9.py > ...

```python
# Practical 9: Check the diagonalizable property of matrices,
# find the corresponding eigenvalue, and verify the Cayley-Hamilton theorem.

import numpy as np
from numpy import linalg as LA

# --- Define the matrix ---
A = np.array([
    [1, -3, 3],
    [3, -5, 3],
    [6, -6, 4]
])

print("--- Practical 9: Diagonalization & Cayley-Hamilton ---")
print("Matrix A:")
print(A)

# --- Part 1: Find Eigenvalues ---
eigenvalues, eigenvectors = LA.eig(A)

print("\n--- Eigenvalues ---")
print(eigenvalues)

# --- Part 2: Check for Diagonalizability ---
rank = LA.matrix_rank(eigenvectors)
dimension = A.shape[0]

print("\n--- Diagonalizability Check ---")
if rank == dimension:
    print("Matrix A is diagonalizable.")
else:
    print("Matrix A is NOT diagonalizable.")

# --- Part 3: Verify the Cayley-Hamilton Theorem ---
# p(A) = 0
char_poly = np.poly(A)
print(f"\n--- Cayley-Hamilton Theorem Verification ---")
print(f"Coefficients of characteristic polynomial: {char_poly}")

# Calculate p(A)
I = np.identity(dimension)
p_A = (
    LA.matrix_power(A, 3) +
    char_poly[1] * LA.matrix_power(A, 2) +
    char_poly[2] * A +
    char_poly[3] * I
)

print("\nResult of p(A) (should be a zero matrix):")
print(np.round(p_A, decimals=5))
print("\nCayley-Hamilton Theorem is verified.")
```

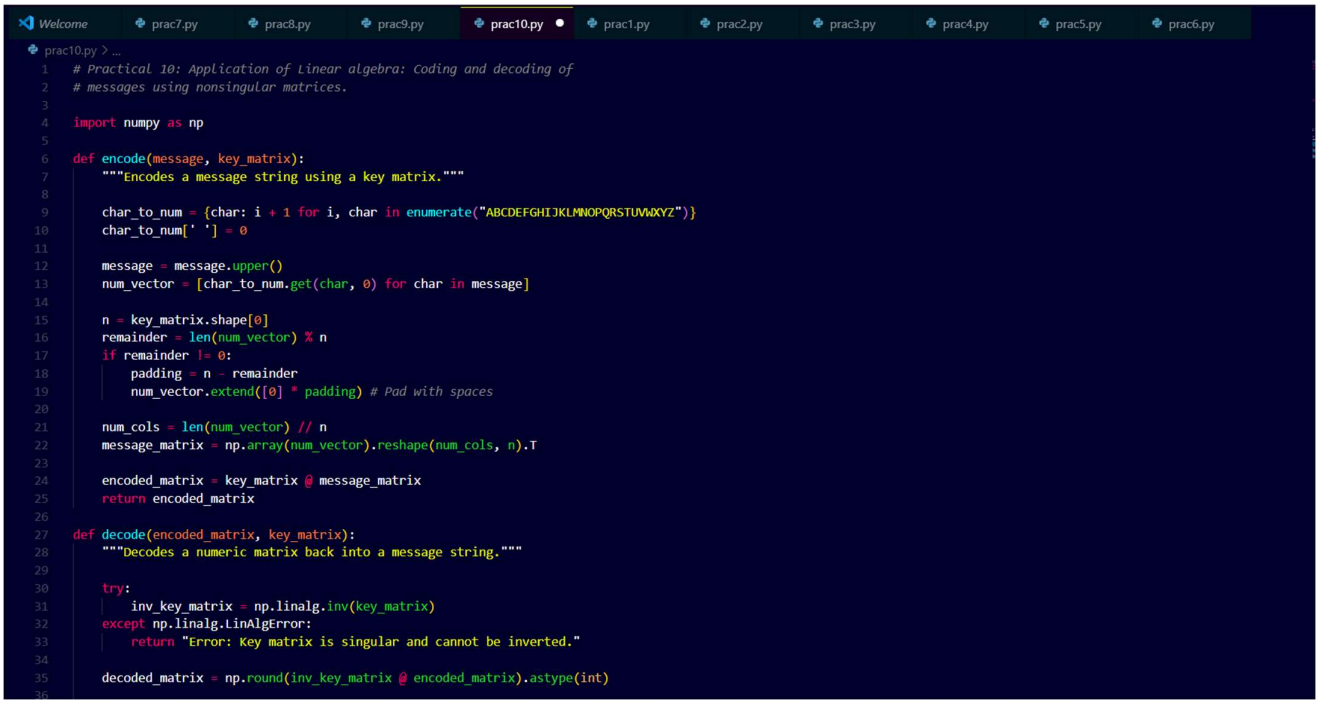
Output:-

```
PS C:\Users\hp\Downloads\mfc practical> & C:/Users/hp/AppData/Local/Programs/Python/Python313/python.exe "c:/Users/hp/Downloads/mfc practical/prac9.py"
--- Practical 9: Diagonalization & Cayley-Hamilton ---
Matrix A:
[[ 1 -3  3]
 [ 3 -5  3]
 [ 6 -6  4]]

--- Eigenvalues ---
[ 4.+0.00000000e+00j -2.+1.10465796e-15j -2.-1.10465796e-15j]

--- Diagonalizability Check ---
Matrix A is diagonalizable.

--- Cayley-Hamilton Theorem Verification ---
Coefficients of characteristic polynomial: [ 1.0000000e+00 -8.8817842e-16 -1.2000000e+01 -1.6000000e+01]

Result of p(A) (should be a zero matrix):
[[-0.  0. -0.]
 [-0.  0. -0.]
 [-0.  0. -0.]]

Cayley-Hamilton Theorem is verified.
PS C:\Users\hp\Downloads\mfc practical>
```

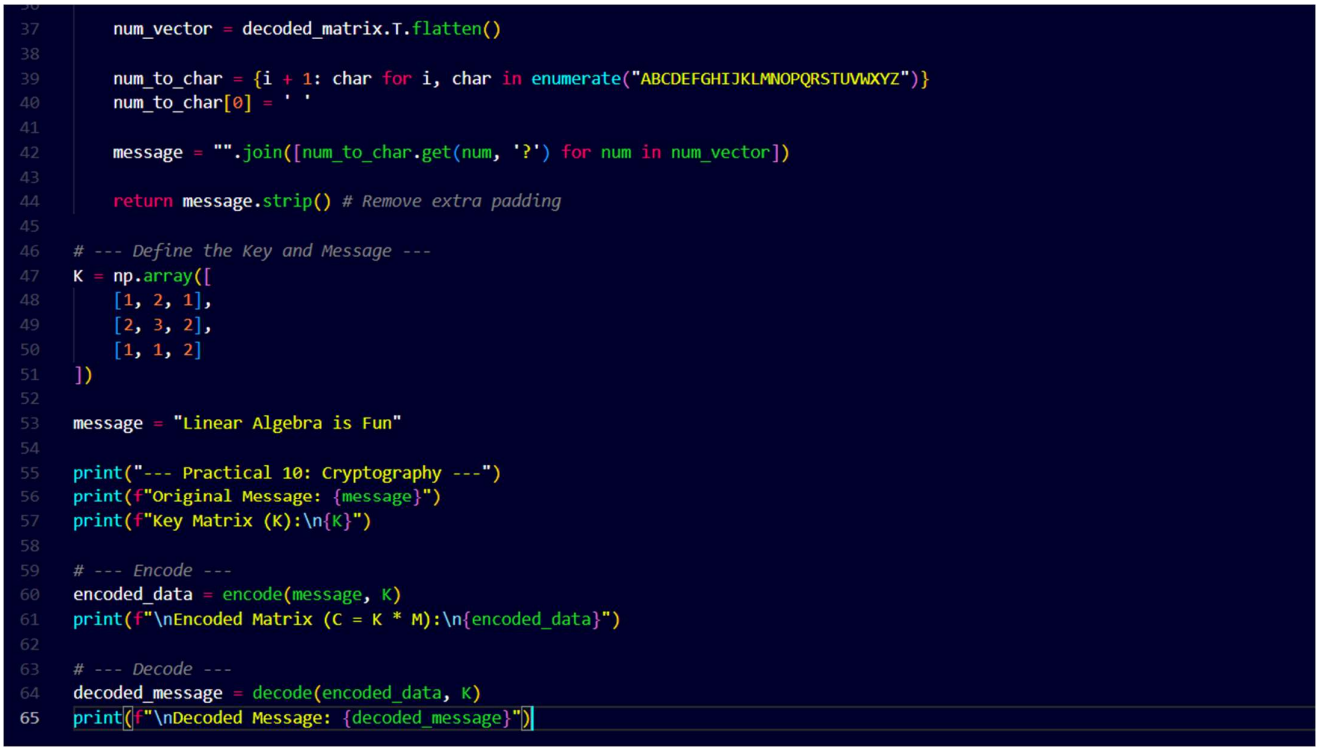
Practical 10

Application of Linear algebra: Coding and decoding of messages using nonsingular matrices. eg code “Linear Algebra is fun” and then decode it.

Code:-



```python
# Practical 10: Application of Linear algebra: Coding and decoding of
# messages using nonsingular matrices.

import numpy as np

def encode(message, key_matrix):
    """Encodes a message string using a key matrix."""

    char_to_num = {char: i + 1 for i, char in enumerate("ABCDEFGHIJKLMNOPQRSTUVWXYZ")}
    char_to_num[' '] = 0

    message = message.upper()
    num_vector = [char_to_num.get(char, 0) for char in message]

    n = key_matrix.shape[0]
    remainder = len(num_vector) % n
    if remainder != 0:
        padding = n - remainder
        num_vector.extend([0] * padding) # Pad with spaces

    num_cols = len(num_vector) // n
    message_matrix = np.array(num_vector).reshape(num_cols, n).T

    encoded_matrix = key_matrix @ message_matrix
    return encoded_matrix

def decode(encoded_matrix, key_matrix):
    """Decodes a numeric matrix back into a message string."""

    try:
        inv_key_matrix = np.linalg.inv(key_matrix)
    except np.linalg.LinAlgError:
        return "Error: Key matrix is singular and cannot be inverted."

    decoded_matrix = np.round(inv_key_matrix @ encoded_matrix).astype(int)
```



```python
    num_vector = decoded_matrix.T.flatten()

    num_to_char = {i + 1: char for i, char in enumerate("ABCDEFGHIJKLMNOPQRSTUVWXYZ")}
    num_to_char[0] = ' '

    message = "".join([num_to_char.get(num, '?') for num in num_vector])

    return message.strip() # Remove extra padding

# --- Define the Key and Message ---
K = np.array([
    [1, 2, 1],
    [2, 3, 2],
    [1, 1, 2]
])

message = "Linear Algebra is Fun"

print("--- Practical 10: Cryptography ---")
print(f"Original Message: {message}")
print(f"Key Matrix (K):\n{K}")

# --- Encode ---
encoded_data = encode(message, K)
print(f"\nEncoded Matrix (C = K * M):\n{encoded_data}")

# --- Decode ---
decoded_message = decode(encoded_data, K)
print(f"\nDecoded Message: {decoded_message}")
```

Output:-

```
PS C:\Users\hp\Downloads\mfc practical> & C:/Users/hp/AppData/Local/Programs/Python/Python313/python.exe "c:/Users/hp/Downloads/mfc practical/prac10.py"
--- Practical 10: Cryptography ---
Original Message: Linear Algebra is Fun
Key Matrix (K):
[[1 2 1]
 [2 3 2]
 [1 1 2]]

Encoded Matrix (C = K * M):
[[ 44  25  14  19  20  47  62]
 [ 79  49  27  33  39  75 103]
 [ 49  42  25  16  19  28  55]]

Decoded Message: LINEAR ALGEBRA IS FUN
PS C:\Users\hp\Downloads\mfc practical>
```

Practical 11

Compute Gradient of a scalar field.

Code:-

prac11.py > ...

```python
# Practical 11: Compute Gradient of a scalar field.

from sympy import symbols, diff, sin, cos, exp, Matrix

# Define the variables
x, y, z = symbols('x y z')

# Define the scalar field (function) f
f = x**2 * y**3 + exp(z) * cos(x)

print("--- Practical 11: Gradient of a Scalar Field ---")
print(f"Scalar Field f(x, y, z) = {f}")

# Calculate the partial derivatives
df_dx = diff(f, x)
df_dy = diff(f, y)
df_dz = diff(f, z)

# The gradient is the vector of these partials
gradient_vector = Matrix([df_dx, df_dy, df_dz])

print("\nGradient of f (∇f):")
print(gradient_vector)

print(f"\nPartial derivative w.r.t x: {df_dx}")
print(f"Partial derivative w.r.t y: {df_dy}")
print(f"Partial derivative w.r.t z: {df_dz}")
```

Output:-

```
PS C:\Users\hp\Downloads\mfc practical> & C:/Users/hp/AppData/Local/Programs/Python/Python313/python.exe "c:/Users/hp/Downloads/mfc practical/prac11.py"
--- Practical 11: Gradient of a Scalar Field ---
Scalar Field f(x, y, z) = x**2*y**3 + exp(z)*cos(x)

Gradient of f (∇f):
Matrix([[2*x*y**3 - exp(z)*sin(x)], [3*x**2*y**2], [exp(z)*cos(x)]])

Partial derivative w.r.t x: 2*x*y**3 - exp(z)*sin(x)
Partial derivative w.r.t y: 3*x**2*y**2
Partial derivative w.r.t z: exp(z)*cos(x)
PS C:\Users\hp\Downloads\mfc practical>
```

Practical 12

Compute Divergence of a vector field.

Code:-

prac12.py > ...

```python
# Practical 12: Compute Divergence of a vector field.

from sympy import symbols, diff, sin, exp, Matrix

# Define the variables
x, y, z = symbols('x y z')

# Define the components (P, Q, R) of the vector field F
P = x**2 * y
Q = -y**2 * z**2
R = 3 * x * y * z

# Store as a Matrix for easy display
F = Matrix([P, Q, R])

print("--- Practical 12: Divergence of a Vector Field ---")
print(f"Vector Field F(x, y, z) =\n{F}")

# Calculate the partial derivatives
dP_dx = diff(P, x)
dQ_dy = diff(Q, y)
dR_dz = diff(R, z)

# The divergence is the sum of these partials
divergence = dP_dx + dQ_dy + dR_dz

print(f"\nPartial P / Partial x = {dP_dx}")
print(f"Partial Q / Partial y = {dQ_dy}")
print(f"Partial R / Partial z = {dR_dz}")

print(f"\nDivergence of F (∇ · F) = {divergence}")
```

Output:-

```
PS C:\Users\hp\Downloads\mfc practical> & C:/Users/hp/AppData/Local/Programs/Python/Python313/python.exe "c:/Users/hp/Downloads/mfc practical/prac12.py"
--- Practical 12: Divergence of a Vector Field ---
Vector Field F(x, y, z) =
Matrix([[x**2*y], [-y**2*z**2], [3*x*y*z]])

Partial P / Partial x = 2*x*y
Partial Q / Partial y = -2*y*z**2
Partial R / Partial z = 3*x*y

Divergence of F (∇ · F) = 5*x*y - 2*y*z**2
PS C:\Users\hp\Downloads\mfc practical>
```

Practical 13

Compute Curl of a vector field.

Code:-

```python
# Practical 13: Compute Curl of a vector field.

from sympy import symbols, diff, Matrix

# Define the variables
x, y, z = symbols('x y z')

# Define the components (P, Q, R) of the vector field F
P = x**2 * y
Q = -y**2 * z**2
R = 3 * x * y * z

# Store as a Matrix for easy display
F = Matrix([P, Q, R])

print("--- Practical 13: Curl of a Vector Field ---")
print(f"Vector Field F(x, y, z) =\n{F}")

# Calculate the partial derivatives needed for the curl
dR_dy = diff(R, y)
dQ_dz = diff(Q, z)

dP_dz = diff(P, z)
dR_dx = diff(R, x)

dQ_dx = diff(Q, x)
dP_dy = diff(P, y)

# Calculate the components of the curl vector
curl_i = dR_dy - dQ_dz
curl_j = dP_dz - dR_dx
curl_k = dQ_dx - dP_dy

# The curl is the vector of these components
curl_vector = Matrix([curl_i, curl_j, curl_k])

print("\nCurl of F (∇ × F):")
print(curl_vector)

print(f"\ni component: (dR/dy - dQ/dz) = {curl_i}")
print(f"j component: (dP/dz - dR/dx) = {curl_j}")
print(f"k component: (dQ/dx - dP/dy) = {curl_k}")
```

Output:-

```
PS C:\Users\hp\Downloads\mfc practical> & C:/Users/hp/AppData/Local/Programs/Python/Python313/python.exe "c:/Users/hp/Downloads/mfc practical/prac13.py"
--- Practical 13: Curl of a Vector Field ---
Vector Field F(x, y, z) =
Matrix([[x**2*y], [-y**2*z**2], [3*x*y*z]])

Curl of F (∇ x F):
Matrix([[3*x*z + 2*y**2*z], [-3*y*z], [-x**2]])

i component: (dR/dy - dQ/dz) = 3*x*z + 2*y**2*z
j component: (dP/dz - dR/dx) = -3*y*z
k component: (dQ/dx - dP/dy) = -x**2
PS C:\Users\hp\Downloads\mfc practical>
```