Q1. Create a vector matrix and generate the transpose of that vector matrix.

Code :-

```
import numpy as np
import sympy as sp

# transpose of vector matrix
NR=int(input('enter no. of rows:-'))
NC=int(input('enter no. of columns:-'))
entries= list(map(float,input().split()))
A=np.array(entries).reshape(NR,NC)
Transpose= np.transpose(A)

print('transpose of matrix A:-', Transpose)
```

OUTPUT :-

```
================================================================================
enter no. of rows:-3
enter no. of columns:-3
1 2 3 4 6 7 8 9 5
transpose of matrix A:- [[1. 4. 8.]
 [2. 6. 9.]
 [3. 7. 5.]]
>>>
```

Q2. Generating the echelon form and rank of the matrix.

Code :-

```
import numpy as np
import sympy as sp

# ECHELON FORM AND RANK OF A MATRIX

NR= int(input('enter no. of rows:-'))
NC=int(input('enter no. of columns:-'))
elements=[]
print('enter elements row by row:-')
for i in range(NR):
  NR=list(map(float,input().split()))
  elements.append(NR)
A= np.array(elements)
A=sp.Matrix(A)
print('user defdined matrix:-')

echelon= A.echelon_form()
rank= A.rank()

print('echelon form :-',echelon)
print('rank:-', rank)
```

OUTPUT :-

```
enter no. of rows:-3
enter no. of columns:-3
enter elements row by row:-
2 2 0
5 0 6
9 8 3
user defdined matrix:-
echelon form :- Matrix([[9.00000000000000, 8.00000000000000, 3.00000000000000], [0, -40.0000000000000, 39.0000000000000], [0, 0, 162.000000000000]])
rank:- 3
>>>
```

Q3. Finding inverse, adjoint, transpose and cofactor of the matrix.

Code :-

```
import numpy as np
import sympy as sp

NR= int(input('enter no. of rows:-'))
NC=int(input('enter no. of columns:-'))
elements=[]
print('enter elements row by row:-')
for i in range(NR):
  row=list(map(float,input().split())) # Use float to avoid type issues later
  elements.append(row)
A_sympy= sp.Matrix(elements)
A_numpy = np.array(elements, dtype=np.float64) # Convert to NumPy array with float type
print('user defined matrix (SymPy):-', A_sympy)
print('user defined matrix (NumPy):-', A_numpy)

try:
    DETERMINANT = np.linalg.det(A_numpy)
    print('\ndeterminant of matrix:-', DETERMINANT)

    if DETERMINANT != 0:
        INVERSE = np.linalg.inv(A_numpy)
        print('\ninverse of matrix:-', INVERSE)

        TRANSPOSE = np.transpose(A_numpy)
        print('\ntranspose of matrix:-', TRANSPOSE)

        # Calculate the cofactor and adjoint matrices
        # The cofactor matrix is the transpose of the adjoint matrix
        # adj(A) = det(A) * inv(A)
        ADJOINT = DETERMINANT * INVERSE
        COFACTOR = np.transpose(ADJOINT)

        print('\ncofactor of matrix:-', COFACTOR)
        print('\nadjoint of matrix:-', ADJOINT)

    else:
        print("\nMatrix is singular, inverse does not exist.")

except np.linalg.LinAlgError:
    print("\nCould not calculate inverse. Matrix might be singular or not square.")
```

OUTPUT :-

```
========== RESTART: C:/Users/user/Downloads/mtc.py ==========
enter no. of rows:-3
enter no. of columns:-3
enter elements row by row:-
6 7 4
3 3 1
0 2 5
user defined matrix (SymPy):- Matrix([[6.00000000000000, 7.00000000000000, 4.00000000000000], [3.00000000000000, 3.00000000000000, 1.00000000000000], [0.0, 2.00000000000000, 5.00000000000000]])
user defined matrix (NumPy):- [[6. 7. 4.]
 [3. 3. 1.]
 [0. 2. 5.]]

determinant of matrix:- -3.0000000000000004

inverse of matrix:- [[ -4.33333333   9.          1.66666667]
 [  5.        -10.         -2.        ]
 [ -2.          4.          1.        ]]

transpose of matrix:- [[6. 3. 0.]
 [7. 3. 2.]
 [4. 1. 5.]]

cofactor of matrix:- [[ 13. -15.   6.]
 [-27.  30. -12.]
 [ -5.   6.  -3.]]

adjoint of matrix:- [[ 13. -27.  -5.]
 [-15.  30.   6.]
 [  6. -12.  -3.]]
>>>
```

Q4. Solving homogenous system of equation using Gauss elimination.

Code :-

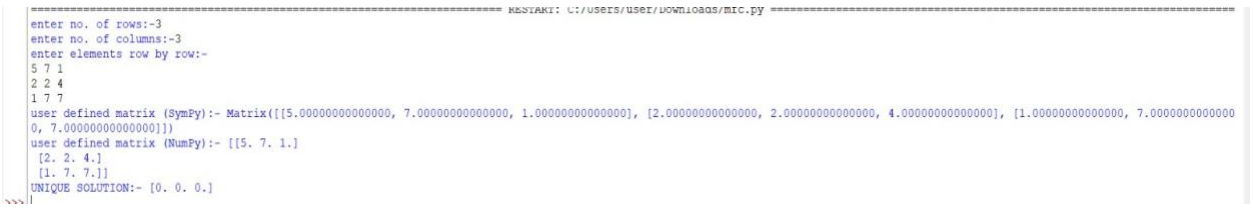
```
import numpy as np
import sympy as sp

# solving homogeneous system of equation using gauss elimination

NR= int(input('enter no. of rows:-'))
NC=int(input('enter no. of columns:-'))
elements=[]
print('enter elements row by row:-')
for i in range(NR):
  row=list(map(float,input().split())) # Use float to avoid type issues later
  elements.append(row)
A_sympy= sp.Matrix(elements)
A_numpy = np.array(elements, dtype=np.float64) # Convert to NumPy array with float type
print('user defined matrix (SymPy):-', A_sympy)
print('user defined matrix (NumPy):-', A_numpy)

Constant_Matrix= np.zeros(NR)
X= np.linalg.solve(A_numpy, Constant_Matrix)
print('UNIQUE SOLUTION:-', X)
```

OUTPUT :-



Q5. Solving homogenous system of equation using Gauss Jordan.

Code :-

```
import numpy as np
import sympy as sp

# SOLVING homogeneous system using gauss jordan

NR= int(input('enter no. of rows:-'))
NC=int(input('enter no. of columns:-'))
elements=[]
print('enter elements row by row:-')
for i in range(NR):
  row=list(map(float,input().split())) # Use float to avoid type issues later
  elements.append(row)
A_sympy= sp.Matrix(elements)
A_numpy = np.array(elements, dtype=np.float64) # Convert to NumPy array with float type
print('user defined matrix (SymPy):-', A_sympy)
print('user defined matrix (NumPy):-', A_numpy)

column_entries= list(map(float,input().split()))
column_matrix= np.array(column_entries).reshape(NR,1)

print('coeffiient matrix A:-', '\n', A_numpy)
print('column matrix B:-', '\n', column_matrix)

INVERSE_A= np.linalg.inv(A_numpy)
solution= np.matmul(INVERSE_A, column_matrix)
print(solution)
```

OUTPUT :-

```
========== RESTART: C:/Users/user/Downloads/mfc.py ==========
enter no. of rows:-3
enter no. of columns:-3
enter elements row by row:-
4 4 3
3 4 2
0 0 9
user defined matrix (SymPy):- Matrix([[4.00000000000000, 4.00000000000000, 3.00000000000000], [3.00000000000000, 4.00000000000000, 2.00000000000000], [0.0, 0.0, 9.00000000000000]])
user defined matrix (NumPy):- [[4. 4. 3.]
 [3. 4. 2.]
 [0. 0. 9.]]
```

Q6(a). Determining Null space and Nullity of the matrix.

Code :-

```
import numpy as np
import sympy as sp

# finding null space and nullity of matrix
NR= int(input('enter no. of rows:-'))
NC=int(input('enter no. of columns:-'))
elements=[]
print('enter elements row by row:-')
for i in range(NR):
  row=list(map(float,input().split())) # Use float to avoid type issues later
  elements.append(row)
A_sympy= sp.Matrix(elements)
A_numpy = np.array(elements, dtype=np.float64) # Convert to NumPy array with float type
print('user defined matrix (SymPy):-', A_sympy)
print('user defined matrix (NumPy):-', A_numpy)

nullspace = A_sympy.nullspace()
# nullspace=Matrix() # This line seems unnecessary, so I've commented it out

NoC= A_numpy.shape[1]
rank = A_sympy.rank()
nullity= NoC-rank
print('nullity of matrix A:-', nullity)
print('null space of matrix A:-', nullspace)
```

OUTPUT :-

```
---------- RESTART: C:/Users/User/Downloads/mrc.py ----------
enter no. of rows:-3
enter no. of columns:-3
enter elements row by row:-
3 2 5
1 2 0
8 8 0
user defined matrix (SymPy):- Matrix([[3.00000000000000, 2.00000000000000, 5.00000000000000], [1.00000000000000, 2.00000000000000, 0.0], [8.00000000000000, 8.00000000000000, 0.0]])
user defined matrix (NumPy):- [[3. 2. 5.]
 [1. 2. 0.]
 [8. 8. 0.]]
nullity of matrix A:- 0
null space of matrix A:- []
>>>
```

Q6(b). Determine column space and row space of the matrix.

Code :-

```
import numpy as np
import sympy as sp

# finding columnspace and rowspace

NR=int(input('enter no. of rows:-'))
NC=int(input('enter no. of columns:-'))
entries= list(map(float,input().split()))
A=np.array(entries).reshape(NR,NC)

m_columnspace= sp.Matrix(A).columnspace()
m_rowspace= sp.Matrix(A).rowspace()

print('columnspace of matrix A:-', m_columnspace)
print('rowspace of matrix A:-', m_rowspace)
```

OUTPUT

```
========== RESTART: C:/Users/user/Downloads/mic.py ==========
enter no. of rows:-3
enter no. of columns:-3
3 2 0 8 1 2 7 6 2
columnspace of matrix A:- [Matrix([
[3.0],
[8.0],
[7.0]]), Matrix([
[2.0],
[1.0],
[6.0]]), Matrix([
[0.0],
[2.0],
[2.0]])]
rowspace of matrix A:- [Matrix([[8.0, 1.0, 2.0]]), Matrix([[0, 41.0, 2.0]]), Matrix([[0, 0, -272.0]])]
>>>
```

Q7. Checking the linear dependency of the matrix and generating linear combination of the same.

Code :-

```
import numpy as np
import sympy as sp

# checking linear dependence of vectors and generate a linear combination of given vectors of matrices

NR= int(input("Enter the number of rows:"))
NC= int(input("Enter the number of columns:"))
print("Enter the entries in a single line (seperated by space)")
entries = list (map(int, input().split()))
A=np.array(entries).reshape(NR,NC)
A=sp.Matrix(A)
rank = A.rank()
print(f"\nRank = {rank}")
if rank == NC:
    print("Linearly independent.")
else:
    print("Linearly dependent.")
    ns = A.nullspace()
    for v in ns:
        expr = " + ".join([f"({v[i]})v{i+1}" for i in range(NC)]) + " = 0"
        print(expr)
```

OUTPUT :-

```
================================================================================
Enter the number of rows:3
Enter the number of columns:3
Enter the entries in a single line (seperated by space)
1 2 3 2 4 6 1 0 1

Rank = 2
Linearly dependent.
(-1)v1 + (-1)v2 + (1)v3 = 0
>>>
```

Q8. Generate the orthonormal basis of given set of vectors using Gram Schmidt orthogonalization process.

Code :-

```
import numpy as np
import sympy as sp

# Finding the orthonormal basis of given vector space using gram-schmidt orthogonalization process


n = int(input("Enter vector dimension: "))
print("Enter 3 vectors (each with", n, "elements):")
v = [np.array(list(map(float, input().split()))) for _ in range(3)]

# Gram-Schmidt process
u = [v[0]]
u.append(v[1] - np.dot(v[1], u[0]) / np.dot(u[0], u[0]) * u[0])
u.append(v[2] - sum(np.dot(v[2], ui) / np.dot(ui, ui) * ui for ui in u[:2]))

# Normalize
e = [ui / np.linalg.norm(ui) for ui in u]

# Output
print("\nOrthonormal basis vectors:")
for i, ei in enumerate(e, 1):
    print(f"e{i} =", np.round(ei, 4))
```

OUTPUT :-

```
========== RESTART: C:/
Enter vector dimension: 3
Enter 3 vectors (each with 3 elements):
3 4 7
1 8 3
2 1 0

Orthonormal basis vectors:
e1 = [0.3487 0.465  0.8137]
e2 = [-0.2259  0.8843 -0.4085]
e3 = [ 0.9096  0.0413 -0.4134]
```

Q9. Checking diagonalizable property of the matrix and finding corresponding eigenvalues, hence verifying Cayley Hamilton theorem.

Code :-

```
import numpy as np
import sympy as sp

# checking the diagonizable property of matrices and finding the corresponding wife values and verify cayley Hamilton theorem

import numpy as np
import sympy as sp

NR= int(input("Enter the number of rows:"))
NC= int(input("Enter the number of columns:"))
print("Enter the entries in a single line (seperated by space)")
entries = list (map(float, input().split()))
A=np.array(entries).reshape(NR,NC)

M = sp.Matrix(A)
try:
    P, D = M.diagonalize()
    print("Matrix is diagonalizable.")
    print("\nP (Eigenvectors):\n", P)
    print("\nD (Diagonal Matrix of Eigenvalues):\n", D)
except:
    print("Matrix is not diagonalizable.")

    p = M.charpoly()
    print("\nCharacteristic Polynomial:", p.as_expr())
    print("Verifying Cayley-Hamilton Theorem...")
    cayley_hamilton_result = p.eval(M)
    print("p(A) =\n", cayley_hamilton_result)
    if cayley_hamilton_result == sp.zeros(*M.shape):
        print("Cayley-Hamilton theorem verified successfully!")
    else:
        print("Cayley-Hamilton theorem not satisfied (possible rounding or symbolic issue).")
```

OUTPUT :-

```
========== RESTART: C:/Users/User/Downloads/mtc.py ==========
Enter the number of rows:3
Enter the number of columns:3
Enter the entries in a single line (seperated by space)
2 5 7 9 3 0 1 1 2
Matrix is diagonalizable.

P (Eigenvectors):
 Matrix([[0.615010950499444, -0.632586244422200, -0.275167087777334], [0.770153296573564, 0.806654772920123, 1.16315600289715], [0.169190515522124, -0.0287342106257025, -0.786440307185
733]])

D (Diagonal Matrix of Eigenvalues):
 Matrix([[10.1870088450193, 0, 0], [0, -4.05788447663914, 0], [0, 0, 0.870875631619812]])
```

Q10. Linear Algebra :- Encoding and Decoding of the message using non singular matrix.

Code :-

```
import numpy as np
import sympy as sp

# Linear algebra :- Coding and Decoding of message using non singular matrices.

import math
from sympy import Matrix

msg = input('enter the message to be encoded:-')
# prepare numeric vector: A=1..Z=26, other -> 0 (space/pad)
nums = [ord(c.upper())-64 if c.isalpha() else 0 for c in msg]
# pad to multiple of 3
pad = (-len(nums)) % 3
nums += [0]*pad

K = Matrix([[2,3,1],[1,1,1],[1,2,2]])
Kinv = K.inv()

# break into blocks of 3 and encode each block (no reshape)
blocks = [Matrix(nums[i:i+3]) for i in range(0, len(nums), 3)]
encoded_blocks = [K * b for b in blocks]
encoded = [int(x) for B in encoded_blocks for x in list(B)]
print("Encoded:", encoded)

# decode blockwise and reassemble
decoded_blocks = [Kinv * B for B in encoded_blocks]
decoded_nums = [round(int(x)) if float(x).is_integer() else round(float(x))
                for D in decoded_blocks for x in list(D)]
decoded_text = ''.join(chr(n+64) if 1 <= n <= 26 else ' ' for n in decoded_nums).strip()
print("Decoded:", decoded_text)
```

OUTPUT :-

```
========== RESTART: C:/Users/user/Downloads/mfc.py ==
enter the message to be encoded:-Linear Algebra IS FUN
Encoded: [65, 35, 58, 31, 24, 43, 15, 13, 26, 31, 14, 21, 39, 19, 20, 75, 28, 47, 89, 41, 76]
Decoded: LINEAR ALGEBRA IS FUN
```

Q11. Computing Gradient of scalar field.

Code :-

```
import numpy as np
import sympy as sp

# gradient of vector field

x, y, z = sp.symbols('x y z')
expr_input = input("Enter scalar function f(x,y,z) [default: x**2*y + sin(z)]: ").strip()
if expr_input == "":
    expr_input = "x**2*y + sin(z)"
f = sp.sympify(expr_input)
df_dx = sp.diff(f, x)
df_dy = sp.diff(f, y)
df_dz = sp.diff(f, z)
print("\nFunction f(x,y,z) =", f)
print("Gradient ∇f = [∂f/∂x, ∂f/∂y, ∂f/∂z]")
print("∂f/∂x =", df_dx)
print("∂f/∂y =", df_dy)
print("∂f/∂z =", df_dz)
```

OUTPUT

```
Enter scalar function f(x,y,z) [default: x**2*y + sin(z)]: x**2+y**4+z**3

Function f(x,y,z) = x**2 + y**4 + z**3
Gradient ∇f = [∂f/∂x, ∂f/∂y, ∂f/∂z]
∂f/∂x = 2*x
∂f/∂y = 4*y**3
∂f/∂z = 3*z**2
```

Q12. Computing Divergence of vector field.

Code :-

```
import numpy as np
import sympy as sp

# divergence of vector field

x, y, z = sp.symbols('x y z')
P = input("Enter P(x,y,z) [default: x*y]: ").strip()
Q = input("Enter Q(x,y,z) [default: y*z]: ").strip()
R = input("Enter R(x,y,z) [default: z*x]: ").strip()

if P == "": P = "x*y"
if Q == "": Q = "y*z"
if R == "": R = "z*x"

P = sp.sympify(P)
Q = sp.sympify(Q)
R = sp.sympify(R)
div = sp.diff(P, x) + sp.diff(Q, y) + sp.diff(R, z)
print("\nVector Field F =", (P, Q, R))
print("Divergence ∇·F =", sp.simplify(div))
```

OUTPUT :-

```
================================================================================
Enter P(x,y,z) [default: x*y]: tan(x)
Enter Q(x,y,z) [default: y*z]: exp(y)**5
Enter R(x,y,z) [default: z*x]: sin(z)+z**2

Vector Field F = (tan(x), exp(5*y), z**2 + sin(z))
Divergence ∇·F = 2*z + 5*exp(5*y) + cos(z) + cos(x)**(-2)
```

Q13. Computing curl of the vector field.

Code :-

```
import numpy as np
import sympy as sp

# curl of a vector field

x, y, z = sp.symbols('x y z')

P = input("Enter P(x,y,z) [default: y*z]: ").strip()
Q = input("Enter Q(x,y,z) [default: z*x]: ").strip()
R = input("Enter R(x,y,z) [default: x*y]: ").strip()

if P == "": P = "y*z"
if Q == "": Q = "z*x"
if R == "": R = "x*y"

P = sp.sympify(P)
Q = sp.sympify(Q)
R = sp.sympify(R)

curl_x = sp.diff(R, y) - sp.diff(Q, z)
curl_y = sp.diff(P, z) - sp.diff(R, x)
curl_z = sp.diff(Q, x) - sp.diff(P, y)

print("\nVector Field F =", (P, Q, R))
print("Curl ∇×F = [∂R/∂y - ∂Q/∂z, ∂P/∂z - ∂R/∂x, ∂Q/∂x - ∂P/∂y]")
print("∂R/∂y - ∂Q/∂z =", curl_x)
print("∂P/∂z - ∂R/∂x =", curl_y)
print("∂Q/∂x - ∂P/∂y =", curl_z)
```

OUTPUT :-

```
========== RESTART: C:/User
Enter P(x,y,z) [default: y*z]: cos(y)+x**3
Enter Q(x,y,z) [default: z*x]: sec(z)+tan(z)
Enter R(x,y,z) [default: x*y]: x**2**y

Vector Field F = (x**3 + cos(y), tan(z) + sec(z), x**(2**y))
Curl ∇×F = [∂R/∂y - ∂Q/∂z, ∂P/∂z - ∂R/∂x, ∂Q/∂x - ∂P/∂y]
∂R/∂y - ∂Q/∂z = 2**y*x**(2**y)*log(2)*log(x) - tan(z)**2 - tan(z)*sec(z) - 1
∂P/∂z - ∂R/∂x = -2**y*x**(2**y)/x
∂Q/∂x - ∂P/∂y = sin(y)
```