1. Create and transform vectors and matrices (the transpose vector (matrix) conjugate transpose of a vector (matrix)).

Code:

```
import numpy as np
R=int(input('Enter no. of rows'))
C=int(input('Enter no. of columns'))
print('Enter entries in single line')
entries=list(map(int,input().split()))
A=np.array(entries).reshape(R,C)

print('Original Matrix:',A)
print('Transpose Matrix:',A.T)
print('Conjugate Transpose Matrix:', np.conjugate(A.T))
```

Output:

```
Enter no. of rows3
Enter no. of columns2
Enter entries in single line
1 2 3 4 5 6
Original Matrix:
 [[1 2]
 [3 4]
 [5 6]]
Transpose Matrix:
 [[1 3 5]
 [2 4 6]]
Conjugate Transpose Matrix:
 [[1 3 5]
 [2 4 6]]
```

2. Generate the matrix into echelon form and find its rank.

Code:

```
import numpy as np
import sympy as sp
R=int(input('Enter no. of rows'))
C=int(input('Enter no. of columns'))
print('Enter entries in single line')
entries=list(map(int,input().split()))
A=np.array(entries).reshape(R,C)
M=sp.Matrix(A)
echelon_M=M.rref()

print('Row Echelon Form:\n', echelon_M)
print('Rank of matrix:',M.rank())
```

Output:

```
================================================
Enter no. of rows2
Enter no. of columns3
Enter entries in single line
1 2 3 4 5 6
Row Echelon Form:
 (Matrix([
[1, 0, -1],
[0, 1,  2]]), (0, 1))
Rank of matrix: 2
```

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

Code:

```
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)
print("Matrix:\n", M)
print("\nCofactor Matrix:\n", M.cofactor_matrix())
print("\nDeterminant:", M.det())
print("\nAdjoint (Adjugate):\n", M.adjugate())
if M.det() != 0:
    print("\nInverse:\n", M.inv())
else:
    print("\nMatrix is singular, inverse does not exist.")
```

Output:

```
Enter the number of rows:3
Enter the number of columns:3
Enter the entries in a single line (seperated by space)
23 77 90 91 36 10 88 69 42
Matrix:
 Matrix([[23.0000000000000, 77.0000000000000, 90.0000000000000], [91.0000000000000, 36.0000000000000, 10.0000000000000], [88.0000000000000, 69.0000000000
000, 42.0000000000000]])

Cofactor Matrix:
 Matrix([[822.000000000000, -2942.00000000000, 3111.00000000000], [2976.00000000000, -6954.00000000000, 5189.00000000000], [-2470.00000000000, 7960.00000
000000, -6179.00000000000]])

Determinant: 72362.0000000000

Adjoint (Adjugate):
 Matrix([[822.000000000000, 2976.00000000000, -2470.00000000000], [-2942.00000000000, -6954.00000000000, 7960.00000000000], [3111.00000000000, 5189.00000
000000, -6179.00000000000]])

Inverse:
 Matrix([[0.0113595533567342, 0.0411265581382494, -0.0341339377021089], [-0.0406566982670463, -0.0961001630690142, 0.110002487493436], [0.042992178215085
3, 0.0717089079903818, -0.0853901218871784]])
```

4. Solve a system of Homogeneous and non-homogeneous equations using Gauss elimination method.

Code:

```
import numpy as np
import sympy as sp

NR = int (input ("Enter number of equations (rows of A): "))
NC = int (input ("Enter number of variables (columns of A): "))
if NR != NC:
    print ("Error: For Gauss/LU solve, A must be a square matrix!")
    exit()
print ("Enter the entries of matrix A (row-wise, space-separated) :")
entries = list (map (float, input () .split ()))
A = np. array (entries) . reshape (NR, NC)
print ("Enter the entries of vector b (one value per row):")
entries2 = list (map (float, input().split()))
if len (entries2) != NR:
    print ("Error: b must have the same number of rows as A!")
    exit ()
b = np. array (entries2) . reshape (NR, 1)
M = sp.Matrix (A)
b_vec = sp. Matrix (b)
print ("\Matrix A:")
print (M)
print ("\nVector b:")
print (b_vec)
try:
    sol = M. LUsolve (b_vec)
    print ("\nSolution x:")
    print (sol)
except Exception as e:
```

Output:

```
========== RESTART: /Users/avanigarg/Documents/python prac.py ==========
Enter number of equations (rows of A): 3
Enter number of variables (columns of A): 3
Enter the entries of matrix A (row-wise, space-separated) :
1 2 3 4 5 6 7 8 9
Enter the entries of vector b (one value per row):
10 11 12
\Matrix A:
Matrix([[1.00000000000000, 2.00000000000000, 3.00000000000000], [4.00000000000000, 5.00000000000000, 6.00000000000000], [7.00000000000000, 8.00000000000000, 9.000000

Vector b:
Matrix([[10.0000000000000], [11.0000000000000], [12.0000000000000]])
\Matrix A is singular or system not solvable using LU decomposition.
```

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

Code:

```
import numpy as np
import sympy as sp

R = int (input ("Enter the number of equations: "))
C = int (input ("Enter the number of variables: "))
print ("Enter the coefficients in a single line separated by space:")
entries = list (map (int, input ().split ()))
A = np. array (entries) . reshape (R,C)
M = sp.Matrix (A)
print ("\nOriginal Coefficient Matrix A:")
print (M)
echelon = M. rref ()
print ("\nReduced Row Echelon Form:")
print (echelon)
print ("\nSolution (Null Space Basis) :")
null_space = M. nullspace ()
if null_space:
    print ("The system has infinitely many solutions:")
    for i, vec in enumerate (null_space, 1) :
        print("\nBasis vector {i):")
        print (vec)
else:
    print("only trivial solution: all variables = 0")
```

Output:

```
================================================================================
Enter the number of equations: 3
Enter the number of variables: 3
Enter the coefficients in a single line separated by space:
1 2 3 4 5 6 7 8 9

Original Coefficient Matrix A:
Matrix([[1, 2, 3], [4, 5, 6], [7, 8, 9]])

Reduced Row Echelon Form:
(Matrix([
[1, 0, -1],
[0, 1,  2],
[0, 0,  0]]), (0, 1))

Solution (Null Space Basis) :
The system has infinitely many solutions:

Basis vector {i):
Matrix([[1], [-2], [1]])
```

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

Code:

```
import numpy as np
import sympy as sp

R= int (input ("Enter the number of rows:"))
C= 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 (R,C)
M = sp. Matrix (A)
print ("Column Space Basis:\n", M. columnspace ())
print ("\nRow Space Basis: \n", M. rowspace ())
print ("\n. Null Space Basis:")
null = M. nullspace ()
print (null if null else "{ 0 ) (only zero vector) ")
print ("\n. Left Null Space Basis:")
left_null = M.T. nullspace ()
print (left_null if left_null else "( 0) (only zero vector) ")
```

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  4 5 6 7 8 9
Column Space Basis:
 [Matrix([
[1],
[4],
[7]]), Matrix([
[2],
[5],
[8]])]

Row Space Basis:
 [Matrix([[1, 2, 3]]), Matrix([[0, -3, -6]])]

. Null Space Basis:
[Matrix([
[ 1],
[-2],
[ 1]])]

. Left Null Space Basis:
[Matrix([
[ 1],
[-2],
[ 1]])]
```

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:

```
import numpy as np
import sympy as sp
R = int (input ("Enter the number of rows: "))
C = int (input ("Enter the number of columns: "))
print ("Enter the entries in a single line separated by space :")
entries = list (map (int, input () . split ()))
A = np. array (entries). reshape (R,C)
M = sp. Matrix (A)
print ("\nMatrix A:")
print (M)
print ("\nl. Linear Dependence:")
if M. rank () == M. shape [1] :
    print ("Linearly independent")
else:
    print ("Linearly dependent")

print ("\n2. Enter {C) coefficients (separated by space) :")
coeffs = list (map (int, input () .split ()))
result = M * sp. Matrix (coeffs)
print ("Linear combination result:")
print (result)

if input ("\n3. Find transition matrix? (y/n): "). lower () == 'y':
    print ("Enter new basis ({R)x{R}) entries (separated by space) :")
    new_entries = list (map (int, input ().split ()))
    B = sp. Matrix (np. array (new_entries) . reshape (R,R) )
    print ("\nTransition matrix:")
    print (B.inv ())
```

Output:

```
================================================================================ RES
Enter the number of rows: 3
Enter the number of columns: 3
Enter the entries in a single line separated by space :
1 2 3 4 5 6 7 8 9

Matrix A:
Matrix([[1, 2, 3], [4, 5, 6], [7, 8, 9]])

l. Linear Dependence:
Linearly dependent

2. Enter {C) coefficients (separated by space) :
10 11 12
Linear combination result:
Matrix([[68], [167], [266]])

3. Find transition matrix? (y/n): y
Enter new basis ({R)x{R}) entries (separated by space) :
20 30 40 50 60 70 80 90 10

Transition matrix:
Matrix([[-19/90, 11/90, -1/90], [17/90, -1/9, 1/45], [-1/90, 1/45, -1/90]])
```

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

Code:

```
import numpy as np
import sympy as sp

R = int(input("Enter the number of rows:"))
C = int(input("Enter the number of columns:"))
print ("Enter the entries in a single line separated by space:")
entries = list(map(float,input().split ()))
A = np.array(entries).reshape (R,C)
M = sp.Matrix (A)

print ("\nOriginal vectors (columns of A) :")
print (M)

Q, R= M.QRdecomposition()
print ("\nOrthonormal Basis (Q):")
print (Q)
print ("\nUpper triangular Matrix (R):")
print (R)
print ("\nVerification (Q * R should equal A) :")
print (Q*R)
```

Output:

```
Enter the number of rows:3
Enter the number of columns:3
Enter the entries in a single line separated by space:
1 2 3 4 5 6 7 8 9

Original vectors (columns of A) :
Matrix([[1.00000000000000, 2.00000000000000, 3.00000000000000], [4.00000000000000, 5.00000000000000, 6.00000000000000], [7.00000000000000, 8.00000000000000, 9.00000000000000]])

Orthonormal Basis (Q):
Matrix([[0.123091490979333, 0.904534033733291, 0.111111111111111], [0.492365963917331, 0.301511344577763, 0.444444444444444], [0.861640436855329, -0.301511344577764, 0.888888888888889]])

Upper triangular Matrix (R):
Matrix([[8.12403840463596, 9.60113629638795, 11.0782341881399], [0, 0.904534033733291, 1.80906806746658], [0, 0, 1.99840144432528e-15]])

Verification (Q * R should equal A) :
Matrix([[1.00000000000000, 2.00000000000000, 3.00000000000000], [4.00000000000000, 5.00000000000000, 6.00000000000000], [7.00000000000000, 8.00000000000000, 9.00000000000000]])
```

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

Code:

```
import numpy as np
import sympy as sp
R = int(input("Enter the number of rows:"))
C = int(input("Enter the number of columns:"))
print ("Enter the entries in a single line separated by space:")
entries = list(map(float,input().split ()))
A = np.array(entries).reshape(R,C)
M = sp.Matrix(A)

try:
    P, D = M. diagonalize()
    print ("Matrix is diagonalizable.")
    print ("\nEigenvectors : \n", P)
    print ("\nEigenvalues : \n", D)
except:
    print ("Matrix is not diagonalizable.")

print ("\nCayley-Hamilton Theorem:")
p = M.charpoly()
print ("Characteristic Polynomial:", p.as_expr())
result = p.as_expr ().subs (sp.Symbol('lambda'), M)
if result == sp.zeros (*M.shape):
    print ("p (A) = 0 / Cayley-Hamilton theorem verified!")
else:
    print ("Cayley-Hamilton theorem not satisfied.")
```

Output:

```
========== RESTART: /Users/avanigarg/Documents/python prac.py ==========
Enter the number of rows:3
Enter the number of columns:3
Enter the entries in a single line separated by space:
1 2 3 4 5 6 7 8 9
Matrix is diagonalizable.

Eigenvectors :
 Matrix([[-0.408248290463863, 0.785830238742067, 0.241161180276058], [0.816496580927726, 0.0867513392566285, 0.546134934329482], [-0.408248290463863, -0.612327560228810, 0.851108688382906]])

Eigenvalues :
 Matrix([[-5.70700919927125e-63, 0, 0], [0, -1.11684396980704, 0], [0, 0, 16.1168439698070]])

Cayley-Hamilton Theorem:
Characteristic Polynomial: 1.0*lambda**3 - 15.0*lambda**2 - 18.0*lambda
p (A) = 0 / Cayley-Hamilton theorem verified!
```

10.Application of Linear algebra: Coding and decoding of messages using nonsingular matrices.

Code:

```
import numpy as np
import sympy as sp
n = int(input("Enter matrix size:"))
print ("Enter {n*n) entries separated by space:")
entries = list (map (int, input (). split ()))
A = np.array (entries).reshape(n,n)
M= sp.Matrix(A)

print ("\nEncoding Matrix:")
print (M)
msg = input ("\nEnter message: ").upper ()
print ("\nOriginal Message:", msg)
nums =[]
for char in msg:
    if char.isalpha () :
        nums. append (ord (char) - ord ('A'))
    elif char == ' ':
        nums. append (26)
while len (nums) % n != 0:
    nums.append (26)
print ("As numbers:",nums)
encoded = []
for i in range(0, len (nums), n) :
    block = sp.Matrix (nums [i:i+n])
    encoded_block = M * block
    encoded.extend ([int (x) for x in encoded_block])
print ("\nEncoded: ", encoded)
M_inv = M.inv ()
decoded = []
for i in range(0, len (encoded), n) :
    block = sp. Matrix (encoded [i:i+n])
    decoded_block = M_inv * block
    decoded. extend ([int (round (x)) for x in decoded_block])
result = ""
for num in decoded:
    if 0 <= num <= 25:
        result += chr (num + ord('A'))
    elif num == 26:
        result +=' '
print ("\nDecoded Message:", result)
```

Output:

```
========== RESTART: /Users/avanigarg/Documents/pythor
Enter matrix size:3
Enter {n*n) entries separated by space:
2 4 6 7 12 25 9 42 18

Encoding Matrix:
Matrix([[2, 4, 6], [7, 12, 25], [9, 42, 18]])

Enter message: LINEAR ALGEBRA IS FUN

Original Message: LINEAR ALGEBRA IS FUN
As numbers: [11, 8, 13, 4, 0, 17, 26, 0, 11, 6, 4, 1, 17, 0, 26, 8, 18, 26, 5, 20, 13]

Encoded:  [132, 498, 669, 110, 453, 342, 118, 457, 432, 34, 115, 240, 190, 769, 621, 244, 922, 1296, 168, 600, 1119]

Decoded Message: LINEAR ALGEBRA IS FUN
```

11.Compute Gradient of a scalar field.

Code:

```
import sympy as sp
x, y, z = sp. symbols ('x y z')

expr_input = input ("Enter scalar function f(x,y,z): "). strip ()
if expr_input == "":
    expr_input = "x**2*y + sin(z) "

f = sp.sympify(expr_input)
print ("\nFunction f(x, y, z) =", f)

df_dx = sp.diff (f, x)
df_dy = sp. diff (f, y)
df_dz= sp. diff (f, z)

print ("\nGradient Vf = [df/dx, df/dy, df/dz]")
print ("df/dx =", df_dx)
print ("df/dy =", df_dy)
print ("df/dz =", df_dz)
print ("\nV =", [df_dx, df_dy, df_dz])
```

Output:

```
============================================================
Enter scalar function f(x,y,z): exp(x)+sin(y)+cos(z)

Function f(x, y, z) = exp(x) + sin(y) + cos(z)

Gradient Vf = [df/dx, df/dy, df/dz]
df/dx = exp(x)
df/dy = cos(y)
df/dz = -sin(z)

V = [exp(x), cos(y), -sin(z)]
```

12.Compute Divergence of a vector field.

Code:

```
import sympy as sp

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

P = input ("Enter P(x,y,z): ").strip()
Q = input ("Enter l(x, y,z): ").strip()
R = input ("Enter R(x,y,z): ").strip()

P= sp. sympify (P)
Q = sp. sympify (Q)
R = sp. sympify (R)
print ("\nVector Field F = [P, Q, R] =", [P, Q, R])
div = sp.diff (P, x) + sp.diff(Q, y) + sp.diff (R, z)
print ("\nDivergence V •F = dP/dx + dQ/dy + dR/dz")
print (" V •F=", sp.simplify (div) )
```

Output:

```
============================================================
Enter P(x,y,z): x
Enter l(x, y,z): y
Enter R(x,y,z): z

Vector Field F = [P, Q, R] = [x, y, z]

Divergence V •F = dP/dx + dQ/dy + dR/dz
 V •F= 3
```

13.Compute Curl of a vector field.

Code:

```
import sympy as sp
x, y, z = sp. symbols ('x y z')
P = input ("Enter P(x,y,z) : ").strip()
Q= input ("Enter Q(x, y,z) : ") . strip ()
R = input ("Enter R(x,Y,Z) : ") .strip()

P = sp. sympify (P)
Q = sp. sympify (Q)
R = sp. sympify (R)
print ("\nVector Field F = [P, Q, R] =", [P, Q, 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 (" \nCurl VxF = [dR/dy - dQ/dz, dP/dz - dR/dx, dQ/dx - dP/dy")
print ("\nVxF =",sp.simplify(curl_x), sp.simplify(curl_y), sp.simplify(curl_z))
```

Output:

```
============================================================
Enter P(x,y,z) : x**2
Enter Q(x, y,z) : y**2
Enter R(x,Y,Z) : z**2

Vector Field F = [P, Q, R] = [x**2, y**2, z**2]

Curl VxF = [dR/dy - dQ/dz, dP/dz - dR/dx, dQ/dx - dP/dy

VxF = 0 0 0
```