Q1-

import numpy as np

matrix = np.array([[5, -1, 3], [7, -1, 1], [-1, 2, 1]]) # Matrix

vector = np.array([1, 2, -3]) # Vector

matrix = matrix + 1j\*matrix

vector = vector + 1j\*vector

print("TRANSPOSE AND CONJUGATE TRANSPOSE OF VECTOR AND MATRIX")

print("\nVector is : ")

print(vector)

print("\nMatrix is : ")

print(matrix)

print("\nTranspose of Vector is : ")

print(np.matrix(vector).transpose())

print("\nTranspose of Matrix is : ")

print(np.matrix(matrix).transpose())

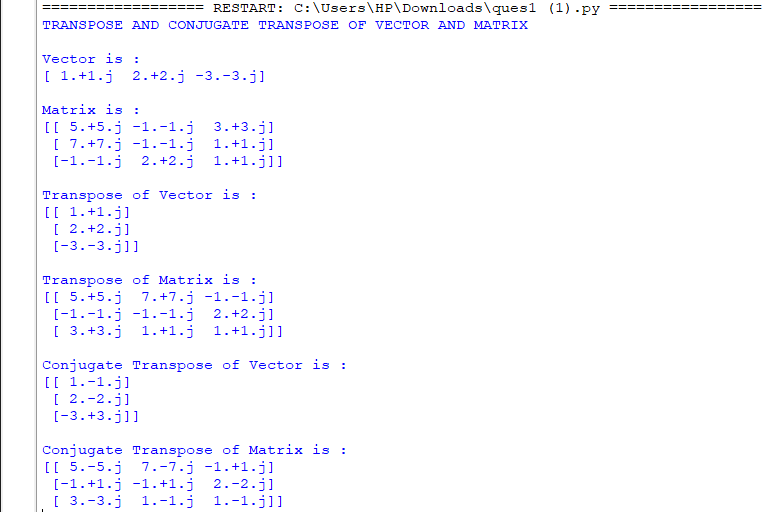
print("\nConjugate Transpose of Vector is : ")

print(np.matrix(vector).getH())

print("\nConjugate Transpose of Matrix is : ")

print(np.matrix(matrix).getH())

Output-



Q2-

import numpy as np

from sympy import Matrix

print("REDUCED ROW ECHELON FORM AND RANK OF MATRIX")

mx = Matrix([[1, 0, 1, 3],

[2, 3, 4, 7],

[-1, -3, -3, -4]])

print("\nMatrix Used is : ")

print(np.array(mx).astype(np.float64))

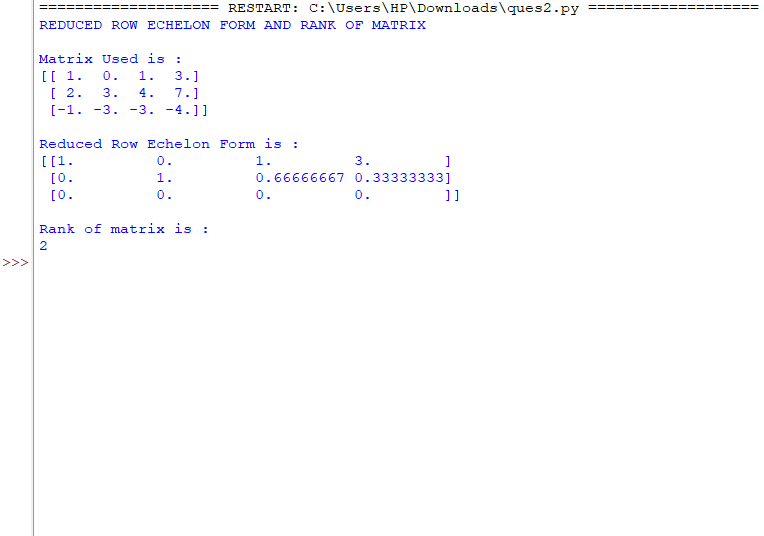
print("\nReduced Row Echelon Form is : ")

print(np.array(mx.rref()[0]).astype(np.float64))

print("\nRank of matrix is : ")

print(np.linalg.matrix\_rank(np.array(mx).astype(np.float64)))

Output-



Q3-

import numpy as np

# Matrix Size

N = 4

def matrix\_cofactor(matrix):

try:

determinant = np.linalg.det(matrix)

if(determinant!=0):

cofactor = None

cofactor = np.linalg.inv(matrix).T \* determinant

# return cofactor matrix of the given matrix

return cofactor

else:

raise Exception("singular matrix")

except Exception as e:

print("Could not find cofactor matrix due to",e)

# Function to get cofactor of

# A[p][q] in temp[][]. n is current

# dimension of A[][]

def getCofactor(A, temp, p, q, n):

i = 0

j = 0

# Looping for each element of the matrix

for row in range(n):

for col in range(n):

# Copying into temporary matrix only those element

# which are not in given row and column

if (row != p and col != q):

temp[i][j] = A[row][col]

j += 1

# Row is filled, so increase row index and

# reset col index

if (j == n - 1):

j = 0

i += 1

# Recursive function for finding determinant of matrix.

# n is current dimension of A[][].

def determinant(A, n):

D = 0 # Initialize result

# Base case : if matrix contains single element

if (n == 1):

return A[0][0]

temp = [] # To store cofactors

for i in range(N):

temp.append([None for \_ in range(N)])

sign = 1 # To store sign multiplier

# Iterate for each element of first row

for f in range(n):

# Getting Cofactor of A[0][f]

getCofactor(A, temp, 0, f, n)

D += sign \* A[0][f] \* determinant(temp, n - 1)

# terms are to be added with alternate sign

sign = -sign

return D

# Function to get adjoint of A[N][N] in adj[N][N].

def adjoint(A, adj):

if (N == 1):

adj[0][0] = 1

return

# temp is used to store cofactors of A[][]

sign = 1

temp = [] # To store cofactors

for i in range(N):

temp.append([None for \_ in range(N)])

for i in range(N):

for j in range(N):

# Get cofactor of A[i][j]

getCofactor(A, temp, i, j, N)

# sign of adj[j][i] positive if sum of row

# and column indexes is even.

sign = [1, -1][(i + j) % 2]

# Interchanging rows and columns to get the

# transpose of the cofactor matrix

adj[j][i] = (sign)\*(determinant(temp, N-1))

# Function to calculate and store inverse, returns false if

# matrix is singular.

def inverse(A, inverse):

# Find determinant of A[][]

det = determinant(A, N)

if (det == 0):

print("Singular matrix, can't find its inverse")

return False

# Find adjoint

adj = []

for i in range(N):

adj.append([None for \_ in range(N)])

adjoint(A, adj)

# Find Inverse using formula "inverse(A) = adj(A)/det(A)"

for i in range(N):

for j in range(N):

inverse[i][j] = adj[i][j] / det

return True

# Driver program

A = [[5, -2, 2, 7], [1, 0, 0, 3], [-3, 1, 5, 0], [3, -1, -9, 4]]

adj = [None for \_ in range(N)]

inv = [None for \_ in range(N)]

for i in range(N):

adj[i] = [None for \_ in range(N)]

inv[i] = [None for \_ in range(N)]

print("COFACTOR, DETERMINANT, ADJOINT AND INVERSE OF MATRIX")

print("\nMatrix Used is :")

print(np.array(A))

print("\nCofactors are :")

print(matrix\_cofactor(np.array(A)))

print("\nMatrix's Determinant : ")

print(round(np.linalg.det(np.array(A))))

print("\nThe Adjoint is :")

adjoint(A, adj)

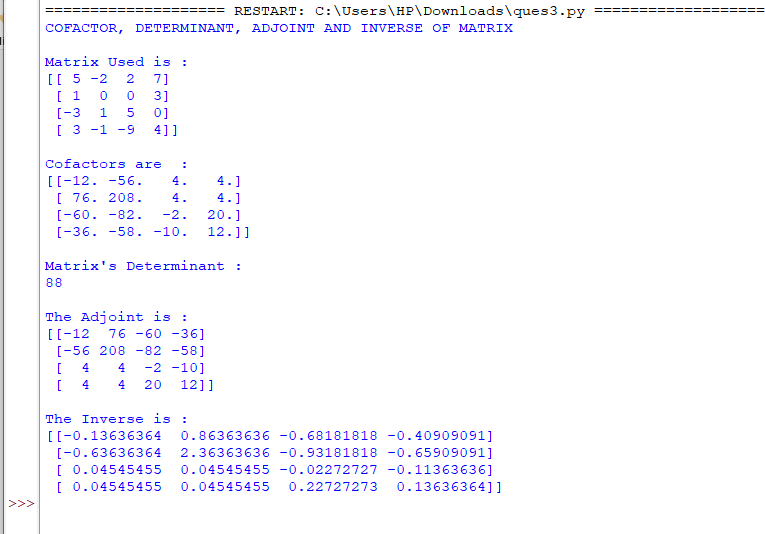
print(np.array(adj).reshape(N,N))

print("\nThe Inverse is :")

if (inverse(A, inv)):

print(np.array(inv).reshape(N,N))

Output-



Q4-

# Importing NumPy Library

import numpy as np

import sys

print("-----GAUSS ELIMINATION METHOD-----")

# Reading number of unknowns

n: int = 3

print('\nNumber of unknowns: ',n)

# Making numpy array of n x (n+1) size

a = np.zeros((n,n+1))

a = np.array([[1.0, 1.0, 1.0, 9.0], [2.0, -3.0, 4.0, 13.0], [3.0, 4.0, 5.0, 40.0]])

print("\nAugmented Matrix Coefficients : ")

print(a)

# Making numpy array of n size and initializing

# to zero for storing solution vector

x = np.zeros(n)

# Applying Gauss Elimination

for i in range(n):

if a[i][i] == 0.0:

sys.exit('\nDivide by zero detected!')

for j in range(i+1, n):

ratio = a[j][i]/a[i][i]

for k in range(n+1):

a[j][k] = a[j][k] - ratio \* a[i][k]

# Back Substitution

x[n-1] = a[n-1][n]/a[n-1][n-1]

for i in range(n-2,-1,-1):

x[i] = a[i][n]

for j in range(i+1,n):

x[i] = x[i] - a[i][j]\*x[j]

x[i] = x[i]/a[i][i]

# Displaying solution

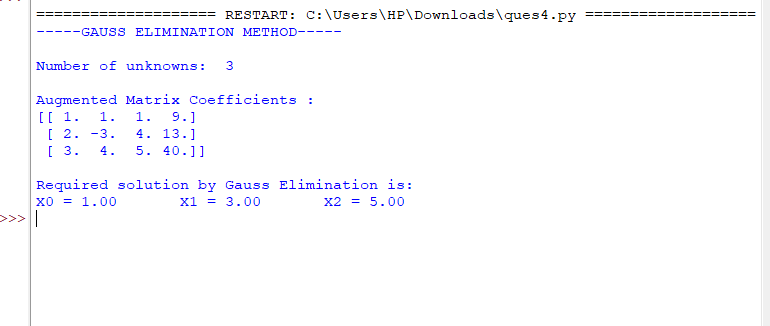
print('\nRequired solution by Gauss Elimination is: ')

for i in range(n):

print('X%d = %0.2f' %(i,x[i]), end = '\t')

print()

Output-



Q5-

# Importing NumPy Library

import numpy as np

import sys

print("-----GAUSS JORDAN METHOD-----")

# Reading number of unknowns

n:int = 3

print('\nNumber of unknowns: ',n)

# Making numpy array of n x (n+1) size

a = np.zeros((n,n+1))

a = np.array([[1.0, 1.0, 1.0, 9.0], [2.0, -3.0, 4.0, 13.0], [3.0, 4.0, 5.0, 40.0]])

print("\nAugmented Matrix Coefficients : ")

print(a)

# Making numpy array of n size and initializing

# to zero for storing solution vector

x = np.zeros(n)

# Applying Gauss Jordan Elimination

for i in range(n):

if a[i][i] == 0.0:

sys.exit('\nDivide by zero detected!')

for j in range(n):

if i != j:

ratio = a[j][i]/a[i][i]

for k in range(n+1):

a[j][k] = a[j][k] - ratio \* a[i][k]

# Obtaining Solution

for i in range(n):

x[i] = a[i][n]/a[i][i]

# Displaying solution

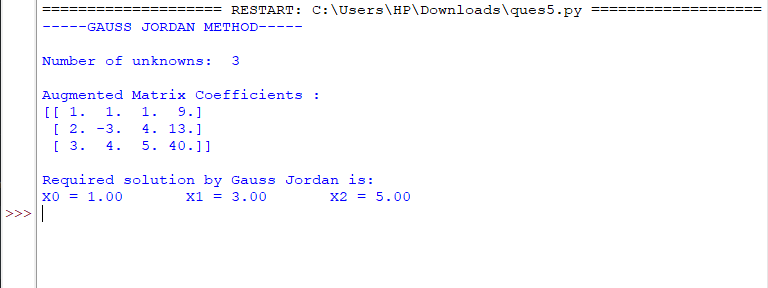
print('\nRequired solution by Gauss Jordan is: ')

for i in range(n):

print('X%d = %0.2f' %(i,x[i]), end = '\t')

print()

Output-



Q6-

# import sympy

from sympy import \*

M = Matrix([[1, 0, 1, 3], [2, 3, 4, 7], [-1, -3, -3, -4]])

print("Matrix Used is : ")

print(M)

# Use sympy.columnspace() method

M\_columnspace = M.columnspace()

print("\nColumn Space of the Matrix is : ")

print(M\_columnspace)

# Use sympy.rowspace() method

M\_rowspace = M.rowspace()

print("\nRow Space of the Matrix is : ")

print(M\_rowspace)

# Use sympy.nullspace() method

M\_nullspace = M.nullspace()

print("\nNull Space of the Matrix is : ")

print(M\_nullspace)

# Use sympy.nullspace() method

MT = M.T

M\_leftnullspace = MT.nullspace()

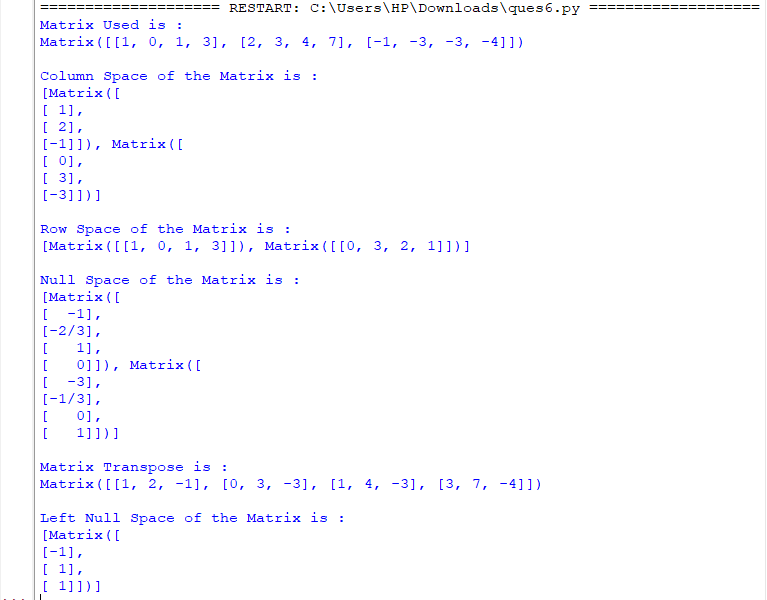
print("\nMatrix Transpose is : ")

print(MT)

print("\nLeft Null Space of the Matrix is : ")

print(M\_leftnullspace)

Output-



Q11-

import numpy as np

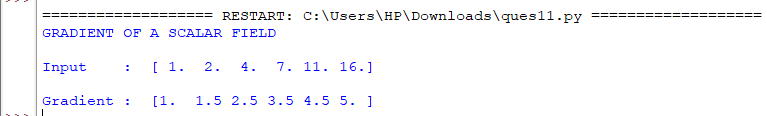
print("GRADIENT OF A SCALAR FIELD")

mx = np.array([1, 2, 4, 7, 11, 16], dtype=float)

print("\nInput : ", mx)

print("\nGradient : ", np.gradient(mx))

Output-



Q12-

import numpy as np

def divergence(F):

""" compute the divergence of n-D vector field `F` """

return np.ufunc.reduce(np.add,np.gradient(F))

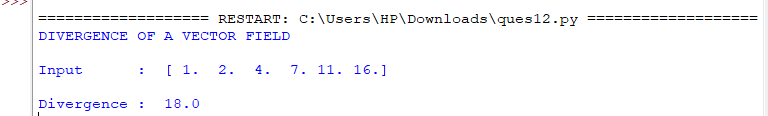
print("DIVERGENCE OF A VECTOR FIELD")

mx = np.array([1, 2, 4, 7, 11, 16], dtype=float)

print("\nInput : ", mx)

print("\nDivergence : ", divergence(mx))

Output-



Q13-

from sympy.physics.vector import ReferenceFrame

from sympy.physics.vector import curl

R = ReferenceFrame('R')

F = R[1]\*\*2 \* R[2] \* R.x - R[0]\*R[1] \* R.y + R[2]\*\*2 \* R.z

print("CURL OF A VECTOR FEILD")

print("\nSuppose F(x,y,z) = (y^2)zi - xyj + z2k, then:")

print("-> x would be R[0], y would be R[1] and z would be R[2]")

print("-> The unit vectors i, j, k of the 3 axes, would be respectively R.x, R.y, R.z")

print("\nVector is : ", F)

CURL = curl(F, R)

print("\nCurl will be : ", CURL)

print("\n-> In other words, CURL = 0i + (y^2)j + (-2yz-y)k")

Output-

