Sahakar Maharshi Bhausaheb Santuji Thorat College Sangamner

DEPARTMENT OF COMPUTER SCIENCE

MATHEMATICS

Remark
Demonstrators Signature
Date :- / /2023

Name :- Prem Vijay Vajare **Batch No. :- D** **Roll No:- 75** **Date:-** / /2023

Title of the:- Practical 15 **Expt. No .** 15 **Class :-** S.Y.BCS

Q.1) Write the python program to find area of the triangle ABC where A[0,0],B[5,0],C[3,3]

Syntax:

```
import math
def calculate_area(x1, y1, x2, y2, x3, y3):
    """Function to calculate area of a triangle given its three vertices."""
    area = abs((x1 * (y2 - y3) + x2 * (y3 - y1) + x3 * (y1 - y2)) / 2)
    return area
# Coordinates of vertices A, B, and C
Ax, Ay = 0, 0
Bx, By = 5, 0
Cx, Cy = 3, 3
# Call the function to calculate the area
area = calculate_area(Ax, Ay, Bx, By, Cx, Cy)
# Print the result
print("Area of triangle ABC is:", area)
```

OUTPUT:

Area of triangle ABC is: 7.5

Q.2) Write the python program to plot the graphs of sin x, cox, e**x and x **2 in [0,5] in one figure with 2X2 subplots

Syntax:

```
import numpy as np
import matplotlib.pyplot as plt
```

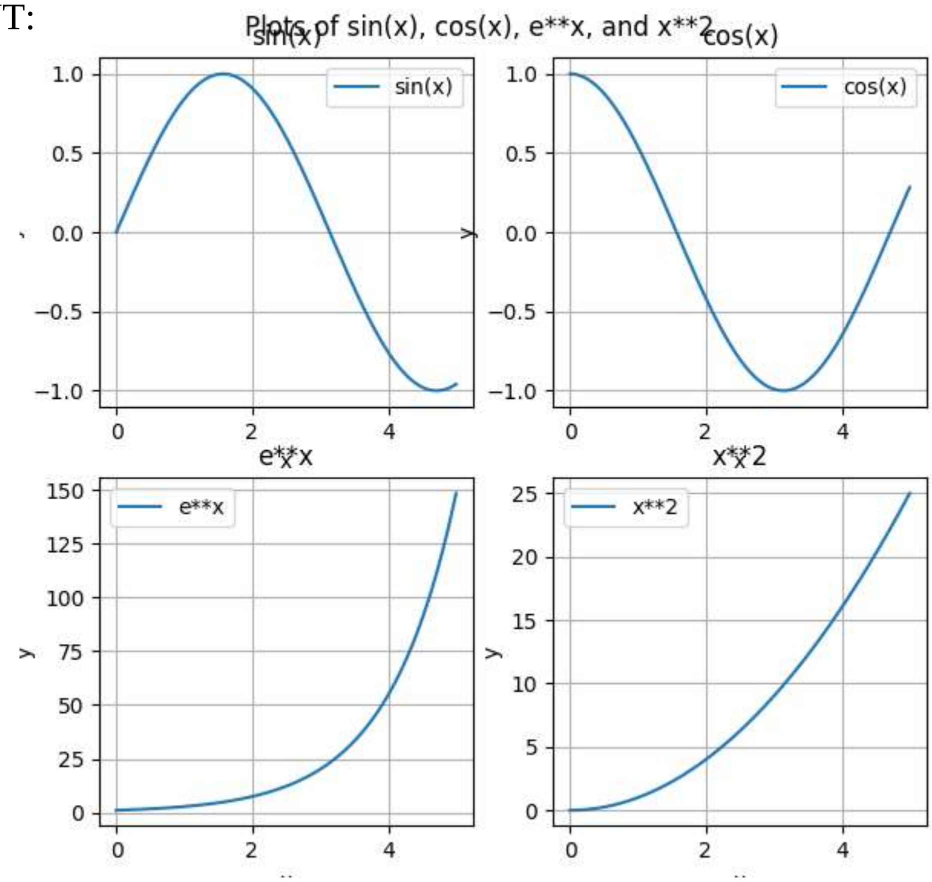
```
# Generate x values in the interval [0, 5]
x = np.linspace(0, 5, 500)
# Evaluate sin(x), cos(x), e**x, and x**2 for the x values
y1 = np.sin(x)
y2 = np.cos(x)
y3 = np.exp(x)
y4 = x**2
# Create a 2x2 subplot figure
fig, axs = plt.subplots(2, 2, figsize=(10, 10))
fig.suptitle('Plots of sin(x), cos(x), e**x, and x**2')
# Plot sin(x) in the top left subplot
axs[0, 0].plot(x, y1, label='sin(x)')
axs[0, 0].set_title('sin(x)')
# Plot cos(x) in the top right subplot
axs[0, 1].plot(x, y2, label='cos(x)')
axs[0, 1].set_title('cos(x)')
# Plot e**x in the bottom left subplot
axs[1, 0].plot(x, y3, label='e**x')
axs[1, 0].set_title('e**x')
# Plot x**2 in the bottom right subplot
axs[1, 1].plot(x, y4, label='x**2')
axs[1, 1].set_title('x**2')
# Add labels, legends, and grids to all subplots
for ax in axs.flat:
    ax.set_xlabel('x')
    ax.set_ylabel('y')
    ax.legend()
    ax.grid(True)
# Adjust spacing between subplots
```

fig.tight_layout()

Show the plot

plt.show()

OUTPUT:



Q.3) Write the python program to plot the graph of the function using def ()

$$f(x) = \begin{cases} x^2 + 4, if & -10 < x < 5 \\ 3x + 9, if & 5 < x \geq 0 \end{cases}$$

Syntax:

import numpy as np

import matplotlib.pyplot as plt

def f(x):

    """Function to define f(x)."""

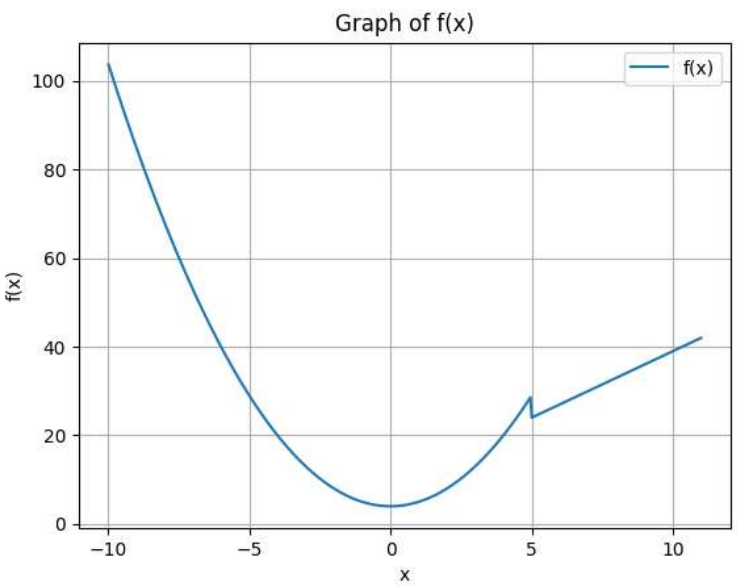
    if -10 < x < 5:

        return x**2 + 4

    elif 5 <= x:

```
        return 3*x + 9
    else:
        return None
# Generate x values
x = np.linspace(-11, 11, 500)  # Generate 500 points between -11 and 11
# Calculate y values using f(x)
y = np.array([f(xi) for xi in x])
# Create the plot
plt.plot(x, y, label='f(x)')
plt.xlabel('x')
plt.ylabel('f(x)')
plt.title('Graph of f(x)')
plt.legend()
plt.grid(True)
plt.show()
```

OUTPUT:



Q.4) write the Python program to rotate the triangle ABC by 180 degree, where A [2,1] B[2, -2] & C[-1, 2].

Syntax:

```
import numpy as np
# Define the original triangle vertices
A = np.array([2, 1])
B = np.array([2, -2])
C = np.array([-1, 2])
# Define the rotation matrix for 180 degrees
rotation_matrix = np.array([[-1, 0],[0, -1]])
# Rotate the triangle vertices using the rotation matrix
A_rotated = np.dot(rotation_matrix, A)
B_rotated = np.dot(rotation_matrix, B)
C_rotated = np.dot(rotation_matrix, C)
# Print the rotated triangle vertices
print("Original Triangle Vertices:")
print("A:", A)
print("B:", B)
print("C:", C)
print("Rotated Triangle Vertices:")
print("A Rotated:", A_rotated)
print("B Rotated:", B_rotated)
print("C Rotated:", C_rotated)
```

Output:

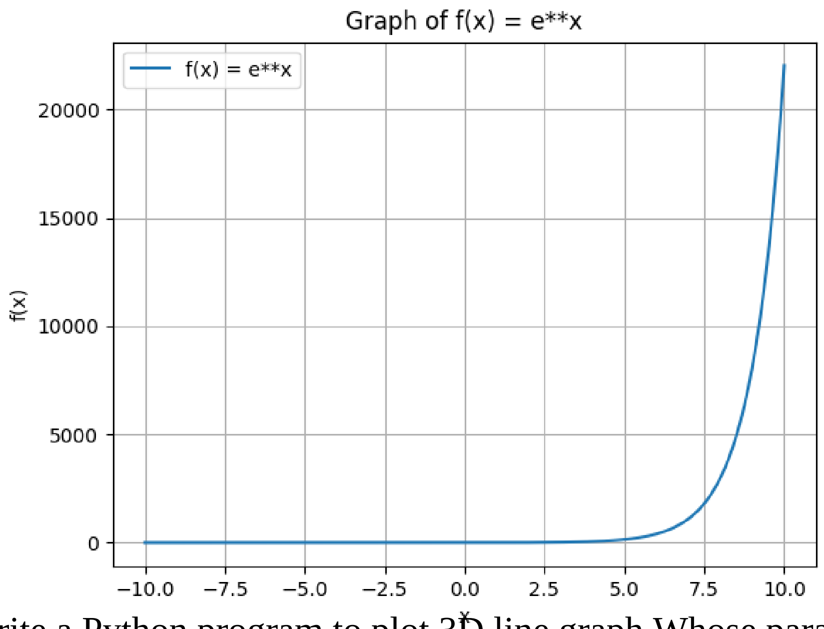
```
Original Triangle Vertices:
A: [2 1]
B: [ 2 -2]
C: [-1  2]
Rotated Triangle Vertices:
A Rotated: [-2 -1]
B Rotated: [-2  2]
C Rotated: [ 1 -2]
```

Q.5) Write the Python program to plot the graph of function f(x) = e**x in the interval [-10, 10].

Syntax:

```
import numpy as np
import matplotlib.pyplot as plt
# Define the function f(x) = e**x
def f(x):
    return np.exp(x)
# Generate x values in the interval [-10, 10]
x = np.linspace(-10, 10, 500)
# Evaluate f(x) for the x values
y = f(x)
# Create a plot
plt.plot(x, y, label='f(x) = e**x')
plt.xlabel('x')
plt.ylabel('f(x)')
plt.title('Graph of f(x) = e**x')
plt.legend()
plt.grid(True)
plt.show()
```

OUTPUT:



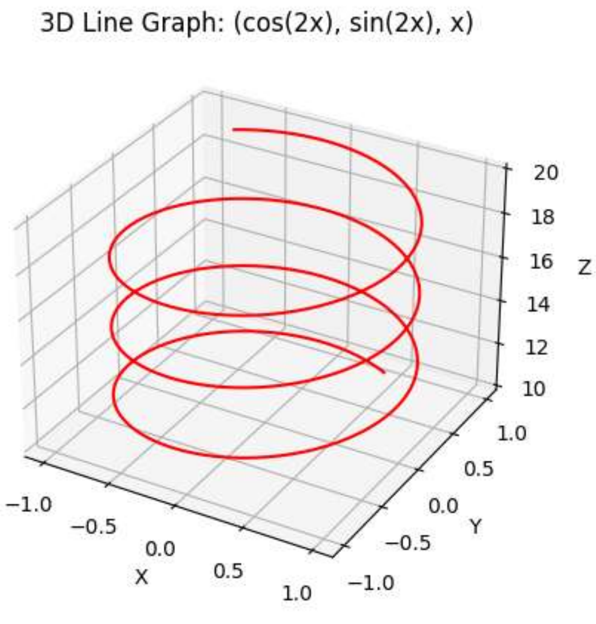
Q.6) Write a Python program to plot 3D line graph Whose parametric equation is (cos(2x),sin(2x),x) for 10 <= x <= 20 (in red color), with title of the graph

Syntax:

```
import numpy as np
```

```
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
# Generate values for x
x = np.linspace(10, 20, 500)
# Calculate parametric equations for x, y, z
y = np.sin(2 * x)
z = x
x = np.cos(2 * x)
# Create a 3D figure
fig = plt.figure()
ax = fig.add_subplot(111, projection='3d')
# Plot the 3D line graph
ax.plot(x, y, z, color='red')
# Set title for the graph
ax.set_title("3D Line Graph: (cos(2x), sin(2x), x)")
# Set labels for x, y, z axes
ax.set_xlabel('X')
ax.set_ylabel('Y')
ax.set_zlabel('Z')
# Show the plot
plt.show()
```

OUTPUT:



Q.7) write a Python program to solve the following LPP

Max Z = 3.5x + 2y

Subjected to

x + y >= 5

x >=4

y<=5

x >= 0,y>= 0.

Syntax:

```
from pulp import *
# Create the LP problem
problem = LpProblem("Maximize Z", LpMaximize)
# Define the decision variables
x = LpVariable('x', lowBound=0)  # x >= 0
y = LpVariable('y', lowBound=0)  # y >= 0
# Define the objective function
problem += 3.5 * x + 2 * y
# Define the constraints
problem += x + y >= 5
problem += x >= 4
problem += y <= 5
# Solve the LP problem
status = problem.solve()
# Check the solution status
if status == 1:
    # Print the optimal solution
    print("Optimal solution:")
    print(f"x = {value(x)}")
```

```
        print(f"y = {value(y)}")
        print(f"Z = {value(problem.objective)}")
    else:
        print("No feasible solution found.")
```

OUTPUT:

No feasible solution found.

Q.8) Write a python program to display the following LPP by using pulp module and simplex method. Find its optimal solution if exist.

Min Z = x+y
subject to
x => 6
y => 6
x + y <= 11
x=>0, y=>0
Syntax:

```
from pulp import *

# Create the LP problem as a minimization problem

problem = LpProblem("LPP", LpMinimize)

# Define the decision variables

x = LpVariable('x', lowBound=0, cat='Continuous')

y = LpVariable('y', lowBound=0, cat='Continuous')

# Define the objective function

problem += x + y, "Z"

# Define the constraints

problem += x >= 6, "Constraint1"

problem += y >= 6, "Constraint2"

problem += x + y <= 11, "Constraint3"

# Solve the LP problem using the simplex method

problem.solve(PULP_CBC_CMD(msg=False))

# Print the status of the solution
```

```
print("Status:", LpStatus[problem.status])
# If the problem has an optimal solution
if problem.status == LpStatusOptimal:
    # Print the optimal values of x and y
    print("Optimal x =", value(x))
    print("Optimal y =", value(y))
    # Print the optimal value of the objective function
    print("Optimal Z =", value(problem.objective))
```

OUTPUT:

Status: Optimal

Status: Infeasible

Q.9) Write a python program to find the combined transformation of the line segment between the points A[5,3] and B[1,4] for the following sequence of transformation

(I) First rotation about origin through an angle pi/e
(II) Followed by scaling in x co-ordinate by 5 units
(III) Followed by reflection through the line y = -x

Syntax:

```
import numpy as np

# Define points A and B as numpy arrays
A = np.array([5, 3])
B = np.array([1, 4])

# Print original points A and B
print("Original Points:")
print("Point A: ({}, {})".format(A[0], A[1]))
print("Point B: ({}, {})".format(B[0], B[1]))

# Transformation I: Rotation about origin through an angle of pi/2
theta = np.pi/2
rotation_matrix = np.array([[np.cos(theta), -np.sin(theta)],
                            [np.sin(theta), np.cos(theta)]])
```

```
A_rotation = np.dot(rotation_matrix, A)
B_rotation = np.dot(rotation_matrix, B)

# Print points A and B after rotation
print("\nPoints after Rotation:")
print("Point A: ({}, {})".format(A_rotation[0], A_rotation[1]))
print("Point B: ({}, {})".format(B_rotation[0], B_rotation[1]))
# Transformation II: Scaling in x-coordinate by 5 units
scaling_matrix = np.array([[5, 0],
                           [0, 1]])
A_scaling = np.dot(scaling_matrix, A_rotation)
B_scaling = np.dot(scaling_matrix, B_rotation)
# Print points A and B after scaling
print("\nPoints after Scaling:")
print("Point A: ({}, {})".format(A_scaling[0], A_scaling[1]))
print("Point B: ({}, {})".format(B_scaling[0], B_scaling[1]))
# Transformation III: Reflection through the line y = -x
reflection_matrix = np.array([[0, -1], [-1, 0]])
A_reflection = np.dot(reflection_matrix, A_scaling)
B_reflection = np.dot(reflection_matrix, B_scaling)
# Print points A and B after reflection
print("\nPoints after Reflection:")
print("Point A: ({}, {})".format(A_reflection[0], A_reflection[1]))
print("Point B: ({}, {})".format(B_reflection[0], B_reflection[1]))
```

OUTPUT:

```
Original Points:
Point A: (5, 3)
Point B: (1, 4)
Points after Rotation:
Point A: (-2.9999999999999996, 5.0)
Point B: (-4.0, 1.0000000000000002)
Points after Scaling:
Point A: (-14.999999999999998, 5.0)
Point B: (-20.0, 1.0000000000000002)
Points after Reflection:
Point A: (-5.0, 14.999999999999998)
Point B: (-1.0000000000000002, 20.0)
```

Q.10) Write the python program to apply each of the following transformation on the point P(-2,4)

(I) Reflection Through the line y = x+1

(II) Scaling in y-Coordinate by factor 1.5

(III) Shearing in x – Direction by 2 unit

(IV) Rotation about origin by an angle 45 degree.

Syntax:

```
import numpy as np
# Define the original point P
P = np.array([-2, 4])
# Print the original point P
print("Original Point:")
print("Point P: ({}, {})".format(P[0], P[1]))
# Transformation I: Reflection through the line y = x + 1
reflection_matrix = np.array([[0, 1], [1, 0]])
P_reflection = np.dot(reflection_matrix, P)
# Print the point P after reflection
print("\nPoint after Reflection:")
print("Point P: ({}, {})".format(P_reflection[0], P_reflection[1]))
# Transformation II: Scaling in y-coordinate by factor 1.5
scaling_matrix = np.array([[1, 0], [0, 1.5]])
P_scaling = np.dot(scaling_matrix, P_reflection)
# Print the point P after scaling
print("\nPoint after Scaling:")
print("Point P: ({}, {})".format(P_scaling[0], P_scaling[1]))
# Transformation III: Shearing in x-direction by 2 units
shearing_matrix = np.array([[1, 2], [0, 1]])
P_shearing = np.dot(shearing_matrix, P_scaling)
# Print the point P after shearing
print("\nPoint after Shearing:")
print("Point P: ({}, {})".format(P_shearing[0], P_shearing[1]))
# Transformation IV: Rotation about origin by an angle of 45 degrees
theta = np.deg2rad(45)
rotation_matrix = np.array([[np.cos(theta), -np.sin(theta)], [np.sin(theta), np.cos(theta)]])
P_rotation = np.dot(rotation_matrix, P_shearing)
# Print the point P after rotation
print("\nPoint after Rotation:")
print("Point P: ({}, {})".format(P_rotation[0], P_rotation[1]))
```

OUTPUT:
Original Point:
Point P: (-2, 4)
Point after Reflection:
Point P: (4, -2)
Point after Scaling:
Point P: (4.0, -3.0)
Point after Shearing:
Point P: (-2.0, -3.0)
Point after Rotation:
Point P: (0.7071067811865477, -3.5355339059327378)