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

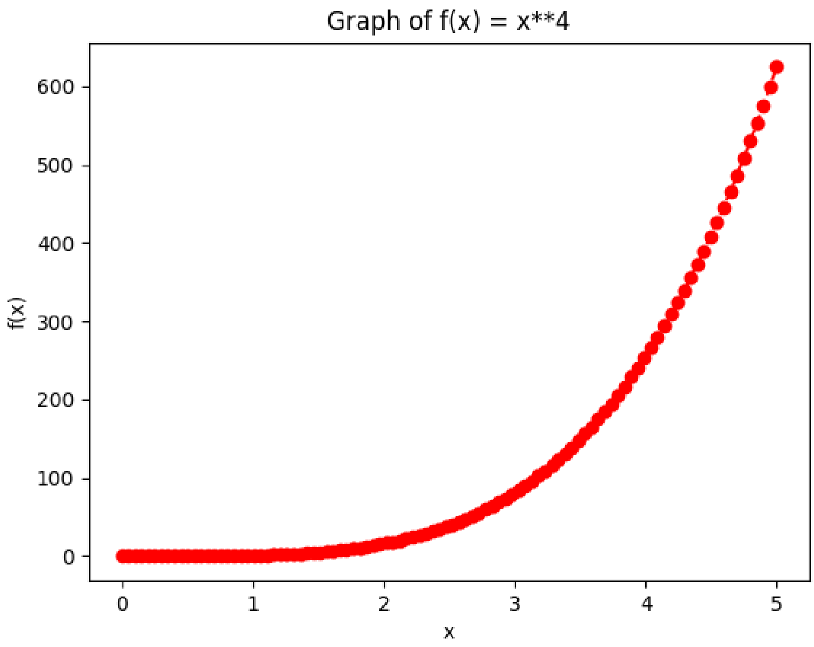
Title of the:- Practical 21 **Expt. No .** 21 **Class :-** S.Y.BCS

Q.1) Plot the graph of f(x) = x**4 in [0, 5] with red dashed line with circle markers.

Syntax:

```
import numpy as np
import matplotlib.pyplot as plt
# Define the function f(x) = x**4
def f(x):
    return x**4
# Generate x values in the interval [0, 5]
x = np.linspace(0, 5, 100)
# Generate y values using the function f(x)
y = f(x)
# Plot the graph with red dashed line and circle markers
plt.plot(x, y, 'r--o', markersize=6)
# Set x-axis label
plt.xlabel('x')
# Set y-axis label
plt.ylabel('f(x)')
# Set title
plt.title('Graph of f(x) = x**4')
# Show the plot
plt.show()
```

OUTPUT:



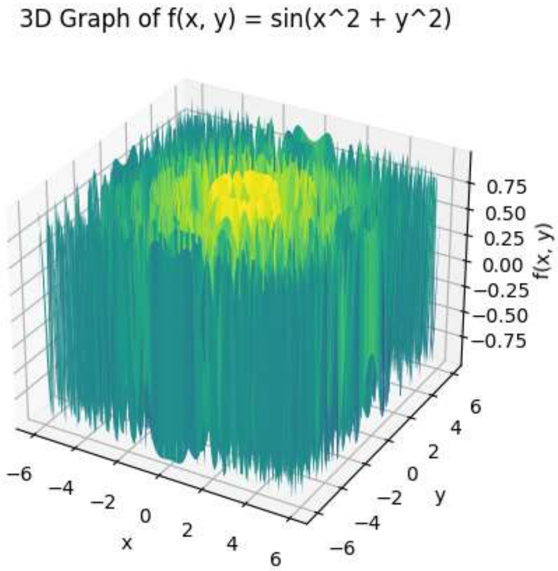
Q.2) Write a Python program to plot the 3D graph of the function f(x) = sin(x^2 + y^2), -6< x,y < 6.

Syntax:

```
import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
# Define the function f(x, y)
def f(x, y):
    return np.sin(x**2 + y**2)
# Generate x, y values in the range -6 to 6 with a step of 0.1
x = np.arange(-6, 6, 0.1)
y = np.arange(-6, 6, 0.1)
# Create a meshgrid from x, y values
X, Y = np.meshgrid(x, y)
# Calculate z values using the function f(x, y)
Z = f(X, Y)
# Create a 3D figure
```

```
fig = plt.figure()
ax = fig.add_subplot(111, projection='3d')
# Plot the 3D surface
ax.plot_surface(X, Y, Z, cmap='viridis')
# Set x, y, z axis labels
ax.set_xlabel('x')
ax.set_ylabel('y')
ax.set_zlabel('f(x, y)')
# Set the title of the graph
ax.set_title('3D Graph of f(x, y) = sin(x^2 + y^2)')
# Show the graph
plt.show()
```

OUTPUT:



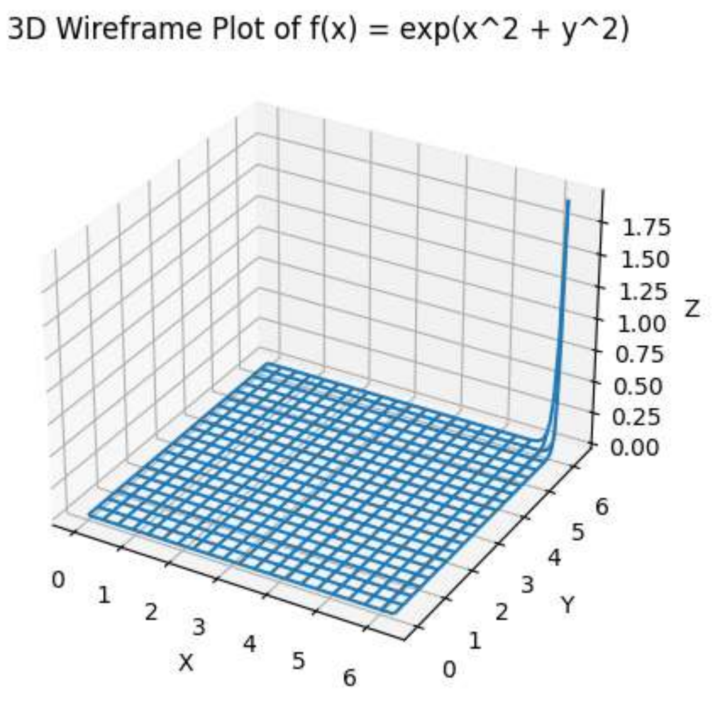
Q.3) Write a Python program to plot the 3D graph of the function f(x) = e(x^2+y^2) for x, y belongs [0, 2*pi] using wireframe.

Syntax:

```
import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
# Define the function f(x, y)
```

```
def f(x, y):
    return np.exp(x**2 + y**2)
# Generate x, y values
x = np.linspace(0, 2*np.pi, 100)
y = np.linspace(0, 2*np.pi, 100)
X, Y = np.meshgrid(x, y)
Z = f(X, Y)
# Create a 3D figure
fig = plt.figure()
ax = fig.add_subplot(111, projection='3d')
# Create a wireframe plot
ax.plot_wireframe(X, Y, Z, rstride=5, cstride=5)
# Set axis labels
ax.set_xlabel('X')
ax.set_ylabel('Y')
ax.set_zlabel('Z')
ax.set_title('3D Wireframe Plot of f(x) = exp(x^2 + y^2)')
# Show the plot
plt.show()
```

OUTPUT:



Q.4) if the line segment joining the points A[2,5] and [4,-13] is transformed to the line segment A'B' by the transformation matrix [T] = $\begin{matrix} 2 & 3 \\ 4 & 1 \end{matrix}$ the using python find the slope and midpoint of the transformed line.

Syntax:

```
import numpy as np
# Define the original line segment points A and B
A = np.array([2, 5])
B = np.array([4, -13])
# Define the transformation matrix [T]
T = np.array([[2, 3], [4, 1]])
# Apply the transformation matrix [T] to points A and B
A_transformed = np.dot(T, A)
B_transformed = np.dot(T, B)
# Calculate the slope of the transformed line
slope_transformed = (B_transformed[1] - A_transformed[1]) / (B_transformed[0] - A_transformed[0])
# Calculate the midpoint of the transformed line
midpoint_transformed = (A_transformed + B_transformed) / 2
# Print the slope and midpoint of the transformed line
print("Slope of the transformed line: ", slope_transformed)
print("Midpoint of the transformed line: ", midpoint_transformed)
```

Output:

```
Slope of the transformed line:  0.2
Midpoint of the transformed line:  [-6.  8.]
```

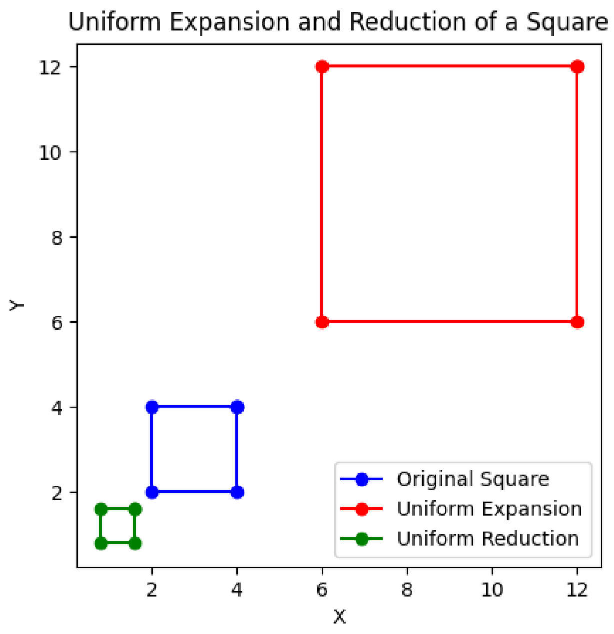
Q.5) Write a python program to plot square with vertices at [4, 4] [2, 4], [2, 2], [4, 2] and find its uniform expansion by factor 3, uniform reduction by factor 0.4.

Syntax:

```
import numpy as np
import matplotlib.pyplot as plt
# Define the vertices of the original square
vertices = np.array([[4, 4], [2, 4], [2, 2], [4, 2], [4, 4]])
```

```
# Create a figure and axis
fig, ax = plt.subplots()
# Plot the original square
ax.plot(vertices[:, 0], vertices[:, 1], 'b-o', label='Original Square')
# Define the uniform expansion and reduction factors
expansion_factor = 3
reduction_factor = 0.4
# Perform uniform expansion
expanded_vertices = vertices * expansion_factor
# Perform uniform reduction
reduced_vertices = vertices * reduction_factor
# Plot the expanded and reduced squares
ax.plot(expanded_vertices[:, 0], expanded_vertices[:, 1], 'r-o', label='Uniform Expansion')
ax.plot(reduced_vertices[:, 0], reduced_vertices[:, 1], 'g-o', label='Uniform Reduction')
# Set axis labels and title
ax.set_xlabel('X')
ax.set_ylabel('Y')
ax.set_title('Uniform Expansion and Reduction of a Square')
# Set legend
ax.legend()
# Set aspect ratio to 'equal' for a square plot
ax.set_aspect('equal')
# Show the plot
plt.show()
```

OUTPUT:



Q.6) write a Python program to find the equation of the transformed line if shearing is applied on the line 2x + y = 3 in x and y direction by 2 and -3 units respectively.

```
import numpy as np

# Define the original line equation

original_line = np.array([2, 1, -3])  # Coefficients of x, y, and constant term

# Define the shear transformation matrices in x and y directions

shear_matrix_x = np.array([[1, 2, 0],

                           [0, 1, 0],

                           [0, 0, 1]])

shear_matrix_y = np.array([[1, 0, 0],

                           [-3, 1, 0],

                           [0, 0, 1]])

# Apply shear transformations to the original line

transformed_line_x = np.dot(shear_matrix_x, original_line)

transformed_line_y = np.dot(shear_matrix_y, original_line)

# Extract the coefficients of x, y, and constant term from the transformed lines

a_x, b_x, c_x = transformed_line_x

a_y, b_y, c_y = transformed_line_y

# Print the equations of the transformed lines

print("Equation of the transformed line after x-direction shear: {}x + {}y = {}".format(a_x, b_x, c_x))

print("Equation of the transformed line after y-direction shear: {}x + {}y = {}".format(a_y, b_y, c_y))
```

OUTPUT:

Equation of the transformed line after x-direction shear: 4x + 1y = -3

Equation of the transformed line after y-direction shear: 2x + -5y = -3

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

Max Z = 4x + 2y
Subjected to
x + y <= 5
x - y >= 2
y <= 2
x > 0 , y > 0
Syntax:

```
from pulp import *

# Create a maximization problem

prob = LpProblem("Maximize Z", LpMaximize)

# Define the decision variables

x = LpVariable("x", lowBound=0, cat='Continuous')  # x >= 0

y = LpVariable("y", lowBound=0, cat='Continuous')  # y >= 0

# Define the objective function

prob += 4 * x + 2 * y, "Z"

# Define the constraints

prob += x + y <= 5, "Constraint 1"

prob += x - y >= 2, "Constraint 2"

prob += y <= 2, "Constraint 3"

# Solve the problem

prob.solve()

# Print the solution status

print("Solution Status: {}".format(LpStatus[prob.status]))

# Print the optimal values of the decision variables

print("Optimal Solution:")

print("x = {}".format(value(x)))

print("y = {}".format(value(y)))

# Print the optimal value of the objective function
```

```
print("Z = {}".format(value(prob.objective)))
```

OUTPUT:

```
Solution Status: Optimal
Optimal Solution:
x = 5.0
y = 0.0
Z = 20.0
```

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 = 2x+4y
subject to
2x + 2y >= 30
x + 2y = 26
x>= 0, y>= 0

Syntax:

```
from pulp import *
# Create a maximization problem
prob = LpProblem("Minimize Z", LpMinimize)
# Define the decision variables
x = LpVariable("x", lowBound=0, cat='Continuous')  # x >= 0
y = LpVariable("y", lowBound=0, cat='Continuous')  # y >= 0
# Define the objective function
obj_func = 2 * x + 4 * y
prob += obj_func
# Define the constraints
constr1 = 2 * x + 2 * y >= 30
constr2 = x + 2 * y == 26
prob += constr1
prob += constr2
# Solve the problem using the simplex method
solver = getSolver('PULP_CBC_CMD')
solver.actualSolve(prob)
# Print the solution status
print("Solution Status: {}".format(LpStatus[prob.status]))
# If the problem has an optimal solution, print the optimal values of the decision variables and the objective function
if prob.status == LpStatusOptimal:
```

```
    print("Optimal Solution:")
    print("x = {}".format(value(x)))
    print("y = {}".format(value(y)))
    print("Z = {}".format(value(obj_func)))
```

OUTPUT:
Solution Status: Optimal
Optimal Solution:
x = 4.0
y = 11.0
Z = 52.0

Q.9) Apply Python. Program in each of the following transformation on the point P[-2,4]
(I) Reflection through line 3x + 4y = 5
(II) Scaling in X coordinate by factor 6.
(III) Scaling in Y coordinate by factor 4.1
(IV) Reflection through the line y = 2x + 3
Syntax:

```
P = [-2, 4]
print("Original Point P: {}".format(P))
A = 3
B = 4
C = -5
# Compute the reflected point
Px_reflect = P[0] - 2 * (A * P[0] + B * P[1] + C) / (A**2 + B**2)
Py_reflect = P[1] - 2 * (A * P[1] - B * P[0] + C) / (A**2 + B**2)
P_reflect = [Px_reflect, Py_reflect]
print("Reflection through line 3x + 4y = 5: {}".format(P_reflect))
# Transformation (II): Scaling in X coordinate by factor 6
scale_factor_x = 6
Px_scaled_x = P[0] * scale_factor_x
Py_scaled_x = P[1]
P_scaled_x = [Px_scaled_x, Py_scaled_x]
print("Scaling in X coordinate by factor 6: {}".format(P_scaled_x))
# Transformation (III): Scaling in Y coordinate by factor 4.1
scale_factor_y = 4.1
Px_scaled_y = P[0]
Py_scaled_y = P[1] * scale_factor_y
P_scaled_y = [Px_scaled_y, Py_scaled_y]
```

```
print("Scaling in Y coordinate by factor 4.1: {}".format(P_scaled_y))
A = 2
B = -1
C = -3
# Compute the reflected point
Px_reflect_y = P[0]
Py_reflect_y = P[1] - 2 * (A * P[0] + B * P[1] + C) / (A**2 + B**2)
P_reflect_y = [Px_reflect_y, Py_reflect_y]
print("Reflection through line y = 2x + 3: {}".format(P_reflect_y))
```

OUTPUT:
Original Point P: [-2, 4]
Reflection through line 3x + 4y = 5: [-2.4, 2.8]
Scaling in X coordinate by factor 6: [-12, 4]
Scaling in Y coordinate by factor 4.1: [-2, 16.4]
Reflection through line y = 2x + 3: [-2, 8.4]

Q.10) Apply the following transformation on the point P[-2,4]
(I) Shearing in Y direction by 7 units.
(II) Scaling in X and Y direction by 4 and 7 units respectively
(III) Rotation about origin by an angle 48 degree.
(IV) Reflection through the line y = x

Syntax:

```
import math
# Point P
P = [-2, 4]
print("Original Point P: {}".format(P))
# Transformation (I): Shearing in Y direction by 7 units
shear_factor_y = 7
Px_shear_y = P[0]
Py_shear_y = P[1] + shear_factor_y * P[0]
P_shear_y = [Px_shear_y, Py_shear_y]
print("Shearing in Y direction by 7 units: {}".format(P_shear_y))
# Transformation (II): Scaling in X and Y direction by 4 and 7 units respectively
scale_factor_x = 4
scale_factor_y = 7
Px_scaled_xy = P[0] * scale_factor_x
Py_scaled_xy = P[1] * scale_factor_y
P_scaled_xy = [Px_scaled_xy, Py_scaled_xy]
print("Scaling in X and Y direction by 4 and 7 units respectively: {}".format(P_scaled_xy))
# Transformation (III): Rotation about origin by an angle of 48 degrees
```

```
angle_degrees = 48
angle_radians = math.radians(angle_degrees)
Px_rotate = P[0] * math.cos(angle_radians) - P[1] * math.sin(angle_radians)
Py_rotate = P[0] * math.sin(angle_radians) + P[1] * math.cos(angle_radians)
P_rotate = [Px_rotate, Py_rotate]
print("Rotation about origin by an angle of 48 degrees: {}".format(P_rotate))
# Transformation (IV): Reflection through the line y = x
Px_reflect = P[1]
Py_reflect = P[0]
P_reflect = [Px_reflect, Py_reflect]
print("Reflection through the line y = x: {}".format(P_reflect))
```

OUTPUT:

```
Original Point P: [-2, 4]
Shearing in Y direction by 7 units: [-2, -10]
Scaling in X and Y direction by 4 and 7 units respectively: [-8, 28]
Rotation about origin by an angle of 48 degrees: [-4.3108405146272935, 1.1902327744806445]
Reflection through the line y = x: [4, -2]
```