EXP 1:

AIM: To implement fuzzy control/ inference system

ALGORITHM:

1. **Define fuzzy sets**: For the input and output variables.
2. **Define fuzzy rules**: These rules determine how the input fuzzy sets combine to produce the output.
3. **Fuzzification**: Convert the crisp input values into fuzzy values based on membership functions.
4. **Apply fuzzy inference**: Use the fuzzy rules to infer output fuzzy values based on the fuzzified inputs.
5. **Defuzzification**: Convert the fuzzy output back into a crisp value for the control system.

CODE:

import numpy as np

import skfuzzy as fuzz

from skfuzzy import control as ctrl

# Step 1: Define fuzzy variables

temperature = ctrl.Antecedent(np.arange(0, 41, 1), 'temperature') # Input variable

fan\_speed = ctrl.Consequent(np.arange(0, 101, 1), 'fan\_speed') # Output variable

# Step 2: Define fuzzy membership functions

temperature['cold'] = fuzz.trapmf(temperature.universe, [0, 0, 15, 20])

temperature['warm'] = fuzz.trimf(temperature.universe, [15, 25, 35])

temperature['hot'] = fuzz.trapmf(temperature.universe, [30, 35, 40, 40])

fan\_speed['low'] = fuzz.trimf(fan\_speed.universe, [0, 0, 50])

fan\_speed['medium'] = fuzz.trimf(fan\_speed.universe, [20, 50, 80])

fan\_speed['high'] = fuzz.trimf(fan\_speed.universe, [50, 100, 100])

# Step 3: Define fuzzy rules

rule1 = ctrl.Rule(temperature['cold'], fan\_speed['low'])

rule2 = ctrl.Rule(temperature['warm'], fan\_speed['medium'])

rule3 = ctrl.Rule(temperature['hot'], fan\_speed['high'])

# Step 4: Control system creation

fan\_speed\_ctrl = ctrl.ControlSystem([rule1, rule2, rule3])

fan\_speed\_system = ctrl.ControlSystemSimulation(fan\_speed\_ctrl)

# Step 5: Provide input and compute output

fan\_speed\_system.input['temperature'] = 28 # Input temperature

fan\_speed\_system.compute()

# Display the output

print(f"Fan speed: {fan\_speed\_system.output['fan\_speed']:.2f}%")

OUTPUT:



EXP 2

AIM: The aim of this exercise is to implement a simple **discrete perceptron** for classification tasks

ALGORITHM:

1. **Initialize weights**: Set random initial weights for each input feature and a bias.
2. **Activation function**: Use a step function (e.g., sign function) to make a classification decision based on the weighted sum of inputs.
3. **Training**:
   1. For each training sample, calculate the predicted output.
   2. If the predicted output differs from the actual output, adjust the weights based on the error.
4. **Repeat**: Continue adjusting the weights over several iterations (epochs) until the model converges (i.e., makes correct classifications).

CODE:

import numpy as np

# Step function (activation function)

def step\_function(x):

return 1 if x >= 0 else 0

# Perceptron training

def perceptron\_train(X, y, epochs=10, learning\_rate=0.1):

weights = np.zeros(X.shape[1]) # Initialize weights (including bias)

for \_ in range(epochs):

for i in range(len(X)):

prediction = step\_function(np.dot(X[i], weights))

error = y[i] - prediction

weights += learning\_rate \* error \* X[i] # Update weights

return weights

# Example dataset (X: features, y: labels)

X = np.array([[0, 0], [0, 1], [1, 0], [1, 1]]) # XOR inputs

y = np.array([0, 0, 0, 1]) # XOR output

# Train the perceptron

weights = perceptron\_train(X, y)

# Test the perceptron

def perceptron\_predict(X, weights):

return [step\_function(np.dot(x, weights)) for x in X]

predictions = perceptron\_predict(X, weights)

print("Predictions:", predictions)

OUTPUT:



RESULT:

EXP 3:

AIM: Implementation of XOR with backpropagation algorithm

ALGORITHM:

1. **Initialize the network**: Define a neural network with an input layer, a hidden layer, and an output layer. Initialize random weights and biases.
2. **Forward pass**: For each input, calculate the activations for each layer starting from the input layer and moving through the hidden layer to the output layer.
3. **Compute loss**: Calculate the loss (e.g., Mean Squared Error) between the predicted output and the true label.
4. **Backpropagation**:
   1. Compute the error for the output layer.
   2. Propagate the error backward through the network, updating the weights and biases using the gradient descent algorithm.
5. **Repeat**: Continue the forward pass and backpropagation for multiple epochs until the model converges.

CODE:

import numpy as np

# Sigmoid activation and its derivative

def sigmoid(x):

return 1 / (1 + np.exp(-x))

def sigmoid\_derivative(x):

return x \* (1 - x)

# XOR input and output

X = np.array([[0, 0], [0, 1], [1, 0], [1, 1]]) # Inputs

y = np.array([[0], [1], [1], [0]]) # XOR output

# Initialize weights and biases

weights\_input\_hidden = np.random.uniform(size=(2, 4)) # 2 inputs, 4 hidden neurons

weights\_hidden\_output = np.random.uniform(size=(4, 1)) # 4 hidden neurons, 1 output

bias\_hidden = np.random.uniform(size=(1, 4))

bias\_output = np.random.uniform(size=(1, 1))

# Training loop

for epoch in range(10000):

# Forward pass

hidden\_input = X.dot(weights\_input\_hidden) + bias\_hidden

hidden\_output = sigmoid(hidden\_input)

output\_input = hidden\_output.dot(weights\_hidden\_output) + bias\_output

predicted\_output = sigmoid(output\_input)

# Backpropagation

error = y - predicted\_output

output\_gradient = error \* sigmoid\_derivative(predicted\_output)

hidden\_gradient = output\_gradient.dot(weights\_hidden\_output.T) \* sigmoid\_derivative(hidden\_output)

# Update weights and biases

weights\_hidden\_output += hidden\_output.T.dot(output\_gradient) \* 0.1

weights\_input\_hidden += X.T.dot(hidden\_gradient) \* 0.1

bias\_output += np.sum(output\_gradient, axis=0, keepdims=True) \* 0.1

bias\_hidden += np.sum(hidden\_gradient, axis=0, keepdims=True) \* 0.1

# Print loss every 1000 epochs

if epoch % 1000 == 0:

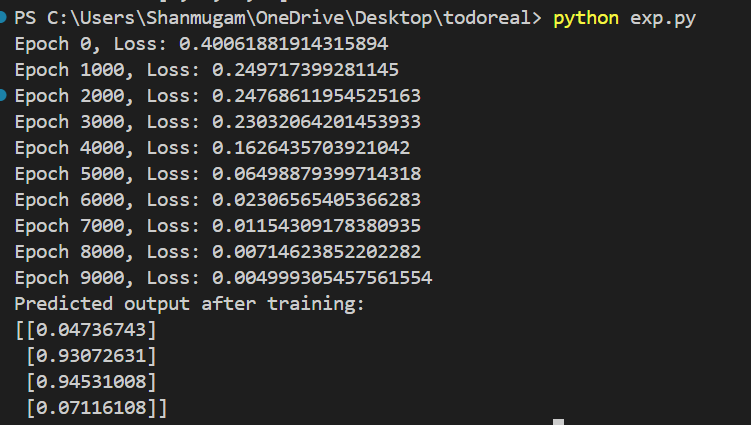
print(f"Epoch {epoch}, Loss: {np.mean(error\*\*2)}")

# Test the network

print("Predicted output after training:")

print(predicted\_output)

OUTPUT:



RESULT:

EXP 4

AIM : Implementation of self organizing maps for a specific application

ALGORITHM :

1. **Initialize weights**: Set random weights for each neuron in the SOM grid. The weights should have the same dimensions as the input features.
2. **Training process**:
   1. For each input, find the Best Matching Unit (BMU), i.e., the neuron whose weight vector is closest to the input vector.
   2. Update the BMU and its neighbors’ weights towards the input vector using a learning rate and neighborhood function.
3. **Repeat**: Iterate over the dataset for multiple epochs, gradually decreasing the learning rate and neighborhood size.
4. **Result**: After training, the SOM grid will cluster the data points based on the similarity of their features.

CODE:

import numpy as np

from sklearn.datasets import load\_iris

import matplotlib.pyplot as plt

# Load Iris dataset

iris = load\_iris()

X = iris.data

# SOM Parameters

grid\_size = 10 # Grid size (10x10)

epochs = 1000 # Number of epochs

learning\_rate = 0.1 # Initial learning rate

radius = grid\_size / 2 # Initial neighborhood radius

# Initialize weights randomly

weights = np.random.rand(grid\_size, grid\_size, X.shape[1])

# Distance function (Euclidean distance)

def distance(a, b):

return np.linalg.norm(a - b)

# Find Best Matching Unit (BMU)

def find\_bmu(input\_vector):

distances = np.linalg.norm(weights - input\_vector, axis=2)

return np.unravel\_index(np.argmin(distances), distances.shape)

# Update weights

def update\_weights(input\_vector, bmu\_idx, learning\_rate, radius):

for i in range(grid\_size):

for j in range(grid\_size):

dist\_to\_bmu = np.sqrt((i - bmu\_idx[0]) \*\* 2 + (j - bmu\_idx[1]) \*\* 2)

if dist\_to\_bmu < radius:

influence = np.exp(-dist\_to\_bmu / (2 \* (radius \*\* 2)))

weights[i, j] += learning\_rate \* influence \* (input\_vector - weights[i, j])

# Train the SOM

for epoch in range(epochs):

for input\_vector in X:

bmu\_idx = find\_bmu(input\_vector)

update\_weights(input\_vector, bmu\_idx, learning\_rate, radius)

# Decay learning rate and radius

learning\_rate \*= 0.99

radius \*= 0.99

# Visualization of the SOM

plt.figure(figsize=(8, 6))

for i in range(grid\_size):

for j in range(grid\_size):

neuron = weights[i, j]

distances\_to\_neuron = np.linalg.norm(X - neuron, axis=1)

bmu\_label = np.argmin(distances\_to\_neuron)

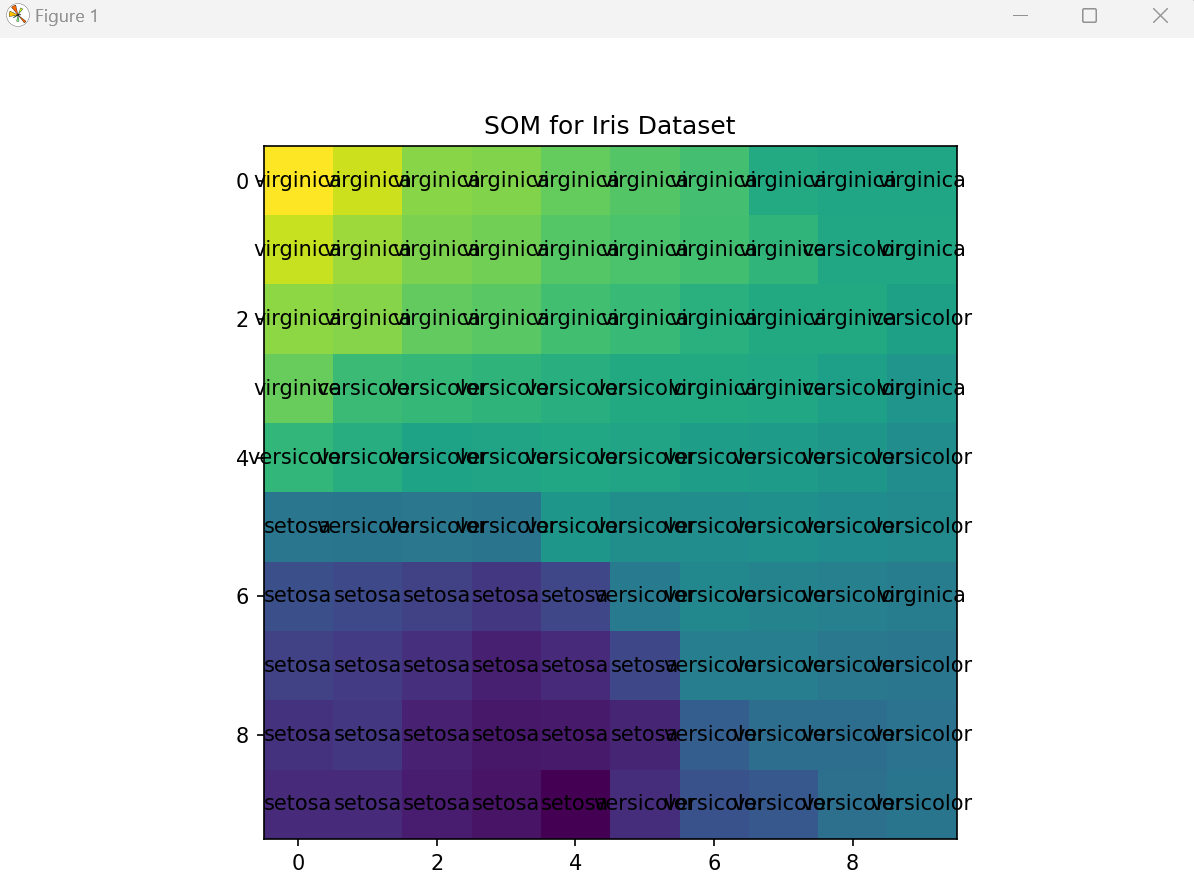
plt.text(j, i, str(iris.target\_names[iris.target[bmu\_label]]), ha='center', va='center')

plt.imshow(np.sum(weights, axis=2), cmap='viridis')

plt.title("SOM for Iris Dataset")

plt.show()

OUTPUT:



RESULT:

EXP 5:

AIM : Programming exercises on maximizing a function using Genetic algorithm.

ALGORITHM :

1. **Initialization**: Generate a random population of candidate solutions (chromosomes).
2. **Fitness Function**: Evaluate the fitness of each candidate solution using the Rastrigin function.
3. **Selection**: Select individuals from the population based on their fitness using a method like **Roulette Wheel Selection** or **Tournament Selection**.
4. **Crossover**: Apply a crossover operator (e.g., single-point crossover) to combine pairs of individuals and create offspring.
5. **Mutation**: Apply a mutation operator (randomly altering bits in the offspring).
6. **Replacement**: Replace the old population with the new generation.
7. **Repeat**: Repeat steps 2-6 for a set number of generations or until convergence.

CODE :

import numpy as np

import matplotlib.pyplot as plt

# Rastrigin function

def rastrigin(x):

return 10 \* len(x) + np.sum(x\*\*2 - 10 \* np.cos(2 \* np.pi \* x))

# Genetic algorithm

def genetic\_algorithm(pop\_size, n\_genes, generations):

population = np.random.uniform(-5.12, 5.12, (pop\_size, n\_genes))

best\_solution, best\_fitness = None, float('inf')

for \_ in range(generations):

fitness = np.apply\_along\_axis(rastrigin, 1, population)

best\_idx = np.argmin(fitness)

if fitness[best\_idx] < best\_fitness:

best\_fitness, best\_solution = fitness[best\_idx], population[best\_idx]

# Create next generation

next\_gen = [np.concatenate((population[np.random.randint(pop\_size)][:n\_genes//2], population[np.random.randint(pop\_size)][n\_genes//2:])) for \_ in range(pop\_size)]

population = np.array(next\_gen)

return best\_solution, best\_fitness

# Run and display results

best\_solution, best\_fitness = genetic\_algorithm(50, 5, 100)

print("Best Solution:", best\_solution)

print("Best Fitness:", best\_fitness)

# Plot

x = np.linspace(-5.12, 5.12, 100)

plt.plot(x, rastrigin(x))

plt.scatter(best\_solution, rastrigin(best\_solution), color='red')

plt.title("Maximizing Rastrigin with Genetic Algorithm")

plt.show()

OUTPUT:

EXP 6

AIM: The aim of this implementation is to compute and visualize the sine function for two inputs, where the result is the sine of the sum of the two inputs xxx and yyy.

ALGORITHM:

1. **Define the Function**:
   1. Define a function that takes two inputs xxx and yyy and returns the sine of their sum, i.e., sin⁡(x+y)\sin(x + y)sin(x+y).
2. **Generate Input Values**:
   1. Create two arrays of input values for xxx and yyy that range from −2π-2\pi−2π to 2π2\pi2π, each having 400 evenly spaced values.
3. **Create a Mesh Grid**:
   1. Use np.meshgrid() to create a 2D grid of xxx and yyy values from the generated arrays. This is necessary for evaluating the function over a grid of points.
4. **Evaluate the Function**:
   1. Apply the sine function to the grid values of xxx and yyy by computing sin⁡(x+y)\sin(x + y)sin(x+y) for each pair of values.
5. **Visualize the Result**:
   1. Use Matplotlib to create a 3D surface plot of the computed values. The plot will display how the sine function behaves for the two inputs across the 2D grid

CODE:

import numpy as np

import matplotlib.pyplot as plt

# Two-input sine function

def two\_input\_sine(x, y):

return np.sin(x + y)

# Create a grid of values for x and y

x = np.linspace(-2 \* np.pi, 2 \* np.pi, 400)

y = np.linspace(-2 \* np.pi, 2 \* np.pi, 400)

X, Y = np.meshgrid(x, y) # Create a mesh grid

Z = two\_input\_sine(X, Y) # Calculate Z = sin(X + Y)

# Plotting the 3D surface

fig = plt.figure(figsize=(8, 6))

ax = fig.add\_subplot(111, projection='3d')

ax.plot\_surface(X, Y, Z, cmap='viridis')

# Labels

ax.set\_xlabel('X-axis')

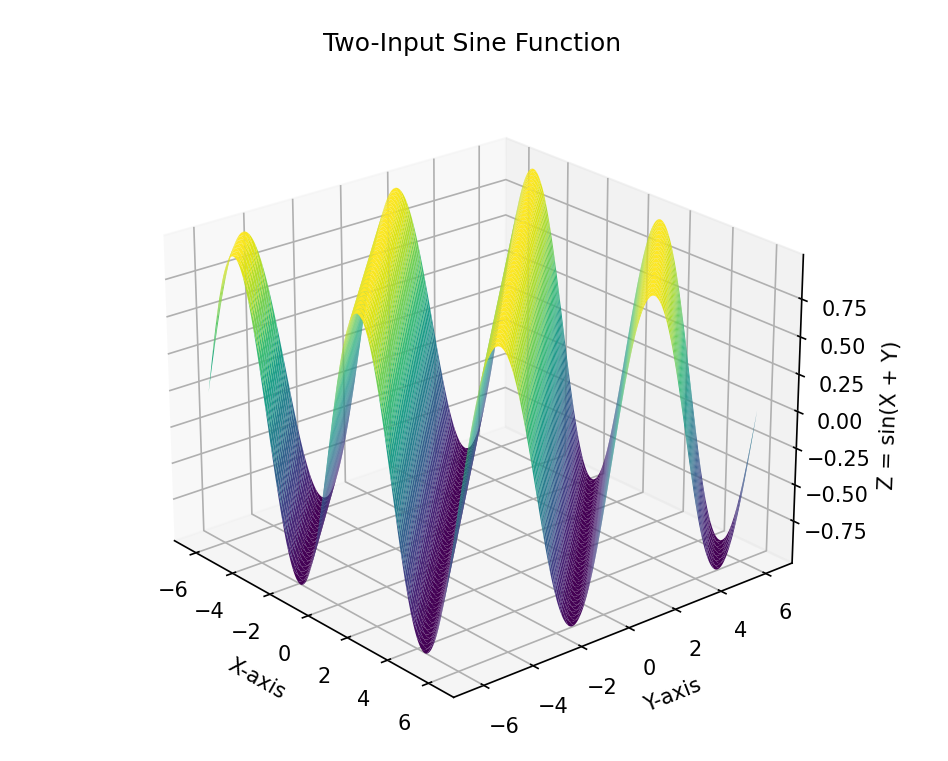
ax.set\_ylabel('Y-axis')

ax.set\_zlabel('Z = sin(X + Y)')

plt.title("Two-Input Sine Function")

plt.show()

OUTPUT:



RESULT:

EXP 7

AIM: The aim of this implementation is to compute and visualize a three-input nonlinear function.

ALGORITHM:

1. **Define the Function**:
   1. Define a function that takes three inputs xxx, yyy, and zzz, and returns the sine of the sum of these inputs, i.e., sin⁡(x+y+z)\sin(x + y + z)sin(x+y+z).
2. **Generate Input Values**:
   1. Create three arrays of input values for xxx, yyy, and zzz that range from −2π-2\pi−2π to 2π2\pi2π, each with a specific number of evenly spaced values.
3. **Create a Grid**:
   1. Use np.meshgrid() to create a 3D grid of values for xxx, yyy, and zzz. This is necessary to evaluate the function over a grid of points in 3D space.
4. **Evaluate the Function**:
   1. Compute the value of the function sin⁡(x+y+z)\sin(x + y + z)sin(x+y+z) for each combination of xxx, yyy, and zzz from the grid.
5. **Visualize the Result**:
   1. Visualizing a 3D surface with three inputs is challenging, but we can slice the data at different values of one of the inputs (e.g., zzz) and plot the resulting 2D surfaces.

CODE:

import numpy as np

import matplotlib.pyplot as plt

# Aim: To compute and visualize a three-input nonlinear function, f(x, y, z) = sin(x + y + z).

# Step 1: Define the three-input nonlinear function

def three\_input\_function(x, y, z):

return np.sin(x + y + z)

# Step 2: Generate input values for x, y, and z in the range [-2π, 2π]

x = np.linspace(-2 \* np.pi, 2 \* np.pi, 50)

y = np.linspace(-2 \* np.pi, 2 \* np.pi, 50)

z = np.linspace(-2 \* np.pi, 2 \* np.pi, 50)

# Step 3: Create mesh grid for X, Y, and Z values

X, Y, Z = np.meshgrid(x, y, z)

# Step 4: Compute the output W = sin(X + Y + Z)

W = three\_input\_function(X, Y, Z)

# Step 5: Visualize the result by plotting slices of the 3D function

# We will plot the function for a fixed value of z (e.g., z = 0)

z\_index = np.argmin(np.abs(z - 0)) # Find the index of z = 0

W\_slice = W[:, :, z\_index] # Slice of W when z = 0

# Plot the 2D slice of the 3D function

plt.figure(figsize=(8, 6))

plt.contourf(X[:, :, 0], Y[:, :, 0], W\_slice, cmap='viridis')

plt.colorbar(label="f(x, y, z=0) = sin(x + y + z)")

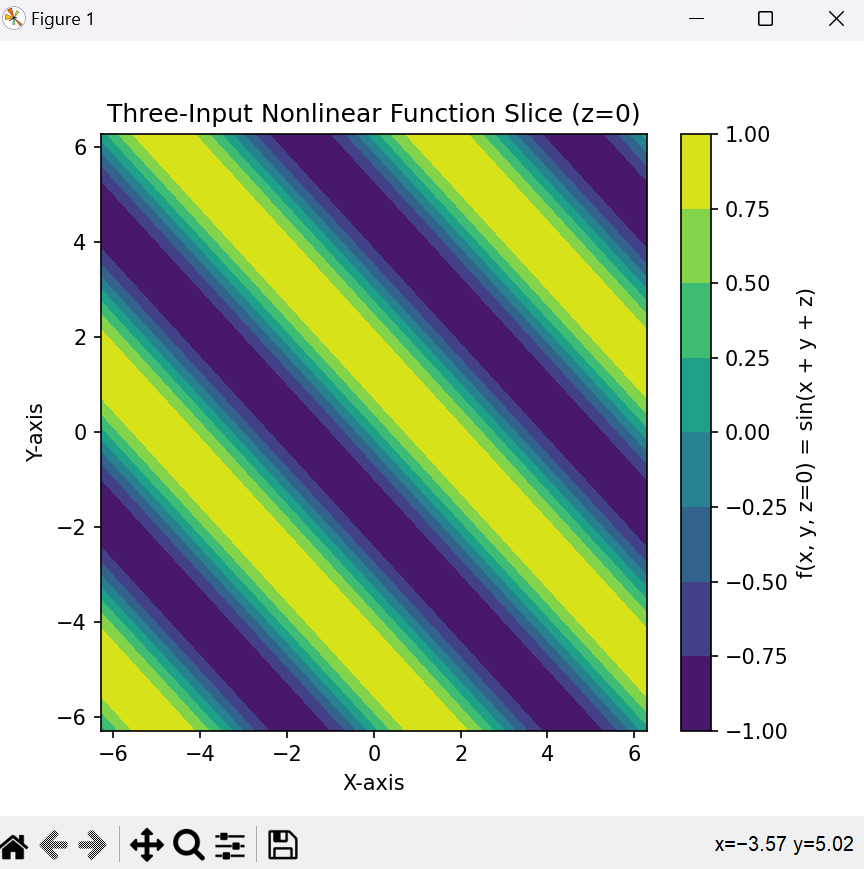
plt.xlabel('X-axis')

plt.ylabel('Y-axis')

plt.title("Three-Input Nonlinear Function Slice (z=0)")

plt.show()

OUTPUT:



RESULT: