AKBAR PEERBHOY COLLEGE OF COMMERCE & ECONOMICS

M. S. Ali Road, Do Taaki, Grant Road (E), Mumbai – 400008.

PRACTICAL JOURNAL

SUBMITTED TO

MASTER OF SCIENCE (INFORMATION TECHNOLOGY) Part-I

Semester – I

Subject: Soft Computing Techniques

University of Mumbai

Submitted by

Mohammad Arbaz

Under the guidance of

Prof. Taskeen Ansari

DEPARTMENT OF INFORMATION TECHNOLOGY

AKBAR PEERBHOY COLLEGE OF COMMERCE & ECONOMICS

Academic Year (2025 – 2026)

Anjuman-I-Islam's
AKBAR PEERBHOY COLLEGE OF COMMERCE & ECONOMICS
NAAC Reaccredited College • Affiliated to University of Mumbai
Maulana Shaukatali Road, Do Taaki, Grant Road (E), Mumbai - 400008.
Tel.: Office: 23074122 Principal: 23083405
E-mail : apcce_college@yahoo.co.in Website : www.apcollege.in

Ref. No. _____

CERTIFICATE

This is to Certify that Mr. / Mrs. Miss_____

Seat No. _____of **M.Sc. IT Part-I, Semester I** has completed the practical work in the subject _____ during the academic year **2025-26** being the partial requirement for the fulfillment of curriculum of Degree in Master of Science in Information Technology, University of Mumbai.

_____
Subject In charge

_____
Coordinator
Prof. Dr. Abdul Sadique

_____
Asst. Director, Professional Course
Prof. Dr. Hanif Lakdawala

_____
Principal
Prof. Dr. Shaukat Ali

_____
External Examiner

College Seal:

Date: _____

INDEX

Sr.No.	Practical Aim	Date	Signature
1	Implement the following: **a.** Design a simple linear neural network model. **b.** Calculate the output of neural net using both binary and bipolar sigmoidal function.		
2	Implement the following: a. Generate AND/NOT function using McCulloch-Pitts neural net. **b.** Generate XOR function using McCulloch-Pitts neural net.		
3	Implement the Following a. Write a program to implement Hebb's rule. b. Write a program to implement of delta rule.		
4	Implement the Following a. Write a program for Back Propagation Algorithm b. Write a program for error Backpropagation algorithm.		
5	Implement the Following a. Write a program for Hopfield Network. b. Write a program for Radial Basis function		
6	Implement the Following a. Kohonen Self organizing map b. Adaptive resonance theory		
7	Implement the Following a. Write a program for Linear separation.		
8	Implement the Following a. Membership and Identity Operators in, not in, b. Membership and Identity Operators is, is not		
9	Implement the Following a. Find ratios using fuzzy logic b. Solve Tipping problem using fuzzy logic		
10	Implement the Following a. Implementation of Simple genetic algorithm b. Create two classes: City and Fitness using Genetic algorithm		

Practical 1

Practical 1 a: Design a simple linear neural network model.

Code:

```
# Simple Linear Neural Network (Single Neuron)
# Take inputs
x = float(input("Enter value of x: "))
w = float(input("Enter value of weight w: "))
b = float(input("Enter value of bias b: "))
# Net input
net = w * x + b
# Activation function (piecewise linear)
if net < 0:
 out = 0
elif 0 <= net <= 1:
 out = net # linear region
else:
 out = 1
# Print results
print("Net input =", net)
print("Output =", out)
```

Output:

```
Enter value of x: 1
Enter value of weight w: 0.5
Enter value of bias b: 0.2
Net input = 0.7
Output = 0.7
```

Practical 1 b: Calculate the output of neural net using both binary and bipolar sigmoidal function

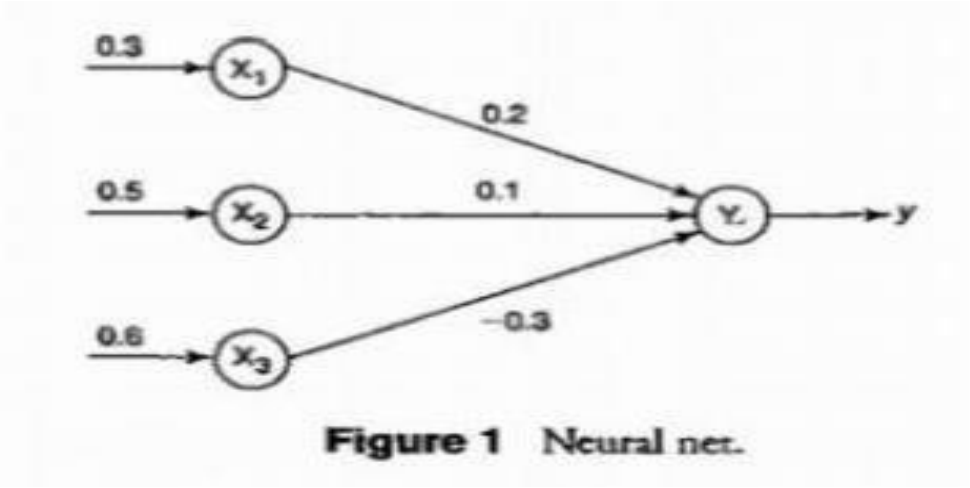


Figure 1 Neural net.

Code:

```
import math

# Number of inputs
n = int(input("Enter number of elements : "))

# Input values
print("Enter the inputs")
inputs = []
for i in range(n):
    ele = float(input())
    inputs.append(ele)

# Weight values
print("Enter the weights")
weights = []
for i in range(n):
    ele = float(input())
    weights.append(ele)

# Bias
b = float(input("Enter bias value: "))

# Net input Yin = Σ (xi * wi) + b
Yin = sum(inputs[i] * weights[i] for i in range(n)) + b
print("\nNet input (Yin) =", round(Yin, 3))

# Binary Sigmoid (0 to 1)
def binary_sigmoid(x):
    return 1 / (1 + math.exp(-x))

# Bipolar Sigmoid (-1 to +1)
def bipolar_sigmoid(x):
    return (1 - math.exp(-x)) / (1 + math.exp(-x))

# Outputs
print("Binary Sigmoid Output  =", round(binary_sigmoid(Yin), 3))
print("Bipolar Sigmoid Output =", round(bipolar_sigmoid(Yin), 3))
```

Output:

```
Enter number of elements : 3

Enter the inputs
0.3
0.5
0.6
```

Enter the weights
0.2
0.1
-0.3
Enter bias value: 0.1
Net input (Yin) = 0.03
Binary Sigmoid Output = 0.507
Bipolar Sigmoid Output = 0.015

Practical 2

Practical 2 a: Implement AND/NOT function using McCulloch-Pits neuron (use binary data representation).

Code:

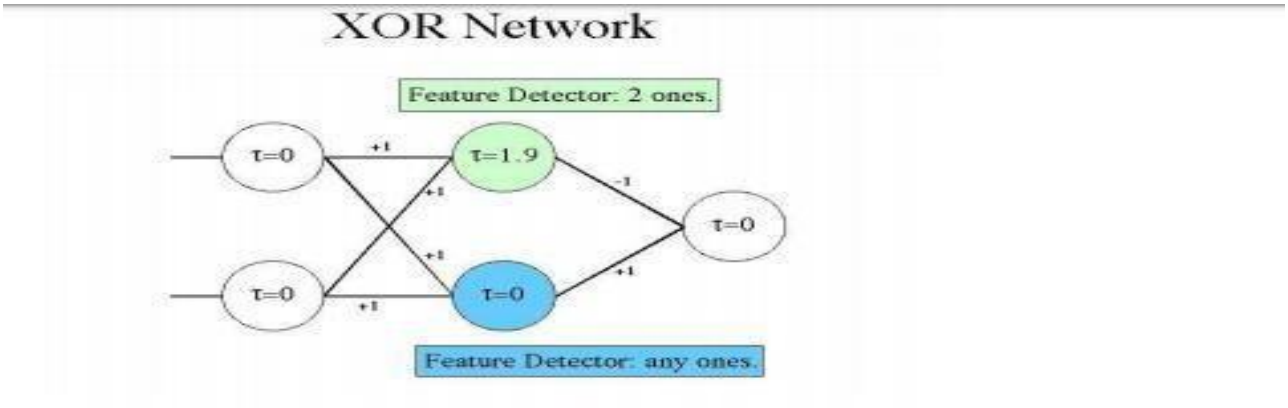
```
# enter the no of inputs
num_ip = int(input("Enter the number of inputs : "))

# Set the weights with value 1
w1 = 1
w2 = 1

print("For the", num_ip, "inputs calculate the net input using yin = x1w1 + x2w2")

x1 = []
x2 = []
for j in range(0, num_ip):
    ele1 = int(input("x1 = "))
    ele2 = int(input("x2 = "))
    x1.append(ele1)
    x2.append(ele2)

print("x1 = ", x1)
print("x2 = ", x2)

n = [val * w1 for val in x1]
m = [val * w2 for val in x2]

Yin = []
for i in range(0, num_ip):
    Yin.append(n[i] + m[i])
print("Yin = ", Yin)

# Assume one weight as excitatory and the other as inhibitory
Yin = []
for i in range(0, num_ip):
    Yin.append(n[i] - m[i])
print("After assuming one weight as excitatory and the other as inhibitory Yin = ", Yin)

# From the calculated net inputs, fire the neuron using threshold θ ≥ 1
Y = []
for i in range(0, num_ip):
    if Yin[i] >= 1:
        ele = 1
        Y.append(ele)
```

```
    if Yin[i] < 1:
        ele = 0
        Y.append(ele)
print("Y = ", Y)
```

Output:

```
Enter the number of inputs : 4
For the 4 inputs calculate the net input using yin = x1w1 + x2w2
x1 = 0
x2 = 0
x1 = 0
x2 = 1
x1 = 1
x2 = 0
x1 = 1
x2 = 1
x1 =  [0, 0, 1, 1]
x2 =  [0, 1, 0, 1]
Yin =  [0, 1, 1, 2]
After assuming one weight as excitatory and the other as inhibitory Yin =  [0, -1, 1, 0]
Y =  [0, 0, 1, 0]
```

Practical 2 b: Generate XOR function using McCulloch-Pitts neural net



The XOR (exclusive or) function is defined by the following truth table:

Input1	Input2	XOR Output
0	0	0
0	1	1
1	0	1
1	1	0

Code:

```
# XOR using McCulloch-Pitts Neural Net

import numpy as np
```

```
# Inputs (XOR truth table)
x1 = np.array([0, 0, 1, 1])
x2 = np.array([0, 1, 0, 1])
expected = np.array([0, 1, 1, 0])  # XOR outputs
# --- McCulloch-Pitts Network Design ---
# Hidden neurons
# z1 = x1 AND (NOT x2)
# z2 = (NOT x1) AND x2
# Output y = z1 OR z2
def mcp_xor(x1, x2, theta=1):
    # Hidden layer
    z1 = np.array([1 if (a == 1 and b == 0) else 0 for a, b in zip(x1, x2)])
    z2 = np.array([1 if (a == 0 and b == 1) else 0 for a, b in zip(x1, x2)])
    # Output neuron
    y = np.array([1 if (z1[i] + z2[i]) >= theta else 0 for i in range(len(z1))])
    return z1, z2, y
# Run the network
z1, z2, y = mcp_xor(x1, x2)
print("Inputs:")
print("x1:", x1)
print("x2:", x2)
print("\nHidden Neurons:")
print("z1:", z1)
print("z2:", z2)
print("\nOutput of Net (XOR):")
print("y :", y)
print("Expected:", expected)
# Check correctness
```

```
if np.array_equal(y, expected):
    print("\n✅ XOR function generated successfully using McCulloch-Pitts Neural Net")
else:
    print("\n❌ Net did not learn XOR")
```

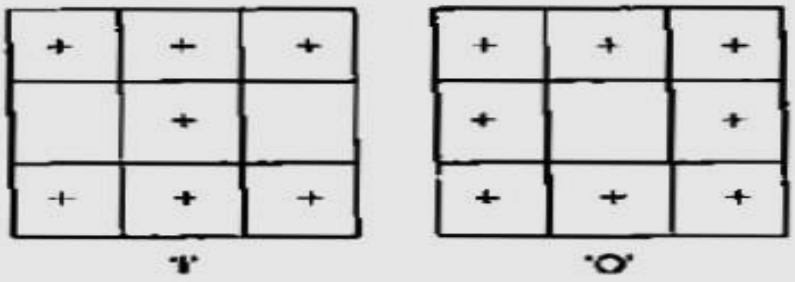
Output:

```
Inputs:
x1: [0 0 1 1]
x2: [0 1 0 1]

Hidden Neurons:
z1: [0 0 1 0]
z2: [0 1 0 0]

Output of Net (XOR):
y : [0 1 1 0]
Expected: [0 1 1 0]

✅ XOR function generated successfully using McCulloch-Pitts Neural Net
```

Practical 3

Practical 3 a: Write a program to implement Hebb's rule

Using the Hebb rule, find the weights required to perform the following classifications of the given input patterns shown in Figure 16. The pattern is shown as 3 × 3 matrix form in the squares. The "+" symbols represent the value "1" and empty squares indicate "−1." Consider "I" belongs to the members of class (so has target value 1) and "O" does not belong to the members of class (so has target value −1).



Code:

```
import numpy as np

# first pattern
x1 = np.array([1, 1, 1, -1, 1, -1, 1, 1, 1])

# second pattern
x2 = np.array([1, 1, 1, 1, -1, 1, 1, 1, 1])

# initialize bias value
b = 0

# define target
y = np.array([1, -1])

wtold = np.zeros((9,))
wtnew = np.zeros((9,))

wtnew = wtnew.astype(int)
wtold = wtold.astype(int)

print("First input with target = 1")
for i in range(0, 9):
    wtold[i] = wtold[i] + x1[i] * y[0]
    wtnew = wtold.copy()
```

```
b = b + y[0]
print("new wt =", wtnew)
print("Bias value =", b)

print("\nSecond input with target = -1")
for i in range(0, 9):
    wtnew[i] = wtold[i] + x2[i] * y[1]

b = b + y[1]
print("new wt =", wtnew)
print("Bias value =", b)
```

Output:

```
First input with target =1
new wt = [ 1  1  1 -1  1 -1  1  1  1]
Bias value 1
Second input with target =-1
new wt = [ 0  0  0 -2  2 -2  0  0  0]
Bias value 0
```

Practical 3 b: Write a program to implement of delta rule

Code:

```
# supervised learning
import numpy as np
import time

np.set_printoptions(precision=2)

# initialize arrays
x = np.zeros((3,))
weights = np.zeros((3,))
desired = np.zeros((3,))
actual = np.zeros((3,))

# input initial values
for i in range(0, 3):
    x[i] = float(input("Initial input x[" + str(i) + "]: "))

for i in range(0, 3):
    weights[i] = float(input("Initial weight w[" + str(i) + "]: "))

for i in range(0, 3):
```

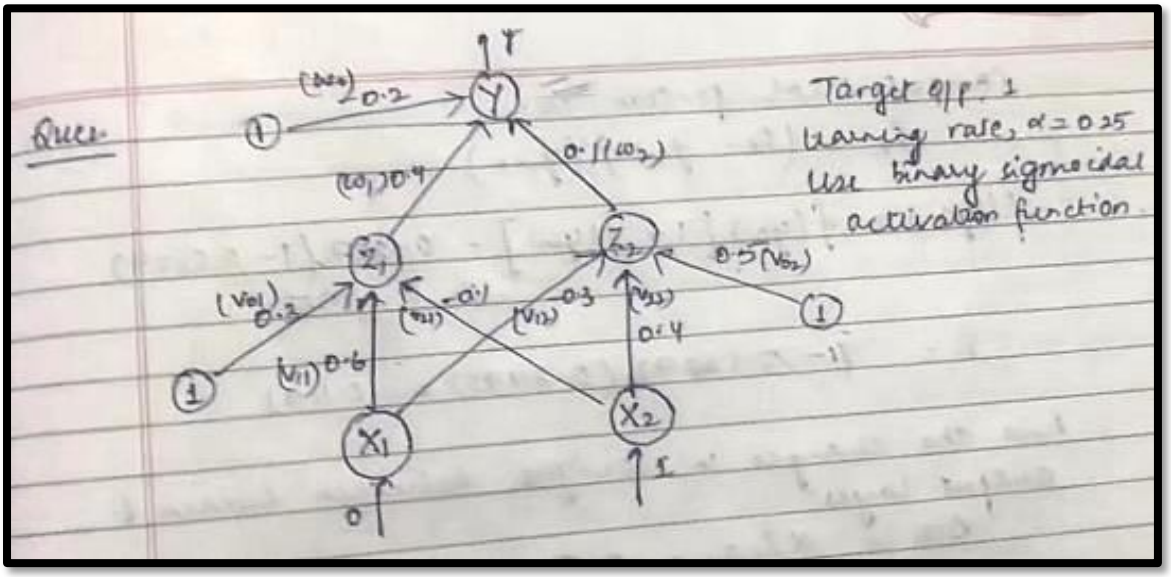
```
    desired[i] = float(input("Desired output d[" + str(i) + "]: "))

a = float(input("Enter learning rate: "))

# calculate initial actual output
actual = x * weights
print("Initial actual:", actual)
print("Desired:", desired)

# learning loop
while True:
    if np.array_equal(desired, actual):
        break  # no change, stop learning
    else:
        for i in range(0, 3):
            weights[i] = weights[i] + a * (desired[i] - actual[i])

        actual = x * weights
        print("Updated weights:", weights)
        print("Updated actual:", actual)
        print("Desired:", desired)
        print("*" * 30)
        time.sleep(1)

print("\nFinal output")
print("Corrected weights:", weights)
print("Final actual:", actual)
print("Desired:", desired)
```

Output:

```
Initial input x[0]: 1
Initial input x[1]: 1
Initial input x[2]: 1
Initial weight w[0]: 1
Initial weight w[1]: 1
Initial weight w[2]: 1
Desired output d[0]: 2
Desired output d[1]: 3
Desired output d[2]: 4
Enter learning rate: 1
Initial actual: [1. 1. 1.]
Desired: [2. 3. 4.]
Updated weights: [2. 3. 4.]
Updated actual: [2. 3. 4.]
Desired: [2. 3. 4.]
******************************

Final output
Corrected weights: [2. 3. 4.]
Final actual: [2. 3. 4.]
Desired: [2. 3. 4.]
```

Practical 4

Practical 4 a: Write a program for Back Propagation Algorithm



Code:

```
import numpy as np
import math
np.set_printoptions(precision=4)

# Initial weights and biases
v1 = np.array([0.6, 0.3])
v2 = np.array([-0.1, 0.4])
w = np.array([-0.2, 0.4, 0.1])
b1 = 0.3
b2 = 0.5
x1 = 0
x2 = 1
alpha = 0.25

print("Calculate net input to z1 layer")
zin1 = b1 + x1 * v1[0] + x2 * v2[0]
print("z1 net =", round(zin1, 4))

print("Calculate net input to z2 layer")
zin2 = b2 + x1 * v1[1] + x2 * v2[1]
print("z2 net =", round(zin2, 4))

print("\nApply activation function (Sigmoid) to calculate outputs")
z1 = 1 / (1 + math.exp(-zin1))
z2 = 1 / (1 + math.exp(-zin2))
print("z1 =", round(z1, 4))
```

```
print("z2 =", round(z2, 4))

print("\nCalculate net input to output layer")
yin = w[0] + z1 * w[1] + z2 * w[2]
print("yin =", round(yin, 4))

print("\nCalculate net output")
y = 1 / (1 + math.exp(-yin))
print("y =", round(y, 4))

# Derivatives
fyin = y * (1 - y)
dk = (1 - y) * fyin
print("\ndk =", round(dk, 6))

# Weight updates for output layer
dw1 = alpha * dk * z1
dw2 = alpha * dk * z2
dw0 = alpha * dk
print("\nWeight changes (hidden → output):")
print("dw1 =", round(dw1, 6))
print("dw2 =", round(dw2, 6))
print("dw0 =", round(dw0, 6))

# Error portion in delta (hidden layer)
din1 = dk * w[1]
din2 = dk * w[2]
print("\ndin1 =", round(din1, 6))
print("din2 =", round(din2, 6))

print("\nError in delta (hidden layer)")
fzin1 = z1 * (1 - z1)
fzin2 = z2 * (1 - z2)
d1 = din1 * fzin1
d2 = din2 * fzin2
print("fzin1 =", round(fzin1, 6), " d1 =", round(d1, 6))
print("fzin2 =", round(fzin2, 6), " d2 =", round(d2, 6))

# Weight updates for input → hidden layer
dv11 = alpha * d1 * x1
dv21 = alpha * d1 * x2
dv01 = alpha * d1
dv12 = alpha * d2 * x1
dv22 = alpha * d2 * x2
dv02 = alpha * d2

print("\nWeight changes (input → hidden):")
print("dv11 =", round(dv11, 6))
print("dv21 =", round(dv21, 6))
print("dv01 =", round(dv01, 6))
```

```
print("dv12 =", round(dv12, 6))
print("dv22 =", round(dv22, 6))
print("dv02 =", round(dv02, 6))

# Final weights update
v1[0] += dv11
v1[1] += dv12
v2[0] += dv21
v2[1] += dv22
w[1] += dw1
w[2] += dw2
w[0] += dw0
b1 += dv01
b2 += dv02

print("\nFinal updated weights of network:")
print("v1 =", v1)
print("v2 =", v2)
print("w =", w)
print("bias b1 =", b1, " b2 =", b2)
```

Output:

```
Calculate net input to z1 layer
z1 net = 0.2
Calculate net input to z2 layer
z2 net = 0.9

Apply activation function (Sigmoid) to calculate outputs
z1 = 0.5498
z2 = 0.7109

Calculate net input to output layer
yin = 0.091

Calculate net output
y = 0.5227

dk = 0.119068

Weight changes (hidden → output):
dw1 = 0.016367
dw2 = 0.021163
dw0 = 0.029767

din1 = 0.047627
din2 = 0.011907

Error in delta (hidden layer)
fzin1 = 0.247517  d1 = 0.011789
fzin2 = 0.2055  d2 = 0.002447

Weight changes (input → hidden):
dv11 = 0.0
dv21 = 0.002947
dv01 = 0.002947
dv12 = 0.0
dv22 = 0.000612
dv02 = 0.000612
```

```
Final updated weights of network:
v1 = [0.6 0.3]
v2 = [-0.0971  0.4006]
w = [-0.1702  0.4164  0.1212]
bias b1 = 0.30294712576780886  b2 = 0.5006117118183497
```

Practical 4 b: Write a Program for Error Back Propagation Algorithm (Ebpa) Learning

Code:

```
import math

a0 = -1

t = -1

# Inputs

w10 = float(input("Enter weight first network: "))

b10 = float(input("Enter bias first network: "))

w20 = float(input("Enter weight second network: "))

b20 = float(input("Enter bias second network: "))

c = float(input("Enter learning coefficient: "))

# Forward pass

n1 = float(w10 * a0 + b10)

a1 = math.tanh(n1)

n2 = float(w20 * a1 + b20)

a2 = math.tanh(n2)

# Error

e = t - a2

# Backpropagation

s2 = -2 * (1 - a2 * a2) * e

s1 = (1 - a1 * a1) * w20 * s2

# Weight updates
```

```
w21 = w20 - (c * s2 * a1)

w11 = w10 - (c * s1 * a0)

# Bias updates

b21 = b20 - (c * s2)

b11 = b10 - (c * s1)

# Results

print("The updated weight of first n/w (w11) =", w11)

print("The updated weight of second n/w (w21) =", w21)

print("The updated bias of first n/w (b11) =", b11)

print("The updated bias of second n/w (b21) =", b21)
```

Output:

```
Enter weight first network: 12
Enter bias first network: 35
Enter weight second network: 23
Enter bias second network: 45
Enter learning coefficient: 11
The updated weight of first n/w (w11) = 12.0
The updated weight of second n/w (w21) = 23.0
The updated bias of first n/w (b11) = 35.0
The updated bias of second n/w (b21) = 45.0
```

Practical 5

Practical 5 a: Write a program for Hopfield Network.

Code:

```
import numpy as np

# Threshold function
def threshold(x):
    return np.where(x >= 0, 1, 0)

class HopfieldNetwork:
    def __init__(self, weights):
        self.weights = np.array(weights)  # weight matrix (4x4)

    def activate(self, pattern):
        pattern = np.array(pattern)
        print("\nActivating with pattern:", pattern)

        # calculate activations = W * X
        activations = self.weights.dot(pattern)
        print("Activations:", activations)

        # apply threshold
        output = threshold(activations)
        print("Output:", output)

        return output


# ---------- MAIN ----------
if __name__== "__main__":
    # patterns to test
    pattern1 = [1, 0, 1, 0]
    pattern2 = [0, 1, 0, 1]

    # weight matrix
    weights = [
        [0, -3,  3, -3],
        [-3, 0, -3,  3],
        [3, -3,  0, -3],
        [-3, 3, -3,  0]
    ]

    print("Hopfield Network with 4 neurons \n")

    # create network
```

```
net = HopfieldNetwork(weights)

# test pattern 1
out1 = net.activate(pattern1)
print("Expected:", pattern1, "Got:", out1.tolist())

# test pattern 2
out2 = net.activate(pattern2)
print("Expected:", pattern2, "Got:", out2.tolist())
```

Output:

```
Hopfield Network with 4 neurons


Activating with pattern: [1 0 1 0]
Activations: [ 3 -6  3 -6]
Output: [1 0 1 0]
Expected: [1, 0, 1, 0] Got: [1, 0, 1, 0]

Activating with pattern: [0 1 0 1]
Activations: [-6  3 -6  3]
Output: [0 1 0 1]
Expected: [0, 1, 0, 1] Got: [0, 1, 0, 1]
```

Practical 5 b: Write a program for Radial Basis function

Code:

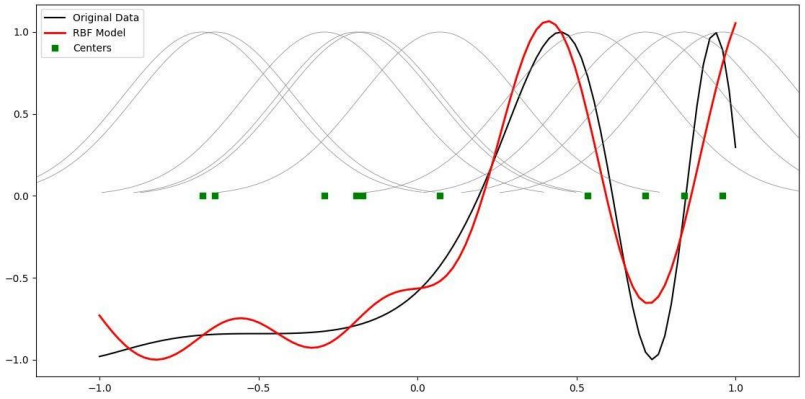
```
import numpy as np

from numpy.linalg import norm, pinv

from matplotlib import pyplot as plt

class RBF:

    def __init__(self, indim, numCenters, outdim):

        self.indim = indim

        self.outdim = outdim

        self.numCenters = numCenters

        self.centers = [np.random.uniform(-1, 1, indim) for _ in range(numCenters)]

        self.beta = 8

        self.W = np.random.random((self.numCenters, self.outdim))
```

```
def _basisfunc(self, c, d):
    assert len(d) == self.indim
    return np.exp(-self.beta * norm(c - d) ** 2)
def _calcAct(self, X):
    """Calculate activations of RBFs"""
    G = np.zeros((X.shape[0], self.numCenters), float)
    for ci, c in enumerate(self.centers):
        for xi, x in enumerate(X):
            G[xi, ci] = self._basisfunc(c, x)
    return G
def train(self, X, Y):
    """ X: matrix of dimensions n x indim
        Y: column vector of dimension n x outdim """
    # choose random center vectors from training set
    rnd_idx = np.random.permutation(X.shape[0])[:self.numCenters]
    self.centers = [X[i, :] for i in rnd_idx]
    print("Centers:", self.centers)
    # calculate activations of RBFs
    G = self._calcAct(X)
    print("Activation matrix G:\n", G)
    # calculate output weights (pseudoinverse)
    self.W = np.dot(pinv(G), Y)
def test(self, X):
    """ X: matrix of dimensions n x indim """
    G = self._calcAct(X)
    Y = np.dot(G, self.W)
```

```
        return Y
# - - - - - - - - - - - MAIN - - - - - - - - - - -
if __name___== "__main__":
    # 1D Example
    n = 100
    x = np.mgrid[-1:1:complex(0, n)].reshape(n, 1)
    # set y
    y = np.sin(3 * (x + 0.5) ** 3 - 1)
    # RBF regression
    rbf = RBF(1, 10, 1)
    rbf.train(x, y)
    z = rbf.test(x)
    # plot original data
    plt.figure(figsize=(12, 8))
    plt.plot(x, y, 'k-', label="Original Data")
    # plot learned model
    plt.plot(x, z, 'r-', linewidth=2, label="RBF Model")
    # plot RBF centers
    plt.plot([c[0] for c in rbf.centers], np.zeros(rbf.numCenters), 'gs', label="Centers")
    # plot RBF basis functions
    for c in rbf.centers:
        cx = np.arange(c[0] - 0.7, c[0] + 0.7, 0.01)
        cy = [rbf._basisfunc(np.array([cx_]), c) for cx_ in cx]
        plt.plot(cx, cy, '-', color='gray', linewidth=0.5)
    plt.xlim(-1.2, 1.2)
    plt.legend()
```

```
plt.show()
```

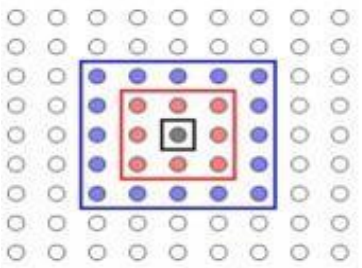
Output:



Practical 6

Practical 6 a: Self-Organizing Maps

The SOM algorithm is used to compress the information to produce a similarity graph while preserving the topologic relationship of the input data space. The basic SOM model construction algorithm can be interpreted as follows: 1) Create and initialize a matrix (weight vector) randomly to hold the neurons. If the matrix can be initialized with order and roughly compiles with the input density function, the map will converge quickly 2) Read the input data space. For each observation (instance), use the optimum fit approach, which is based on the Euclidean distance c = $argi$ min || x – mi || to find the neuron which best matches this observation. Let x denote the training vector from the observation and mi denote a single neuron in the matrix. Update that neuron to resemble that observation using the following equation: $mi\ (t + 1) = mi\ (t) + h(t)[x(t) - mi\ (t)]$ (4) $mi(t)$: the weight vector before the neuron is updated. $(t + 1)$: the weight vector after the neuron is updated. (t): the training vector from the observation. $h(t)$: the neighborhood function (a smoothing kernel defined over the lattice points), defined though the following equation: $h(t) = \{\ \alpha(t), i \in Nc\ 0, i \in Nc$
(5) : the neighborhood set, which decreases with time. (t): the learning-rate factor which can be linear, exponential or inversely proportional. It is a monotonically decreasing function of time (t)



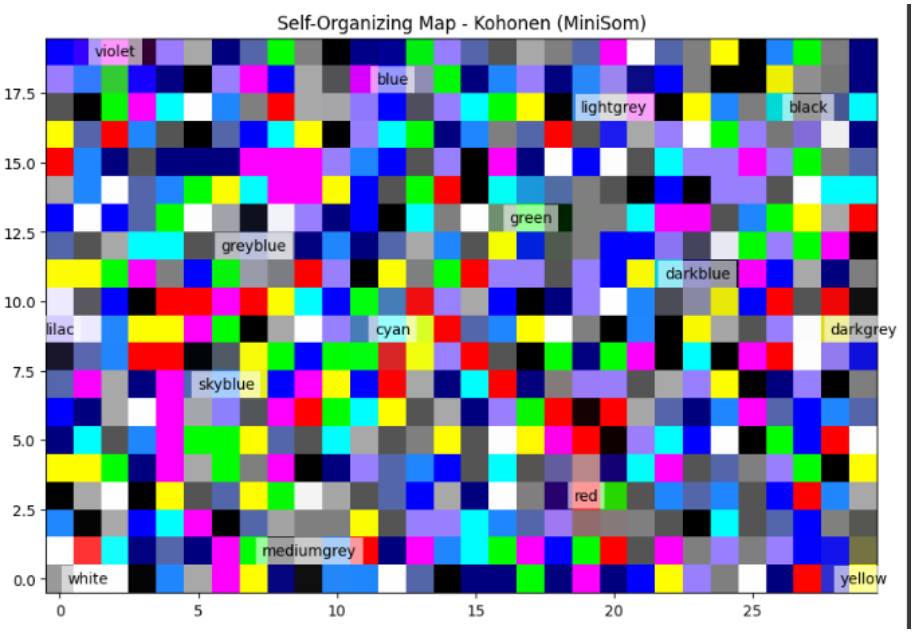
In general, SOMs might be useful for visualizing high-dimensional data in terms of its similarity structure. Especially large SOMs (i.e. with large number of Kohonen units) are known to perform mappings that preserve the topology of the original data, i.e. neighboring data points in input space will also be represented in adjacent locations on the SOM. The following code shows the ‘classic’ color mapping example, i.e. the SOM will map a number of colors into a rectangular area.

Code:

```
!pip install minisom

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

```
from minisom import MiniSom

# Define colors as RGB values
colors = np.array([
    [0., 0., 0.],        # black
    [0., 0., 1.],        # blue
    [0., 0., 0.5],       # darkblue
    [0.125, 0.529, 1.0], # skyblue
    [0.33, 0.4, 0.67],   # greyblue
    [0.6, 0.5, 1.0],     # lilac
    [0., 1., 0.],        # green
    [1., 0., 0.],        # red
    [0., 1., 1.],        # cyan
    [1., 0., 1.],        # violet
    [1., 1., 0.],        # yellow
    [1., 1., 1.],        # white
    [0.33, 0.33, 0.33],  # darkgrey
    [0.5, 0.5, 0.5],     # mediumgrey
    [0.66, 0.66, 0.66]   # lightgrey
])

# Names of colors for visualization
color_names = [
    'black','blue','darkblue','skyblue',
    'greyblue','lilac','green','red',
    'cyan','violet','yellow','white',
    'darkgrey','mediumgrey','lightgrey'
]

# Initialize MiniSom (20x30 grid, input_len = 3 for RGB)
som = MiniSom(x=20, y=30, input_len=3, sigma=1.0, learning_rate=0.05)
som.random_weights_init(colors)
som.train_random(colors, 400)  # 400 iterations

# Plot the SOM weight map
plt.figure(figsize=(10, 7))
weights = som.get_weights()

# Display SOM as image
plt.imshow(weights, origin='lower')
plt.title("Self-Organizing Map - Kohonen (MiniSom)")

# Map training data onto SOM
for i, color in enumerate(colors):
    w = som.winner(color)
    plt.text(w[1], w[0], color_names[i],
             ha='center', va='center',
             bbox=dict(facecolor='white', alpha=0.6, lw=0))

plt.show()
```

Output:



Practical 6 b: ADAPTIVE RESONANCE THEORY

Code:

```
pip install artlib
import numpy as np

class ART1:
    def __init__(self, n_features, n_categories, rho=0.8):
        # Initialize bottom-up weights (W) to all ones (max vigilance)
        self.W = np.ones((n_categories, n_features), dtype=int)
        self.rho = rho  # vigilance parameter
        self.n_categories = n_categories

    def train(self, patterns):
        category_used = 0
        for pattern in patterns:
            for i in range(self.n_categories):
                # Check match: pattern should fit within category with vigilance
                match = np.all((pattern & self.W[i]) == pattern)
                similarity = np.sum(pattern & self.W[i]) / np.sum(pattern)
                if match and similarity >= self.rho:
                    # Update weights by intersection (AND operation)
                    self.W[i] = self.W[i] & pattern
                    break
```

```
            else:
                # Create new category if vigilance test fails in all categories
                if category_used < self.n_categories:
                    self.W[category_used] = pattern
                    category_used += 1

    def predict(self, pattern):
        # Find category matching pattern based on vigilance
        for i in range(self.n_categories):
            match = np.all((pattern & self.W[i]) == pattern)
            similarity = np.sum(pattern & self.W[i]) / np.sum(pattern)
            if match and similarity >= self.rho:
                return i
        return -1  # no suitable category found

# Example usage
if __name___== "__main__":
    # Binary input patterns
    patterns = np.array([
        [1, 0, 1, 0],
        [1, 1, 1, 0],
        [0, 1, 0, 1]
    ])

    # Initialize ART1 with 4 features and up to 3 categories
    art = ART1(n_features=4, n_categories=3, rho=0.8)

    # Train the network on patterns
    art.train(patterns)

    print("Trained category weights (bottom-up):")
    print(art.W)

    # Predict category of new input
    new_pattern = np.array([1, 0, 1, 0])
    category = art.predict(new_pattern)
    print(f"Pattern {new_pattern} belongs to category {category}")
```

Output:

```
Trained category weights (bottom-up):
[[1 0 1 0]
 [1 1 1 0]
 [0 1 0 1]]
Pattern [1 0 1 0] belongs to category 0
```

Practical 7

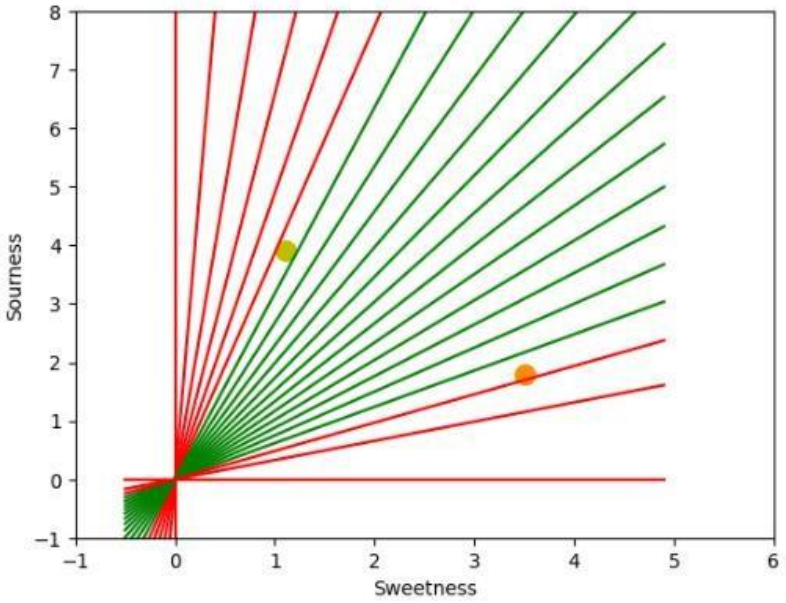
Practical 7 a: Line Separation

Code:

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

# Function to create a line equation ax + by + c = 0
def create_distance_function(a, b, c):
    """
    Returns a distance function for line ax + by + c = 0
    """
    def distance(x, y):
        """
        Returns tuple (d, pos):
        - d is the distance from point to line
        - pos = -1 → point below line
        - pos = 0  → point on line
        - pos = +1 → point above line
        """
        nom = a * x + b * y + c
        if nom == 0:
            pos = 0
        elif (nom < 0 and b < 0) or (nom > 0 and b > 0):
            pos = -1
        else:
            pos = 1
        return (np.abs(nom) / np.sqrt(a ** 2 + b ** 2), pos)
    return distance

# Sample points (two classes)
points = [(3.5, 1.8), (1.1, 3.9)]

# Plot setup
fig, ax = plt.subplots()
ax.set_xlabel("Sweetness")
ax.set_ylabel("Sourness")
ax.set_xlim([-1, 6])
ax.set_ylim([-1, 8])
X = np.arange(-0.5, 5, 0.1)

# Plot sample points
size = 10
for index, (x, y) in enumerate(points):
    if index == 0:
        ax.plot(x, y, "o", color="darkorange", markersize=size)
    else:
```

```
        ax.plot(x, y, "oy", markersize=size)

# Try different separating lines
step = 0.05
for x in np.arange(0, 1 + step, step):
    slope = np.tan(np.arccos(x))
    dist4line1 = create_distance_function(slope, -1, 0)

    Y = slope * X
    results = []

    for point in points:
        results.append(dist4line1(*point))

    # If points lie on different sides of line → green line
    if (results[0][1] != results[1][1]):
        ax.plot(X, Y, "g-")
    else:
        ax.plot(X, Y, "r-")

plt.show()
```

Output:



Practical 7 b: Write a program for Hopfield network model for associative memory

Code:

```
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.gridspec as gridspec
from matplotlib.widgets import Button
import time

class HopfieldNetwork:
    def __init__(self, n_neurons):
        self.n_neurons = n_neurons
        self.weights = np.zeros((n_neurons, n_neurons))
        self.patterns = []

    def train(self, patterns):
        """Train the network with patterns using Hebbian learning"""
        self.patterns = np.array(patterns)
        n_patterns = len(patterns)

        # Initialize weights matrix
        self.weights = np.zeros((self.n_neurons, self.n_neurons))

        # Hebbian learning rule
        for pattern in patterns:
            pattern = np.array(pattern).reshape(-1, 1)
            self.weights += np.dot(pattern, pattern.T)

        # Remove self-connections (set diagonal to zero)
        np.fill_diagonal(self.weights, 0)

        # Normalize weights
        self.weights /= n_patterns

    def recall(self, input_pattern, max_iterations=100):
        """Recall a pattern from noisy input"""
        pattern = np.array(input_pattern).copy()
        history = [pattern.copy()]

        for _ in range(max_iterations):
            # Update neurons asynchronously (random order)
            update_order = np.random.permutation(self.n_neurons)

            for i in update_order:
                # Calculate net input
                net_input = np.dot(self.weights[i], pattern)
```

```
                # Update neuron state
                pattern[i] = 1 if net_input >= 0 else -1

            history.append(pattern.copy())

            # Check for convergence
            if np.array_equal(history[-1], history[-2]):
                break

        return pattern, history

    def calculate_energy(self, pattern):
        """Calculate the energy of a state"""
        pattern = np.array(pattern)
        return -0.5 * np.dot(pattern.T, np.dot(self.weights, pattern))

    def add_noise(self, pattern, noise_level=0.3):
        """Add random noise to a pattern"""
        noisy_pattern = pattern.copy()
        n_flips = int(noise_level * len(pattern))
        flip_indices = np.random.choice(len(pattern), n_flips, replace=False)
        noisy_pattern[flip_indices] *= -1
        return noisy_pattern

def create_patterns(grid_size, num_patterns):
    """Create random patterns for training"""
    patterns = []
    for _ in range(num_patterns):
        pattern = np.random.choice([-1, 1], size=grid_size*grid_size)
        patterns.append(pattern)
    return patterns

def pattern_to_grid(pattern, grid_size):
    """Convert 1D pattern to 2D grid for visualization"""
    return pattern.reshape((grid_size, grid_size))

def visualize_hopfield(network, patterns, recalled_pattern, history, grid_size):
    """Visualize the Hopfield network process"""
    fig = plt.figure(figsize=(15, 10))
    gs = gridspec.GridSpec(3, 4, figure=fig)

    # Original patterns
    ax1 = fig.add_subplot(gs[0, 0])
    ax1.imshow(pattern_to_grid(patterns[0], grid_size), cmap='binary', interpolation='nearest')
    ax1.set_title('Pattern 1')
    ax1.set_xticks([])
    ax1.set_yticks([])

    ax2 = fig.add_subplot(gs[0, 1])
    ax2.imshow(pattern_to_grid(patterns[1], grid_size), cmap='binary', interpolation='nearest')
```

```
    ax2.set_title('Pattern 2')
    ax2.set_xticks([])
    ax2.set_yticks([])

    ax3 = fig.add_subplot(gs[0, 2])
    ax3.imshow(pattern_to_grid(patterns[2], grid_size), cmap='binary', interpolation='nearest')
    ax3.set_title('Pattern 3')
    ax3.set_xticks([])
    ax3.set_yticks([])

    # Noisy input
    ax4 = fig.add_subplot(gs[0, 3])
    ax4.imshow(pattern_to_grid(history[0], grid_size), cmap='binary', interpolation='nearest')
    ax4.set_title('Noisy Input')
    ax4.set_xticks([])
    ax4.set_yticks([])

    # Final output
    ax5 = fig.add_subplot(gs[1, :2])
    ax5.imshow(pattern_to_grid(recalled_pattern, grid_size), cmap='binary',
interpolation='nearest')
    ax5.set_title('Final Output')
    ax5.set_xticks([])
    ax5.set_yticks([])

    # Energy landscape
    ax6 = fig.add_subplot(gs[1, 2:])
    energy_history = [network.calculate_energy(state) for state in history]
    ax6.plot(energy_history, 'b-', linewidth=2)
    ax6.set_xlabel('Iteration')
    ax6.set_ylabel('Energy')
    ax6.set_title('Energy Convergence')
    ax6.grid(True)

    # Convergence animation
    ax7 = fig.add_subplot(gs[2, :])
    ax7.set_title('Convergence Process (Last few iterations)')
    ax7.set_xticks([])
    ax7.set_yticks([])

    # Show last few iterations
    n_frames = min(8, len(history))
    for i, state in enumerate(history[-n_frames:]):
        offset = i * (grid_size + 2)
        ax7.imshow(pattern_to_grid(state, grid_size),
                   cmap='binary',
                   interpolation='nearest',
                   extent=[offset, offset + grid_size, 0, grid_size])

    ax7.set_xlim(0, n_frames * (grid_size + 2))
```

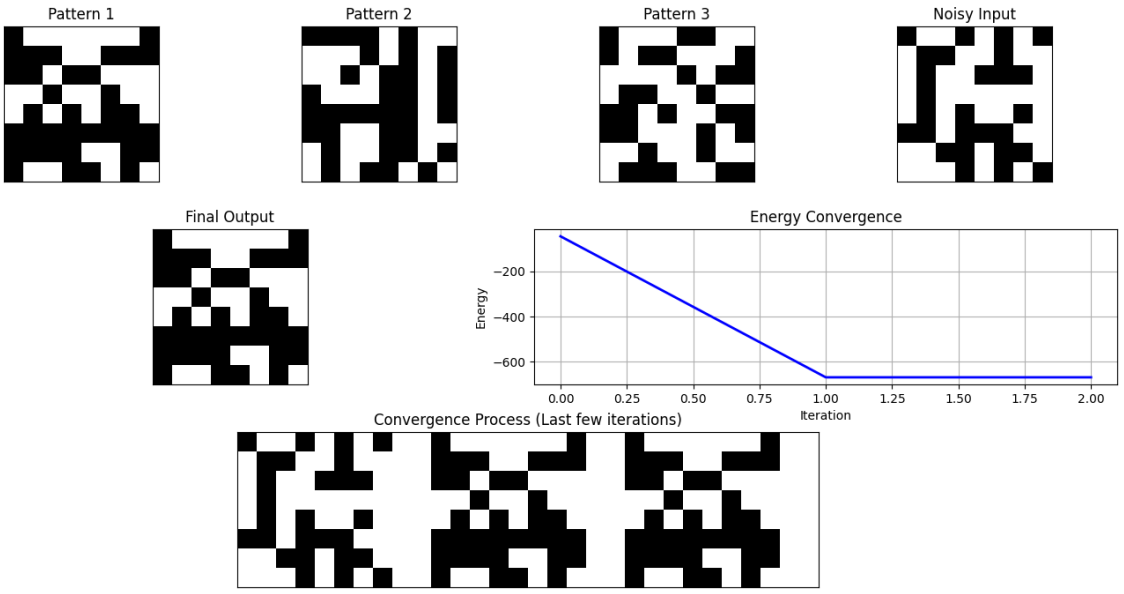
```
    ax7.set_ylim(0, grid_size)

    plt.tight_layout()
    plt.show()

def run_demo():
    """Run a complete demonstration of the Hopfield network"""
    # Parameters
    grid_size = 8
    n_neurons = grid_size * grid_size
    num_patterns = 3
    noise_level = 0.4

    # Create Hopfield network
    network = HopfieldNetwork(n_neurons)

    # Generate and train patterns
    patterns = create_patterns(grid_size, num_patterns)
    network.train(patterns)

    # Select a pattern to recall and add noise
    target_pattern = patterns[0]
    noisy_input = network.add_noise(target_pattern, noise_level)

    # Recall the pattern
    recalled_pattern, history = network.recall(noisy_input)

    # Calculate accuracy
    accuracy = np.mean(target_pattern == recalled_pattern) * 100

    print(f"Pattern size: {grid_size}x{grid_size} ({n_neurons} neurons)")
    print(f"Number of stored patterns: {num_patterns}")
    print(f"Noise level: {noise_level * 100}%")
    print(f"Recall accuracy: {accuracy:.1f}%")
    print(f"Converged in {len(history)} iterations")

    # Visualize results - FIXED: pass network as parameter
    visualize_hopfield(network, patterns, recalled_pattern, history, grid_size)

    return network, patterns, noisy_input, recalled_pattern, history

def interactive_demo():
    """Interactive version with buttons to try different patterns"""
    fig, axes = plt.subplots(2, 3, figsize=(12, 8))
    plt.subplots_adjust(bottom=0.2)

    # Parameters
    grid_size = 8
    n_neurons = grid_size * grid_size
    num_patterns = 3
```

```
# Create network and patterns
network = HopfieldNetwork(n_neurons)
patterns = create_patterns(grid_size, num_patterns)
network.train(patterns)

current_pattern_idx = 0
noise_level = 0.4

def update_display(pattern_idx, noise_level):
    """Update the display with new results"""
    for ax in axes.flat:
        ax.clear()
        ax.set_xticks([])
        ax.set_yticks([])

    # Get target pattern and add noise
    target_pattern = patterns[pattern_idx]
    noisy_input = network.add_noise(target_pattern, noise_level)

    # Recall pattern
    recalled_pattern, history = network.recall(noisy_input)
    accuracy = np.mean(target_pattern == recalled_pattern) * 100

    # Display patterns
    axes[0,0].imshow(pattern_to_grid(patterns[0], grid_size), cmap='binary')
    axes[0,0].set_title('Pattern 1')

    axes[0,1].imshow(pattern_to_grid(patterns[1], grid_size), cmap='binary')
    axes[0,1].set_title('Pattern 2')

    axes[0,2].imshow(pattern_to_grid(patterns[2], grid_size), cmap='binary')
    axes[0,2].set_title('Pattern 3')

    # Display noisy input and result
    axes[1,0].imshow(pattern_to_grid(noisy_input, grid_size), cmap='binary')
    axes[1,0].set_title(f'Noisy Input ({noise_level*100:.0f}% noise)')

    axes[1,1].imshow(pattern_to_grid(recalled_pattern, grid_size), cmap='binary')
    axes[1,1].set_title(f'Recalled ({accuracy:.1f}% accuracy)')

    # Energy plot
    energy_history = [network.calculate_energy(state) for state in history]
    axes[1,2].plot(energy_history, 'b-')
    axes[1,2].set_title('Energy Convergence')
    axes[1,2].set_xlabel('Iterations')
    axes[1,2].set_ylabel('Energy')
    axes[1,2].grid(True)

    plt.draw()
```

```
    # Create buttons
    ax_prev = plt.axes([0.2, 0.05, 0.1, 0.075])
    ax_next = plt.axes([0.35, 0.05, 0.1, 0.075])
    ax_more_noise = plt.axes([0.5, 0.05, 0.1, 0.075])
    ax_less_noise = plt.axes([0.65, 0.05, 0.1, 0.075])

    btn_prev = Button(ax_prev, 'Previous')
    btn_next = Button(ax_next, 'Next')
    btn_more_noise = Button(ax_more_noise, 'More Noise')
    btn_less_noise = Button(ax_less_noise, 'Less Noise')

    def next_pattern(event):
        nonlocal current_pattern_idx
        current_pattern_idx = (current_pattern_idx + 1) % num_patterns
        update_display(current_pattern_idx, noise_level)

    def prev_pattern(event):
        nonlocal current_pattern_idx
        current_pattern_idx = (current_pattern_idx - 1) % num_patterns
        update_display(current_pattern_idx, noise_level)

    def increase_noise(event):
        nonlocal noise_level
        noise_level = min(0.8, noise_level + 0.1)
        update_display(current_pattern_idx, noise_level)

    def decrease_noise(event):
        nonlocal noise_level
        noise_level = max(0.1, noise_level - 0.1)
        update_display(current_pattern_idx, noise_level)

    btn_next.on_clicked(next_pattern)
    btn_prev.on_clicked(prev_pattern)
    btn_more_noise.on_clicked(increase_noise)
    btn_less_noise.on_clicked(decrease_noise)

    # Initial display
    update_display(current_pattern_idx, noise_level)
    plt.show()

if __name__== "__main__":
    print("Hopfield Network Demo - Associative Memory")
    print("=" * 50)

    # Run simple demo
    network, patterns, noisy_input, recalled_pattern, history = run_demo()

    # Uncomment the line below to run the interactive demo instead
    # interactive_demo()
```

Output:

Hopfield Network Demo - Associative Memory
==========
Pattern size: 8x8 (64 neurons)
Number of stored patterns: 3
Noise level: 40.0%
Recall accuracy: 100.0%
Converged in 3 iterations



Practical 8

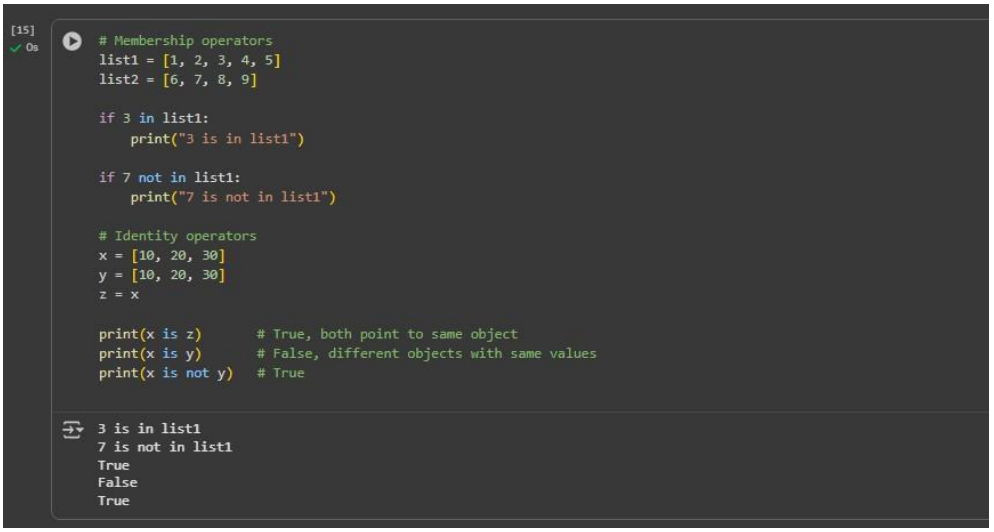
Practical 8 a: Membership and Identity operators in, not in

Code:

```
# Membership operators
list1 = [1, 2, 3, 4, 5]
list2 = [6, 7, 8, 9]

if 3 in list1:
    print("3 is in list1")

if 7 not in list1:
    print("7 is not in list1")

# Identity operators
x = [10, 20, 30]
y = [10, 20, 30]
z = x

print(x is z)       # True, both point to same object
print(x is y)       # False, different objects with same values
print(x is not y)   # True
```

Output:

```
3 is in list1
7 is not in list1
True
False
True
```



Practical 8 b: Membership and Identity Operators is, is not

Code:

```
# Membership operators
list1 = [1, 2, 3, 4, 5]

if 3 in list1:
    print("3 is in list1")

if 7 not in list1:
    print("7 is not in list1")

# Identity operators
x = 5
if type(x) is int:
    print("x is an integer")

y = 5.2
if type(y) is not int:
    print("y is not an integer")
```

Output:

```
3 is in list1
7 is not in list1
x is an integer
y is not an integer
```

Practical 9

Practical 9 a: Find the ratios using fuzzy logic

Code:

```
!pip install fuzzywuzzy
!pip install python-Levenshtein
# Python code showing all the ratios together,
# make sure you have installed fuzzywuzzy module
from fuzzywuzzy import fuzz
from fuzzywuzzy import process

s1 = "I love fuzzysforfuzzys"
s2 = "I am loving fuzzysforfuzzys"

print ("FuzzyWuzzy Ratio:", fuzz.ratio(s1, s2))
print ("FuzzyWuzzyPartialRatio: ", fuzz.partial_ratio(s1, s2))
print ("FuzzyWuzzyTokenSortRatio: ", fuzz.token_sort_ratio(s1, s2))
print ("FuzzyWuzzyTokenSetRatio: ", fuzz.token_set_ratio(s1, s2))
print ("FuzzyWuzzyWRatio: ", fuzz.WRatio(s1, s2),'\n\n')

# for process library,
query = 'fuzzys for fuzzys'
choices = ['fuzzy for fuzzy', 'fuzzy fuzzy', 'g. for fuzzys']

print ("List of ratios: ")
print (process.extract(query, choices), '\n')
print ("Best among the above list: ",process.extractOne(query, choices))
```

Output:

```
FuzzyWuzzy Ratio: 86
FuzzyWuzzyPartialRatio:  86
FuzzyWuzzyTokenSortRatio:  86
FuzzyWuzzyTokenSetRatio:  87
FuzzyWuzzyWRatio:  86

List of ratios:
[('g. for fuzzys', 95), ('fuzzy for fuzzy', 94), ('fuzzy fuzzy', 86)]
Best among the above list:  ('g. for fuzzys', 95)
```

```
# Python code showing all the ratios together,
# make sure you have installed fuzzywuzzy module
from fuzzywuzzy import fuzz
from fuzzywuzzy import process

s1 = "I love fuzzysforfuzzys"
s2 = "I am loving fuzzysforfuzzys"

print ("FuzzyWuzzy Ratio:", fuzz.ratio(s1, s2))
print ("FuzzyWuzzyPartialRatio: ", fuzz.partial_ratio(s1, s2))
print ("FuzzyWuzzyTokenSortRatio: ", fuzz.token_sort_ratio(s1, s2))
print ("FuzzyWuzzyTokenSetRatio: ", fuzz.token_set_ratio(s1, s2))
print ("FuzzyWuzzyWRatio: ", fuzz.WRatio(s1, s2),'\n\n')

# for process library,
query = 'fuzzys for fuzzys'
choices = ['fuzzy for fuzzy', 'fuzzy fuzzy', 'g. for fuzzys']

print ("List of ratios: ")
print (process.extract(query, choices), '\n')
print ("Best among the above list: ",process.extractOne(query, choices))
```

```
FuzzyWuzzy Ratio: 86
FuzzyWuzzyPartialRatio:  86
FuzzyWuzzyTokenSortRatio:  86
FuzzyWuzzyTokenSetRatio:  87
FuzzyWuzzyWRatio:  86


List of ratios:
[('g. for fuzzys', 95), ('fuzzy for fuzzy', 94), ('fuzzy fuzzy', 86)]

Best among the above list:  ('g. for fuzzys', 95)
```

Practical 9 b: Solve Tipping Problem using fuzzy logic

Code:

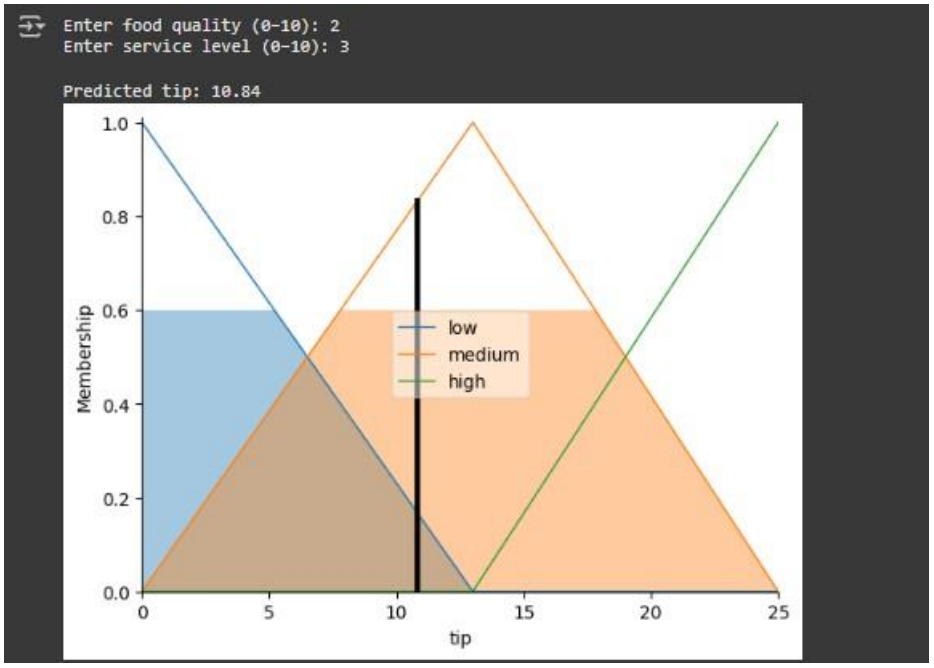
```
!pip install scikit-fuzzy
import numpy as np
import skfuzzy as fuzz
from skfuzzy import control as ctrl
import matplotlib.pyplot as plt

# Antecedent/Consequent objects
quality = ctrl.Antecedent(np.arange(0, 11, 1), 'quality')
service = ctrl.Antecedent(np.arange(0, 11, 1), 'service')
tip = ctrl.Consequent(np.arange(0, 26, 1), 'tip')

# Auto-membership function population with 3 terms: poor/average/good
quality.automf(3)
service.automf(3)

# Custom membership functions for tip
tip['low'] = fuzz.trimf(tip.universe, [0, 0, 13])
tip['medium'] = fuzz.trimf(tip.universe, [0, 13, 25])
tip['high'] = fuzz.trimf(tip.universe, [13, 25, 25])

# Fuzzy rules
rule1 = ctrl.Rule(quality['poor'] | service['poor'], tip['low'])
```

```
rule2 = ctrl.Rule(service['average'], tip['medium'])
rule3 = ctrl.Rule(service['good'] | quality['good'], tip['high'])

# Control system creation and simulation
tipping_ctrl = ctrl.ControlSystem([rule1, rule2, rule3])
tipping = ctrl.ControlSystemSimulation(tipping_ctrl)

# Take interactive input from user
q = float(input("Enter food quality (0–10): "))
s = float(input("Enter service level (0–10): "))

# Pass inputs to the system
tipping.input['quality'] = q
tipping.input['service'] = s

# Compute the result
tipping.compute()

# Show result
print("\nPredicted tip: {:.2f}".format(tipping.output['tip']))

# Plot membership function and result
tip.view(sim=tipping)
plt.show()
```

Output:

Enter food quality (0–10): 2
Enter service level (0–10): 3

Predicted tip: 10.84



Practical 10

Practical 10 a: Implementation of simple genetic algorithm

Code:

```
import random
import matplotlib.pyplot as plt

# Number of individuals in each generation
POPULATION_SIZE = 100

# Valid genes
GENES = '''abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ
1234567890, .-;:_!"#%&/()=?@${[]}'''

# Target string to be generated
TARGET = "I love GeeksforGeeks"


class Individual(object):
    """
    Class representing individual in population
    """
    def __init__(self, chromosome):
        self.chromosome = chromosome
        self.fitness = self.cal_fitness()

    @classmethod
    def mutated_genes(self):
        """
        Create random genes for mutation
        """
        global GENES
        gene = random.choice(GENES)
        return gene

    @classmethod
    def create_gnome(self):
        """
        Create chromosome or string of genes
        """
        global TARGET
        gnome_len = len(TARGET)
        return [self.mutated_genes() for _ in range(gnome_len)]

    def mate(self, par2):
        """
```

```
        Perform mating and produce new offspring
        """
        child_chromosome = []
        for gp1, gp2 in zip(self.chromosome, par2.chromosome):
            prob = random.random()
            # 45% genes from parent1
            if prob < 0.45:
                child_chromosome.append(gp1)
            # 45% genes from parent2
            elif prob < 0.90:
                child_chromosome.append(gp2)
            # 10% mutation
            else:
                child_chromosome.append(self.mutated_genes())
        return Individual(child_chromosome)

    def cal_fitness(self):
        """
        Calculate fitness score.
        Fitness is the number of characters in which
        the chromosome differs from the target string.
        """
        global TARGET
        fitness = 0
        for gs, gt in zip(self.chromosome, TARGET):
            if gs != gt:
                fitness += 1
        return fitness


# Driver code
def main():
    global POPULATION_SIZE

    # Current generation
    generation = 1
    found = False
    population = []

    # Store history for visualization
    fitness_history = []

    # Create initial population
    for _ in range(POPULATION_SIZE):
        gnome = Individual.create_gnome()
        population.append(Individual(gnome))

    while not found:
        # Sort the population in increasing order of fitness score
        population = sorted(population, key=lambda x: x.fitness)
```

```
    # Record best fitness for visualization
    fitness_history.append(population[0].fitness)

    # Print best result of current generation
    print("Generation: {}\tBest String: {}\tFitness: {}".format(
        generation,
        "".join(population[0].chromosome),
        population[0].fitness
    ))

    # If the individual having lowest fitness score is 0,
    # we know we have reached the target
    if population[0].fitness <= 0:
        found = True
        break

    # Otherwise, generate new offsprings for new generation
    new_generation = []

    # Perform Elitism — 10% of fittest population goes to next generation
    s = int((10 * POPULATION_SIZE) / 100)
    new_generation.extend(population[:s])

    # From 50% of fittest population, generate offspring
    s = int((90 * POPULATION_SIZE) / 100)
    for _ in range(s):
        parent1 = random.choice(population[:50])
        parent2 = random.choice(population[:50])
        child = parent1.mate(parent2)
        new_generation.append(child)

    population = new_generation
    generation += 1

# Final output
print("\n🎯 Target Reached in Generation {}!".format(generation))
print("Final String: {}\tFitness: {}".format(
    "".join(population[0].chromosome),
    population[0].fitness
))

# Plot convergence
plt.plot(fitness_history, label="Best Fitness per Generation")
plt.xlabel("Generations")
plt.ylabel("Fitness (Lower is Better)")
plt.title("Genetic Algorithm Convergence")
plt.legend()
plt.show()
```

```
if __name___== "__main__":
    main()
```

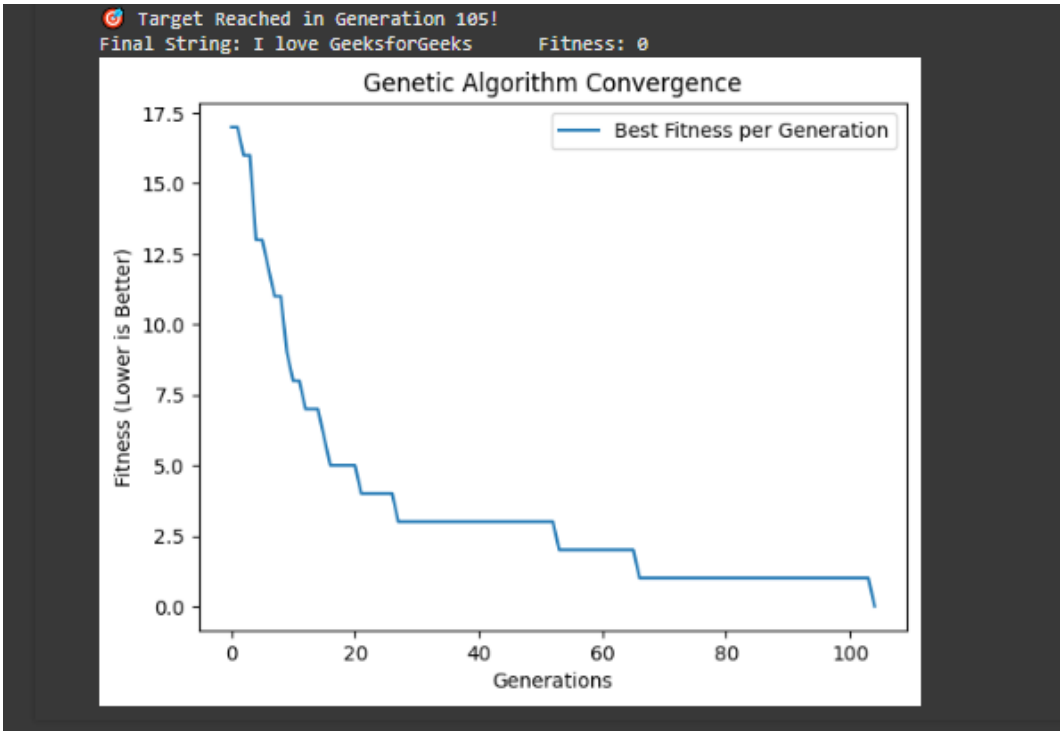
Output:

```
Generation: 1  Best String: Z UyveN@)bb8XJOiwVST       Fitness: 17
Generation: 2  Best String: Z UyveN@)bb8XJOiwVST       Fitness: 17
Generation: 3  Best String: R Hu}q;[=U[]fxr_weJ_  Fitness: 16
Generation: 4  Best String: R Hu}q;[=U[]fxr_weJ_  Fitness: 16
Generation: 5  Best String: g tovenG)[Ksfr0_z;SD    Fitness: 13
Generation: 6  Best String: g tovenG)[Ksfr0_z;SD    Fitness: 13
Generation: 7  Best String: abtovenG)eNsxrH4z#ks Fitness: 12
Generation: 8  Best String: 3%UPveNGVgbsfohJ/eks         Fitness: 11
Generation: 9  Best String: 3%UPveNGVgbsfohJ/eks         Fitness: 11
Generation: 10Best String: g tove KoeN:foH4e;ks    Fitness: 9
Generation: 11Best String: t VoveNGl[bs4orG]eks  Fitness: 8
Generation: 12Best String: t VoveNGl[bs4orG]eks  Fitness: 8
Generation: 13Best String: 3 VoveNGKXbsforGzeks         Fitness: 7
Generation: 14Best String: 3 VoveNGKXbsforGzeks         Fitness: 7
Generation: 15Best String: 3 VoveNGKXbsforGzeks         Fitness: 7
Generation: 16Best String: j Vove G;eNsfoHGeeSs  Fitness: 6
Generation: 17Best String: j tove G)eisfoHGeeks     Fitness: 5
Generation: 18Best String: j tove G)eisfoHGeeks     Fitness: 5
Generation: 19Best String: j tove G)eisfoHGeeks     Fitness: 5
Generation: 20Best String: j tove G)eisfoHGeeks     Fitness: 5
Generation: 21Best String: j tove G)eisfoHGeeks     Fitness: 5
Generation: 22Best String: j Vove GheisforGeeks    Fitness: 4
Generation: 23Best String: j Vove GheisforGeeks    Fitness: 4
Generation: 24Best String: j Vove GheisforGeeks    Fitness: 4
Generation: 25Best String: j Vove GheisforGeeks    Fitness: 4
Generation: 26Best String: j Vove GheisforGeeks    Fitness: 4
Generation: 27Best String: j Vove GheisforGeeks    Fitness: 4
Generation: 28Best String: j Vove G eksforGeeks    Fitness: 3
Generation: 29Best String: j Vove G eksforGeeks    Fitness: 3
Generation: 30Best String: j Vove G eksforGeeks    Fitness: 3
Generation: 31Best String: j Vove G eksforGeeks    Fitness: 3
Generation: 32Best String: j Vove G eksforGeeks    Fitness: 3
Generation: 33Best String: j Vove G eksforGeeks    Fitness: 3
Generation: 34Best String: j Vove G eksforGeeks    Fitness: 3
Generation: 35Best String: j Vove G eksforGeeks    Fitness: 3
Generation: 36Best String: j Vove G eksforGeeks    Fitness: 3
Generation: 37Best String: j Vove G eksforGeeks    Fitness: 3
Generation: 38Best String: j Vove G eksforGeeks    Fitness: 3
Generation: 39Best String: j Vove G eksforGeeks   Fitness: 3
Generation: 40Best String: j Vove G eksforGeeks   Fitness: 3
Generation: 41Best String: j Vove G eksforGeeks   Fitness: 3
Generation: 42Best String: j Vove G eksforGeeks   Fitness: 3
Generation: 43Best String: j Vove G eksforGeeks   Fitness: 3
```

Generation: 44Best String: j Vove G eksforGeeks Fitness: 3
Generation: 45Best String: j Vove G eksforGeeks Fitness: 3
Generation: 46Best String: j Vove G eksforGeeks Fitness: 3
Generation: 47Best String: j Vove G eksforGeeks Fitness: 3
Generation: 48Best String: j Vove G eksforGeeks Fitness: 3
Generation: 49Best String: j Vove G eksforGeeks Fitness: 3
Generation: 50Best String: j Vove G eksforGeeks Fitness: 3
Generation: 51Best String: j Vove G eksforGeeks Fitness: 3
Generation: 52Best String: j Vove G eksforGeeks Fitness: 3
Generation: 53Best String: j Vove G eksforGeeks Fitness: 3
Generation: 54Best String: k love GBeksforGeeks Fitness: 2
Generation: 55Best String: k love GBeksforGeeks Fitness: 2
Generation: 56Best String: k love GBeksforGeeks Fitness: 2
Generation: 57Best String: k love GBeksforGeeks Fitness: 2
Generation: 58Best String: k love GBeksforGeeks Fitness: 2
Generation: 59Best String: k love GBeksforGeeks Fitness: 2
Generation: 60Best String: k love GBeksforGeeks Fitness: 2
Generation: 61Best String: k love GBeksforGeeks Fitness: 2
Generation: 62Best String: k love GBeksforGeeks Fitness: 2
Generation: 63Best String: k love GBeksforGeeks Fitness: 2
Generation: 64Best String: k love GBeksforGeeks Fitness: 2
Generation: 65Best String: k love GBeksforGeeks Fitness: 2
Generation: 66Best String: k love GBeksforGeeks Fitness: 2
Generation: 67Best String: I love G eksforGeeks Fitness: 1
Generation: 68Best String: I love G eksforGeeks Fitness: 1
Generation: 69Best String: I love G eksforGeeks Fitness: 1
Generation: 70Best String: I love G eksforGeeks Fitness: 1
Generation: 71Best String: I love G eksforGeeks Fitness: 1
Generation: 72Best String: I love G eksforGeeks Fitness: 1
Generation: 73Best String: I love G eksforGeeks Fitness: 1
Generation: 74Best String: I love G eksforGeeks Fitness: 1
Generation: 75Best String: I love G eksforGeeks Fitness: 1
Generation: 76Best String: I love G eksforGeeks Fitness: 1
Generation: 77Best String: I love G eksforGeeks Fitness: 1
Generation: 78Best String: I love G eksforGeeks Fitness: 1
Generation: 79Best String: I love G eksforGeeks Fitness: 1
Generation: 80Best String: I love G eksforGeeks Fitness: 1
Generation: 81Best String: I love G eksforGeeks Fitness: 1
Generation: 82Best String: I love G eksforGeeks Fitness: 1
Generation: 83Best String: I love G eksforGeeks Fitness: 1
Generation: 84Best String: I love G eksforGeeks Fitness: 1
Generation: 85Best String: I love G eksforGeeks Fitness: 1
Generation: 86Best String: I love G eksforGeeks Fitness: 1
Generation: 87Best String: I love G eksforGeeks Fitness: 1
Generation: 88Best String: I love G eksforGeeks Fitness: 1
Generation: 89Best String: I love G eksforGeeks Fitness: 1
Generation: 90Best String: I love G eksforGeeks Fitness: 1
Generation: 91Best String: I love G eksforGeeks Fitness: 1
Generation: 92Best String: I love G eksforGeeks Fitness: 1
Generation: 93Best String: I love G eksforGeeks Fitness: 1

Generation: 94Best String: I love G eksforGeeks Fitness: 1
Generation: 95Best String: I love G eksforGeeks Fitness: 1
Generation: 96Best String: I love G eksforGeeks Fitness: 1
Generation: 97Best String: I love G eksforGeeks Fitness: 1
Generation: 98Best String: I love G eksforGeeks Fitness: 1
Generation: 99Best String: I love G eksforGeeks Fitness: 1
Generation: 100 Best String: I love G eksforGeeks Fitness: 1
Generation: 101 Best String: I love G eksforGeeks Fitness: 1
Generation: 102 Best String: I love G eksforGeeks Fitness: 1
Generation: 103 Best String: I love G eksforGeeks Fitness: 1
Generation: 104 Best String: I love G eksforGeeks Fitness: 1
Generation: 105 Best String: I love GeeksforGeeks Fitness: 0

🎯 Target Reached in Generation 105!
Final String: I love GeeksforGeeks Fitness: 0



Practical 10 b: Create two classes: City and Fitness using Genetic algorithm

Code:

```
import numpy as np
import random
import operator
import pandas as pd
```

```
import matplotlib.pyplot as plt

# ============================
# City Class
# ============================
class City:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def distance(self, city):
        """Calculate Euclidean distance between two cities"""
        xDis = abs(self.x - city.x)
        yDis = abs(self.y - city.y)
        distance = np.sqrt((xDis ** 2) + (yDis ** 2))
        return distance

    def __repr__(self):
        return f"({self.x}, {self.y})"

# ============================
# Fitness Class
# ============================
class Fitness:
    def __init__(self, route):
        self.route = route
        self.distance = 0
        self.fitness = 0.0

    def routeDistance(self):
        if self.distance == 0:
            pathDistance = 0
            for i in range(len(self.route)):
                fromCity = self.route[i]
                toCity = self.route[(i + 1) % len(self.route)]
                pathDistance += fromCity.distance(toCity)
            self.distance = pathDistance
        return self.distance

    def routeFitness(self):
        if self.fitness == 0:
            self.fitness = 1 / float(self.routeDistance())
        return self.fitness

# ============================
# Genetic Algorithm Functions
# ============================
def createRoute(cityList):
    return random.sample(cityList, len(cityList))
```

```
def initialPopulation(popSize, cityList):
    return [createRoute(cityList) for _ in range(popSize)]

def rankRoutes(population):
    fitnessResults = {i: Fitness(pop).routeFitness() for i, pop in enumerate(population)}
    return sorted(fitnessResults.items(), key=operator.itemgetter(1), reverse=True)

def selection(popRanked, eliteSize):
    selectionResults = []
    df = pd.DataFrame(np.array(popRanked), columns=["Index", "Fitness"])
    df['cum_sum'] = df.Fitness.cumsum()
    df['cum_perc'] = 100 * df.cum_sum / df.Fitness.sum()

    # Elitism
    for i in range(eliteSize):
        selectionResults.append(popRanked[i][0])

    # Roulette Wheel Selection
    for _ in range(len(popRanked) - eliteSize):
        pick = 100 * random.random()
        for i in range(len(popRanked)):
            if pick <= df.iat[i, 3]:
                selectionResults.append(popRanked[i][0])
                break
    return selectionResults

def matingPool(population, selectionResults):
    return [population[i] for i in selectionResults]

def breed(parent1, parent2):
    childP1, childP2 = [], []
    geneA, geneB = int(random.random() * len(parent1)), int(random.random() * len(parent1))
    startGene, endGene = min(geneA, geneB), max(geneA, geneB)

    for i in range(startGene, endGene):
        childP1.append(parent1[i])

    childP2 = [item for item in parent2 if item not in childP1]
    return childP1 + childP2

def breedPopulation(matingpool, eliteSize):
    children = []
    length = len(matingpool) - eliteSize
    pool = random.sample(matingpool, len(matingpool))

    # Keep elites
    children.extend(matingpool[:eliteSize])

    # Breed rest
    for i in range(length):
```

```
        child = breed(pool[i], pool[len(matingpool) - i - 1])
        children.append(child)
    return children

def mutate(individual, mutationRate):
    for swapped in range(len(individual)):
        if random.random() < mutationRate:
            swapWith = int(random.random() * len(individual))
            individual[swapped], individual[swapWith] = individual[swapWith],
individual[swapped]
    return individual

def mutatePopulation(population, mutationRate):
    return [mutate(ind, mutationRate) for ind in population]

def nextGeneration(currentGen, eliteSize, mutationRate):
    popRanked = rankRoutes(currentGen)
    selectionResults = selection(popRanked, eliteSize)
    matingpool = matingPool(currentGen, selectionResults)
    children = breedPopulation(matingpool, eliteSize)
    nextGen = mutatePopulation(children, mutationRate)
    return nextGen

def geneticAlgorithm(population, popSize, eliteSize, mutationRate, generations):
    pop = initialPopulation(popSize, population)
    print("Initial distance: " + str(1 / rankRoutes(pop)[0][1]))

    for i in range(generations):
        pop = nextGeneration(pop, eliteSize, mutationRate)

    print("Final distance: " + str(1 / rankRoutes(pop)[0][1]))
    bestRouteIndex = rankRoutes(pop)[0][0]
    bestRoute = pop[bestRouteIndex]
    return bestRoute

def geneticAlgorithmPlot(population, popSize, eliteSize, mutationRate, generations):
    pop = initialPopulation(popSize, population)
    progress = [1 / rankRoutes(pop)[0][1]]

    for i in range(generations):
        pop = nextGeneration(pop, eliteSize, mutationRate)
        progress.append(1 / rankRoutes(pop)[0][1])

    plt.plot(progress)
    plt.ylabel('Distance')
    plt.xlabel('Generation')
    plt.title("Genetic Algorithm - TSP Optimization")
    plt.show()

# ==========================
```

```
# Main Function
# ============================
def main():
    cityList = [City(x=int(random.random() * 200), y=int(random.random() * 200)) for _ in
range(25)]
    bestRoute = geneticAlgorithm(population=cityList, popSize=100, eliteSize=20,
                                 mutationRate=0.01, generations=500)
    print("Best Route: ", bestRoute)
    geneticAlgorithmPlot(population=cityList, popSize=100, eliteSize=20,
                         mutationRate=0.01, generations=500)

if __name___== "__main__":
    main()
```

Output:

Initial distance: 2085.318808614946
Final distance: 923.011100368927
Best Route: [(14, 63), (51, 29), (89, 24), (160, 10), (159, 16), (177, 42), (185, 52), (194, 99), (184, 123), (176, 175), (132, 171), (125, 154), (102, 144), (97, 111), (137, 117), (135, 76), (88, 82), (61, 52), (55, 57), (34, 109), (11, 150), (55, 174), (38, 135), (41, 130), (1, 80)]