INDEX

PRACTICAL 1A

Aim: Design a **simple linear neural network model**.

Code:

```
#1part take input from user
n = int(input("Enter no. of elements : "))

print("Enter the inputs : ")
inputs = []

for i in range(0,n):
        ele = float(input())
        inputs.append(ele)

print(inputs)

print("Enter the weights")
weights = []

for i in range(0,n):
        ele = float(input())
        weights.append(ele)

print(weights)

print("The net input can be calculated as Yin = x1w1 + x2w2 + x3w3")

Yin =  []
for i in range(0,n):
        Yin.append(inputs[i]*weights[i])
print(round(sum(Yin),3))
```

Output:

```
"C:\Users\ADITYA KUMAR\PyCharmMiscProject\.venv\Scripts\python.exe" "C:\Users\ADITYA KUMAR\PyCharmMiscProject\.venv\Lib\1a.py"
Enter no. of elements : 3
Enter the inputs :
1.2
3.5
-2.1
[1.2, 3.5, -2.1]
Enter the weights
3
2
1
[3.0, 2.0, 1.0]
The net input can be calculated as Yin = x1w1 + x2w2 + x3w3
8.5

Process finished with exit code 0
```

PRACTICAL 1B

Aim: Calculate the output of **neural net** using **both binary and bipolar sigmoidal function.**

Code:

```
import numpy as np

def binary_sigmoid(x):

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

def bipolar_sigmoid(x):

   return (2 / (1 + np.exp(-x))) - 1

def calculate_neural_net_output(inputs, weights, biases, activation_function):

   # Compute the net input to the neurons
   net_input = np.dot(inputs, weights) + biases

   # Apply the activation function
   return activation_function(net_input)

# Example usage
if __name__ == "__main__":
   # Define inputs, weights, and biases
   inputs = np.array([0.5, -0.2, 0.1])
   weights = np.array([
      [0.4, 0.3, 0.5],
      [-0.3, 0.8, -0.6]
   ]).T  # Transpose to match dimensions
   biases = np.array([0.1, -0.1])

   # Calculate outputs using binary sigmoid
   binary_output = calculate_neural_net_output(inputs, weights, biases, binary_sigmoid)
print("Binary Sigmoid Output:", binary_output)

   # Calculate outputs using bipolar sigmoid
   bipolar_output = calculate_neural_net_output(inputs, weights, biases, bipolar_sigmoid)
   print("Bipolar Sigmoid Output:", bipolar_output)
```

Output:

```
"C:\Users\ADITYA KUMAR\PyCharmMiscProject\.venv\Scripts\python.exe" "C:\Users\ADITYA KUMAR\PyCharmMiscProject\.venv\Lib\1b.py"
Binary Sigmoid Output: [0.57199613 0.38461624]
Bipolar Sigmoid Output: [ 0.14399227 -0.23076751]

Process finished with exit code 0
```

PRACTICAL 2A

Aim: Generate **AND/NOT** function using **McCulloch-Pitts neural net**.

Code:

```
num_ip = int(input("Enter the number of inputs : "))

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 = x1 * w1
m = x2 * w2

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

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)

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

```
For the  1  inputs calculate the net input using yin = x1w1 + x2w2
x1 = 1
x2 = 2
x1 =  [1]
x2 =  [2]
Yin =  [3]
After assuming one weight as excitatory and the other as inhibitory Yin =  [-1]
Y =  [0]

Process finished with exit code 0
```

PRACTICAL 2B

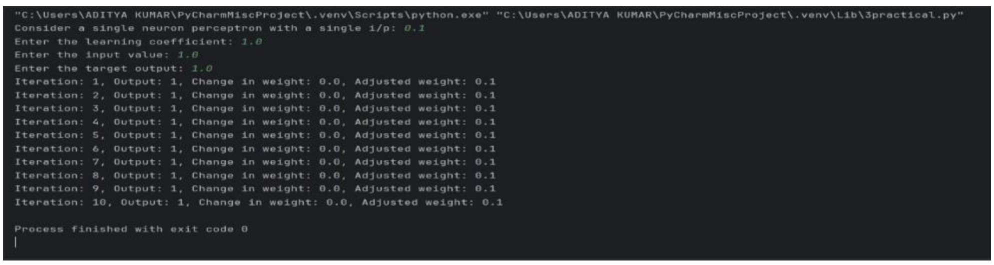
Aim: Generate **XOR** function using **McCulloch-Pitts neural net.**

Code:

```
import math
import numpy
import random
INPUT_NODES = 2
OUTPUT_NODES = 1
HIDDEN_NODES = 2
MAX_ITERATIONS = 130000
LEARNING_RATE = .2
print
"Neural Network Program"
class network:
    def __init__(self, input_nodes, hidden_nodes, output_nodes, learning_rate):
        self.input_nodes = input_nodes
        self.hidden_nodes = hidden_nodes
        self.output_nodes = output_nodes
        self.total_nodes = input_nodes + hidden_nodes + output_nodes
        self.learning_rate = learning_rate
        self.values = numpy.zeros(self.total_nodes)
        self.expectedValues = numpy.zeros(self.total_nodes)
        self.thresholds = numpy.zeros(self.total_nodes)
        self.weights = numpy.zeros((self.total_nodes, self.total_nodes))
        random.seed(10000)
        for i in range(self.input_nodes, self.total_nodes):
            self.thresholds[i] = random.random() / random.random()
            for j in range(i + 1, self.total_nodes):
                self.weights[i][j] = random.random() * 2
    def process(self):
        for i in range(self.input_nodes, self.input_nodes + self.hidden_nodes):
            # sum weighted input nodes for each hidden node, compare threshold, apply sigmoid
            W_i = 0.0
            for j in range(self.input_nodes):
                W_i += self.weights[j][i] * self.values[j]
            W_i -= self.thresholds[i]
            self.values[i] = 1 / (1 + math.exp(-W_i))

        for i in range(self.input_nodes + self.hidden_nodes, self.total_nodes):
            # sum weighted hidden nodes for each output node, compare threshold, apply sigmoid
            W_i = 0.0
            for j in range(self.input_nodes, self.input_nodes + self.hidden_nodes):
                W_i += self.weights[j][i] * self.values[j]
            W_i -= self.thresholds[i]
            self.values[i] = 1 / (1 + math.exp(-W_i))

    def processErrors(self):
        sumOfSquaredErrors = 0.0
       for i in range(self.input_nodes + self.hidden_nodes, self.total_nodes):
            error = self.expectedValues[i] - self.values[i]
            sumOfSquaredErrors += math.pow(error, 2)
            outputErrorGradient = self.values[i] * (1 - self.values[i]) * error
```

```
 for j in range(self.input_nodes, self.input_nodes + self.hidden_nodes):
                delta = self.learning_rate * self.values[j] * outputErrorGradient
                self.weights[j][i] += delta
                hiddenErrorGradient = self.values[j] * (1 - self.values[j]) * outputErrorGradient * 
self.weights[j][i]
                for k in range(self.input_nodes):
                    delta = self.learning_rate * self.values[k] * hiddenErrorGradient
                    self.weights[k][j] += delta
                    delta = self.learning_rate * -1 * hiddenErrorGradient
                self.thresholds[j] += delta
            delta = self.learning_rate * -1 * outputErrorGradient
            self.thresholds[i] += delta
        return sumOfSquaredErrors
class sampleMaker:
    def __init__(self, network):
        self.counter = 0
        self.network = network
    def setXor(self, x):
        if x == 0:
            self.network.values[0] = 1
            self.network.values[1] = 1
            self.network.expectedValues[4] = 0
        elif x == 1:
            self.network.values[0] = 0
            self.network.values[1] = 1
            self.network.expectedValues[4] = 1
        elif x == 2:
            self.network.values[0] = 1
            self.network.values[1] = 0
            self.network.expectedValues[4] = 1
        else:
            self.network.values[0] = 0
            self.network.values[1] = 0
            self.network.expectedValues[4] = 0
    def setNextTrainingData(self):
        self.setXor(self.counter % 4)
        self.counter += 1
net = network(INPUT_NODES, HIDDEN_NODES, OUTPUT_NODES, LEARNING_RATE)
samples = sampleMaker(net)
for i in range(MAX_ITERATIONS):
    samples.setNextTrainingData()
    net.process()
    error = net.processErrors()
    if i > (MAX_ITERATIONS - 5):
        output = (net.values[0], net.values[1], net.values[4], net.expectedValues[4], error)
        print(output)
print(net.weights)
```

Output:



```
"C:\Users\ADITYA KUMAR\PyCharmMiscProject\.venv\Scripts\python.exe" "C:\Users\ADITYA KUMAR\PyCharmMiscProject\.venv\Lib\3practical.py"
Consider a single neuron perceptron with a single i/p: 0.1
Enter the learning coefficient: 1.0
Enter the input value: 1.0
Enter the target output: 1.0
Iteration: 1, Output: 1, Change in weight: 0.0, Adjusted weight: 0.1
Iteration: 2, Output: 1, Change in weight: 0.0, Adjusted weight: 0.1
Iteration: 3, Output: 1, Change in weight: 0.0, Adjusted weight: 0.1
Iteration: 4, Output: 1, Change in weight: 0.0, Adjusted weight: 0.1
Iteration: 5, Output: 1, Change in weight: 0.0, Adjusted weight: 0.1
Iteration: 6, Output: 1, Change in weight: 0.0, Adjusted weight: 0.1
Iteration: 7, Output: 1, Change in weight: 0.0, Adjusted weight: 0.1
Iteration: 8, Output: 1, Change in weight: 0.0, Adjusted weight: 0.1
Iteration: 9, Output: 1, Change in weight: 0.0, Adjusted weight: 0.1
Iteration: 10, Output: 1, Change in weight: 0.0, Adjusted weight: 0.1

Process finished with exit code 0
```

PRACTICAL 3A

Aim: Write a program to implement **Hebb's Rule.**

Code:

```
def main()

    w = float(input("Consider a single neuron perceptron with a single i/p: "))

    d = float(input("Enter the learning coefficient: "))

    x = float(input("Enter the input value: "))  # Get the input value from the user

    t = float(input("Enter the target output: "))  # Get the target output from the user

    for i in range(10):

        net = x + w

        a = 1 if net >= 0 else 0

        div = d * (t - a)

        w = w + div

        print(f"Iteration: {i+1}, Output: {a}, Change in weight: {div}, Adjusted weight: {w}")

    if __name__ == "__main__":

        main()
```

Output:

```
"C:\Users\ADITYA KUMAR\PyCharmMiscProject\.venv\Scripts\python.exe" "C:\Users\ADITYA KUMAR\PyCharmMiscProject\.venv\Lib\2b.py"
Neural Network Program
(np.float64(1.0), np.float64(1.0), np.float64(0.007078309998574103), np.float64(0.0), 5.010247243591412e-05)
(np.float64(0.0), np.float64(1.0), np.float64(0.9828882680512363), np.float64(1.0), 0.00029281137028634024)
(np.float64(1.0), np.float64(0.0), np.float64(0.9750988347041768), np.float64(1.0), 0.0006200680330899098)
(np.float64(0.0), np.float64(0.0), np.float64(0.03243291886327582), np.float64(0.0), 0.0010518942259918323)
[[ 0.          0.          4.77121623 -5.98166087  0.        ]
 [ 0.          0.         -5.95518495  7.66710815  0.        ]
 [ 0.          0.          0.          1.93019719 10.85155468]
 [ 0.          0.          0.          0.         11.46595604]
 [ 0.          0.          0.          0.          0.        ]]
[0.         0.         0.75829229 4.43821716 6.99158132]

Process finished with exit code 0
```

PRACTICAL 3B

Aim: Write a program to implement **Delta rule.**

Code:

```
def main():
    inputs = []
    weights = []
    desired_output = 0.0
    # Initialize weights
    for i in range(3):
        weight = float(input(f"Initialize weight vector {i}: "))
        weights.append(weight)
    # Get desired output
    desired_output = float(input("Enter the desired output: "))
    # Perceptron training loop
    while True:
        # Calculate net input (simplified for this example)
        net_input = sum(w * x for w, x in zip(weights, inputs))
        # Calculate output (simplified for this example)
        output = 1 if net_input >= 0 else 0
        # Calculate error
        delta = desired_output - output
        if delta == 0:
            print("\nOutput is correct")
            break
   # Adjust weights based on error
        for i in range(3):
            weights[i] = weights[i] + delta * inputs[i]
        print(f"\nValue of delta is: {delta}")
        print("Weights have been adjusted")
if __name__ == "__main__":
    main()
```

Output:

```
"C:\Users\ADITYA KUMAR\PyCharmMiscProject\.venv\Scripts\python.exe" "C:\Users\ADITYA KUMAR\PyCharmMiscProject\.venv\Lib\3bpractical.py"
Initialize weight vector 0: 0.5
Initialize weight vector 1: -0.2
Initialize weight vector 2: 0.1
Enter the desired output: 1

Output is correct

Process finished with exit code 0
```

PRACTICAL 4A

Aim: Write a program for **Back Propagation Algorithm.**

Code:

```
import math

def main():

#Initial setup
    coeff = 0.1
    s = [{'val': 0, 'out': 0, 'wo': 0, 'wi': 0, 'top': 0} for _ in range(3)]

    #Taking input values
    for i in range(3):
        s[i]['val'] = float(input("Enter the input value to target output: "))
        s[i]['top'] = int(input("Enter the target value: "))

    i = 0
    while i != 3:
        if i == 0:
            s[i]['wo'] = -1.0
            s[i]['wi'] = -0.3
        else:
            s[i]['wo'] = s[i-1]['wo']
            s[i]['wi'] = s[i-1]['wi']

            s[i]['aop'] = s[i]['wo'] + (s[i]['wi'] * s[i]['val'])
            s[i]['out'] = s[i]['aop']
            delta = (s[i]['top'] - s[i]['out']) * s[i]['out'] * (1 - s[i]['out'])
            corr = coeff * delta * s[i]['out']
            s[i]['wo'] += corr
            s[i]['wi'] += corr
            i += 1

        print("VALUE\tTarget\tActual\two\twi")
        for i in range(3):
            print(f"{s[i]['val']}\t{s[i]['top']}\t{s[i]['out']}\t{s[i]['wo']}\t{s[i]['wi']}")

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

Output:

```
"C:\Users\ADITYA KUMAR\PyCharmMiscProject\.venv\Scripts\python.exe" "C:\Users\ADITYA KUMAR\PyCharmMiscProject\.venv\Lib\4a.py"
Enter the input value to target output: 0.1
Enter the target value: 3
Enter the input value to target output: 0.2
Enter the target value: 3
Enter the input value to target output: 03
Enter the target value: 3
VALUE   Target  Actual  wo  wi
0.1 3   -1.03   -0.13208831899999984    0.5679116810000002
0.2 3   -0.018505982799999793   -0.13198303074358506    0.568016969256415
3.0 3   1.57206787702566    -0.33386508378225666    0.3661349162177434

Process finished with exit code 0
```

PRACTICAL 4B

Aim: Write a program for **error Backpropagation algorithm.**

Code:

```
import math

def main():

    c = float(input("Enter the learning coefficient of network c: "))
    w10, b10 = map(float, input("Enter the input weights/base of first network: ").split())
    w20, b20 = map(float, input("Enter the input weights/base of second network: ").split())

    p = float(input("Enter the input value p: "))
    t = float(input("Enter the target value t: "))

    # Step 1: Propagation of signal through network
    n1 = w10 * p + b10
    a1 = math.tanh(n1)
    n2 = w20 * a1 + b20
    a2 = math.tanh(n2)
    e = t - a2

    # Back Propagation of Sensitivities
    s2 = -2 * (1 - a2 ** 2) * e
    s1 = (1 - a1 ** 2) * w20 * s2

    # Updation of weights and bases
    w21 = w20 - (c * s2 * a1)
    w11 = w10 - (c * s1 * -1)
    b21 = b20 - (c * s2)
    b11 = b10 - (c * s1)
    print("The updated weight of first network w11 =", w11)
    print("The updated weight of second network w21 =", w21)
    print("The updated base of first network b11 =", b11)
    print("The updated base of second network b21 =", b21)

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

Output:

```
C:\Users\RPIMS\PycharmProjects\pythonProject\.venv\Scripts\python.exe C:\Users\RPIMS\PycharmProjects\pythonProject\.venv\practicle4b.py
Enter the input weight/base of second n/w (w10): 1
Enter the input base of second n/w (b10): 2
Enter the input weight/base of second n/w (w20): 3
Enter the input base of second n/w (b20): 1
Enter the learning coefficient of n/w (c): 1
Enter the value of p: 1
Enter the value of t: 1
The updated weight of first n/w w11 = 0.9999999435070259
The updated weight of second n/w w21 = 3.0000018992293604
The updated base of first n/w b11 = 2.000000056492974
The updated base of second n/w b21 = 1.000001908668195

Process finished with exit code 0
```

PRACTICAL 5A

Aim: Write a program for **Hopfield Network.**

Code:

```
class Neuron:
    def __init__(self, weights):
        self.weightv = weights

    def act(self, m, x):
        a = 0
        for I in range(m):
            a += x[i] * self.weightv[i]
        return a

class Network:
    def __init__(self, a, b, c, d):
        self.nrn = [Neuron(a), Neuron(b), Neuron(c), Neuron(d)]
        self.output = [0] * 4

    def threshld(self, k):
        return 1 if k >= 0 else 0

    def activation(self, patrn):
        for I in range(4):
            for j in range(4):
                print(f"\n nrn[{i}].weightv[{j}] is {self.nrn[i].weightv[j]}")
            self.nrn[i].activation = self.nrn[i].act(4, patrn)
            print(f"\nactivation is {self.nrn[i].activation}")
            self.output[i] = self.threshld(self.nrn[i].activation)
            print(f"\noutput value is {self.output[i]}\n")

def main():
    patrn1 = [1, 0, 1, 0]
    wt1 = [0, -3, 3, -3]
    wt2 = [-3, 0, -3, 3]
    wt3 = [3, -3, 0, -3]
    wt4 = [-3, 3, -3, 0]

    print("\nTHIS PROGRAM IS FOR A HOPFIELD NETWORK WITH A SINGLE LAYER OF")
    print("4 FULLY INTERCONNECTED NEURONS. THE NETWORK SHOULD RECALL THE")
    print("PATTERNS 1010 AND 0101 CORRECTLY.\n")

    # Create the network by calling its constructor.
    H1 = Network(wt1, wt2, wt3, wt4)

    # Present a pattern to the network and get the activations of the neurons
    h1.activation(patrn1)

    # Check if the pattern given is correctly recalled and give message
    for I in range(4):
        if h1.output[i] == patrn1[i]:
            print(f"\n pattern= {patrn1[i]}  output = {h1.output[i]}  component matches")
        else:
            print(f"\n pattern= {patrn1[i]}  output = {h1.output[i]}  discrepancy occurred")
```

```
    print("\n\n")
    patrn2 = [0, 1, 0, 1]
    h1.activation(patrn2)
    for I in range(4):
        if h1.output[i] == patrn2[i]:
            print(f"\n pattern= {patrn2[i]}  output = {h1.output[i]}  component matches")
        else:
            print(f"\n pattern= {patrn2[i]}  output = {h1.output[i]}  discrepancy occurred")

if __name__ == "__main__":
    main()

import math

class Neuron:
    def __init__(self, weights=None):
        if weights is None:
            weights = [0] * 4
        self.weightv = weights
        self.activation = 0

    def act(self, m, x):
        a = 0
        for I in range(m):
            a += x[i] * self.weightv[i]
        return a

class Network:
    def __init__(self, a, b, c, d):
        self.nrn = [Neuron(a), Neuron(b), Neuron(c), Neuron(d)]
        self.output = [0] * 4

    def threshld(self, k):
        return 1 if k >= 0 else 0

    def activation(self, patrn):
        for I in range(4):
            for j in range(4):
                print(f"\n nrn[{i}].weightv[{j}] is {self.nrn[i].weightv[j]}")
            self.nrn[i].activation = self.nrn[i].act(4, patrn)
            print(f"\nactivation is {self.nrn[i].activation}")
            self.output[i] = self.threshld(self.nrn[i].activation)
            print(f"\noutput value is {self.output[i]}\n")

def main():
    patrn1 = [1, 0, 1, 0]
    wt1 = [0, -3, 3, -3]
    wt2 = [-3, 0, -3, 3]
    wt3 = [3, -3, 0, -3]
    wt4 = [-3, 3, -3, 0]

    print("\nTHIS PROGRAM IS FOR A HOPFIELD NETWORK WITH A SINGLE LAYER OF")
    print("4 FULLY INTERCONNECTED NEURONS. THE NETWORK SHOULD RECALL THE")
    print("PATTERNS 1010 AND 0101 CORRECTLY.\n")
```

```
    # Create the network by calling its constructor.
    H1 = Network(wt1, wt2, wt3, wt4)

    # Present a pattern to the network and get the activations of the neurons
    h1.activation(patrn1)

    # Check if the pattern given is correctly recalled and give message
    for I in range(4):
        if h1.output[i] == patrn1[i]:
            print(f"\n pattern= {patrn1[i]}  output = {h1.output[i]}  component matches")
        else:
            print(f"\n pattern= {patrn1[i]}  output = {h1.output[i]}  discrepancy occurred")
        print("\n\n")
        patrn2 = [0, 1, 0, 1]
        h1.activation(patrn2)
        for I in range(4):
            if h1.output[i] == patrn2[i]:
                print(f"\n pattern= {patrn2[i]}  output = {h1.output[i]}  component matches")
            else:
                print(f"\n pattern= {patrn2[i]}  output = {h1.output[i]}  discrepancy occurred")
if __name__ == "__main__":
    main()
```

Output:

THIS PROGRAM IS FOR A HOPFIELD NETWORK WITH A SINGLE LAYER OF

4 FULLY INTERCONNECTED NEURONS. THE NETWORK SHOULD RECALL THE

PATTERNS 1010 AND 0101 CORRECTLY.

nrn[0].weightv[0] is 0

nrn[0].weightv[1] is -3

nrn[0].weightv[2] is 3

nrn[0].weightv[3] is -3

activation is 3

output value is 1

nrn[1].weightv[0] is -3

nrn[1].weightv[1] is 0

nrn[1].weightv[2] is -3

nrn[1].weightv[3] is 3

activation is -6

output value is 0

nrn[2].weightv[0] is 3

nrn[2].weightv[1] is -3

nrn[2].weightv[2] is 0

nrn[2].weightv[3] is -3

activation is 3

output value is 1

nrn[3].weightv[0] is -3

nrn[3].weightv[1] is 3

nrn[3].weightv[2] is -3

nrn[3].weightv[3] is 0

activation is -6

output value is 0

pattern= 1 output = 1 component matches

pattern= 0 output = 0 component matches

pattern= 1 output = 1 component matches

pattern= 0 output = 0 component matches

nrn[0].weightv[0] is 0

nrn[0].weightv[1] is -3

nrn[0].weightv[2] is 3

nrn[0].weightv[3] is -3

activation is 3

output value is 1

nrn[2].weightv[0] is 3

nrn[2].weightv[1] is -3

nrn[2].weightv[2] is 0

nrn[2].weightv[3] is -3

activation is -6

output value is 0

nrn[3].weightv[0] is -3

nrn[3].weightv[1] is 3

nrn[3].weightv[2] is -3

nrn[3].weightv[3] is 0

activation is 3

output value is 1

pattern= 0 output = 0 component matches

pattern= 1 output = 1 component matches

pattern= 0 output = 0 component matches

pattern= 1 output = 1 component matches

nrn[0].weightv[1] is -3

nrn[0].weightv[2] is 3

nrn[0].weightv[3] is -3

activation is -6

output value is 0

nrn[2].weightv[0] is 3

nrn[2].weightv[1] is -3

nrn[2].weightv[2] is 0

nrn[2].weightv[3] is -3

output value is 1

nrn[3].weightv[0] is -3

nrn[3].weightv[1] is 3

nrn[3].weightv[3] is 0

activation is -6

output value is 0

pattern= 1 output = 1 component matches

pattern= 0 output = 0 component matches

pattern= 1 output = 1 component matches

pattern= 0 output = 0 component matches

nrn[0].weightv[0] is 0

nrn[0].weightv[1] is -3

nrn[0].weightv[2] is 3

nrn[0].weightv[3] is -3

activation is -6

output value is 0

nrn[1].weightv[0] is -3

nrn[1].weightv[1] is 0

nrn[1].weightv[3] is 3

activation is 3

activation is -6

output value is 0

nrn[3].weightv[0] is -3

nrn[3].weightv[1] is 3

nrn[3].weightv[2] is -3

nrn[3].weightv[3] is 0

activation is 3

output value is 1

pattern= 0 output = 0 component matches

pattern= 1 output = 1 component matches

pattern= 0 output = 0 component matches

pattern= 1 output = 1 component matches

PRACTICAL 5B

Aim: Write a program for **Radial Basis function.**

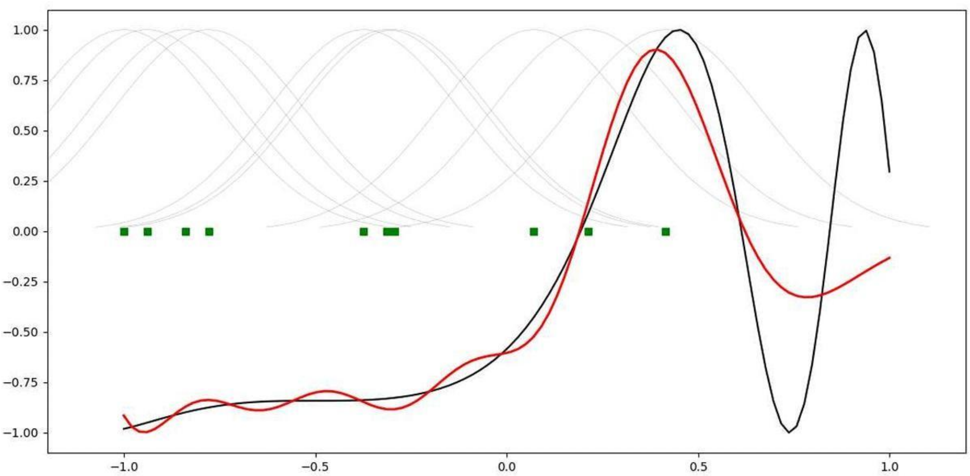
Code:

```
import numpy as np
from scipy.linalg import norm, pinv
import matplotlib.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 I 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 1 “””

        # 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(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
```

```
if __name__ == '__main__':
    # ----- 1D Example ------------------------------------------------
    n = 100
    x = np.mgrid[-1:1:complex(0, n)].reshape(n, 1)
    # set y and add random noise
    y = np.sin(3 * (x + 0.5) ** 3 – 1)
    # y += np.random.normal(0, 0.1, y.shape)

    # 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-')

    # plot learned model
    plt.plot(x, z, 'r-', linewidth=2)

    # plot rbfs
    plt.plot([c[0] for c in rbf.centers], np.zeros(rbf.numCenters), 'gs')

    for c in rbf.centers:
        # RF prediction lines
        cx = np.arange(c – 0.7, c + 0.7, 0.01)
        cy = [rbf._basisfunc(np.array([cx_]), np.array([c])) for cx_ in cx]
        plt.plot(cx, cy, '-', color='gray', linewidth=0.2)

    plt.xlim(-1.2, 1.2)
    plt.show()
```

Output:

```
[
[0.0229, 0.0001, 0.8114, 0.0000, 0.9710, 0.0000, 0.0434, 0.0183, 1.0000, 0.6736],
[0.0286, 0.0001, 0.8522, 0.0000, 0.9870, 0.0000, 0.0529, 0.0229, 0.9967, 0.7214],
[0.0353, 0.0002, 0.8891, 0.0000, 0.9967, 0.0000, 0.0642, 0.0286, 0.9870, 0.7676],
[0.0434, 0.0003, 0.9216, 0.0000, 1.0000, 0.0000, 0.0773, 0.0353, 0.9710, 0.8114],
[0.0529, 0.0004, 0.9491, 0.0000, 0.9967, 0.0000, 0.0925, 0.0434, 0.9491, 0.8522],]
```



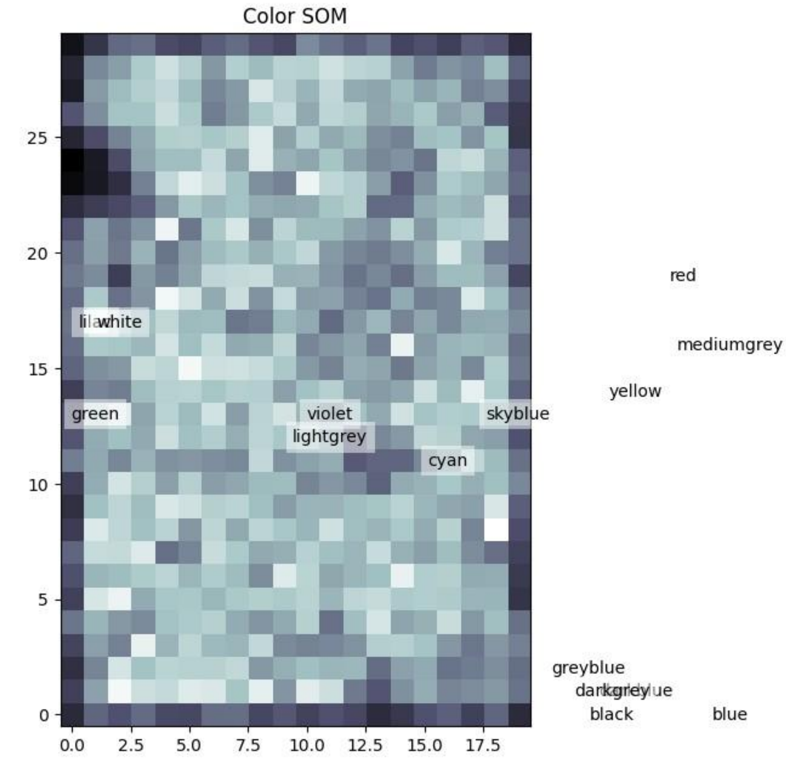
PRACTICAL 6A

Aim: Kohonen Self organizing map.

Code:

```
import numpy as np
import matplotlib.pyplot as pl
from minisom import MiniSom

# Define the color data (RGB values)
colors = np.array(
    [[0., 0., 0.],
    [0., 0., 1.],
    [0., 0., 0.5],
    [0.125, 0.529, 1.0],
    [0.33, 0.4, 0.67],
    [0.6, 0.5, 1.0],
    [0., 1., 0.],
    [1., 0., 0.],
    [0., 1., 1.],
    [1., 0., 1.],
    [1., 1., 0.],
    [1., 1., 1.],
    [.33, .33, .33],
    [.5, .5, .5],
    [.66, .66, .66]])

    # Define corresponding color names
    color_names = [
        'black', 'blue', 'darkblue', 'skyblue',
        'greyblue', 'lilac', 'green', 'red',
        'cyan', 'violet', 'yellow', 'white',
        'darkgrey', 'mediumgrey', 'lightgrey'
    ]

    # Initialize and train the SOM (Self-Organizing Map)
    som = MiniSom(20, 30, 3, sigma=1.0, learning_rate=0.5)  # Grid size (20, 30) and 3 input features (RGB)
    som.train(colors, 100)  # Train the SOM for 100 iterations

    # Plot the distance map of the SOM
    pl.imshow(som.distance_map().T, cmap='bone', origin='lower')

    # Map each color to its corresponding position on the SOM
    for i, color in enumerate(colors):
        x, y = som.winner(color)  # Find the best matching unit for the color
        pl.text(y, x, color_names[i], ha='center',
    va='center', bbox=dict(facecolor='white',
    alpha=0.5, lw=0))

    # Display the plot with the color names
    pl.title('Color SOM')
    pl.show()
```

Output:


Color SOM
white
green
violet
lightgrey
cyan
skyblue
red
mediumgrey
yellow
greyblue
black
blue

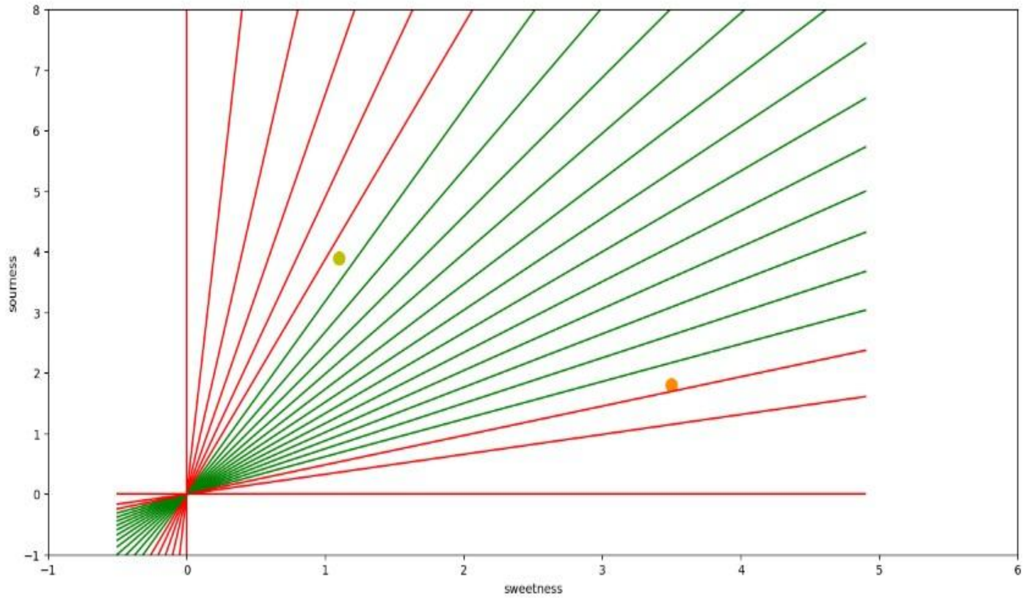
PRACTICAL 7A

Aim: Write a program for **Linear separation.**

Code:

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

def create_distance_function(a, b, c):

    def distance(x, y):
        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.absolute(nom) / np.sqrt(a ** 2 + b ** 2), pos)

    return distance

points = [(3.5, 1.8), (1.1, 3.9)]
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)
colors = ["r", ""]  # for the samples
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)
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)
    # print("x: ", x, "slope: ", slope)
    Y = slope * X

    results = []
    for point in points:
        results.append(dist4line1(*point))
    # print(slope, results)
    if (results[0][1] != results[1][1]):
        ax.plot(X, Y, "g-")
    else:
        ax.plot(X, Y, "r-")
plt.show()
```

Output:



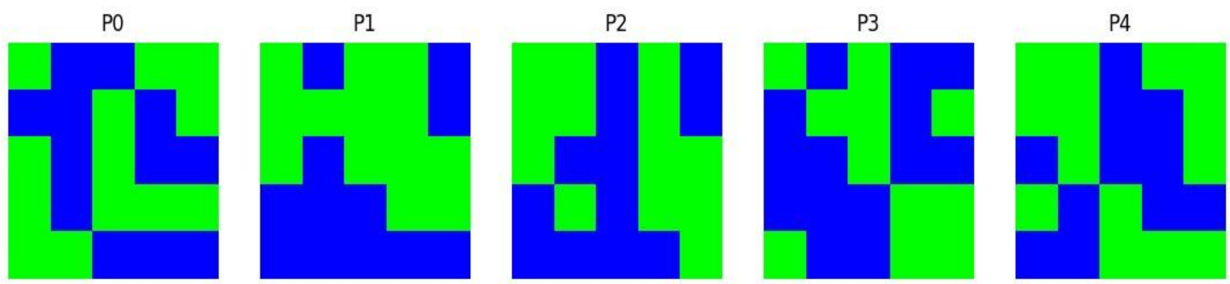
PRACTICAL 7B

Aim: Write a program for **Hopfield network model** for **associative memory.**

Code:

```
from neurodynex.hopfield_network import network, pattern_tools, plot_tools
import numpy as np
pattern_size = 5  # for a 5x5 grid, which is 25 neurons
hopfield_net = network.HopfieldNetwork(nr_neurons=pattern_size**2)
factory = pattern_tools.PatternFactory(pattern_size, pattern_size)
random_patterns = factory.create_random_pattern_list(nr_patterns=5, on_probability=0.5)
plot_tools.plot_pattern_list(random_patterns)
overlap_matrix = pattern_tools.compute_overlap_matrix(random_patterns)
plot_tools.plot_overlap_matrix(overlap_matrix)
hopfield_net.store_patterns(random_patterns)
noisy_pattern = pattern_tools.flip_percentage(random_patterns[0], percentage=0.3)
hopfield_net.set_state_from_pattern(noisy_pattern)
states = hopfield_net.run_with_monitoring(nr_steps=6)
states_as_patterns = factory.reshape_patterns(states)
plot_tools.plot_state_sequence_and_overlap(states_as_patterns, random_patterns, reference_idx=0,
suptitle="Network Dynamics with Noisy Pattern")
```

Output:



PRACTICAL 8A

Aim: Membership and Identity Operators | in, not in,

Code:

```
def overlapping(list1,list2):
    c=0
    d=0
    for i in list1:
        c+=1
    for i in list2:
        d+=1
    for i in range(0,c):
        for j in range(0,d):
            if(list1[i]==list2[j]):
                return 1
        return 0
    list1=[1,2,3,4,5]
    list2=[6,7,8,9]
    if(overlapping(list1,list2)):
        print("overlapping")
    else:
        print("not overlapping")
```

Output:

```
"C:\Users\RPIMS\PycharmProjects\project 1\.venv\Scripts\python.exe" "C:\Users\RPIMS\PycharmProjects\project 1\.venv\Lib\pract8a.py"
not overlapping

Process finished with exit code 0
```

PRACTICAL 8B

Aim: Membership and Identity Operators is, is not

Code:

```
x = 5
if (type(x) is int):
    print ("true")
else:
    print ("false")
```

Output:

```
"C:\Users\RPIMS\PycharmProjects\project 1\.venv\Scripts\python.exe" "C:\Users\RPIMS\PycharmProjects\project 1\.venv\Lib\pract8b.py"
true

Process finished with exit code 0
```

PRACTICAL 9A

Aim: Find ratios using **fuzzy logic.**

Code:

```
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 ("FuzzyWuzzy PartialRatio: ", fuzz.partial_ratio(s1, s2))
print ("FuzzyWuzzy TokenSortRatio: ", fuzz.token_sort_ratio(s1, s2))
print ("FuzzyWuzzy TokenSetRatio: ", fuzz.token_set_ratio(s1, s2))
print ("FuzzyWuzzy WRatio: ", 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:

```
"C:\Users\ADITYA KUMAR\PyCharmMiscProject\.venv\Scripts\python.exe" "C:\Users\ADITYA KUMAR\PyCharmMiscProject\.venv\Lib\9a.py"
FuzzyWuzzy Ratio: 86
FuzzyWuzzy PartialRatio:  86
FuzzyWuzzy TokenSortRatio:  86
FuzzyWuzzy TokenSetRatio:  87
FuzzyWuzzy WRatio:  86


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

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

Process finished with exit code 0
```

PRACTICAL 9B

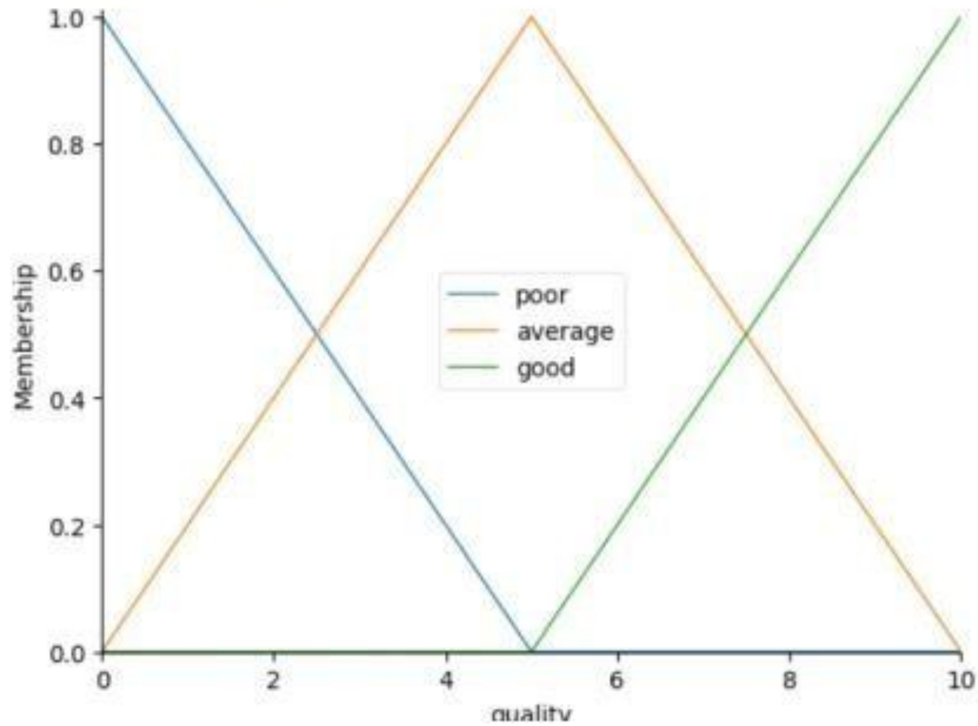
Aim: Solve **Tipping problem** using **fuzzy logic.**

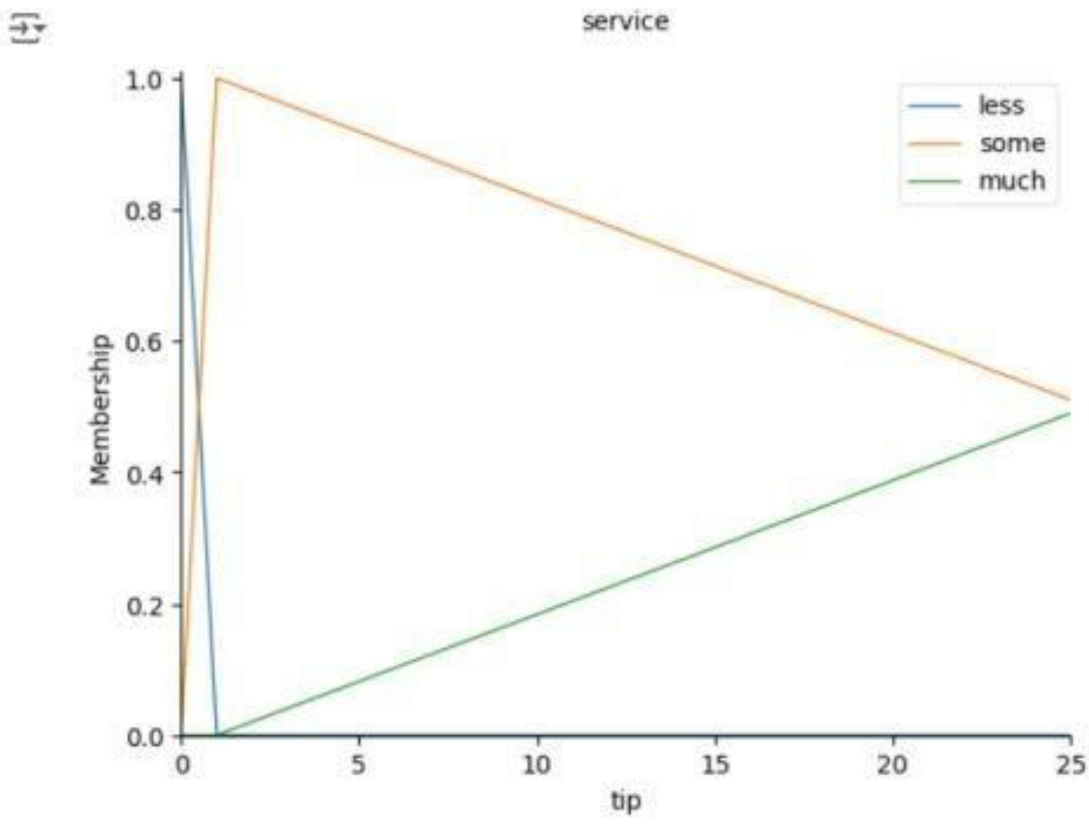
Code:

```
import numpy as np
import skfuzzy as fuzz
from skfuzzy import control as ctrl

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')

quality['poor']=fuzz.trimf(quality.universe,[0,0,5])
quality['average']=fuzz.trimf(quality.universe,[0,5,10])
quality['good']=fuzz.trimf(quality.universe,[5,10,10])

service['poor']=fuzz.trimf(service.universe,[0,0,5])
service['average']=fuzz.trimf(service.universe,[0,5,10])
service['good']=fuzz.trimf(service.universe,[5,10,10])

tip['less']=fuzz.trimf(tip.universe,[0,0,1])
tip['some']=fuzz.trimf(tip.universe,[0,1,50])
tip['much']=fuzz.trimf(tip.universe,[1,50,100])
rule1=ctrl.Rule(quality['poor']|service['poor'],tip['less'])
rule2=ctrl.Rule(service['average'],tip['some'])
rule3=ctrl.Rule(service['good']|quality['good'],tip['much'])
tipping_ctrl=ctrl.ControlSystem([rule1,rule2,rule3])
tipping=ctrl.ControlSystemSimulation(tipping_ctrl)
tipping.input['quality']= float(input(" : "))
tipping.input['service']= float(input(" : "))
tipping.compute()
print("Recommended tip:", tipping.output['tip'])
quality.view()
service.view()
tip.view()
```

Output:

```
: 6
: 10
Recommended tip: 17.000000000000004
```





PRACTICAL 10A

Aim: Implementation of **Simple genetic algorithm.**

Code:

```
import random
  POPULATION_SIZE = 100
GENES = '''abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOP
QRSTUVWXYZ 1234567890, .-;:_!"#%&/()=?@${[]}'''
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
        '''
        # chromosome for offspring
        child_chromosome = []
        for gp1, gp2 in zip(self.chromosome, par2.chromosome):

            # random probability
            prob = random.random()
                child_chromosome.append(gp1)

            elif prob < 0.90:
                child_chromosome.append(gp2)
              else:
                child_chromosome.append(self.mutated_genes())
        return Individual(child_chromosome)
    def cal_fitness(self):
        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
    generation = 1
    found = False
    population = []
    # create initial population
    for _ in range(POPULATION_SIZE):
                gnome = Individual.create_gnome()
                population.append(Individual(gnome))
    while not found:
        population = sorted(population, key = lambda x:x.fitness)
        if population[0].fitness <= 0:
            found = True
            break
        new_generation = []
```

```
        s = int((10*POPULATION_SIZE)/100)
        new_generation.extend(population[:s])
        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
        print("Generation: {}\tString: {}\tFitness: {}".\
              format(generation,
              "".join(population[0].chromosome),
              population[0].fitness))
        generation += 1
    print("Generation: {}\tString: {}\tFitness: {}".\
          format(generation,
          "".join(population[0].chromosome),
          population[0].fitness))
if __name__ == '__main__':
    main()
```

Output:

```
Generation: 1    String: tO{"-?=jH[k8=B4]Oe@}    Fitness: 18
Generation: 2    String: tO{"-?=jH[k8=B4]Oe@}    Fitness: 18
Generation: 3    String: .#lRWf9k_Ifslw #O$k_    Fitness: 17
Generation: 4    String: .-1Rq?9mHqk3Wo]3rek_    Fitness: 16
Generation: 5    String: .-1Rq?9mHqk3Wo]3rek_    Fitness: 16
Generation: 6    String: A#ldW) #lIkslw cVek)    Fitness: 14
Generation: 7    String: A#ldW) #lIkslw cVek)    Fitness: 14
Generation: 8    String: (, o x _x%Rs=, 6Peek3   Fitness: 13
                 .
                 .
Generation: 29    String: I lope Geeks#o, Geeks   Fitness: 3
Generation: 30    String: I loMe GeeksfoBGeeks    Fitness: 2
Generation: 31    String: I love Geeksfo0Geeks    Fitness: 1
Generation: 32    String: I love Geeksfo0Geeks    Fitness: 1
Generation: 33    String: I love Geeksfo0Geeks    Fitness: 1
Generation: 34    String: I love GeeksforGeeks    Fitness:
```

PRACTICAL 10B

Aim: Create two classes: **City and Fitness** using **Genetic algorithm.**

Code:

```
import numpy as np
import random

class City:
    """
    Represents a city with x and y coordinates.
    """
    def __init__(self, x: float, y: float):
        self.x = x
        self.y = y

    def distance_to(self, other_city: 'City') -> float:
        """
        Calculates the Euclidean distance to another city.
        """
        return np.sqrt((self.x - other_city.x) ** 2 + (self.y - other_city.y) ** 2)

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

class Fitness:
    """
    Calculates and stores the fitness of a route in a genetic algorithm.
    """
    def __init__(self, route: list):
        self.route = route
        self.distance = 0
        self.fitness = 0.0
    def calculate_distance(self) -> float:
        """
        Calculates the total distance of the route.
        """
        if self.distance == 0:
```

```
    total_distance = 0
    for i in range(len(self.route)):
        from_city = self.route[i]
        to_city = self.route[(i + 1) % len(self.route)]  # Loop back to the start
        total_distance += from_city.distance_to(to_city)
    self.distance = total_distance
return self.distance

def calculate_fitness(self) -> float:
    if self.fitness == 0:
        self.fitness = 1 / float(self.calculate_distance())
    return self.fitness

def __repr__(self):
    return f"Fitness(distance={self.distance}, fitness={self.fitness})"
# Example usage
if __name__ == "__main__":
    # Create example cities
    city1 = City(0, 0)
    city2 = City(3, 4)
    city3 = City(6, 0)
    # Create a route
    route = [city1, city2, city3]
    # Calculate fitness
    fitness = Fitness(route)
    print("Route:", route)
    print("Total Distance:", fitness.calculate_distance())
    print("Fitness Score:", fitness.calculate_fitness())
```

Output:

```
"C:\Users\ADITYA KUMAR\PyCharmMiscProject\.venv\Scripts\python.exe" "C:\Users\ADITYA KUMAR\PyCharmMiscProject\.venv\Lib\10b.py"
Route: [City(x=0, y=0), City(x=3, y=4), City(x=6, y=0)]
Total Distance: 16.0
Fitness Score: 0.0625

Process finished with exit code 0
```