INDEX

Sr. No.	Practical	Date	Signature
1A	Design a Simple Linear Neural Network Model	24-08-2024	
1B	Calculate Neural Net using both binary and bipolar sigmoidal function	24-08-2024	
2A	Implement AND/NOT function using McCulloch-Pits neuron	21-09-2024	
2B	Generate XOR function using McCulloch-Pits neuron net	21-09-2024	
3A	Program to implement Hebb's Rule	19-11-2024	
3B	Implement Delta Rule	19-11-2024	
4A	Back Propagation Algorithm	19-11-2024	
4B	Error Back Propagation Algorithm Learning	18-12-2024	
5A	Hopfield Network	18-12-2024	
5B	Radial Basis function	18-12-2024	
6A	Kohonen Self-Organizing Map	18-12-2024	
6B	Adaptive Resonance Theory	18-12-2024	
7A	Line Separation	18-12-2024	
8A	Membership and Identity operators in, not in	18-12-2024	
8B	Membership and Identity operators is, is not	18-12-2024	
9A	Ratios using Fuzzy Logic	03-01-2025	
9B	Tipping problem using Fuzzy Logic	03-01-2025	
10A	Simple Genetic Algorithm	11-01-2025	
10B	Classes creation using Genetic Algorithm	11-01-2025	

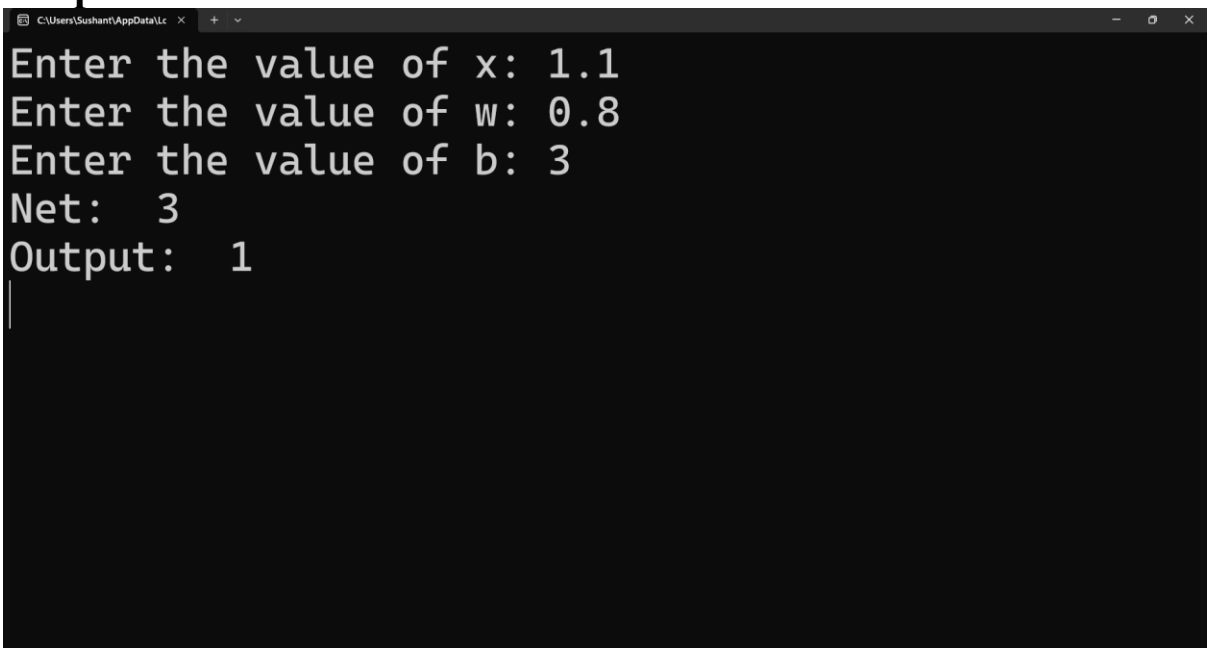
Practical 1

Practical 1A:
Aim: Design a Simple Linear Neural Network Model

Code:

```
x = float(input('Enter the value of x: '))
w = float(input('Enter the value of w: '))
b = float(input('Enter the value of b: '))
net = int(w*x+b)
if (net<0):
	out=0
elif(net>=0)&(net<=1):
	out=net
else:
	out=1
print('Net: ', net)
print('Output: ', out)
```

Output:

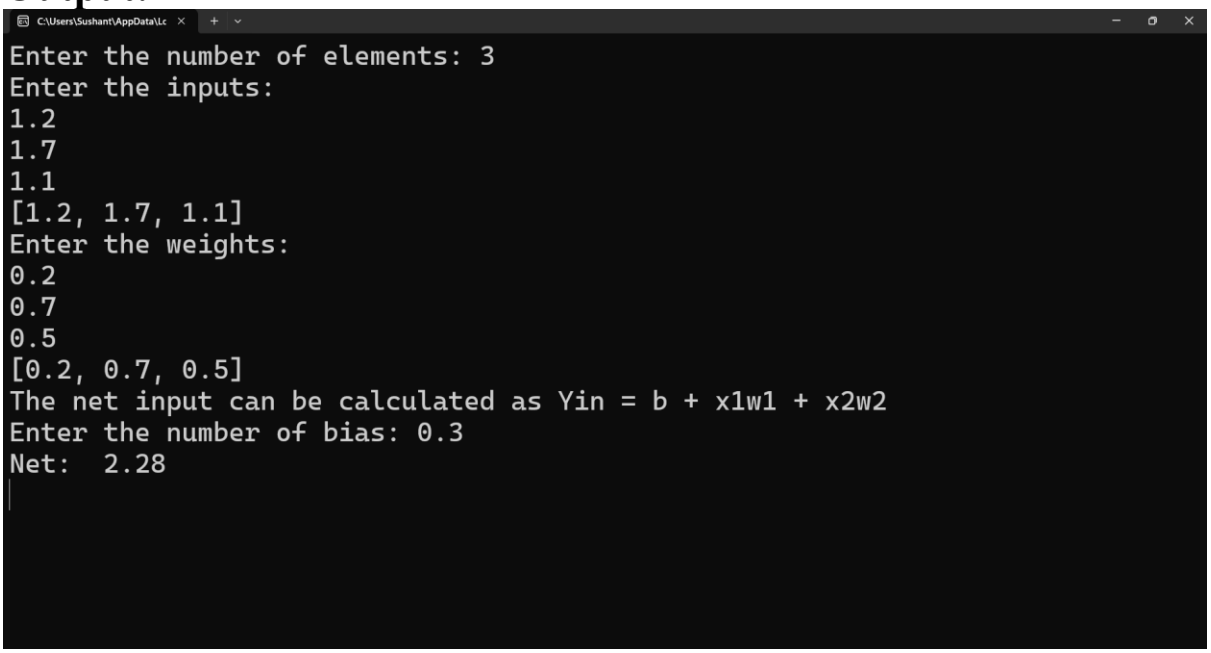


Practical 1B:
Aim: Calculate Neural Net using both binary and bipolar sigmoidal function

Code:

```
n = int(input('Enter the number 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 = b + x1w1 + x2w2')
b = float(input('Enter the number of bias: '))
Yin = []
for i in range(0, n):
    Yin.append(inputs[i]*weights[i])
net = round((sum(Yin) + b), 2)
print('Net: ', net)
```

Output:



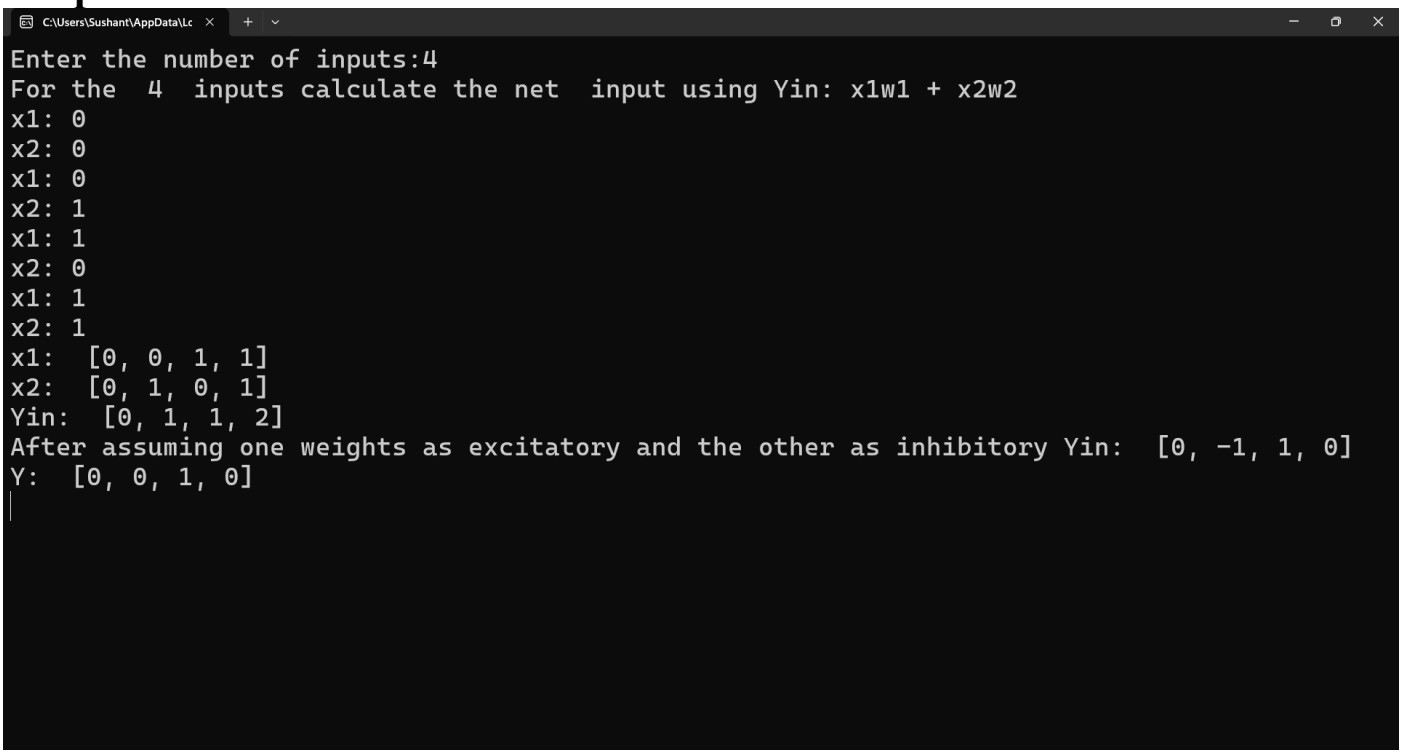
Practical 2

Practical 2A:
Aim: Implement AND/NOT function using McCulloch-Pits neuron.

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 weights as the 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)
    elif(Yin[i] < 1):
        ele = 0
        Y.append(ele)
print('Y: ', Y)
```

Output:


C:\Users\Sushant\AppData\Lc
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 weights as excitatory and the other as inhibitory Yin: [0, -1, 1, 0]
Y: [0, 0, 1, 0]

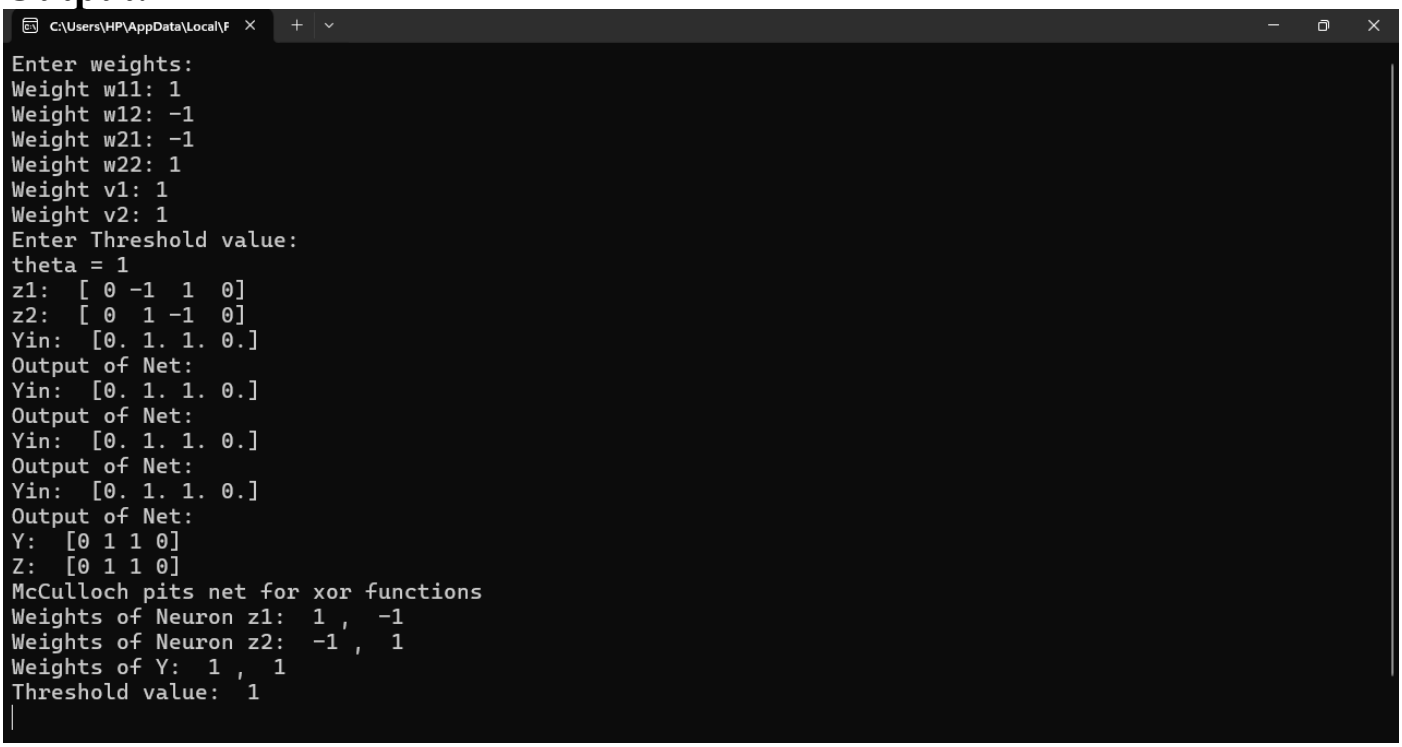
Practical 2B:
Aim: Generate XOR function using McCulloch-Pits neuron net.

Code:

```
import numpy as np
print('Enter weights:')
w11 = int(input('Weight w11: '))
w12 = int(input('Weight w12: '))
w21 = int(input('Weight w21: '))
w22 = int(input('Weight w22: '))
v1 = int(input('Weight v1: '))
v2 = int(input('Weight v2: '))
print('Enter Threshold value:')
theta = int(input('theta = '))
x1 = np.array([0,0,1,1])
x2 = np.array([0,1,0,1])
z = np.array([0,1,1,0])
con = 1
y1 = np.zeros((4, ))
y2 = np.zeros((4, ))
y = np.zeros((4, ))
while con == 1:
    zin1 = np.zeros((4, ))
    zin2 = np.zeros((4, ))
    zin1 = x1 * w11 + x2 * w21
    zin2 = x1 * w21 + x2 * w22
    print('z1: ', zin1)
    print('z2: ', zin2)
    for i in range(0, 4):
        if zin1[i] >= theta:
            y1[i] = 1
        else:
            y1[i] = 0
        if zin2[i] >= theta:
            y2[i] = 1
        else:
            y2[i] = 0
        yin = np.array([])
        yin = y1 * v1 + y2 * v2
    for i in range(0, 4):
        if yin[i] >= theta:
```

```
            y[i] = 1
        else:
            y[i] = 0
        print('Yin: ', yin)
        print('Output of Net: ')
    y = y.astype(int)
    print('Y: ', y)
    print('Z: ', z)
    if np.array_equal(y, z):
        con = 0
    else:
        print('Net isn\'t learning enter another set of weights and threshold value')
        w11 = input('Weights w11: ')
        w12 = input('Weight w12: ')
        w21 = input('Weight w21: ')
        w22 = input('Weight w22: ')
        v1 = input('Weiht v1: ')
        v2 = input('Weight v2: ')
        theta = input('theta: ')
print('McCulloch pits net for xor functions')
print('Weights of Neuron z1: ', w11, ', ', w21)
print('Weights of Neuron z2: ', w12, ', ', w22)
print('Weights of Y: ', v1, ', ', v2)
print('Threshold value: ', theta)
```

Output:



Practical 3

Practical 3A:
Aim: Program to implement Hebb's Rule

Code:

```
pip install numpy
import numpy as np
x1 = np.array([1, 1, 1, -1, 1, -1, 1, 1, 1])
x2 = np.array([1, 1, 1, 1, -1, 1, 1, 1, 1])
b = 0
y = np.array([1, -1])
wtold = np.zeros((9, ))
wtnew = np.zeros((9, ))
bias = 0
print('First input with target: 1')
for i in range(0, 9):
    wtold[i] = wtold[i] + x1[i] * y[0]
    wtnew = wtold
    b = b+ y[0]
print('New wt: ', wtnew)
print('Bias Value: ', b)
print('Second 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: 9
Second input with target = -1
New wt: [ 0. 0. 0. -2. 2. -2. 0. 0. 0.]
Bias Value: 0
```

Practical 3B:
Aim: Implement Delta Rule

Code:

```
import math
print('Using 3 inputs, 3 weights, 1 output')
x1 = [0.3, 0.5, 0.8]
w1 = [0.1, 0.1, 0.1]
t = 1
a = 0.1
diff = 1
yin = 0
while(diff > 0.4):
    for i in range(0, 3):
        yin = yin + (x1[i] * w1[i])
    yin = yin + 0.25
    yin = round(yin, 3)
    print('Yin: ', yin)
    print('Target: ', t)
    diff = t - yin
    diff = round(diff, 3)
    diff = math.fabs(diff)
print('Error: ', diff)
new1 = []
for i in range(0, 3):
    wnew1 = w1[i] + a * diff * x1[i]
    wnew1 = round(wnew1, 2)
    new1.append(wnew1)
w1 = new1
print('w1new=', w1)
```

Output:

```
Using 3 inputs, 3 weights, 1 output
Yin: 0.41
Target: 1
Yin: 0.82
Target: 1
Error: 0.18
w1new = [0.11, 0.11, 0.11]
```

Practical 4

Practical 4A:
Aim: Back Propagation Algorithm

Code:

```
import numpy as np
x = np.array(([2, 9], [1, 5], [3, 6]), dtype = float)
y = np.array(([92], [86], [89]), dtype = float)
x = x/np.amax(x, axis = 0)
y = y/100
class NN(object):
    def __init__(self):
        self.inputsize = 2
        self.outputsize = 1
        self.hiddensize = 3
        self.w1 = np.random.randn(self.inputsize, self.hiddensize)
        self.w2 = np.random.randn(self.hiddensize, self.outputsize)
    def forward(self, x):
        self.z1 = np.dot(x, self.w1)
        self.z2 = self.sigmoidal(self.z1)
        self.z3 = np.dot(self.z2, self.w2)
        op = self.sigmoidal(self.z3)
        return op
    def sigmoidal(self, s):
        return 1/(1/np.exp(-s))
obj = NN()
op = obj.forward(x)
print('Actual Output: ', str(op))
print('Target Output: ', str(y))
```

Output:

```
Actual Output: [[11.44977396] [ 3.39433893] [ 5.98908991]]
Target Output: [[0.92] [0.86] [0.89]]
```

Practical 4B:
Aim: Error Back Propagation Algorithm Learning

Code:

```
import math
a0 = -1
t = -1
w10 = float(input('Enter weight of first network: '))
b10 = float(input('Enter base of first network: '))
w20 = float(input('Enter weight of second network: '))
b20 = float(input('Enter base of second network: '))
c = float(input('Enter learning coefficient: '))
n1 = float(w10 * c + b10)
a1 = math.tanh(n1)
n2 = float(w20 * c + b20)
a2 = math.tanh(float(n2))
e = t - a2
s2 = 2 * (1 - a2 * a2) * e
s1 = (1 - a2 * a2) * w20 * s2
w21 = w20 - (c * s2 * a1)
w11 = w10 - (c * s1 * a0)
b21 = b20 - (c * s2)
b11 = b10 - (c * s1)
print('Updated weight of first n/w w11: ', w11)
print('Updated base of first n/w b10: ', b10)
print('Updated weight of first n/w w21: ', w21)
print('Updated base of first n/w b20: ', b20)
```

Output:

```
Enter weight of first network: 12
Enter base of first network: 35
Enter weight of second network: 23
Enter base of second network: 45
Enter learning coefficient: 11
Updated weight of first n/w w11: 12.0
Updated base of first n/w b10: 35.0
Updated weight of first n/w w21: 23.0
Updated base of first n/w b20: 45.0
```

Practical 5

Practical 5A:
Aim: Hopfield Network

Code:

```
#include "hop.h"
#include <iostream>
using namespace std;
neuron::neuron(int *j)
{
   int i;
   for (i = 0; i < 4; i++)
   {
      weightv[i] = *(j + i);
   }
}
int neuron::act(int m, int *x)
{
   int i;
   int a = 0;
   for (i = 0; i < m; i++)
   {
      a += x[i] * weightv[i];
   }
   return a;
}
int network::threshld(int k)
{
   if (k >= 0)
      return (1);
   else
      return (0);
}
network::network(int a[4], int b[4], int c[4], int d[4])
{
   nrn[0] = neuron(a);
   nrn[1] = neuron(b);
   nrn[2] = neuron(c);
   nrn[3] = neuron(d);
}
```

```
void network::activation(int *patrn)
{
   int i, j;
   for (i = 0; i < 4; i++)
   {
      for (j = 0; j < 4; j++)
      {
         cout << "\n nrn[" << i << "].weightv[" << j << "] is " <<
nrn[i].weightv[j];
      }
      nrn[i].activation = nrn[i].act(4, patrn);
      cout << "\nActivation is " << nrn[i].activation;
      output[i] = threshld(nrn[i].activation);
      cout << "\nOutput value is " << output[i] << "\n";
   }
}
int main()
{
   int patrn1[] = {1, 0, 1, 0}, i;
   int wt1[] = {0, -3, 3, -3};
   int wt2[] = {-3, 0, -3, 3};
   int wt3[] = {3, -3, 0, -3};
   int wt4[] = {-3, 3, -3, 0};
   cout << "THIS PROGRAM IS FOR A HOPFIELD NETWORK WITH A
SINGLE LAYER OF 4 FULLY INTERCONNECTED NEURONS. NETWORK
SHOULD RECALL THE PATTERNS 1010 AND 0101 CORRECTLY.\n";
   network h1(wt1, wt2, wt3, wt4);
   h1.activation(patrn1);
   for (i = 0; i < 4; i++)
   {
      if (h1.output[i] == patrn1[i])
      {
         cout << "\n pattern = " << patrn1[i] << " output = " << h1.output[i] << "
component matches";
      }
      else
      {
         cout << "\n pattern = " << patrn1[i] << " output = " << h1.output[i] << "
discrepancy occurred";
      }
   }
```

```
    cout << "\n\n";
    int patrn2[] = {0, 1, 0, 1};
    h1.activation(patrn2);
    for (i = 0; i < 4; i++)
    {
        if (h1.output[i] == patrn2[i])
        {
            cout << "\n pattern = " << patrn2[i] << " output = " << h1.output[i] << "
component matches";
        }
        else
        {
            cout << "\n pattern = " << patrn2[i] << " output = " << h1.output[i] << "
discrepancy occurred";
        }
    }
}
```

Code for hop.h:

```
#include <stdio.h>
#include <iostream>
#include <math.h>
class neuron
{
protected:
    int activation;
    friend class network;
public:
    int weightv[4];
    neuron() {};
    neuron(int *j);
    int act(int, int *);
};
class network
{
public:
    neuron nrn[4];
    int output[4];
    int threshld(int);
    void activation(int j[4]);
    network(int *, int *, int *, int *);
```

```
};
```

Output:

```
THIS PROGRAM IS FOR A HOPFIELD NETWORK WITH A SINGLE
LAYER OF 4 FULLY INTERCONNECTED NEURONS. 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 -3
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 -3
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 0

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 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 -3
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
```

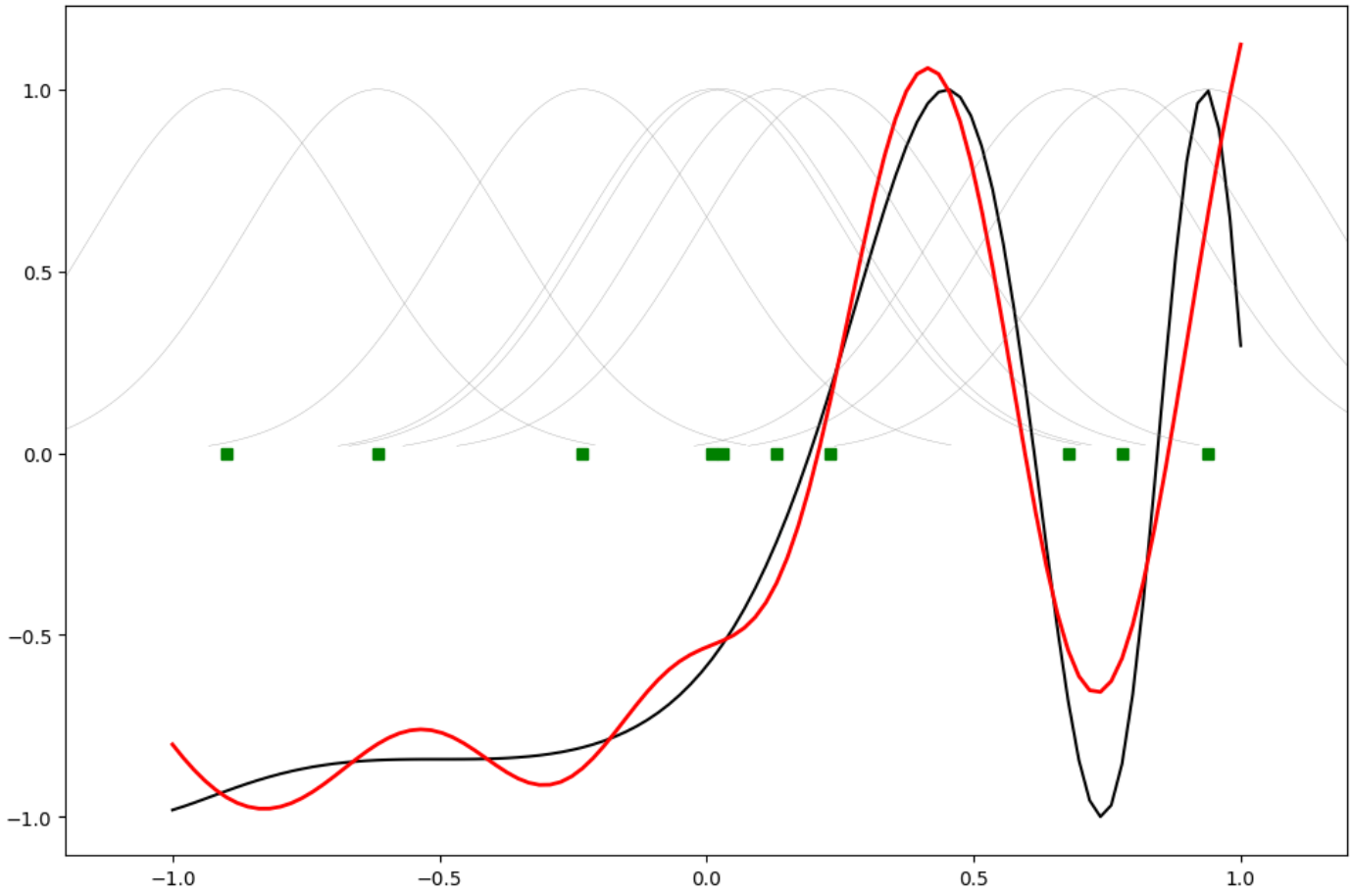
Practical 5B:
Aim: Radial Basis Function

Code:

```
import numpy as np
from scipy.linalg import norm, pinv
from matplotlib import pyplot as plt
from numpy import random, exp, zeros, dot, array, sin, arange
class RBF:
    def __init__(self, indim, numCenters, outdim):
        self.indim = indim
        self.outdim = outdim
        self.numCenters = numCenters
        self.centers = [random.uniform(-1, 1, indim) for i in range(numCenters)]
        self.beta = 8
        self.W = random.random((self.numCenters, self.outdim))
    def _basisfunc(self, c, d):
        assert len(d) == self.indim
        return exp(-self.beta * norm(c - d) ** 2)
    def _calcAct(self, X):
        G = 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):
        rnd_idx = random.permutation(X.shape[0])[:self.numCenters]
        self.centers = [X[i, :] for i in rnd_idx]
        G = self._calcAct(X)
        self.W = dot(pinv(G), Y)
    def test(self, X):
        G = self._calcAct(X)
        Y = dot(G, self.W)
        return Y
if __name__ == '__main__':
    n = 100
    x = np.mgrid[-1:1:complex(0, n)].reshape(n, 1)
    y = sin(3 * (x + 0.5) ** 3 - 1)
    rbf = RBF(1, 10, 1)
    rbf.train(x, y)
    z = rbf.test(x)
```

```
plt.figure(figsize=(12, 8))
plt.plot(x, y, 'k-')
plt.plot(x, z, 'r-', linewidth=2)
plt.plot(rbf.centers, zeros(rbf.numCenters), 'gs')
for c in rbf.centers:
    cx = arange(c - 0.7, c + 0.7, 0.01)
    cy = [rbf._basisfunc(array([cx_]), array([c])) for cx_ in cx]
    plt.plot(cx, cy, '-', color='gray', linewidth=0.2)
plt.xlim(-1.2, 1.2)
plt.show()
```

Output:



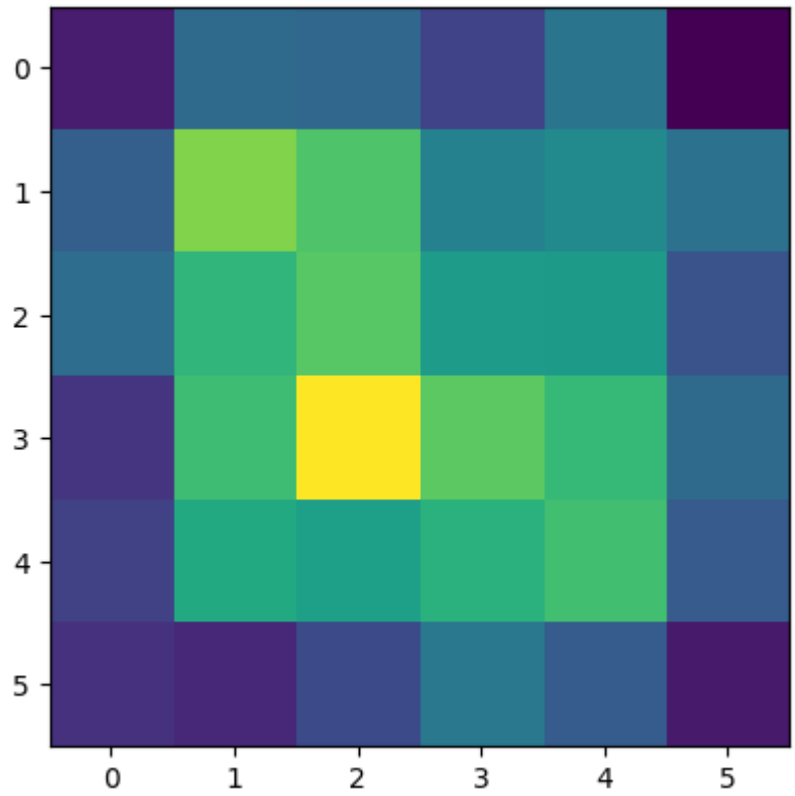
Practical 6

Practical 6A:
Aim: Kohonen Self Organizing Map

Code:

```
pip install minisom
pip install matplotlib
from minisom import MiniSom
import matplotlib.pyplot as plt
data = [[0.8, 0.55, 0.22, 0.03], [0.82, 0.5, 0.23, 0.03], [0.8, 0.54, 0.22, 0.03], [0.8, 0.53, 0.26, 0.03], [0.79, 0.56, 0.22, 0.03], [0.75, 0.6, 0.25, 0.03], [0.77, 0.59, 0.22, 0.03]]
som = MiniSom(6, 6, 4, sigma = 0.4, learning_rate = 0.5)
som.train_random(data, 100)
plt.imshow(som.distance_map())
plt.show()
```

Output:



Practical 6B:
Aim: Adaptive Response Theory

Code:

```
from __future__ import print_function, division
import numpy as np
class ART:
    def _init_(self, n=5, m=10, rho=0.5):
        self.F1 = np.ones(n)
        self.F2 = np.ones(m)
        self.Wf = np.random.random((m, n))
        self.Wb = np.random.random((n, m))
        self.rho = rho
        self.active = 0
    def learn(self, X):
        ''' Learn input X '''
        self.F2[...] = np.dot(self.Wf, X)
        I = np.argsort(self.F2[:self.active].ravel())[::-1]
        for i in I:
            d = (self.Wb[:, i] * X).sum() / X.sum()
            if d >= self.rho:
                self.Wb[:, i] *= X
                self.Wf[i, :] = self.Wb[:, i] / (0.5 + self.Wb[:, i].sum())
                return self.Wb[:, i], i
        if self.active < self.F2.size:
            i = self.active
            self.Wb[:, i] *= X
            self.Wf[i, :] = self.Wb[:, i] / (0.5 + self.Wb[:, i].sum())
            self.active += 1
            return self.Wb[:, i], i
        return None, None
def letter_to_array(letter):
    ''' Convert a letter to a numpy array '''
    shape = len(letter), len(letter[0])
    Z = np.zeros(shape, dtype=int)
    for row in range(Z.shape[0]):
        for column in range(Z.shape[1]):
            if letter[row][column] == '#':
                Z[row][column] = 1
    return Z
def print_letter(Z):
```

```
    ''' Print an array as if it was a letter '''
    for row in range(Z.shape[0]):
        for col in range(Z.shape[1]):
            if Z[row, col]:
                print('#', end="")
            else:
                print(' ', end="")
        print()
if __name__ == '__main__':
    np.random.seed(1)
    network = ART(5, 10, rho=0.5)
    data = [" O   ", " O O ", "  O  ", " O O ", "  O  ", " O O ", "  O  ", " OO O", "
OO  ", " OO O", " OO  ", "OOO  ", "OO   ", "O    ", "OO   ", "OOO  ", "OOOO ",
"OOOOO"]
    X = np.zeros(len(data[0]))
    for i in range(len(data)):
        for j in range(len(data[i])):
            X[j] = (data[i][j] == 'O')
        Z, k = network.learn(X)
        print("|%s|" % data[i], "-> class", k)
    A = letter_to_array([' #### ', '#    #', '#    #', '######', '#    #', '#    #', '#    #'])
    B = letter_to_array(['##### ', '#    #', '#    #', '##### ', '#    #', '#    #', '##### '])
    C = letter_to_array([' #### ', '#    #', '#     ', '#     ', '#     ', '#    #', ' #### '])
    D = letter_to_array(['##### ', '#    #', '#    #', '#    #', '#    #', '#    #', '##### '])
    E = letter_to_array(['######', '#     ', '#     ', '####  ', '#     ', '#     ', '######'])
    F = letter_to_array(['######', '#     ', '#     ', '####  ', '#     ', '#     ', '#     '])
    samples = [A, B, C, D, E, F]
    network = ART(6 * 7, 10, rho=0.15)
    for i in range(len(samples)):
        Z, k = network.learn(samples[i].ravel())
        print("%c" % (ord('A') + i), "-> class", k)
        print_letter(Z.reshape(7, 6))
```

Output:

```
| O   | -> class 0
| O O | -> class 1
|  O  | -> class 2
| O O | -> class 1
|  O  | -> class 3
| O O | -> class 1
|  O  | -> class 3
| OO O| -> class 4
| OO  | -> class 5
| OO O| -> class 6
| OO  | -> class 6
|OOO  | -> class 6
|OO   | -> class 7
|O    | -> class 8
|OO   | -> class 9
|OOO  | -> class 6
|OOOO | -> class None
|OOOOO| -> class None
A -> class 0
  ####
#       #
#       #
######
#       #
#       #
#       #
B -> class 0
  ####
#       #
#       #
#####
#       #
#       #
#
C -> class 0
  ####
#       #
#
#
#
```

# #

D -> class 0

 ####

# #

#

#

#

# #

E -> class 0

 ####

#

#

#

#

#

F -> class 0

 ####

#

#

#

#

#

Practical 7

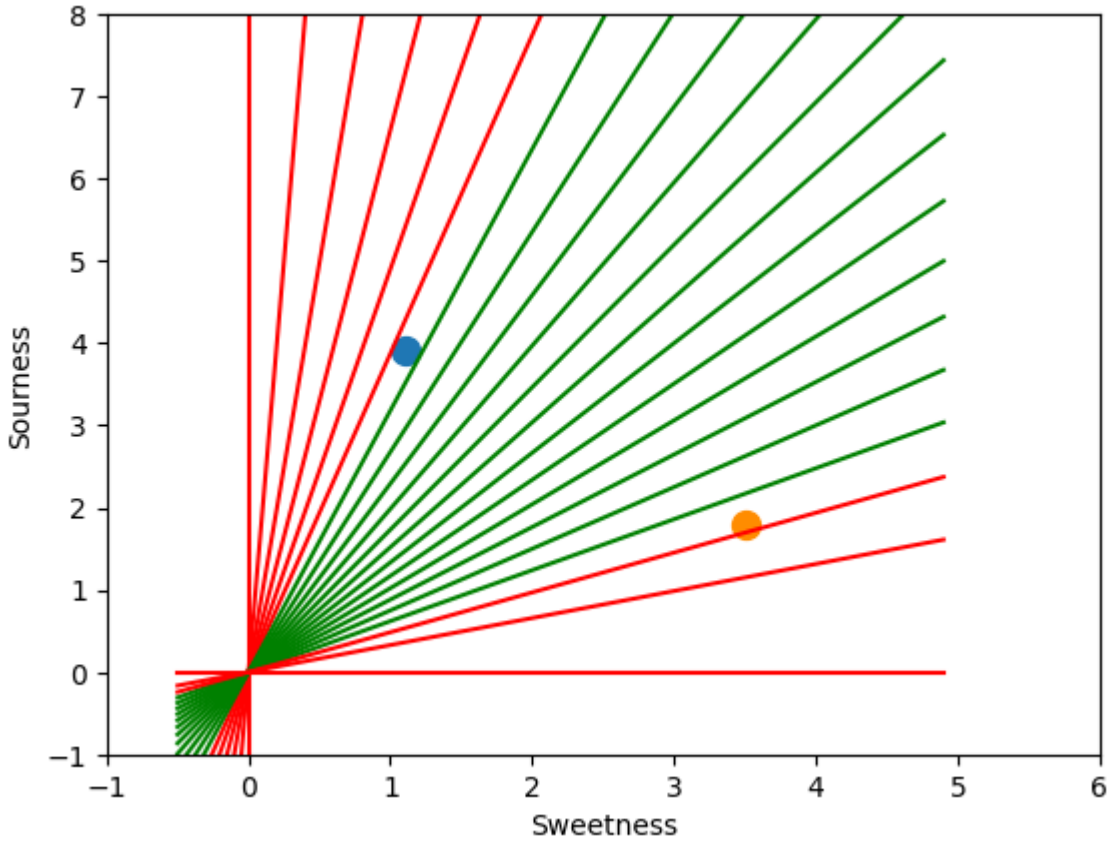
Practical 7A:
Aim: Line Separation

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", ""]
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, "o", 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)
    Y = slope * X
    results = []
    for point in points:
        results.append(dist4line1(*point))
```

```
    if (results[0][1] != results[1][1]):
        ax.plot(X, Y, 'g-')
    else:
        ax.plot(X, Y, 'r-')
plt.show()
```

Output:



Practical 8

Practical 8A:
Aim: Membership and Identity operators in, not in

Code:
```
list1 = []
print("Enter 5 numbers: ")
for i in range(0, 5):
    v = input()
    list1.append(v)
    list2 = []
print("Enter 5 numbers: ")
for i in range(0, 5):
    v = input()
    list2.append(v)
    flag = []
for i in list1:
    if i in list2:
        flag = 1
    if(flag == 1):
        print("Lists Overlap")
    else:
        print("Lists don't Overlap")
```

Output:
Enter 5 numbers: 1, 2, 3, 4, 5
Enter 5 numbers: 6, 7, 8, 9, 0
Lists don’t Overlap
Lists don’t Overlap
Lists don’t Overlap
Lists don’t Overlap
Lists don’t Overlap

Practical 8B:
Aim: Membership and Identity operators is, is not

Code:
```
x = 5
if(type(x) is not int):
   print('True')
else:
   print('False')
```
Output:
False

Code:
```
x = 5
if(type(x) is int):
   print('True')
else:
   print('False')
```
Output:
True

Code:
```
x = int(input('Enter value of x: '))
if(type(x) is int):
   print('True')
else:
   print('False')
```
Output:
Enter value of x: 0.01
False

Code:
```
x = int(input('Enter value of x: '))
if(type(x) is int):
   print('True')
else:
   print('False')
```
Output:
Enter value of x: 1
False

Practical 9

Practical 9A:
Aim: 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 Partial Ratio: ", fuzz.partial_ratio(s1, s2))
print ("FuzzyWuzzy Token Sort Ratio: ", fuzz.token_sort_ratio(s1, s2))
print ("FuzzyWuzzy Token Set Ratio: ", fuzz.token_set_ratio(s1, s2))
print ("FuzzyWuzzy W Ratio: ", fuzz.WRatio(s1, s2), '\n')
query = 'fuzzys for fuzzys'
choices = ['fuzzy for fuzzy', 'fuzzy fuzzy', 'g. for fuzzys']
print ("List of ratios: ", process.extract(query, choices), '\n')
print ("Best among the above list: ", process.extractOne(query, choices))
```

Output:

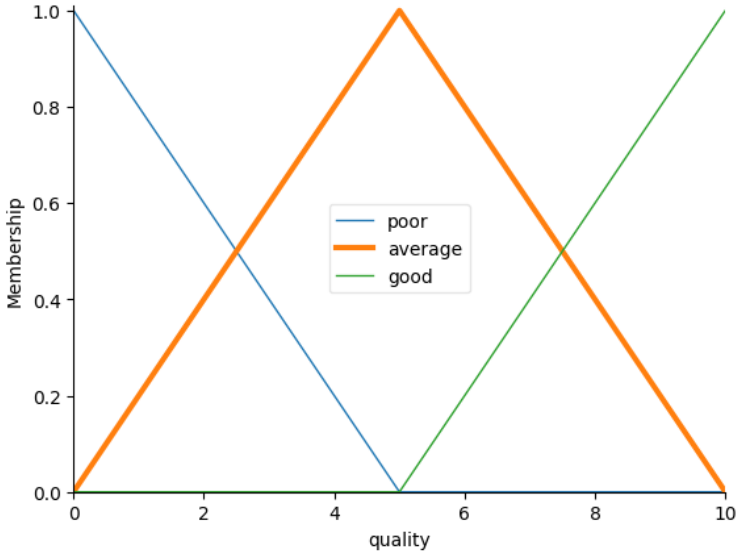
```
FuzzyWuzzy Ratio: 86
FuzzyWuzzy Partial Ratio:  86
FuzzyWuzzy Token Sort Ratio:  86
FuzzyWuzzy Token Set Ratio:  87
FuzzyWuzzy W Ratio:  86

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

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

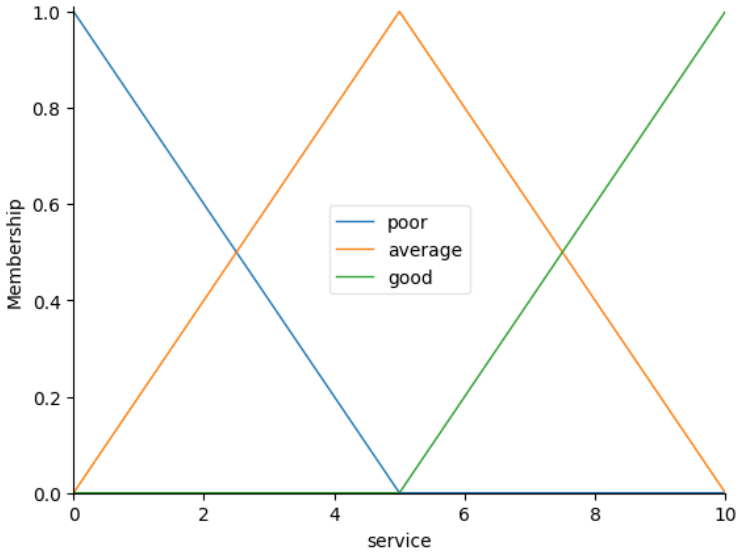
Practical 9B:
Aim: Tipping Problem using Fuzzy Logic

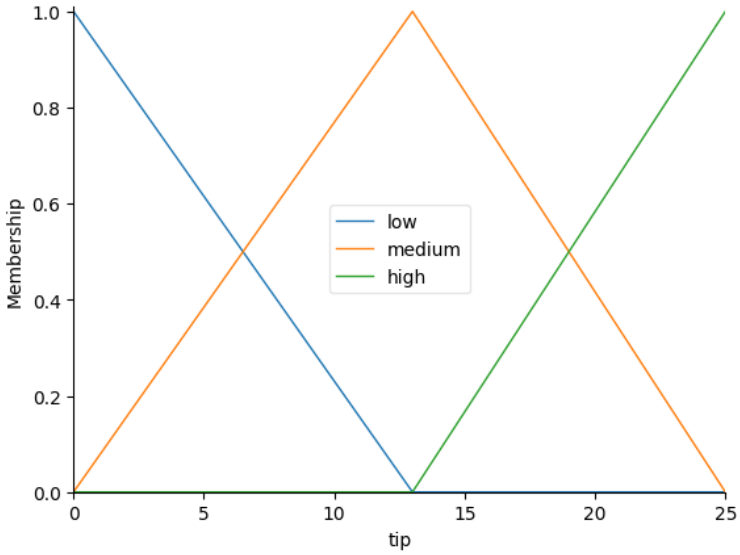
Code:

```
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.automf(3)
service.automf(3)
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])
quality['average'].view()
service.view()
tip.view()
```

Output:


1.0
0.8
0.6
0.4
0.2
0.0
Membership
poor
average
good
0
2
4
6
8
10
quality


1.0
0.8
0.6
0.4
0.2
0.0
Membership
poor
average
good
0
2
4
6
8
10
service


1.0
0.8
0.6
0.4
0.2
0.0
Membership
low
medium
high
0
5
10
15
20
25
tip

Practical 10

Practical 10A:
Aim: Simple Genetic Algorithm

Code:

```
import random
POPULATION_SIZE = 100
GENES =
'''abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ
1234567890, .-;:_!"#%&/()=?@${[]}'''
TARGET = "I love Coding"
class Individual(object):
    def __init__(self, chromosome):
        self.chromosome = chromosome
        self.fitness = self.cal_fitness()
    @classmethod
    def mutated_genes(cls):
        global GENES
        gene = random.choice(GENES)
        return gene
    @classmethod
    def create_gnome(cls):
        global TARGET
        gnome_len = len(TARGET)
        return [cls.mutated_genes() for _ in range(gnome_len)]
    def mate(self, par2):
        child_chromosome = []
        for gp1, gp2 in zip(self.chromosome, par2.chromosome):
            prob = random.random()
            if prob < 0.45:
                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
def main():
    global POPULATION_SIZE
    generation = 1
    found = False
    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: Y 3,v3Zg#K@Em    Fitness: 11
Generation: 2    String: Y 3,v3Zg#K@Em    Fitness: 11
Generation: 3    String: I}S/8P p@ui?w    Fitness: 10
Generation: 4    String: I}S/8P p@ui?w    Fitness: 10
Generation: 5    String: zZ#ovshCZuXLg    Fitness: 9
Generation: 6    String: zZ#ovshCZuXLg    Fitness: 9
```

Generation: 7 String: IjlCFe C@&i?g Fitness: 6
Generation: 8 String: IjlCFe C@&i?g Fitness: 6
Generation: 9 String: I lVLe C_&i:g Fitness: 5
Generation: 10 String: I lVve C@&i:g Fitness: 4
Generation: 11 String: I lVve C@&i:g Fitness: 4
Generation: 12 String: I lVve C@&i:g Fitness: 4
Generation: 13 String: I love CIGi?g Fitness: 3
Generation: 14 String: I love CIGi?g Fitness: 3
Generation: 15 String: I love CGdirg Fitness: 2
Generation: 16 String: I love CGdirg Fitness: 2
Generation: 17 String: I love CGdirg Fitness: 2
Generation: 18 String: I love CGdirg Fitness: 2
Generation: 19 String: I love CGdirg Fitness: 2
Generation: 20 String: I love CGdirg Fitness: 2
Generation: 21 String: I love CGdirg Fitness: 2
Generation: 22 String: I love CGdirg Fitness: 2
Generation: 23 String: I love CGdirg Fitness: 2
Generation: 24 String: I love CGdirg Fitness: 2
Generation: 25 String: I love CGdirg Fitness: 2
Generation: 26 String: I love CGdirg Fitness: 2
Generation: 27 String: I love CGdirg Fitness: 2
Generation: 28 String: I love CGdirg Fitness: 2
Generation: 29 String: I love CGdirg Fitness: 2
Generation: 30 String: I love CGdirg Fitness: 2
Generation: 31 String: I love CGdirg Fitness: 2
Generation: 32 String: I love CGdirg Fitness: 2
Generation: 33 String: I love CGdirg Fitness: 2
Generation: 34 String: I love CGdirg Fitness: 2
Generation: 35 String: I love CGdirg Fitness: 2
Generation: 36 String: I love CGding Fitness: 1
Generation: 37 String: I love CGding Fitness: 1
Generation: 38 String: I love CGding Fitness: 1
Generation: 39 String: I love CGding Fitness: 1
Generation: 40 String: I love CGding Fitness: 1
Generation: 41 String: I love Coding Fitness: 0

Practical 10B:
Aim: Classes creation using Genetic Algorithm

Code:

```
import numpy as np, random, operator, pandas as pd, matplotlib.pyplot as plt
from tkinter import Tk, Canvas, Frame, BOTH, Text
import math
class City:
    def __init__(self, x, y):
        self.x = x
        self.y = y
    def distance(self, city):
        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 "(" + str(self.x) + "," + str(self.y) + ")"
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(0, len(self.route)):
                fromCity = self.route[i]
                toCity = None
                if i + 1 < len(self.route):
                    toCity = self.route[i + 1]
                else:
                    toCity = self.route[0]
                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
def createRoute(cityList):
```

```
    route = random.sample(cityList, len(cityList))
    return route
def initialPopulation(popSize, cityList):
    population = []
    for i in range(0, popSize):
        population.append(createRoute(cityList))
    return population
def rankRoutes(population):
    fitnessResults = {}
    for i in range(0, len(population)):
        fitnessResults[i] = Fitness(population[i]).routeFitness()
    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()
    for i in range(0, eliteSize):
        selectionResults.append(popRanked[i][0])
    for i in range(0, len(popRanked) - eliteSize):
        pick = 100 * random.random()
        for i in range(0, len(popRanked)):
            if pick <= df.iat[i, 3]:
                selectionResults.append(popRanked[i][0])
                break
    return selectionResults
def matingPool(population, selectionResults):
    matingpool = []
    for i in range(0, len(selectionResults)):
        index = selectionResults[i]
        matingpool.append(population[index])
    return matingpool
def breed(parent1, parent2):
    child = []
    childP1 = []
    childP2 = []
    geneA = int(random.random() * len(parent1))
    geneB = int(random.random() * len(parent1))
    startGene = min(geneA, geneB)
    endGene = max(geneA, geneB)
    for i in range(startGene, endGene):
```

```
        childP1.append(parent1[i])
    childP2 = [item for item in parent2 if item not in childP1]
    child = childP1 + childP2
    return child
def breedPopulation(matingpool, eliteSize):
    children = []
    length = len(matingpool) - eliteSize
    pool = random.sample(matingpool, len(matingpool))
    for i in range(0, eliteSize):
        children.append(matingpool[i])
    for i in range(0, 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))
            city1 = individual[swapped]
            city2 = individual[swapWith]
            individual[swapped] = city2
            individual[swapWith] = city1
    return individual
def mutatePopulation(population, mutationRate):
    mutatedPop = []
    for ind in range(0, len(population)):
        mutatedInd = mutate(population[ind], mutationRate)
        mutatedPop.append(mutatedInd)
    return mutatedPop
def nextGeneration(currentGen, eliteSize, mutationRate):
    popRanked = rankRoutes(currentGen)
    selectionResults = selection(popRanked, eliteSize)
    matingpool = matingPool(currentGen, selectionResults)
    children = breedPopulation(matingpool, eliteSize)
    nextGeneration = mutatePopulation(children, mutationRate)
    return nextGeneration
def geneticAlgorithm(population, popSize, eliteSize, mutationRate, generations):
    pop = initialPopulation(popSize, population)
    print("Initial distance: " + str(1 / rankRoutes(pop)[0][1]))
    for i in range(0, 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 = []
    progress.append(1 / rankRoutes(pop)[0][1])
    for i in range(0, generations):
        pop = nextGeneration(pop, eliteSize, mutationRate)
        progress.append(1 / rankRoutes(pop)[0][1])
    plt.plot(progress)
    plt.ylabel('Distance')
    plt.xlabel('Generation')
    plt.show()
def main():
    cityList = []
    for i in range(0, 25):
        cityList.append(City(x=int(random.random() * 200), y =
int(random.random() * 200)))
    geneticAlgorithmPlot(population=cityList, popSize=100, eliteSize=20,
mutationRate=0.01, generations=500)
if __name__ == '__main__':
    main()
```

Output: