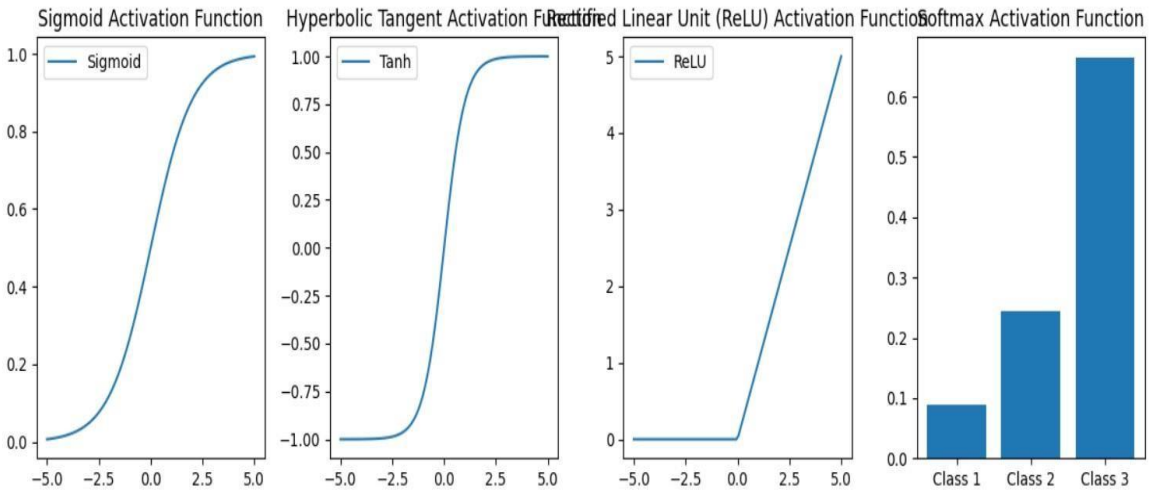
Experiment 1:- Plot Different types of activationFunctions.

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

# Define the activation functions
def sigmoid(x):
    return 1 / (1 + np.exp(-x))

def tanh(x):
    return np.tanh(x)

def relu(x):
    return np.maximum(0, x)

def softmax(x):
    exp_x = np.exp(x - np.max(x))
    return exp_x / exp_x.sum(axis=0)

# Generate x values
x = np.linspace(-5, 5, 100)

# Plot sigmoid activation function
plt.figure(figsize=(12, 4))
plt.subplot(1, 4, 1)
plt.plot(x, sigmoid(x), label='Sigmoid')
plt.title('Sigmoid Activation Function')
plt.legend()

# Plot tanh activation function
plt.subplot(1, 4, 2)
plt.plot(x, tanh(x), label='Tanh')
plt.title('Hyperbolic Tangent Activation Function')
plt.legend()
```

```
# Plot ReLU activation function
plt.subplot(1, 4, 3)
plt.plot(x, relu(x), label='ReLU')
plt.title('Rectified Linear Unit (ReLU) Activation Function')
plt.legend()

# Plot softmax activation function (for multiple classes)
x_softmax = np.array([1, 2, 3])
plt.subplot(1, 4, 4)
plt.bar(range(len(x_softmax)), softmax(x_softmax))
plt.title('Softmax Activation Function')
plt.xticks(range(len(x_softmax)), ['Class 1', 'Class 2', 'Class 3'])

# Adjust layout and show the plots
plt.tight_layout()
plt.show()
```



Experiment 2:- Program for single perceptron.

```
import numpy as np

class Perceptron:
    def __init__(self, num_inputs, learning_rate=0.01, epochs=100):
        self.weights = np.random.rand(num_inputs)
        self.threshold = 0
        self.learning_rate = learning_rate
        self.epochs = epochs

    # An epoch means training the neural network with all the training data for one cycle.
    def activation_function(self, x):
        return 1 if x > self.threshold else 0

    def train(self, X_train, y_train):
        for _ in range(self.epochs):
            for inputs, label in zip(X_train, y_train):
                prediction = self.predict(inputs)
                error = label - prediction
                self.weights += self.learning_rate * error * inputs

    def predict(self, inputs):
        weighted_sum = np.dot(inputs, self.weights)
        return self.activation_function(weighted_sum)

# Sample training data (OR gate)
X_train = np.array([[0, 0], [0, 1], [1, 0], [1, 1]])
y_train = np.array([0, 1, 1, 1])

# Create and train the perceptron
perceptron = Perceptron(num_inputs=2)
perceptron.train(X_train, y_train)
```

```
# Test the perceptron
test_inputs = np.array([[0, 0], [0, 1], [1, 0], [1, 1]])
for inputs in test_inputs:
    prediction = perceptron.predict(inputs)
    print(f"Input: {inputs}, Predicted Output: {prediction}")
```

```
Input: [0 0], Predicted Output: 0
Input: [0 1], Predicted Output: 1
Input: [1 0], Predicted Output: 1
Input: [1 1], Predicted Output: 1
```

Experiment 3:- Write a program to implement hetero-associative memory using hebb rule.

```
import numpy as np
class HeteroAssociativeMemory:
    def __init__(self, input_patterns, output_patterns):
        self.input_patterns = np.array(input_patterns)
        self.output_patterns = np.array(output_patterns)
        self.weights = np.zeros((self.input_patterns.shape[1], self.output_patterns.shape[1]))
    def train(self):
        for input_pattern, output_pattern in zip(self.input_patterns, self.output_patterns):
            self.weights += np.outer(input_pattern, output_pattern)
    def recall(self, input_pattern):
        return np.dot(input_pattern, self.weights)
```

Experiment 4:- Write a program to implements a hetero-associative memory (HAM).

```
import numpy as np
class HeteroAssociativeMemory:
    def __init__(self, input_patterns, output_patterns):
        self.input_patterns = np.array(input_patterns)
        self.output_patterns = np.array(output_patterns)
        self.weights = np.zeros((self.input_patterns.shape[1], self.output_patterns.shape[1]))
    def train(self):
        for input_pattern, output_pattern in zip(self.input_patterns, self.output_patterns):
            self.weights += np.outer(input_pattern, output_pattern)
    def recall(self, input_pattern):
        return np.dot(input_pattern, self.weights)
# Example usage:
input_patterns = [[1, 1, 0],
                  [1, 0, 1],
                  [0, 1, 1]]
output_patterns = [[1, 0],
                   [0, 1],
                   [1, 1]]
ham = HeteroAssociativeMemory(input_patterns, output_patterns)
ham.train()
input_pattern = [1, 0, 0]
retrieved_output = ham.recall(input_pattern)
print("Retrieved Output:", retrieved_output)
```

```
Retrieved Output: [1. 1.]
```

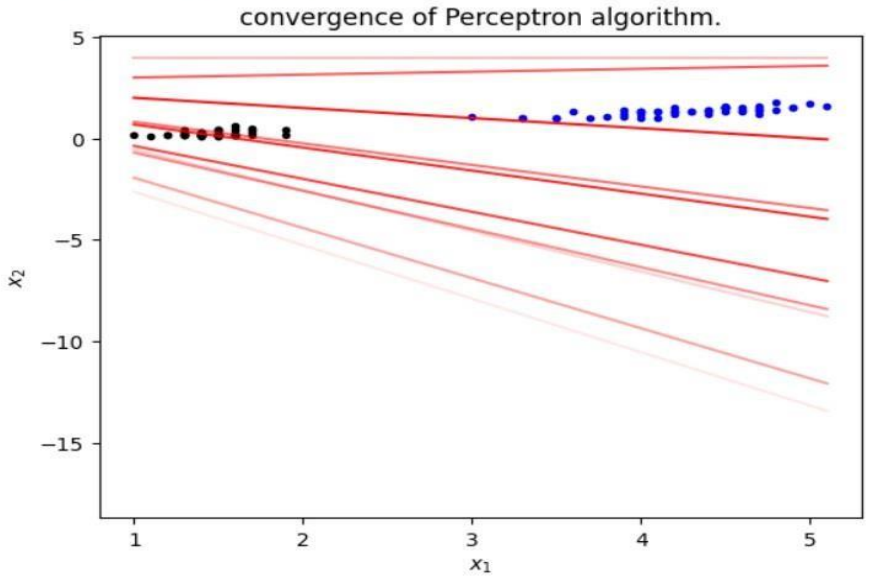
Experiment 5:- Write a program to implements a perceptron algorithm for binary classification.

```
import numpy as np
#%matplotlib notebook
%matplotlib inline
import matplotlib.pyplot as plt
from IPython.core.debugger import set_trace
import warnings
warnings.filterwarnings('ignore')
class Perceptron:

    def __init__(self, add_bias=True, max_iters=10000, record_updates=False):
        self.max_iters = max_iters
        self.add_bias = add_bias
        self.record_updates = record_updates
        if record_updates:
            self.w_hist = [] # records the weight
            self.n_hist = [] # records the data-point selected

    def fit(self, x, y):
        if x.ndim == 1:
            x = x[:, None]
        if self.add_bias:
            N = x.shape[0]
            x = np.column_stack([x,np.ones(N)])
        N,D = x.shape
        w = np.zeros(D)                                 #initialize the weights
        if self.record_updates:
            w_hist = [w]
        #y = np.sign(y -.1)                              #to get +1 for class 1 and -1 for class 0
        y = 2*y - 1                                     # converting 0,1 to -1,+1
        t = 0
        change = True                                   #if the weight does not change the algorithm has converged
        while change and t < self.max_iters:
            change = False
```

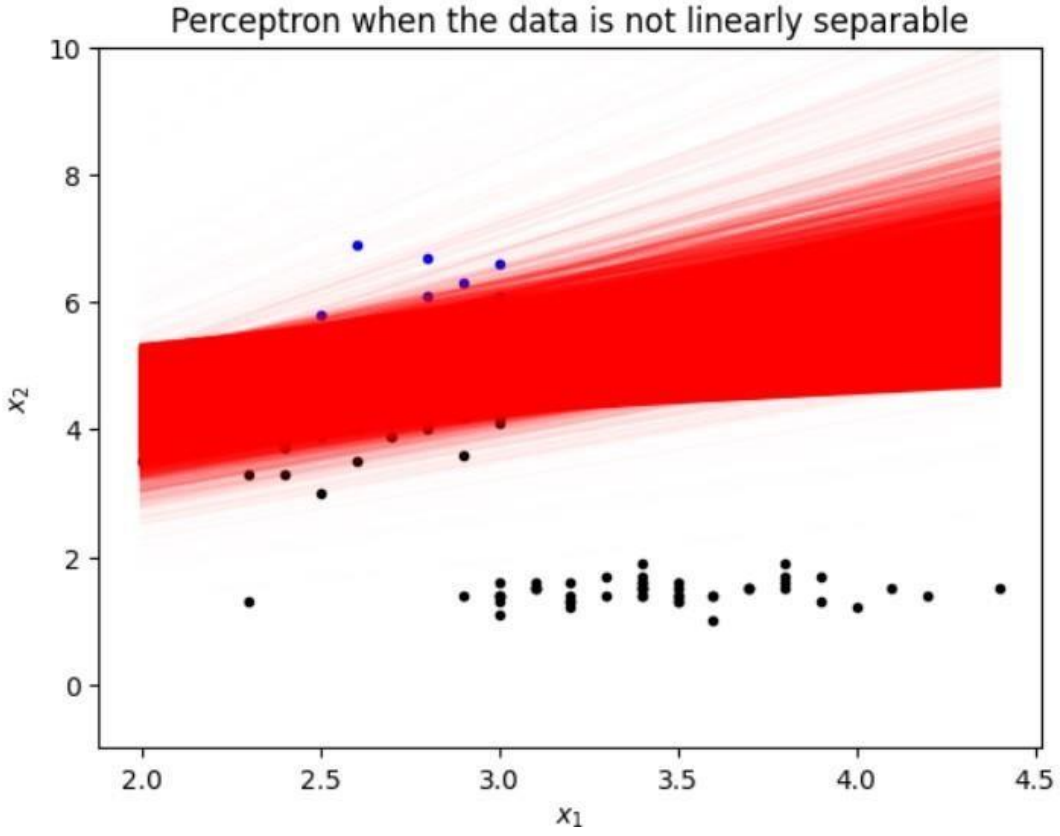
```
            for n in np.random.permutation(N):
                yh = np.sign(np.dot(x[n,:], w))        #predict the output of the training sample
                if yh == y[n]:
                    continue                           #skip the samples which are correctly classified
                #w = w + (y[n]-yh)*x[n,:]               #update the weights
                w = w + y[n]*x[n,:]
                if self.record_updates:
                    self.w_hist.append(w)
                    self.n_hist.append(n)
                change = True
                t += 1
                if t >= self.max_iters:
                    break
        if change:
            print(f'did not converge after {t} updates')
        else:
            print(f'converged after {t} iterations!')
        self.w = w
        return self

    def predict(self, x):
        if x.ndim == 1:
            x = x[:, None]
        Nt = x.shape[0]
        if self.add_bias:
            x = np.column_stack([x,np.ones(Nt)])
        yh = np.sign(np.dot(self.w, x))
        return (yh + 1)//2                         # converting -/+1 to classes 0,1
from sklearn import datasets
dataset = datasets.load_iris()
x, y = dataset['data'][:,2:], dataset['target']      #slice last two features of Iris dataset
x, y = x[y < 2, :], y[y< 2]                          #slice class 0 and 1

model = Perceptron(record_updates=True)
```

```
yh = model.fit(x,y)
plt.plot(x[y==0,0], x[y==0,1], 'k.' )
plt.plot(x[y==1,0], x[y==1,1], 'b.' )
x_line = np.linspace(np.min(x[:,0]), np.max(x[:,0]), 100)
for t,w in enumerate(model.w_hist):
    coef = -w[0]/w[1]                                    #slope of the decision boundary
    plt.plot(x_line, coef*x_line - w[2]/w[1], 'r-',
             alpha=t/len(model.w_hist), label=f't={t}')
plt.xlabel(r'$x_1$')
plt.ylabel(r'$x_2$')
plt.title('convergence of Perceptron algorithm.')
plt.show()
dataset = datasets.load_iris()
x, y = dataset['data'][:,:], dataset['target']
print(x.shape)
dataset = datasets.load_iris()
x, y = dataset['data'][:,[1,2]], dataset['target']                    #slice feature 1 and 2 of Iris dataset
y =  y > 1
model = Perceptron(record_updates=True)
yh = model.fit(x,y)
plt.plot(x[y==0,0], x[y==0,1], 'k.' )
plt.plot(x[y==1,0], x[y==1,1], 'b.' )
x_line = np.linspace(np.min(x[:,0]), np.max(x[:,0]), 100)
for t,w in enumerate(model.w_hist):
    coef = -w[0]/w[1]
    plt.plot(x_line, coef*x_line - w[2]/w[1], 'r-', alpha=t/len(model.w_hist), label=f't={t}')
plt.xlabel(r'$x_1$')
plt.ylabel(r'$x_2$')
plt.ylim(-1,10)
plt.title('Perceptron when the data is not linearly separable')
plt.show()
```

```
converged after 11 iterations!
```



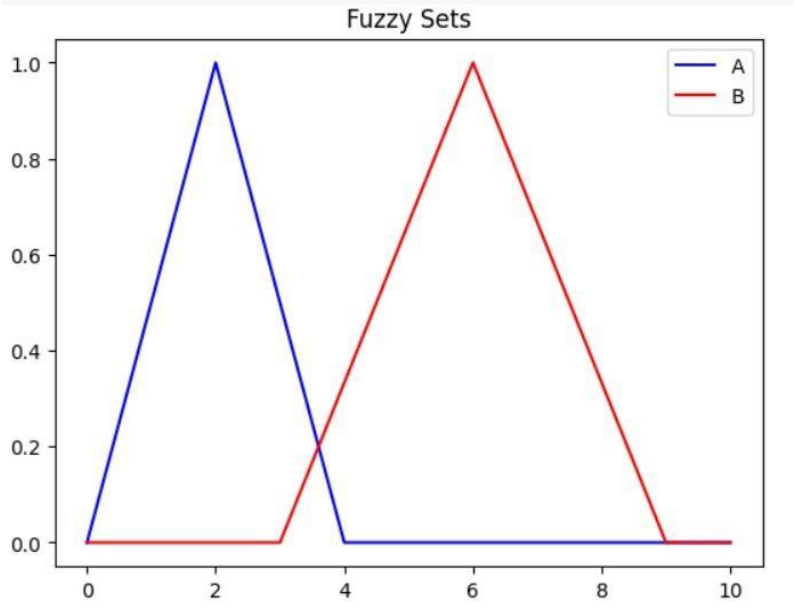

```
(150, 4)
did not converge after 10000 updates
```

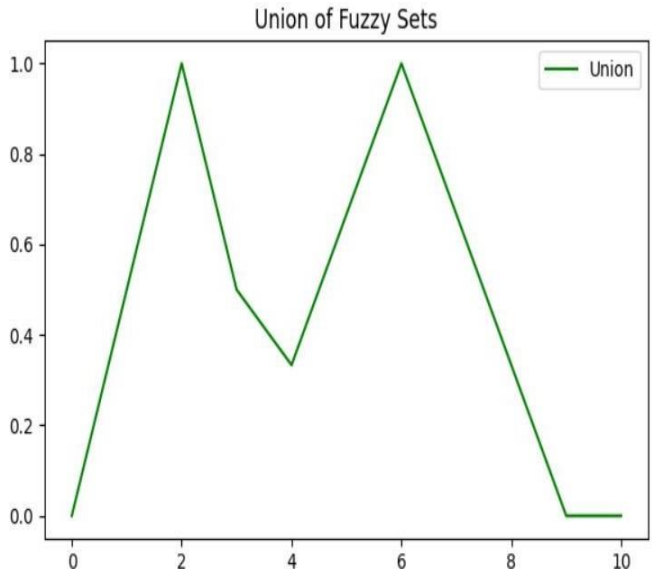


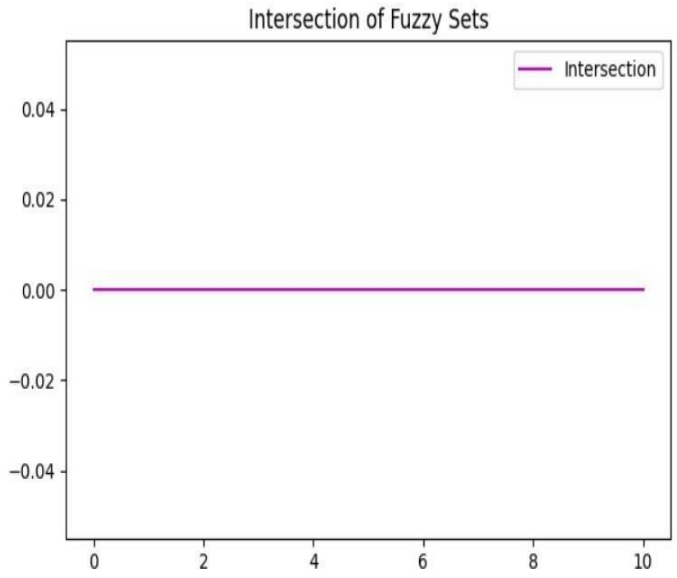
Experiment 6:- Demonstrates basic operations on fuzzy sets.

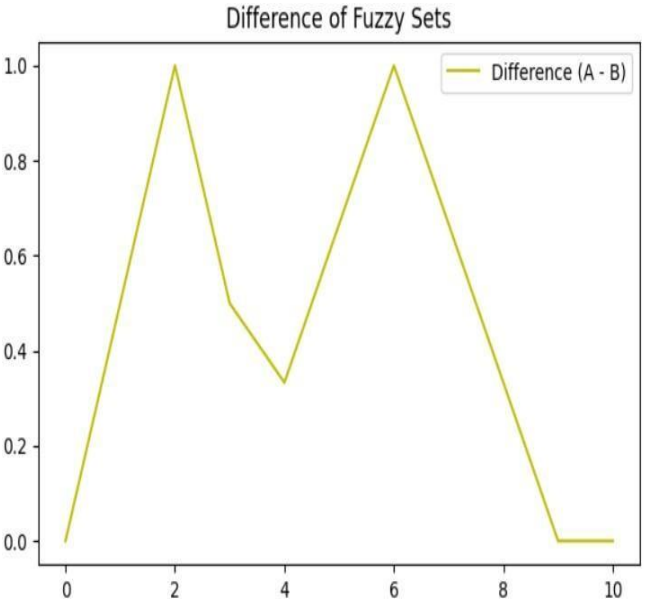
```
!pip install -U scikit-fuzzy
# Import necessary libraries
import numpy as np
import skfuzzy as fuzz
import matplotlib.pyplot as plt

# Define input universe
x = np.arange(0, 11, 1)

# Define fuzzy sets
A = fuzz.trimf(x, [0, 2, 4])  # Fuzzy set A
B = fuzz.trimf(x, [3, 6, 9])  # Fuzzy set B

# Plot fuzzy sets
plt.figure()
plt.plot(x, A, 'b', linewidth=1.5, label='A')
plt.plot(x, B, 'r', linewidth=1.5, label='B')
plt.title('Fuzzy Sets')
plt.legend()
plt.show()

# Perform union operation
union = np.fmax(A, B)

# Perform intersection operation
intersection = np.fmin(A, B)

# Perform difference operation (A - B)
difference = np.fmax(A, np.fmin(B, 1 - A))

# Perform complement operation
complement_A = 1 - A
complement_B = 1 - B
```

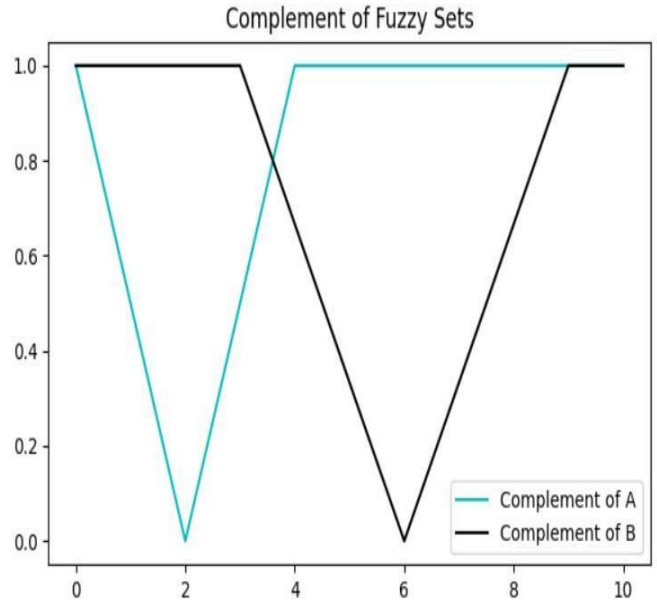
```
# Plot union, intersection, difference, and complement
plt.figure(figsize=(12, 8))

plt.subplot(2, 2, 1)
plt.plot(x, union, 'g', linewidth=1.5, label='Union')
plt.title('Union of Fuzzy Sets')
plt.legend()

plt.subplot(2, 2, 2)
plt.plot(x, intersection, 'm', linewidth=1.5, label='Intersection')
plt.title('Intersection of Fuzzy Sets')
plt.legend()

plt.subplot(2, 2, 3)
plt.plot(x, difference, 'y', linewidth=1.5, label='Difference (A - B)')
plt.title('Difference of Fuzzy Sets')
plt.legend()

plt.subplot(2, 2, 4)
plt.plot(x, complement_A, 'c', linewidth=1.5, label='Complement of A')
plt.plot(x, complement_B, 'k', linewidth=1.5, label='Complement of B')
plt.title('Complement of Fuzzy Sets')
plt.legend()

plt.tight_layout()
plt.show()
```




Union of Fuzzy Sets
Union
1.0
0.8
0.6
0.4
0.2
0.0
0
2
4
6
8
10


Intersection of Fuzzy Sets
Intersection
0.04
0.02
0.00
−0.02
−0.04
0
2
4
6
8
10

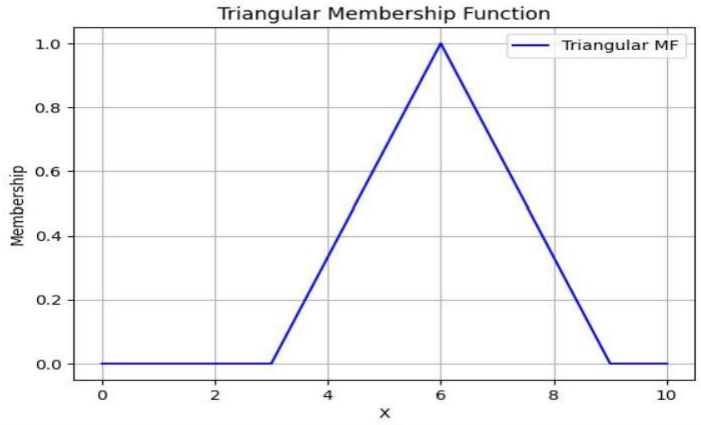

Difference of Fuzzy Sets
Difference (A - B)
1.0
0.8
0.6
0.4
0.2
0.0
0
2
4
6
8
10

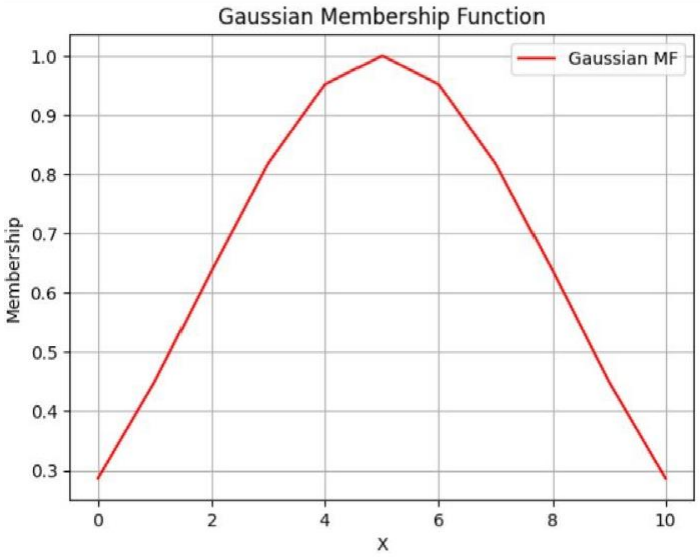

Complement of Fuzzy Sets
Complement of A
Complement of B
1.0
0.8
0.6
0.4
0.2
0.0
0
2
4
6
8
10

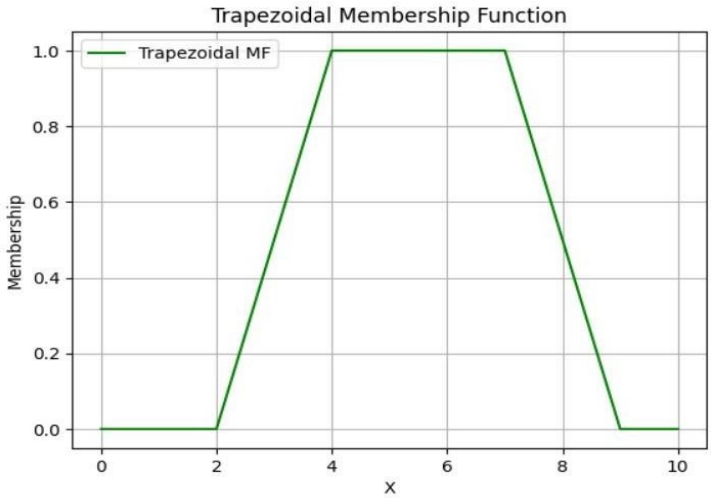
Experiment 7:- Demonstrates how to generate and plot different types of membership functions.

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

# Generate universe variables
x = np.arange(0, 11, 1)

# Generate triangular membership function
triangular_mf = fuzz.trimf(x, [3, 6, 9])

# Plot the triangular membership function
plt.figure()
plt.plot(x, triangular_mf, 'b', label='Triangular MF')
plt.title('Triangular Membership Function')
plt.xlabel('X')
plt.ylabel('Membership')
plt.legend()
plt.grid()
plt.show()

# Generate trapezoidal membership function
trapezoidal_mf = fuzz.trapmf(x, [2, 4, 7, 9])

# Plot the trapezoidal membership function
plt.figure()
plt.plot(x, trapezoidal_mf, 'g', label='Trapezoidal MF')
plt.title('Trapezoidal Membership Function')
plt.xlabel('X')
plt.ylabel('Membership')
plt.legend()
plt.grid()
plt.show()
```

```
# Generate Gaussian membership function
gaussian_mf = fuzz.gaussmf(x, np.mean(x), np.std(x))

# Plot the Gaussian membership function
plt.figure()
plt.plot(x, gaussian_mf, 'r', label='Gaussian MF')
plt.title('Gaussian Membership Function')
plt.xlabel('X')
plt.ylabel('Membership')
plt.legend()
plt.grid()
plt.show()
```


Triangular Membership Function
Triangular MF
Membership
X
0.0
0.2
0.4
0.6
0.8
1.0
0
2
4
6
8
10


Gaussian Membership Function
Gaussian MF
Membership
X
0.3
0.4
0.5
0.6
0.7
0.8
0.9
1.0
0
2
4
6
8
10


Trapezoidal Membership Function
Trapezoidal MF
Membership
X
0.0
0.2
0.4
0.6
0.8
1.0
0
2
4
6
8
10

Experiment 8:- Program to find a relation using Max-Min Composition, enter the two vectors whose relation is to be found.

```
import numpy as np
# Define the max-min composition function
def max_min_composition(A, B):
    n = len(A)
    C = np.zeros((n, n))
    for i in range(n):
        for j in range(n):
            min_max = min(max(A[i, k], B[k, j]) for k in range(n))
            C[i, j] = min_max
    return C
# Get input vectors from the user
n = int(input("Enter the size of the vectors: "))
print("Enter elements of vector A separated by space:")
A = np.array(list(map(float, input().split()))).reshape((n, n))
print("Enter elements of vector B separated by space:")
B = np.array(list(map(float, input().split()))).reshape((n, n))
# Perform max-min composition
relation = max_min_composition(A, B)
# Display the resulting relation
print("\nThe resulting relation after max-min composition is:")
print(relation)
```

```
Enter the size of the vectors: 2
Enter elements of vector A separated by space:
1 2 3 4
Enter elements of vector B separated by space:
5 4 3 6

The resulting relation after max-min composition is:
[[3. 4.]
 [4. 4.]]
```

Experiment 9:- WAP to implement Defuzzification methods such as Centroid, Centre of Sum and Mean and Maxima.

```
import numpy as np

# Define the membership function for a fuzzy set
def membership_function(x, a, b, c):
    if x <= a or x >= c:
        return 0
    elif a < x <= b:
        return (x - a) / (b - a)
    elif b < x < c:
        return (c - x) / (c - b)

# Define the defuzzification methods
def centroid_defuzzification(membership, universe):
    numerator = np.sum(membership * universe)
    denominator = np.sum(membership)
    return numerator / denominator

def center_of_sum_defuzzification(membership, universe):
    return np.sum(membership * universe) / np.sum(membership)

def mean_of_maxima_defuzzification(membership, universe):
    maxima = universe[membership == np.max(membership)]
    return np.mean(maxima)

# Define the fuzzy set parameters
a = 0
b = 5
c = 10
universe = np.linspace(0, 10, 100)
membership = np.array([membership_function(x, a, b, c) for x in universe])
```

```
# Perform defuzzification using centroid method
centroid_result = centroid_defuzzification(membership, universe)

# Perform defuzzification using center of sum method
center_of_sum_result = center_of_sum_defuzzification(membership, universe)

# Perform defuzzification using mean of maxima method
mean_of_maxima_result = mean_of_maxima_defuzzification(membership, universe)

# Display results
print("Defuzzification using Centroid:", centroid_result)
print("Defuzzification using Center of Sum:", center_of_sum_result)
print("Defuzzification using Mean of Maxima:", mean_of_maxima_result)
```

```
Defuzzification using Centroid: 4.999999999999999
Defuzzification using Center of Sum: 4.999999999999999
Defuzzification using Mean of Maxima: 5.0
```