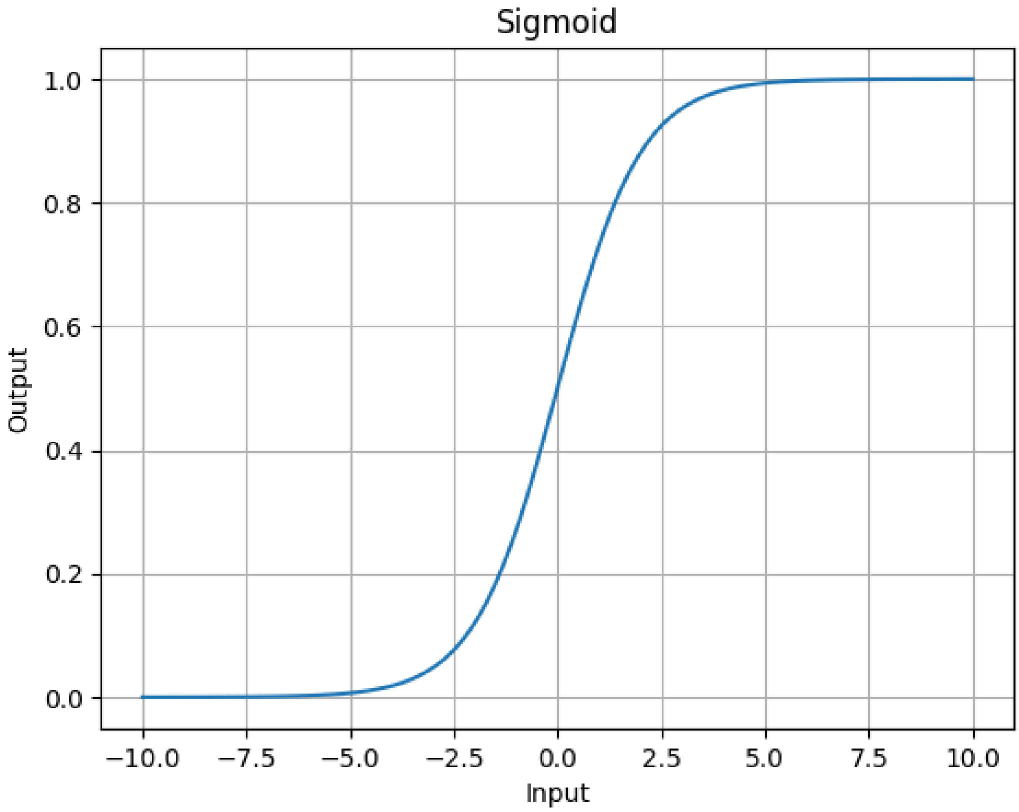
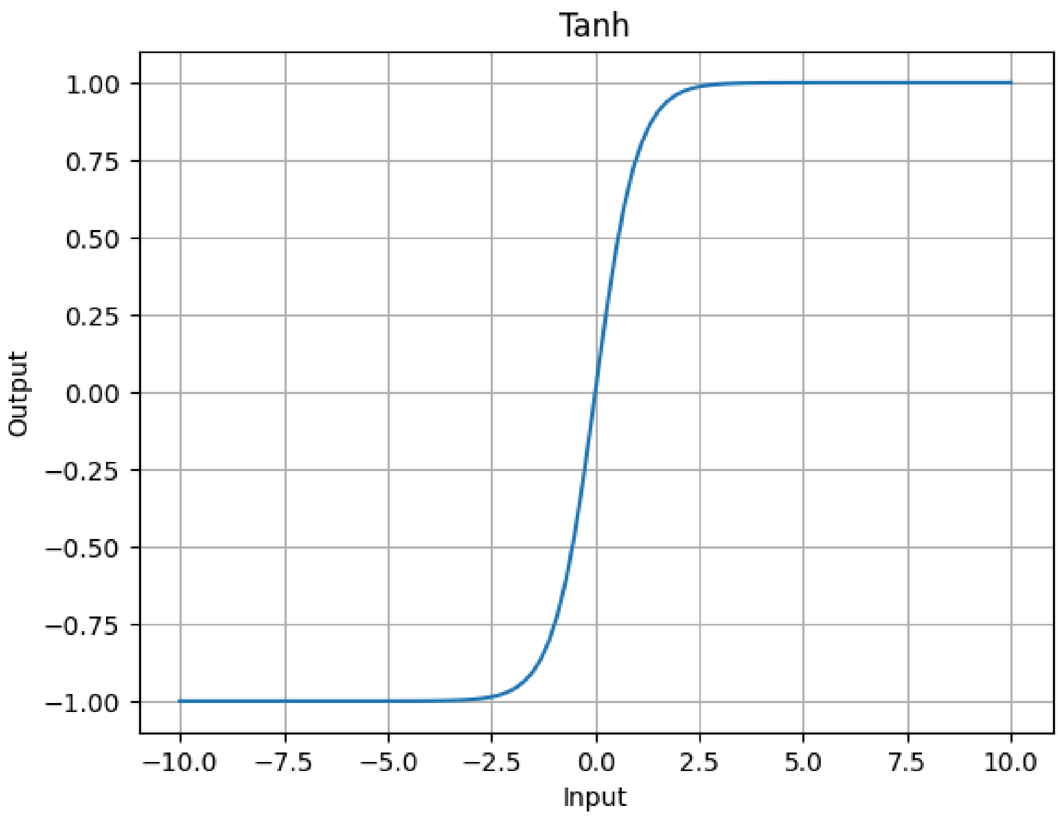
1. Program to implement basic activation functions.

In [4]:
```python
import numpy as np
import matplotlib.pyplot as plt

# 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 leaky_relu(x, alpha=0.01):
    return np.where(x > 0, x, alpha * x)

def softmax(x):
    exp_x = np.exp(x - np.max(x))  # Stability adjustment
    return exp_x / np.sum(exp_x)

# Plot Activation Functions
def plot_activation_function(func, x, title):
    y = func(x)
    plt.plot(x, y)
    plt.title(title)
    plt.xlabel("Input")
    plt.ylabel("Output")
    plt.grid(True)
    plt.show()

# Input Range for Visualization
x = np.linspace(-10, 10, 100)

# Sigmoid Function
print("Sigmoid Activation Function")
plot_activation_function(sigmoid, x, "Sigmoid")

# Tanh Function
print("Tanh Activation Function")
plot_activation_function(tanh, x, "Tanh")

# ReLU Function
print("ReLU Activation Function")
plot_activation_function(relu, x, "ReLU")

# Leaky ReLU Function
print("Leaky ReLU Activation Function")
plot_activation_function(lambda x: leaky_relu(x, alpha=0.1), x, "Leaky ReLU")

# Softmax Function (Demonstrated on a small vector)
print("Softmax Activation Function")
input_vector = np.array([1.0, 2.0, 3.0, 4.0, 5.0])
softmax_output = softmax(input_vector)
print(f"Input Vector: {input_vector}")
print(f"Softmax Output: {softmax_output}")
```

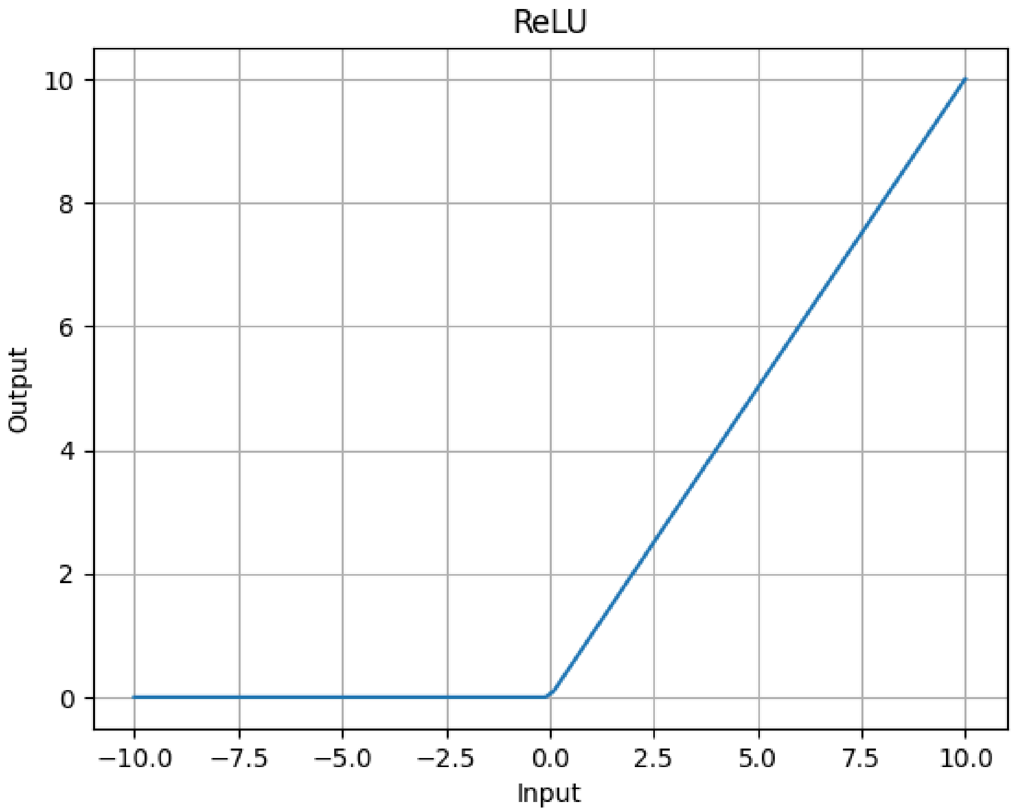
```
Sigmoid Activation Function
```



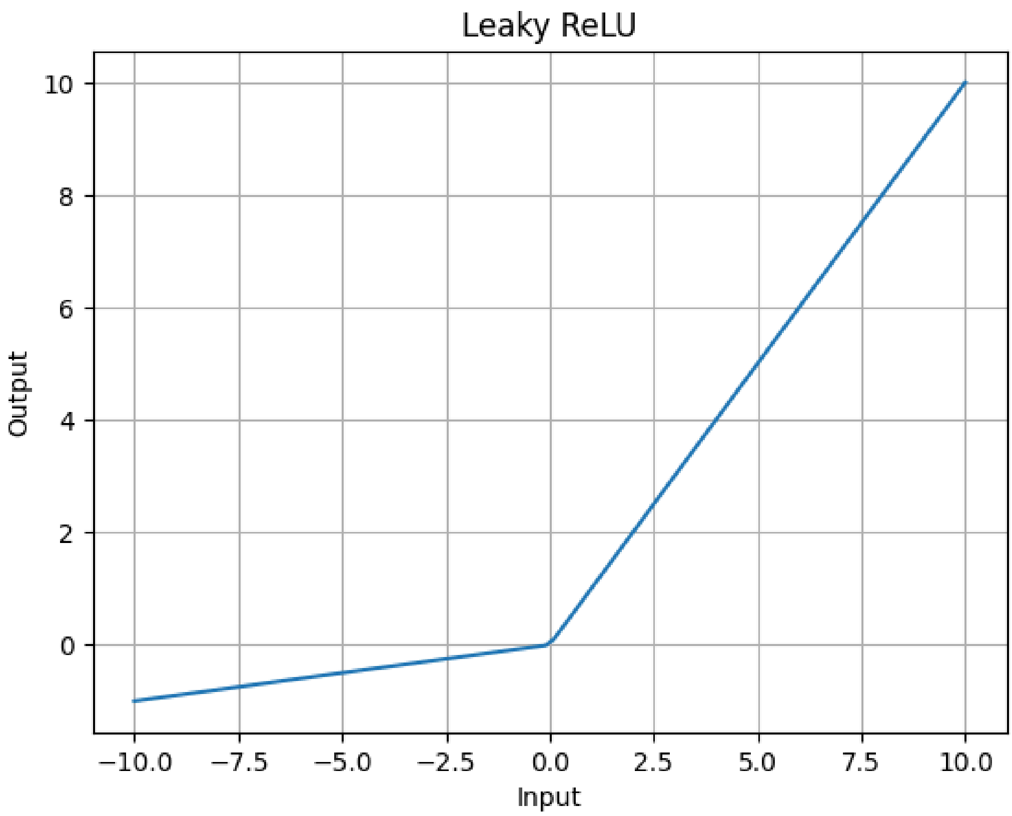

Tanh Activation Function




ReLU Activation Function




```
Leaky ReLU Activation Function
```




```
Softmax Activation Function
Input Vector: [1. 2. 3. 4. 5.]
Softmax Output: [0.01165623 0.03168492 0.08612854 0.23412166 0.63640865]
```

2. Program to implement McCulloch-Pitts Neuron.

In [ ]:

```
# McCulloch-Pitts Neuron Implementation

def mcculloch_pitts(inputs, weights, threshold):
```

```
    weighted_sum = sum(i * w for i, w in zip(inputs, weights))
    return 1 if weighted_sum >= threshold else 0

# Example: AND Gate
inputs_list = [
    [0, 0],
    [0, 1],
    [1, 0],
    [1, 1]
]
weights = [1, 1]  # Weights for the AND gate
threshold = 2     # Threshold for the AND gate

print("AND Gate using McCulloch-Pitts Neuron:")
for inputs in inputs_list:
    output = mcculloch_pitts(inputs, weights, threshold)
    print(f"Inputs: {inputs}, Output: {output}")

# Example: OR Gate
weights = [1, 1]  # Weights for the OR gate
threshold = 1     # Threshold for the OR gate

print("\nOR Gate using McCulloch-Pitts Neuron:")
for inputs in inputs_list:
    output = mcculloch_pitts(inputs, weights, threshold)
    print(f"Inputs: {inputs}, Output: {output}")

# Example: NOT Gate
inputs_list = [
    [0],
    [1]
]
weights = [-1]    # Weight for the NOT gate
threshold = 0     # Threshold for the NOT gate

print("\nNOT Gate using McCulloch-Pitts Neuron:")
for inputs in inputs_list:
    output = mcculloch_pitts(inputs, weights, threshold)
    print(f"Inputs: {inputs}, Output: {output}")
```

```
AND Gate using McCulloch-Pitts Neuron:
Inputs: [0, 0], Output: 0
Inputs: [0, 1], Output: 0
Inputs: [1, 0], Output: 0
Inputs: [1, 1], Output: 1

OR Gate using McCulloch-Pitts Neuron:
Inputs: [0, 0], Output: 0
Inputs: [0, 1], Output: 1
Inputs: [1, 0], Output: 1
Inputs: [1, 1], Output: 1

NOT Gate using McCulloch-Pitts Neuron:
Inputs: [0], Output: 1
Inputs: [1], Output: 0
```

3. Program to implement a simple Neuron with various activation functions.

In [6]:
```
import numpy as np

# Activation Functions
def sigmoid(x):
    return 1 / (1 + np.exp(-x))
```

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

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

def leaky_relu(x, alpha=0.01):
    return np.where(x > 0, x, alpha * x)

def softmax(x):
    exp_x = np.exp(x - np.max(x))  # Stability adjustment
    return exp_x / np.sum(exp_x)

# Simple Neuron Implementation
def simple_neuron(inputs, weights, bias, activation_func):
    """
    Simulates a single neuron.

    Args:
        inputs: List of input values.
        weights: List of weights corresponding to the inputs.
        bias: Bias value.
        activation_func: Activation function to apply (e.g., sigmoid, relu).

    Returns:
        Output of the neuron after applying the activation function.
    """
    weighted_sum = np.dot(inputs, weights) + bias
    return activation_func(weighted_sum)

# Example Inputs
inputs = np.array([0.5, 0.3, 0.2])  # Example input values
weights = np.array([0.8, 0.4, 0.3])  # Example weights
bias = 0.5  # Example bias

# Sigmoid Activation
print("Neuron Output with Sigmoid Activation:")
output_sigmoid = simple_neuron(inputs, weights, bias, sigmoid)
print(f"Output: {output_sigmoid}")

# Tanh Activation
print("\nNeuron Output with Tanh Activation:")
output_tanh = simple_neuron(inputs, weights, bias, tanh)
print(f"Output: {output_tanh}")

# ReLU Activation
print("\nNeuron Output with ReLU Activation:")
output_relu = simple_neuron(inputs, weights, bias, relu)
print(f"Output: {output_relu}")

# Leaky ReLU Activation
print("\nNeuron Output with Leaky ReLU Activation:")
output_leaky_relu = simple_neuron(inputs, weights, bias, lambda x: leaky_relu(x, alpha=0
print(f"Output: {output_leaky_relu}")

# Softmax Activation
print("\nNeuron Output with Softmax Activation:")
# Softmax requires multiple outputs, so we simulate a vector of weighted sums
weighted_sums = np.array([np.dot(inputs, weights) + bias for _ in range(3)])
output_softmax = softmax(weighted_sums)
print(f"Output: {output_softmax}")
```

```
Neuron Output with Sigmoid Activation:
Output: 0.7464939833376621

Neuron Output with Tanh Activation:
Output: 0.7931990970835009

Neuron Output with ReLU Activation:
Output: 1.08

Neuron Output with Leaky ReLU Activation:
Output: 1.08

Neuron Output with Softmax Activation:
Output: [0.33333333 0.33333333 0.33333333]
```

4. Program to implement a simple perceptron using the perceptron learning algorithm.

In [ ]:
```python
import numpy as np

class Perceptron:
    def __init__(self, input_size, learning_rate=0.1, epochs=10):

        self.weights = np.zeros(input_size + 1)  # +1 for the bias
        self.learning_rate = learning_rate
        self.epochs = epochs

    def activation_function(self, x):

        return 1 if x >= 0 else 0

    def predict(self, inputs):
        """
        Predict the output for the given inputs.
        """
        weighted_sum = np.dot(inputs, self.weights[1:]) + self.weights[0]
        return self.activation_function(weighted_sum)

    def train(self, training_inputs, labels):
        """
        Train the perceptron using the perceptron learning algorithm.
        """
        for epoch in range(self.epochs):
            print(f"\nEpoch {epoch + 1}")
            for inputs, label in zip(training_inputs, labels):
                prediction = self.predict(inputs)
                error = label - prediction
                # Update weights and bias
                self.weights[1:] += self.learning_rate * error * inputs
                self.weights[0] += self.learning_rate * error
                print(f"Inputs: {inputs}, Label: {label}, Prediction: {prediction}, Weig

# Example Dataset: AND Gate
training_inputs = np.array([
    [0, 0],
    [0, 1],
    [1, 0],
    [1, 1]
])
labels = np.array([0, 0, 0, 1])  # Output for AND gate

# Initialize Perceptron
perceptron = Perceptron(input_size=2, learning_rate=0.1, epochs=10)

# Train Perceptron
```

```python
print("Training the Perceptron on AND Gate:")
perceptron.train(training_inputs, labels)

# Test Perceptron
print("\nTesting the Perceptron:")
for inputs in training_inputs:
    prediction = perceptron.predict(inputs)
    print(f"Inputs: {inputs}, Prediction: {prediction}")
```

```
Training the Perceptron on AND Gate:

Epoch 1
Inputs: [0 0], Label: 0, Prediction: 1, Weights: [-0.1  0.   0. ]
Inputs: [0 1], Label: 0, Prediction: 0, Weights: [-0.1  0.   0. ]
Inputs: [1 0], Label: 0, Prediction: 0, Weights: [-0.1  0.   0. ]
Inputs: [1 1], Label: 1, Prediction: 0, Weights: [0.  0.1 0.1]

Epoch 2
Inputs: [0 0], Label: 0, Prediction: 1, Weights: [-0.1  0.1  0.1]
Inputs: [0 1], Label: 0, Prediction: 1, Weights: [-0.2  0.1  0. ]
Inputs: [1 0], Label: 0, Prediction: 0, Weights: [-0.2  0.1  0. ]
Inputs: [1 1], Label: 1, Prediction: 0, Weights: [-0.1  0.2  0.1]

Epoch 3
Inputs: [0 0], Label: 0, Prediction: 0, Weights: [-0.1  0.2  0.1]
Inputs: [0 1], Label: 0, Prediction: 1, Weights: [-0.2  0.2  0. ]
Inputs: [1 0], Label: 0, Prediction: 1, Weights: [-0.3  0.1  0. ]
Inputs: [1 1], Label: 1, Prediction: 0, Weights: [-0.2  0.2  0.1]

Epoch 4
Inputs: [0 0], Label: 0, Prediction: 0, Weights: [-0.2  0.2  0.1]
Inputs: [0 1], Label: 0, Prediction: 0, Weights: [-0.2  0.2  0.1]
Inputs: [1 0], Label: 0, Prediction: 0, Weights: [-0.2  0.2  0.1]
Inputs: [1 1], Label: 1, Prediction: 1, Weights: [-0.2  0.2  0.1]

Epoch 5
Inputs: [0 0], Label: 0, Prediction: 0, Weights: [-0.2  0.2  0.1]
Inputs: [0 1], Label: 0, Prediction: 0, Weights: [-0.2  0.2  0.1]
Inputs: [1 0], Label: 0, Prediction: 0, Weights: [-0.2  0.2  0.1]
Inputs: [1 1], Label: 1, Prediction: 1, Weights: [-0.2  0.2  0.1]

Epoch 6
Inputs: [0 0], Label: 0, Prediction: 0, Weights: [-0.2  0.2  0.1]
Inputs: [0 1], Label: 0, Prediction: 0, Weights: [-0.2  0.2  0.1]
Inputs: [1 0], Label: 0, Prediction: 0, Weights: [-0.2  0.2  0.1]
Inputs: [1 1], Label: 1, Prediction: 1, Weights: [-0.2  0.2  0.1]

Epoch 7
Inputs: [0 0], Label: 0, Prediction: 0, Weights: [-0.2  0.2  0.1]
Inputs: [0 1], Label: 0, Prediction: 0, Weights: [-0.2  0.2  0.1]
Inputs: [1 0], Label: 0, Prediction: 0, Weights: [-0.2  0.2  0.1]
Inputs: [1 1], Label: 1, Prediction: 1, Weights: [-0.2  0.2  0.1]

Epoch 8
Inputs: [0 0], Label: 0, Prediction: 0, Weights: [-0.2  0.2  0.1]
Inputs: [0 1], Label: 0, Prediction: 0, Weights: [-0.2  0.2  0.1]
Inputs: [1 0], Label: 0, Prediction: 0, Weights: [-0.2  0.2  0.1]
Inputs: [1 1], Label: 1, Prediction: 1, Weights: [-0.2  0.2  0.1]

Epoch 9
Inputs: [0 0], Label: 0, Prediction: 0, Weights: [-0.2  0.2  0.1]
Inputs: [0 1], Label: 0, Prediction: 0, Weights: [-0.2  0.2  0.1]
Inputs: [1 0], Label: 0, Prediction: 0, Weights: [-0.2  0.2  0.1]
Inputs: [1 1], Label: 1, Prediction: 1, Weights: [-0.2  0.2  0.1]

Epoch 10
Inputs: [0 0], Label: 0, Prediction: 0, Weights: [-0.2  0.2  0.1]
Inputs: [0 1], Label: 0, Prediction: 0, Weights: [-0.2  0.2  0.1]
Inputs: [1 0], Label: 0, Prediction: 0, Weights: [-0.2  0.2  0.1]
Inputs: [1 1], Label: 1, Prediction: 1, Weights: [-0.2  0.2  0.1]

Testing the Perceptron:
Inputs: [0 0], Prediction: 0
Inputs: [0 1], Prediction: 0
```

```
Inputs: [1 0], Prediction: 0
Inputs: [1 1], Prediction: 1
```

5. Program to implement delta and back propagation algorithm.

In [10]:
```python
import numpy as np

# Activation Functions and Derivatives
def sigmoid(x):
    return 1 / (1 + np.exp(-x))

def sigmoid_derivative(x):
    return x * (1 - x)

# Backpropagation Algorithm Implementation
class NeuralNetwork:
    def __init__(self, input_size, hidden_size, output_size, learning_rate=0.1):
        # Initialize weights and biases
        self.weights_input_hidden = np.random.rand(input_size, hidden_size)
        self.bias_hidden = np.random.rand(hidden_size)
        self.weights_hidden_output = np.random.rand(hidden_size, output_size)
        self.bias_output = np.random.rand(output_size)
        self.learning_rate = learning_rate

    def feedforward(self, inputs):
        # Forward pass
        self.hidden_input = np.dot(inputs, self.weights_input_hidden) + self.bias_hidden
        self.hidden_output = sigmoid(self.hidden_input)
        self.final_input = np.dot(self.hidden_output, self.weights_hidden_output) + self
        self.final_output = sigmoid(self.final_input)
        return self.final_output

    def backpropagation(self, inputs, expected_output, actual_output):
        # Calculate errors
        output_error = expected_output - actual_output
        output_delta = output_error * sigmoid_derivative(actual_output)

        hidden_error = np.dot(output_delta, self.weights_hidden_output.T)
        hidden_delta = hidden_error * sigmoid_derivative(self.hidden_output)

        # Update weights and biases
        self.weights_hidden_output += self.learning_rate * np.dot(self.hidden_output.resl
        self.bias_output += self.learning_rate * output_delta
        self.weights_input_hidden += self.learning_rate * np.dot(inputs.reshape(-1, 1), l
        self.bias_hidden += self.learning_rate * hidden_delta

    def train(self, inputs, expected_output, epochs):
        for epoch in range(epochs):
            for i in range(len(inputs)):
                actual_output = self.feedforward(inputs[i])
                self.backpropagation(inputs[i], expected_output[i], actual_output)
            print(f"Epoch {epoch + 1}/{epochs} completed.")

    def predict(self, inputs):
        return self.feedforward(inputs)

# Dataset: XOR Gate
inputs = np.array([[0, 0], [0, 1], [1, 0], [1, 1]])
expected_outputs = np.array([[0], [1], [1], [0]])

# Initialize Neural Network
nn = NeuralNetwork(input_size=2, hidden_size=2, output_size=1, learning_rate=0.1)

# Train Neural Network
```

```python
print("Training Neural Network on XOR Gate Data:")
nn.train(inputs, expected_outputs, epochs=20)

# Testing
print("\nTesting Neural Network:")
for input_data in inputs:
    prediction = nn.predict(input_data)
    print(f"Input: {input_data}, Predicted Output: {prediction}")
```

```
Training Neural Network on XOR Gate Data:
Epoch 1/20 completed.
Epoch 2/20 completed.
Epoch 3/20 completed.
Epoch 4/20 completed.
Epoch 5/20 completed.
Epoch 6/20 completed.
Epoch 7/20 completed.
Epoch 8/20 completed.
Epoch 9/20 completed.
Epoch 10/20 completed.
Epoch 11/20 completed.
Epoch 12/20 completed.
Epoch 13/20 completed.
Epoch 14/20 completed.
Epoch 15/20 completed.
Epoch 16/20 completed.
Epoch 17/20 completed.
Epoch 18/20 completed.
Epoch 19/20 completed.
Epoch 20/20 completed.

Testing Neural Network:
Input: [0 0], Predicted Output: [0.65650937]
Input: [0 1], Predicted Output: [0.68600799]
Input: [1 0], Predicted Output: [0.69718526]
Input: [1 1], Predicted Output: [0.71862378]
```

6. Program to implement a simple perceptron with radial basis function (RBF) activation function.

In [12]:
```python
import numpy as np

# Radial Basis Function (RBF) and its derivative
def rbf(x, c, sigma):
    """Radial Basis Function (Gaussian)"""
    return np.exp(-np.linalg.norm(x - c)**2 / (2 * sigma**2))

def rbf_derivative(x, c, sigma):
    """Derivative of Radial Basis Function"""
    return -np.exp(-np.linalg.norm(x - c)**2 / (2 * sigma**2)) * (x - c) / (sigma**2)

# Perceptron with RBF Activation Function
class RBFPerceptron:
    def __init__(self, input_size, num_centroids, sigma=1.0, learning_rate=0.1, epochs=1
        self.input_size = input_size
        self.num_centroids = num_centroids
        self.sigma = sigma
        self.learning_rate = learning_rate
        self.epochs = epochs

        # Initialize centroids and weights
        self.centroids = np.random.rand(num_centroids, input_size)
        self.weights = np.random.rand(num_centroids)
        self.bias = np.random.rand(1)
```

```python
    def feedforward(self, inputs):
        """Feedforward pass through the RBF network"""
        rbf_outputs = np.array([rbf(inputs, self.centroids[i], self.sigma) for i in range
        return np.dot(rbf_outputs, self.weights) + self.bias

    def train(self, training_inputs, labels):
        """Train the perceptron using the RBF activation function"""
        for epoch in range(self.epochs):
            for inputs, label in zip(training_inputs, labels):
                output = self.feedforward(inputs)
                error = label - output
                # Update weights and bias
                rbf_outputs = np.array([rbf(inputs, self.centroids[i], self.sigma) for i
                self.weights += self.learning_rate * error * rbf_outputs
                self.bias += self.learning_rate * error
            print(f"Epoch {epoch + 1}/{self.epochs} completed.")

    def predict(self, inputs):
        """Make a prediction with the trained perceptron"""
        return self.feedforward(inputs)

# Example Dataset: XOR Gate
training_inputs = np.array([[0, 0], [0, 1], [1, 0], [1, 1]])
labels = np.array([0, 1, 1, 0])  # XOR output

# Initialize the RBF Perceptron
rbf_perceptron = RBFPerceptron(input_size=2, num_centroids=2, sigma=1.0, learning_rate=0

# Train the Perceptron
print("Training the RBF Perceptron on XOR Gate Data:")
rbf_perceptron.train(training_inputs, labels)

# Test the Perceptron
print("\nTesting the RBF Perceptron:")
for inputs in training_inputs:
    prediction = rbf_perceptron.predict(inputs)
    print(f"Input: {inputs}, Predicted Output: {prediction}")
```

```
Training the RBF Perceptron on XOR Gate Data:
Epoch 1/20 completed.
Epoch 2/20 completed.
Epoch 3/20 completed.
Epoch 4/20 completed.
Epoch 5/20 completed.
Epoch 6/20 completed.
Epoch 7/20 completed.
Epoch 8/20 completed.
Epoch 9/20 completed.
Epoch 10/20 completed.
Epoch 11/20 completed.
Epoch 12/20 completed.
Epoch 13/20 completed.
Epoch 14/20 completed.
Epoch 15/20 completed.
Epoch 16/20 completed.
Epoch 17/20 completed.
Epoch 18/20 completed.
Epoch 19/20 completed.
Epoch 20/20 completed.

Testing the RBF Perceptron:
Input: [0 0], Predicted Output: [0.48232989]
Input: [0 1], Predicted Output: [0.48389524]
Input: [1 0], Predicted Output: [0.49534759]
Input: [1 1], Predicted Output: [0.4950207]
```

7. Program to implement all the fuzzy set operations like max, min, complement, union, intersection.

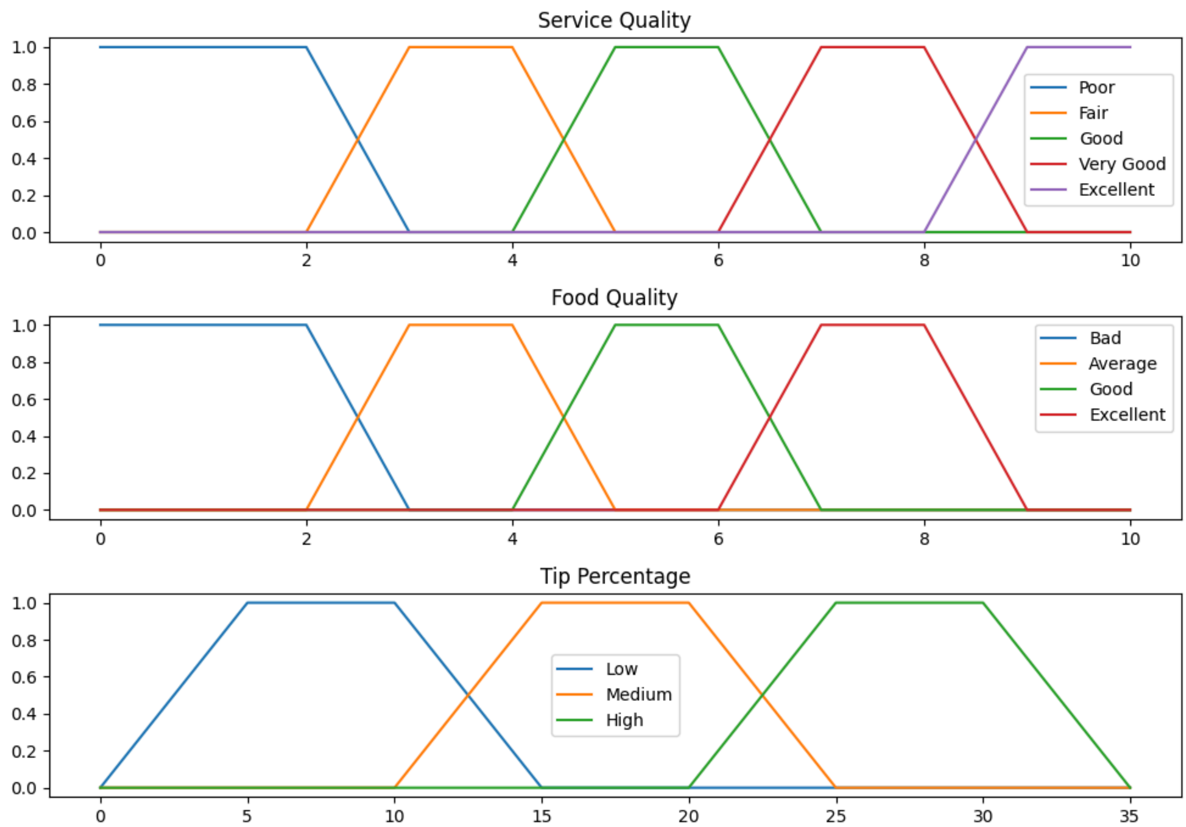
In [13]:
```python
import numpy as np

# Fuzzy Set Operations

# Max Operation: Returns the maximum membership value
def max_operation(set1, set2):
    return np.maximum(set1, set2)

# Min Operation: Returns the minimum membership value
def min_operation(set1, set2):
    return np.minimum(set1, set2)

# Complement Operation: Returns the complement of the fuzzy set
def complement(set1):
    return 1 - set1

# Union Operation: Returns the union of two fuzzy sets (max operation)
def union(set1, set2):
    return np.maximum(set1, set2)

# Intersection Operation: Returns the intersection of two fuzzy sets (min operation)
def intersection(set1, set2):
    return np.minimum(set1, set2)

# Example Usage of Fuzzy Set Operations
if __name__ == "__main__":
    # Defining two fuzzy sets A and B
    A = np.array([0.1, 0.4, 0.7, 1.0, 0.5])
    B = np.array([0.3, 0.6, 0.2, 0.8, 0.9])

    print("Fuzzy Set A: ", A)
    print("Fuzzy Set B: ", B)

    # Max Operation
    print("\nMax Operation (max(A, B)): ", max_operation(A, B))

    # Min Operation
    print("\nMin Operation (min(A, B)): ", min_operation(A, B))

    # Complement Operation (A complement)
    print("\nComplement of A (1 - A): ", complement(A))

    # Union Operation (A union B)
    print("\nUnion of A and B (max(A, B)): ", union(A, B))

    # Intersection Operation (A intersection B)
    print("\nIntersection of A and B (min(A, B)): ", intersection(A, B))
```

```
Fuzzy Set A:  [0.1 0.4 0.7 1.  0.5]
Fuzzy Set B:  [0.3 0.6 0.2 0.8 0.9]

Max Operation (max(A, B)):  [0.3 0.6 0.7 1.  0.9]

Min Operation (min(A, B)):  [0.1 0.4 0.2 0.8 0.5]

Complement of A (1 - A):  [0.9 0.6 0.3 0.  0.5]

Union of A and B (max(A, B)):  [0.3 0.6 0.7 1.  0.9]

Intersection of A and B (min(A, B)):  [0.1 0.4 0.2 0.8 0.5]
```

8. Program to design fuzzy control system form restaurant tipping problem.

In [18]:
```
import numpy as np
import skfuzzy as fuzz
import matplotlib.pyplot as plt

# Define the fuzzy sets for the inputs and output

# Service quality membership functions
service_poor = np.array([0, 1, 2, 3])
service_fair = np.array([2, 3, 4, 5])
service_good = np.array([4, 5, 6, 7])
service_very_good = np.array([6, 7, 8, 9])
service_excellent = np.array([8, 9, 10, 10])

# Food quality membership functions
food_bad = np.array([0, 1, 2, 3])
food_average = np.array([2, 3, 4, 5])
food_good = np.array([4, 5, 6, 7])
food_excellent = np.array([6, 7, 8, 9])

# Tip percentage membership functions
tip_low = np.array([0, 5, 10, 15])
tip_medium = np.array([10, 15, 20, 25])
tip_high = np.array([20, 25, 30, 35])

# Create fuzzy membership functions using the skfuzzy library
service_quality = np.arange(0, 11, 1)
food_quality = np.arange(0, 11, 1)
tip_percentage = np.arange(0, 36, 1)

# Define the fuzzy membership for the inputs and output
service_poor_mf = fuzz.trapmf(service_quality, [0, 0, 2, 3])
service_fair_mf = fuzz.trapmf(service_quality, [2, 3, 4, 5])
service_good_mf = fuzz.trapmf(service_quality, [4, 5, 6, 7])
service_very_good_mf = fuzz.trapmf(service_quality, [6, 7, 8, 9])
service_excellent_mf = fuzz.trapmf(service_quality, [8, 9, 10, 10])

food_bad_mf = fuzz.trapmf(food_quality, [0, 0, 2, 3])
food_average_mf = fuzz.trapmf(food_quality, [2, 3, 4, 5])
food_good_mf = fuzz.trapmf(food_quality, [4, 5, 6, 7])
food_excellent_mf = fuzz.trapmf(food_quality, [6, 7, 8, 9])

tip_low_mf = fuzz.trapmf(tip_percentage, [0, 5, 10, 15])
tip_medium_mf = fuzz.trapmf(tip_percentage, [10, 15, 20, 25])
tip_high_mf = fuzz.trapmf(tip_percentage, [20, 25, 30, 35])

# Visualize the fuzzy sets for Service Quality, Food Quality, and Tip Percentage
plt.figure(figsize=(10, 7))

plt.subplot(3, 1, 1)
plt.plot(service_quality, service_poor_mf, label="Poor")
plt.plot(service_quality, service_fair_mf, label="Fair")
plt.plot(service_quality, service_good_mf, label="Good")
plt.plot(service_quality, service_very_good_mf, label="Very Good")
plt.plot(service_quality, service_excellent_mf, label="Excellent")
plt.title("Service Quality")
plt.legend()

plt.subplot(3, 1, 2)
plt.plot(food_quality, food_bad_mf, label="Bad")
plt.plot(food_quality, food_average_mf, label="Average")
plt.plot(food_quality, food_good_mf, label="Good")
```

```
plt.plot(food_quality, food_excellent_mf, label="Excellent")
plt.title("Food Quality")
plt.legend()

plt.subplot(3, 1, 3)
plt.plot(tip_percentage, tip_low_mf, label="Low")
plt.plot(tip_percentage, tip_medium_mf, label="Medium")
plt.plot(tip_percentage, tip_high_mf, label="High")
plt.title("Tip Percentage")
plt.legend()

plt.tight_layout()
plt.show()

# Fuzzification
def fuzzify_inputs(service_score, food_score):
    service_level_poor = fuzz.interp_membership(service_quality, service_poor_mf, servic
    service_level_fair = fuzz.interp_membership(service_quality, service_fair_mf, servic
    service_level_good = fuzz.interp_membership(service_quality, service_good_mf, servic
    service_level_very_good = fuzz.interp_membership(service_quality, service_very_good_
    service_level_excellent = fuzz.interp_membership(service_quality, service_excellent_

    food_level_bad = fuzz.interp_membership(food_quality, food_bad_mf, food_score)
    food_level_average = fuzz.interp_membership(food_quality, food_average_mf, food_scor
    food_level_good = fuzz.interp_membership(food_quality, food_good_mf, food_score)
    food_level_excellent = fuzz.interp_membership(food_quality, food_excellent_mf, food_

    return service_level_poor, service_level_fair, service_level_good, service_level_very
           food_level_bad, food_level_average, food_level_good, food_level_excellent

# Rule base for fuzzy inference
def inference(service_level_poor, service_level_fair, service_level_good, service_level_
              food_level_bad, food_level_average, food_level_good, food_level_excellent)
    rule1 = min(service_level_poor, food_level_bad)
    rule2 = min(service_level_fair, food_level_bad)
    rule3 = min(service_level_good, food_level_bad)
    rule4 = min(service_level_very_good, food_level_good)
    rule5 = min(service_level_excellent, food_level_excellent)

    return rule1, rule2, rule3, rule4, rule5

# Defuzzification using centroid method
def defuzzify_output(rule1, rule2, rule3, rule4, rule5):
    tip_output = (
            (rule1 * 10) +
            (rule2 * 15) +
            (rule3 * 20) +
            (rule4 * 25) +
            (rule5 * 35)
    ) / (rule1 + rule2 + rule3 + rule4 + rule5)
    return tip_output

# Example usage: Service score = 8 (Good), Food score = 7 (Good)
service_score = 8
food_score = 7

service_level_poor, service_level_fair, service_level_good, service_level_very_good, ser
food_level_bad, food_level_average, food_level_good, food_level_excellent = fuzzify_inpu

rule1, rule2, rule3, rule4, rule5 = inference(service_level_poor, service_level_fair, se
                                              service_level_excellent, food_level_bad, 

tip = defuzzify_output(rule1, rule2, rule3, rule4, rule5)

print(f"The recommended tip for service score {service_score} and food score {food_score
```




```
The recommended tip for service score 8 and food score 7 is: nan%
```

```
C:\Users\ayush\AppData\Local\Temp\ipykernel_11920\3386839164.py:105: RuntimeWarning: inva
lid value encountered in scalar divide
  tip_output = (
```

9. Program to design fuzzy control system form AC temperature.

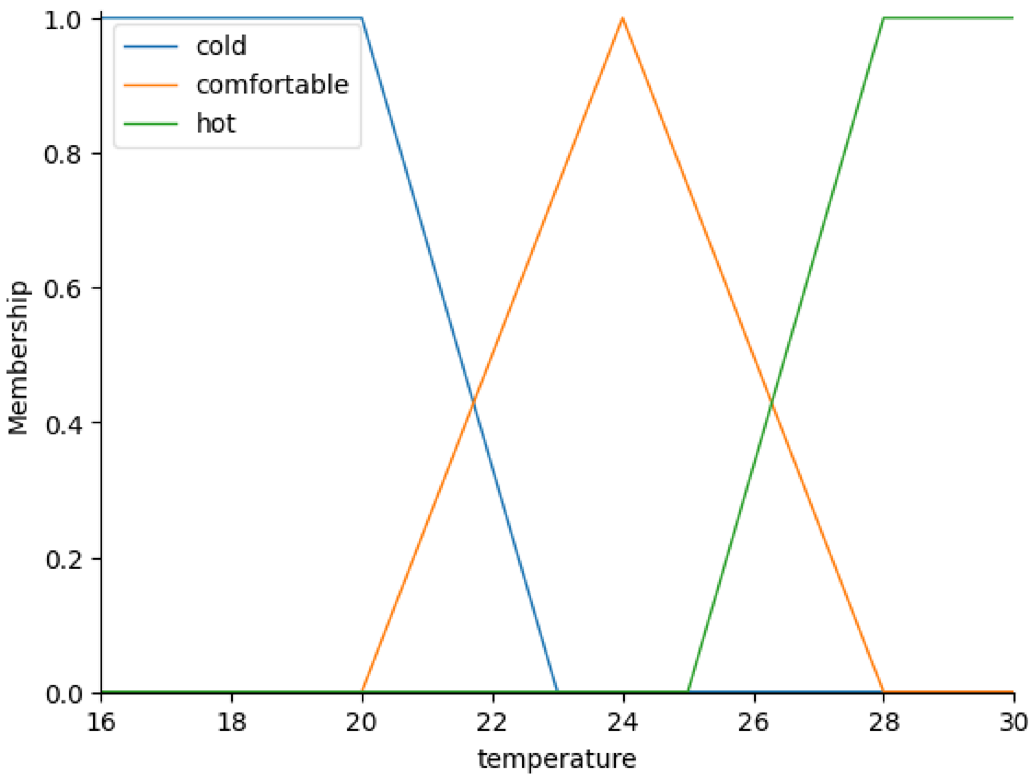
In [27]:

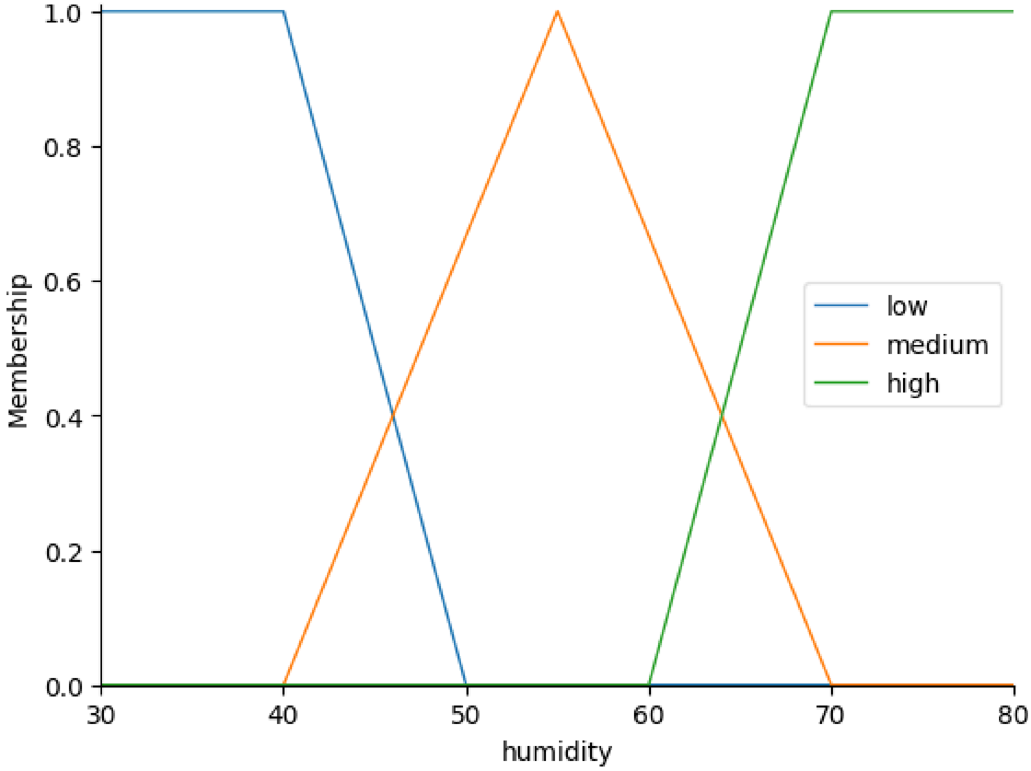
```python
# Import necessary libraries
import numpy as np
import skfuzzy as fuzz
from skfuzzy import control as ctrl

# Define fuzzy variables
temperature = ctrl.Antecedent(np.arange(16, 31, 1), 'temperature')  # Input: Temperature
humidity = ctrl.Antecedent(np.arange(30, 81, 1), 'humidity')  # Input: Humidity (30% to 8
fan_speed = ctrl.Consequent(np.arange(0, 11, 1), 'fan_speed')  # Output: Fan Speed (0 to

# Membership functions for temperature
temperature['cold'] = fuzz.trapmf(temperature.universe, [16, 16, 20, 23])
temperature['comfortable'] = fuzz.trimf(temperature.universe, [20, 24, 28])
temperature['hot'] = fuzz.trapmf(temperature.universe, [25, 28, 30, 30])

# Membership functions for humidity
humidity['low'] = fuzz.trapmf(humidity.universe, [30, 30, 40, 50])
humidity['medium'] = fuzz.trimf(humidity.universe, [40, 55, 70])
humidity['high'] = fuzz.trapmf(humidity.universe, [60, 70, 80, 80])

# Membership functions for fan speed
fan_speed['low'] = fuzz.trimf(fan_speed.universe, [0, 2, 4])
fan_speed['medium'] = fuzz.trimf(fan_speed.universe, [3, 5, 7])
fan_speed['high'] = fuzz.trimf(fan_speed.universe, [6, 8, 10])

# Define fuzzy rules
rule1 = ctrl.Rule(temperature['cold'] & humidity['low'], fan_speed['low'])
rule2 = ctrl.Rule(temperature['cold'] & humidity['medium'], fan_speed['low'])
rule3 = ctrl.Rule(temperature['cold'] & humidity['high'], fan_speed['medium'])
rule4 = ctrl.Rule(temperature['comfortable'] & humidity['low'], fan_speed['low'])
```

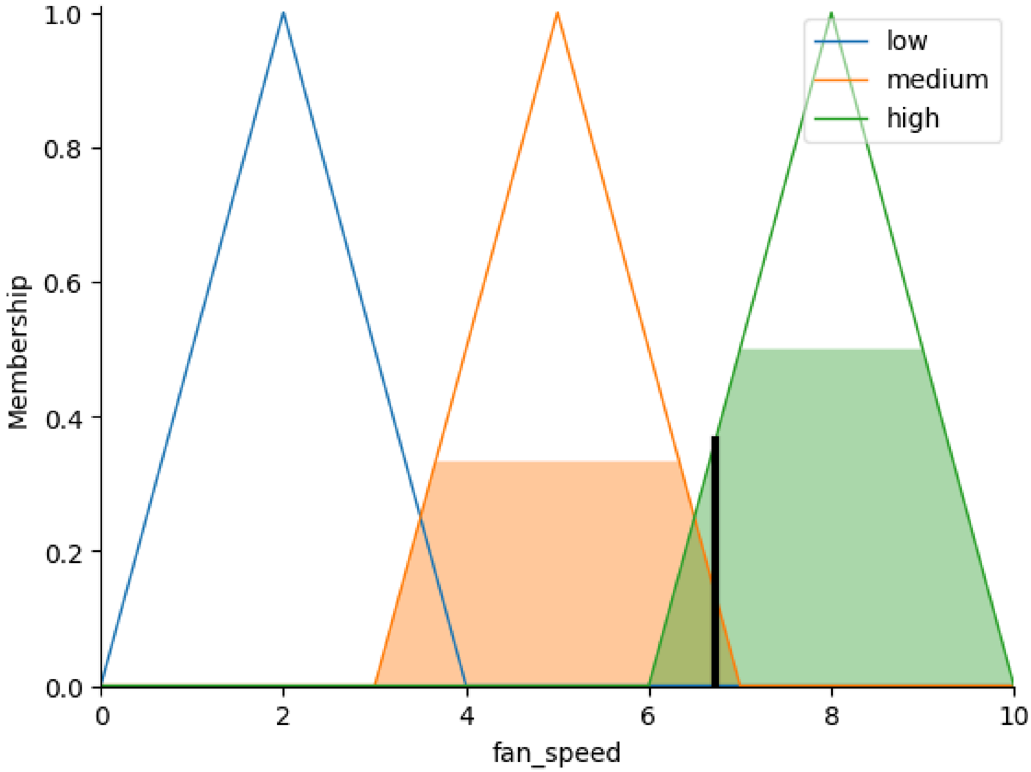
```
rule5 = ctrl.Rule(temperature['comfortable'] & humidity['medium'], fan_speed['medium'])
rule6 = ctrl.Rule(temperature['comfortable'] & humidity['high'], fan_speed['high'])
rule7 = ctrl.Rule(temperature['hot'] & humidity['low'], fan_speed['medium'])
rule8 = ctrl.Rule(temperature['hot'] & humidity['medium'], fan_speed['high'])
rule9 = ctrl.Rule(temperature['hot'] & humidity['high'], fan_speed['high'])

# Create and simulate the fuzzy control system
ac_ctrl = ctrl.ControlSystem([rule1, rule2, rule3, rule4, rule5, rule6, rule7, rule8, ru
ac_simulation = ctrl.ControlSystemSimulation(ac_ctrl)

# Test the fuzzy system
ac_simulation.input['temperature'] = 26  # Input temperature
ac_simulation.input['humidity'] = 65  # Input humidity

# Compute the result
ac_simulation.compute()
print(f"Recommended fan speed: {ac_simulation.output['fan_speed']}")

# Plot results
temperature.view()
humidity.view()
fan_speed.view(sim=ac_simulation)
```

```
Recommended fan speed: 6.732600732600733
```

```
c:\Users\ayush\AppData\Local\Programs\Python\Python312\Lib\site-packages\skfuzzy\control
\fuzzyvariable.py:125: UserWarning: FigureCanvasAgg is non-interactive, and thus cannot b
e shown
  fig.show()
```






10. Program to implement various Genetic operators like crossover, mutation and selection.

In [28]:
```
import random

# Define a fitness function
def fitness_function(individual):
    # Fitness is the sum of bits (maximize 1's)
    return sum(individual)

# Create initial population
def create_population(size, chromosome_length):
    return [[random.randint(0, 1) for _ in range(chromosome_length)] for _ in range(size
```

```python
# Selection: Roulette Wheel Selection
def select_parents(population, fitnesses):
    total_fitness = sum(fitnesses)
    selection_probs = [f / total_fitness for f in fitnesses]
    parents = random.choices(population, weights=selection_probs, k=2)
    return parents

# Crossover: Single-Point Crossover
def crossover(parent1, parent2):
    point = random.randint(1, len(parent1) - 1)
    child1 = parent1[:point] + parent2[point:]
    child2 = parent2[:point] + parent1[point:]
    return child1, child2

# Mutation: Flip Bit Mutation
def mutate(individual, mutation_rate):
    for i in range(len(individual)):
        if random.random() < mutation_rate:
            individual[i] = 1 - individual[i]  # Flip the bit
    return individual

# Genetic Algorithm
def genetic_algorithm(pop_size, chromosome_length, generations, mutation_rate):
    # Initialize population
    population = create_population(pop_size, chromosome_length)

    for generation in range(generations):
        # Calculate fitness for each individual
        fitnesses = [fitness_function(ind) for ind in population]

        # Create new population
        new_population = []
        while len(new_population) < pop_size:
            # Select parents
            parent1, parent2 = select_parents(population, fitnesses)

            # Perform crossover
            child1, child2 = crossover(parent1, parent2)

            # Perform mutation
            child1 = mutate(child1, mutation_rate)
            child2 = mutate(child2, mutation_rate)

            # Add children to new population
            new_population.extend([child1, child2])

        # Replace old population with new one
        population = new_population[:pop_size]

        # Print best fitness in each generation
        best_fitness = max(fitnesses)
        print(f"Generation {generation + 1}: Best Fitness = {best_fitness}")

    # Return the best solution
    best_individual = max(population, key=fitness_function)
    return best_individual

# Parameters
population_size = 10
chromosome_length = 8
num_generations = 20
mutation_rate = 0.1

# Run Genetic Algorithm
```

```
best_solution = genetic_algorithm(population_size, chromosome_length, num_generations, mu
print("Best Solution:", best_solution)
print("Fitness of Best Solution:", fitness_function(best_solution))
```

```
Generation 1: Best Fitness = 6
Generation 2: Best Fitness = 6
Generation 3: Best Fitness = 6
Generation 4: Best Fitness = 7
Generation 5: Best Fitness = 6
Generation 6: Best Fitness = 7
Generation 7: Best Fitness = 8
Generation 8: Best Fitness = 8
Generation 9: Best Fitness = 8
Generation 10: Best Fitness = 8
Generation 11: Best Fitness = 7
Generation 12: Best Fitness = 7
Generation 13: Best Fitness = 7
Generation 14: Best Fitness = 7
Generation 15: Best Fitness = 7
Generation 16: Best Fitness = 8
Generation 17: Best Fitness = 7
Generation 18: Best Fitness = 7
Generation 19: Best Fitness = 7
Generation 20: Best Fitness = 6
Best Solution: [1, 0, 1, 1, 1, 1, 1, 1]
Fitness of Best Solution: 7
```

11. Program to implement Genetic Algorithm to maximize the objective function such as f(x)=x^2 where x can have values from 0 to 31.

In [29]:
```
import random

# Objective function
def objective_function(x):
    return x**2

# Convert binary to integer
def binary_to_int(binary):
    return int("".join(map(str, binary)), 2)

# Fitness function
def fitness_function(individual):
    x = binary_to_int(individual)
    return objective_function(x)

# Generate initial population
def create_population(size, chromosome_length):
    return [[random.randint(0, 1) for _ in range(chromosome_length)] for _ in range(size

# Selection: Roulette Wheel Selection
def select_parents(population, fitnesses):
    total_fitness = sum(fitnesses)
    selection_probs = [f / total_fitness for f in fitnesses]
    parents = random.choices(population, weights=selection_probs, k=2)
    return parents

# Crossover: Single-Point Crossover
def crossover(parent1, parent2):
    point = random.randint(1, len(parent1) - 1)
    child1 = parent1[:point] + parent2[point:]
    child2 = parent2[:point] + parent1[point:]
    return child1, child2

# Mutation: Flip Bit Mutation
```

```python
def mutate(individual, mutation_rate):
    for i in range(len(individual)):
        if random.random() < mutation_rate:
            individual[i] = 1 - individual[i]  # Flip the bit
    return individual

# Genetic Algorithm
def genetic_algorithm(pop_size, chromosome_length, generations, mutation_rate):
    # Initialize population
    population = create_population(pop_size, chromosome_length)

    for generation in range(generations):
        # Calculate fitness for each individual
        fitnesses = [fitness_function(ind) for ind in population]

        # Create new population
        new_population = []
        while len(new_population) < pop_size:
            # Select parents
            parent1, parent2 = select_parents(population, fitnesses)

            # Perform crossover
            child1, child2 = crossover(parent1, parent2)

            # Perform mutation
            child1 = mutate(child1, mutation_rate)
            child2 = mutate(child2, mutation_rate)

            # Add children to new population
            new_population.extend([child1, child2])

        # Replace old population with new one
        population = new_population[:pop_size]

        # Print best fitness in each generation
        best_fitness = max(fitnesses)
        best_individual = population[fitnesses.index(best_fitness)]
        print(f"Generation {generation + 1}: Best Fitness = {best_fitness}, Best x = {bi

    # Return the best solution
    best_individual = max(population, key=fitness_function)
    best_x = binary_to_int(best_individual)
    return best_individual, best_x

# Parameters
population_size = 10
chromosome_length = 5  # Binary representation for numbers 0-31
num_generations = 20
mutation_rate = 0.1

# Run Genetic Algorithm
best_solution, best_x = genetic_algorithm(population_size, chromosome_length, num_genera
print("\nBest Solution (Binary):", best_solution)
print("Best x:", best_x)
print("Maximized f(x) = x^2:", objective_function(best_x))
```

```
Generation 1: Best Fitness = 961, Best x = 10
Generation 2: Best Fitness = 784, Best x = 4
Generation 3: Best Fitness = 841, Best x = 25
Generation 4: Best Fitness = 900, Best x = 24
Generation 5: Best Fitness = 841, Best x = 13
Generation 6: Best Fitness = 961, Best x = 17
Generation 7: Best Fitness = 961, Best x = 29
Generation 8: Best Fitness = 961, Best x = 30
Generation 9: Best Fitness = 900, Best x = 21
Generation 10: Best Fitness = 841, Best x = 25
Generation 11: Best Fitness = 841, Best x = 21
Generation 12: Best Fitness = 900, Best x = 31
Generation 13: Best Fitness = 961, Best x = 21
Generation 14: Best Fitness = 961, Best x = 31
Generation 15: Best Fitness = 961, Best x = 20
Generation 16: Best Fitness = 961, Best x = 31
Generation 17: Best Fitness = 961, Best x = 31
Generation 18: Best Fitness = 961, Best x = 25
Generation 19: Best Fitness = 961, Best x = 30
Generation 20: Best Fitness = 961, Best x = 31

Best Solution (Binary): [1, 1, 1, 1, 1]
Best x: 31
Maximized f(x) = x^2: 961
```