Amity School of Engineering and Technology

Amity University Noida

Soft Computing and Applications

[CSE320]

Lab File

Submitted To: -

Dr Bedatri Moulik

Submitted by: -

Ojaswin Aggarwal

7CSE 2X

A2305222103

Index

Experiment No	Experiments	Remarks
1	To perform basic operations on crisp and fuzzy sets	
2	To implement triangular, trapezoidal, gaussian membership functions graphically and perform union, intersection, complement on them	
3	To perform max-min and max product composition on crisp and fuzzy relations	
4	Design a Fuzzy Logic Controller (FLC) for controlling heater power based on room temperature	
5	Train a single perceptron using the Sigmoid activation and MSE loss to update parameters via gradient descent A) Compute updated weights and bias after one training step B) Write Python code to train for multiple epochs until convergence, showing loss and final parameters	
6	Train a 2-layer neural network (1 hidden layer + 1 output layer) A) Compute updated parameters after one training step B) Implement the training in Python for multiple epochs until convergence	
7	Implement and demonstrate various crossover operations in Python for genetic algorithms	
8	Implement and demonstrate various mutation operations in Python for genetic algorithms	
9	Implement a GA from scratch that takes the objective function and iteratively solves it, returning the best solutions	

Experiment 1

Aim: To perform basic operations on crisp and fuzzy sets

Theory: Fuzzy sets allow elements to have partial membership between 0 and 1, representing uncertainty and vagueness.
Operations like union, intersection, complement, and difference are based on max, min, and inversion of membership values.
The probabilistic sum and product extend fuzzy logic to model combined or joint memberships.
These operations form the foundation for fuzzy logic systems used in AI, control, and decision-making.

Code:

```
# Example to Demonstrate the Union of Two Fuzzy Sets
A = dict()
B = dict()
Y = dict()

A = {"a": 0.2, "b": 0.3, "c": 0.6, "d": 0.6}
B = {"a": 0.9, "b": 0.9, "c": 0.4, "d": 0.5}

print('The First Fuzzy Set is :', A)
print('The Second Fuzzy Set is :', B)

for A_key, B_key in zip(A, B):
    A_value = A[A_key]
    B_value = B[B_key]

    if A_value > B_value:
        Y[A_key] = A_value
    else:
        Y[B_key] = B_value

print('Fuzzy Set Union is :', Y)

# Example to Demonstrate the Intersection Between Two Fuzzy Sets

A = dict()
B = dict()
Y = dict()

A = {"a": 0.2, "b": 0.3, "c": 0.6, "d": 0.6}
B = {"a": 0.9, "b": 0.9, "c": 0.4, "d": 0.5}
print("")
print('The First Fuzzy Set is :', A)
print('The Second Fuzzy Set is :', B)
```

```
for A_key, B_key in zip(A, B):
    A_value = A[A_key]
    B_value = B[B_key]

    if A_value < B_value:
        Y[A_key] = A_value
    else:
        Y[B_key] = B_value
print('Fuzzy Set Intersection is :', Y)
print("")
# Example to Demonstrate the Complement Between Two Fuzzy Sets
A = dict()
Y = dict()

A = {"a": 0.2, "b": 0.3, "c": 0.6, "d": 0.6}

print('The Fuzzy Set is :', A)

for A_key in A:
   Y[A_key]= 1-A[A_key]

print('Fuzzy Set Complement is :', Y)

# Example to Demonstrate the Difference Between Two Fuzzy Sets
A = dict()
B = dict()
Y = dict()

A = {"a": 0.2, "b": 0.3, "c": 0.6, "d": 0.6}
B = {"a": 0.9, "b": 0.9, "c": 0.4, "d": 0.5}

print("")
print('The First Fuzzy Set is :', A)
print('The Second Fuzzy Set is :', B)

for A_key, B_key in zip(A, B):
    A_value = A[A_key]
    B_value = B[B_key]
    B_value = 1 - B_value

    if A_value < B_value:
        Y[A_key] = A_value
    else:
        Y[B_key] = B_value
print('Fuzzy Set Difference is :', Y)

# Example to Demonstrate the Sum Between Two Fuzzy Sets
A = {"a": 0.2, "b": 0.3, "c": 0.6, "d": 0.6}
B = {"a": 0.9, "b": 0.9, "c": 0.4, "d": 0.5}
```

```
Y = dict()

# Probabilistic sum: A + B = A + B - A*B
for key in A:
    Y[key] = A[key] + B[key] - A[key]*B[key]
print("")
print("Fuzzy Set A:", A)
print("Fuzzy Set B:", B)
print("")
print("Fuzzy Set Probabilistic Sum (A + B):", Y)

# Example to Demonstrate the Product Between Two Fuzzy Sets
A_list = [0.2, 0.3, 0.6, 0.6]
B_list = [0.9, 0.9, 0.4, 0.5]
Y_list = [A_list[i] * B_list[i] for i in range(len(A_list))]

print("Fuzzy Set Product (A * B) using lists:", Y_list)
```

Output:

```
The First Fuzzy Set is : {'a': 0.2, 'b': 0.3, 'c': 0.6, 'd': 0.6}
The Second Fuzzy Set is : {'a': 0.9, 'b': 0.9, 'c': 0.4, 'd': 0.5}
Fuzzy Set Union is : {'a': 0.9, 'b': 0.9, 'c': 0.6, 'd': 0.6}

The First Fuzzy Set is : {'a': 0.2, 'b': 0.3, 'c': 0.6, 'd': 0.6}
The Second Fuzzy Set is : {'a': 0.9, 'b': 0.9, 'c': 0.4, 'd': 0.5}
Fuzzy Set Intersection is : {'a': 0.2, 'b': 0.3, 'c': 0.4, 'd': 0.5}

The Fuzzy Set is : {'a': 0.2, 'b': 0.3, 'c': 0.6, 'd': 0.6}
Fuzzy Set Complement is : {'a': 0.8, 'b': 0.7, 'c': 0.4, 'd': 0.4}

The First Fuzzy Set is : {'a': 0.2, 'b': 0.3, 'c': 0.6, 'd': 0.6}
The Second Fuzzy Set is : {'a': 0.9, 'b': 0.9, 'c': 0.4, 'd': 0.5}
Fuzzy Set Difference is : {'a': 0.09999999999999998, 'b': 0.09999999999999998, 'c': 0.6, 'd': 0.5}

Fuzzy Set A: {'a': 0.2, 'b': 0.3, 'c': 0.6, 'd': 0.6}
Fuzzy Set B: {'a': 0.9, 'b': 0.9, 'c': 0.4, 'd': 0.5}

Fuzzy Set Probabilistic Sum (A + B): {'a': 0.92, 'b': 0.9299999999999999, 'c': 0.76, 'd': 0.8}
Fuzzy Set Product (A * B) using lists: [0.18000000000000002, 0.27, 0.24, 0.3]


...Program finished with exit code 0
Press ENTER to exit console.
```

Experiment 2

Aim: To implement triangular, trapezoidal, gaussian membership functions graphically and perform union, intersection, complement on them.

Theory: Fuzzy sets model uncertainty by assigning each element a membership value between 0 and 1. The membership functions define how input values map to these degrees of belonging. The triangular, trapezoidal, and Gaussian functions are commonly used due to their smooth and intuitive shapes. The union of fuzzy sets takes the maximum membership value, the intersection takes the minimum, and the complement is computed as one minus the membership. These operations form the mathematical basis for fuzzy logic systems and decision-making applications.

Code:

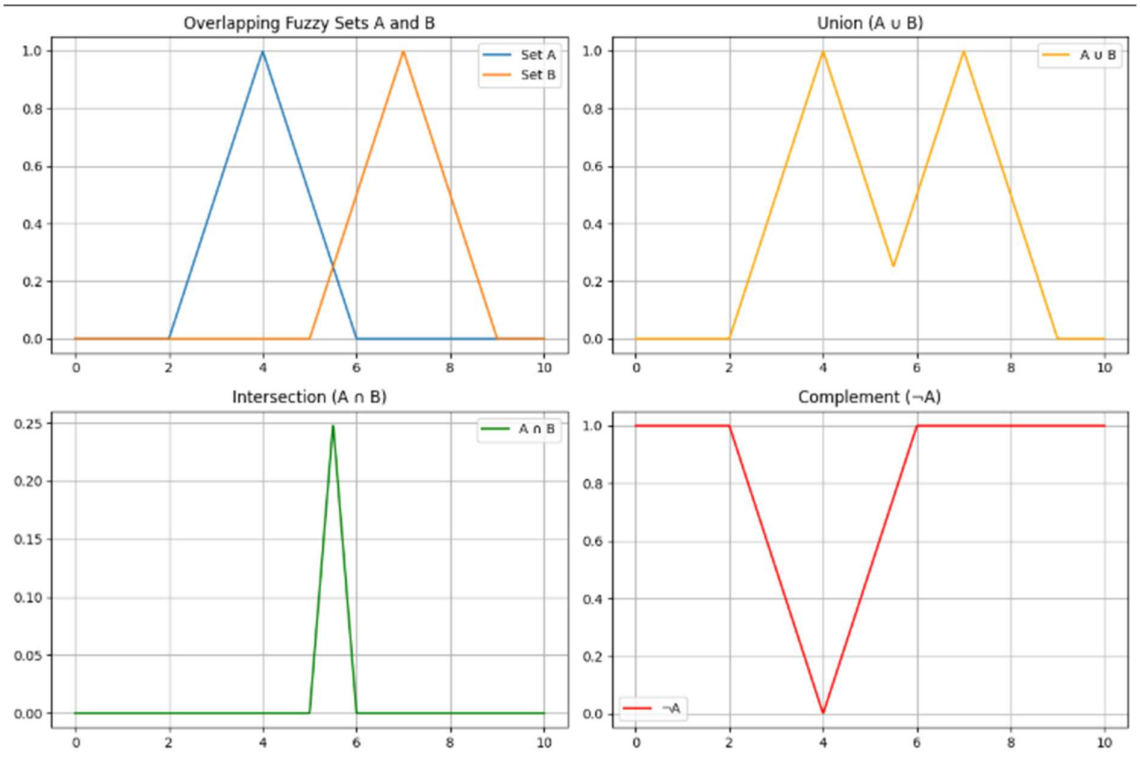
```
import numpy as np
import matplotlib.pyplot as plt

# Triangular membership function
def triangle(x, a, b, c):
    return np.maximum(
        np.minimum((x - a) / (b - a), (c - x) / (c - b)),
        0
    )

# Define the universe
x = np.linspace(0, 10, 1000)

# Define overlapping fuzzy sets
mu_A = triangle(x, 2, 4, 6)  # Centered at 4
mu_B = triangle(x, 5, 7, 9)  # Centered at 7

# Fuzzy operations
mu_union = np.maximum(mu_A, mu_B)
mu_intersection = np.minimum(mu_A, mu_B)
mu_complement_A = 1 - mu_A

# Plotting
fig, axs = plt.subplots(2, 2, figsize=(12, 8))

# Plot A and B
axs[0, 0].plot(x, mu_A, label='Set A')
axs[0, 0].plot(x, mu_B, label='Set B')
axs[0, 0].set_title('Overlapping Fuzzy Sets A and B')
axs[0, 0].legend()
axs[0, 0].grid(True)

# Plot union
axs[0, 1].plot(x, mu_union, color='orange', label='A ∪ B')
axs[0, 1].set_title('Union (A ∪ B)')
```

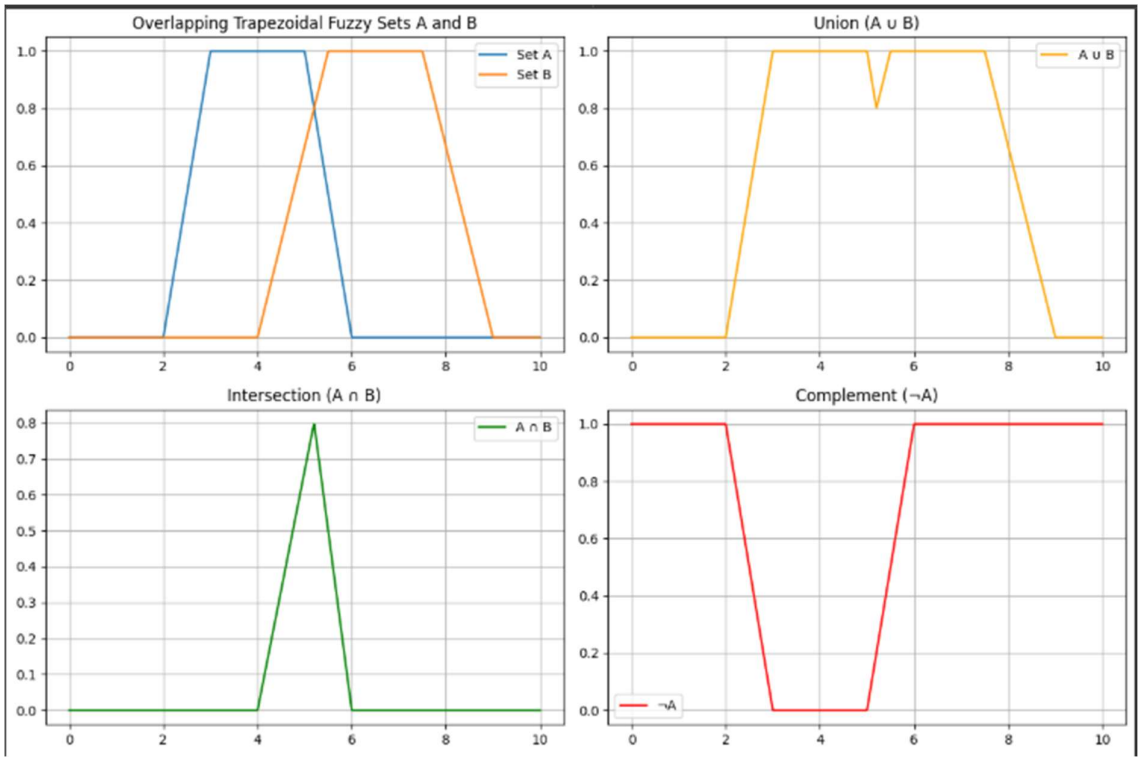
```
axs[0, 1].legend()
axs[0, 1].grid(True)

# Plot intersection
axs[1, 0].plot(x, mu_intersection, color='green', label='A ∩ B')
axs[1, 0].set_title('Intersection (A ∩ B)')
axs[1, 0].legend()
axs[1, 0].grid(True)

# Plot complement
axs[1, 1].plot(x, mu_complement_A, color='red', label='¬A')
axs[1, 1].set_title('Complement (¬A)')
axs[1, 1].legend()
axs[1, 1].grid(True)

plt.tight_layout()
plt.show()

# Trapezoidal membership function
def trapezoid(x, a, b, c, d):
    return np.maximum(
        np.minimum( np.minimum((x - a) / (b - a), 1),(d - x) / (d - c) ), 0 )

# Define the universe
x = np.linspace(0, 10, 1000)

# Define overlapping fuzzy sets (trapezoids)
mu_A = trapezoid(x, 2, 3, 5, 6)  # Plateau from 3 to 5
mu_B = trapezoid(x, 4, 5.5, 7.5, 9)  # Plateau from 5.5 to 7.5

# Fuzzy operations
mu_union = np.maximum(mu_A, mu_B)
mu_intersection = np.minimum(mu_A, mu_B)
mu_complement_A = 1 - mu_A

# Plotting
fig, axs = plt.subplots(2, 2, figsize=(12, 8))

# Plot A and B
axs[0, 0].plot(x, mu_A, label='Set A')
axs[0, 0].plot(x, mu_B, label='Set B')
axs[0, 0].set_title('Overlapping Trapezoidal Fuzzy Sets A and B')
axs[0, 0].legend()
axs[0, 0].grid(True)

# Plot union
axs[0, 1].plot(x, mu_union, color='orange', label='A ∪ B')
axs[0, 1].set_title('Union (A ∪ B)')
axs[0, 1].legend()
axs[0, 1].grid(True)
```

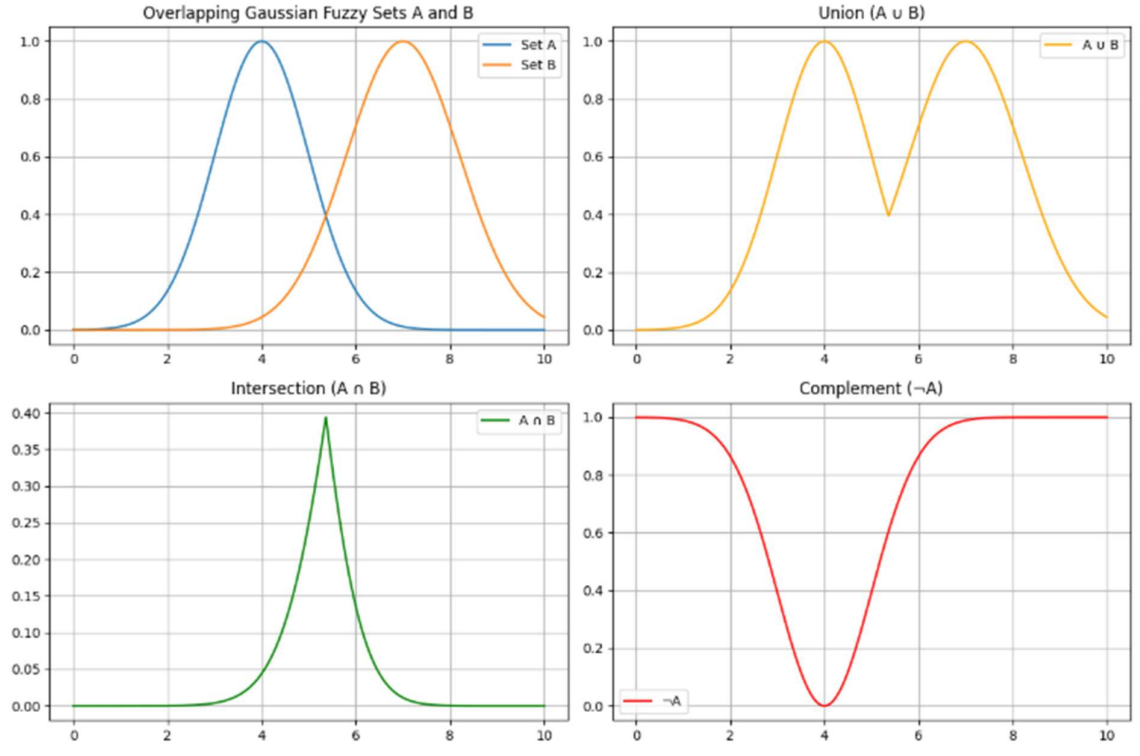
```
# Plot intersection
axs[1, 0].plot(x, mu_intersection, color='green', label='A ∩ B')
axs[1, 0].set_title('Intersection (A ∩ B)')
axs[1, 0].legend()
axs[1, 0].grid(True)

# Plot complement
axs[1, 1].plot(x, mu_complement_A, color='red', label='¬A')
axs[1, 1].set_title('Complement (¬A)')
axs[1, 1].legend()
axs[1, 1].grid(True)

plt.tight_layout()
plt.show()

# Gaussian membership function
def gaussian(x, c, sigma):
    return np.exp(-0.5 * ((x - c) / sigma) ** 2)

# Define the universe
x = np.linspace(0, 10, 1000)

# Define overlapping fuzzy sets (Gaussian)
mu_A = gaussian(x, c=4, sigma=1)   # Center 4, spread 1
mu_B = gaussian(x, c=7, sigma=1.2) # Center 7, spread 1.2

# Fuzzy operations
mu_union = np.maximum(mu_A, mu_B)
mu_intersection = np.minimum(mu_A, mu_B)
mu_complement_A = 1 - mu_A

# Plotting
fig, axs = plt.subplots(2, 2, figsize=(12, 8))

# Plot A and B
axs[0, 0].plot(x, mu_A, label='Set A')
axs[0, 0].plot(x, mu_B, label='Set B')
axs[0, 0].set_title('Overlapping Gaussian Fuzzy Sets A and B')
axs[0, 0].legend()
axs[0, 0].grid(True)

# Plot union
axs[0, 1].plot(x, mu_union, color='orange', label='A ∪ B')
axs[0, 1].set_title('Union (A ∪ B)')
axs[0, 1].legend()
axs[0, 1].grid(True)

# Plot intersection
axs[1, 0].plot(x, mu_intersection, color='green', label='A ∩ B')
```

```
axs[1, 0].set_title('Intersection (A ∩ B)')
axs[1, 0].legend()
axs[1, 0].grid(True)

# Plot complement
axs[1, 1].plot(x, mu_complement_A, color='red', label='¬A')
axs[1, 1].set_title('Complement (¬A)')
axs[1, 1].legend()
axs[1, 1].grid(True)

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

Output:




Overlapping Trapezoidal Fuzzy Sets A and B
Set A
Set B
Union (A ∪ B)
A ∪ B
Intersection (A ∩ B)
A ∩ B
Complement (¬A)
¬A


Overlapping Gaussian Fuzzy Sets A and B
Set A
Set B
Union (A ∪ B)
A ∪ B
Intersection (A ∩ B)
A ∩ B
Complement (¬A)
¬A

Experiment 3

Aim: To perform max-min and max product composition on crisp and fuzzy relations.

Theory: Fuzzy relations extend classical relations by allowing partial associations between elements, represented by values in the range [0, 1]. The **composition of fuzzy relations** combines two fuzzy matrices to establish indirect relationships. In the **max–min composition**, each element of the result is obtained by taking the maximum of all minimum pairs between corresponding elements of the two matrices. Mathematically:

$$T(i,j) = \max_k [\min (R(i,k), S(k,j))]$$

In the **max–product composition**, the minimum operator is replaced with multiplication to capture weighted influence:

$$T(i,j) = \max_k [R(i,k) \times S(k,j)]$$

These operations are widely used in fuzzy inference systems, control, and decision-making processes where relationships between uncertain variables need to be evaluated.

Code:

```
import numpy as np

def min_max_composition(R, S):
    # R: m x n matrix
    # S: n x p matrix
    m, n = R.shape
    n2, p = S.shape
    assert n == n2, "Inner dimensions must match for composition."

    T = np.zeros((m, p))

    for i in range(m):
        for j in range(p):
            # Calculate max over k of min(R[i,k], S[k,j])
            min_values = [min(R[i, k], S[k, j]) for k in range(n)]
            T[i, j] = max(min_values)

    return T

# Example usage with your input matrices and n=2:

n = 2
R = np.array([
    [0.7, 0.6],
    [0.8, 0.3],

])
```

```
S = np.array([
    [0.8, 0.5, 0.4],
    [0.1, 0.6, 0.7],

])

T = min_max_composition(R, S)

print("Matrix R:")
print(R)
print("\nMatrix S:")
print(S)
print("\nMin-Max Composition T = R ∘ S:")
print(T)

def max_product_composition(R, S):
    # R: m x n matrix
    # S: n x p matrix
    m, n = R.shape
    n2, p = S.shape
    assert n == n2, "Inner dimensions must match for composition."

    T = np.zeros((m, p))

    for i in range(m):
        for j in range(p):
            # Calculate max over k of R[i,k] * S[k,j]
            prod_values = [R[i, k] * S[k, j] for k in range(n)]
            T[i, j] = max(prod_values)

    return T

# Example usage with your matrices:
R = np.array([
    [0.7, 0.6],
    [0.8, 0.3],
])

S = np.array([
    [0.8, 0.5, 0.4],
    [0.1, 0.6, 0.7],
])

T = max_product_composition(R, S)

print("Matrix R:")
print(R)
print("\nMatrix S:")
print(S)
```

```
print("\nMax-Product Composition T = R ∘ S:")
print(T)
```

Output:

```
Matrix R:
[[0.7 0.6]
 [0.8 0.3]]

Matrix S:
[[0.8 0.5 0.4]
 [0.1 0.6 0.7]]

Min-Max Composition T = R ∘ S:
[[0.7 0.6 0.6]
 [0.8 0.5 0.4]]
Matrix R:
[[0.7 0.6]
 [0.8 0.3]]

Matrix S:
[[0.8 0.5 0.4]
 [0.1 0.6 0.7]]

Max-Product Composition T = R ∘ S:
[[0.56 0.36 0.42]
 [0.64 0.4  0.32]]
```

Experiment 4

Aim: Design a Fuzzy Logic Controller (FLC) for controlling heater power based on room temperature.

- Input: Temperature (°C)
- Output: Heater Power (0–100%)

Use triangular membership functions for simplicity with the following fuzzy sets:

- Temperature: Cold, Warm, Hot
- Heater Power: Low, Medium, High

Define the fuzzy rules as:

1. IF Temperature is Cold THEN Heater is High
2. IF Temperature is Warm THEN Heater is Medium
3. IF Temperature is Hot THEN Heater is Low

For an input temperature of 22°C, determine:

1. The fuzzy output (membership values for each output fuzzy set)
2. The crisp output (defuzzified heater power).

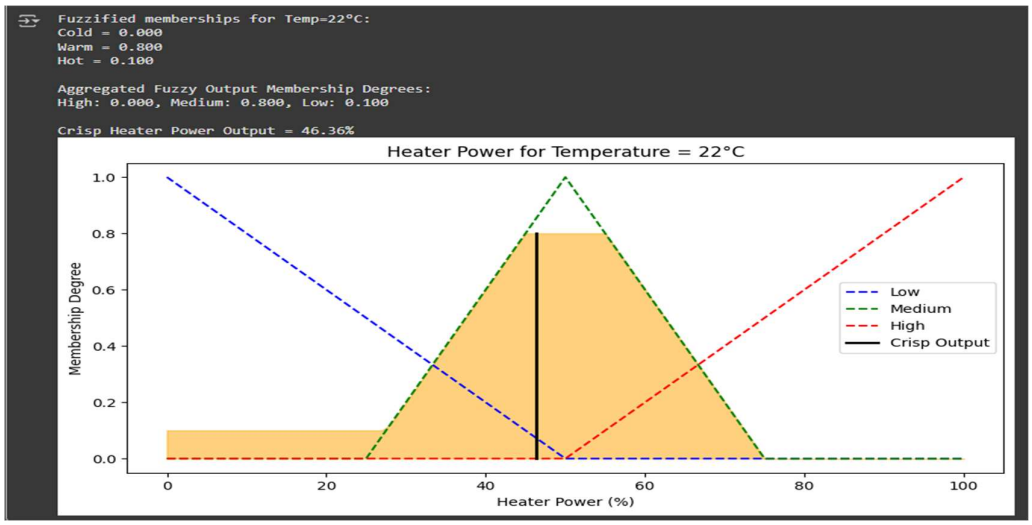
Theory:

Code:

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

# Define universes
temp = np.arange(0, 41, 1)       # Temperature (°C)
heater = np.arange(0, 101, 1)    # Heater Power (%)

# Define triangular membership functions
temp_cold = fuzz.trimf(temp, [0, 0, 20])
temp_warm = fuzz.trimf(temp, [10, 20, 30])
temp_hot = fuzz.trimf(temp, [20, 40, 40])

heater_low = fuzz.trimf(heater, [0, 0, 50])
heater_medium = fuzz.trimf(heater, [25, 50, 75])
heater_high = fuzz.trimf(heater, [50, 100, 100])

# Input temperature
temp_input = 22

# Fuzzification
μ_cold = fuzz.interp_membership(temp, temp_cold, temp_input)
μ_warm = fuzz.interp_membership(temp, temp_warm, temp_input)
μ_hot = fuzz.interp_membership(temp, temp_hot, temp_input)

print(f"Fuzzified memberships for Temp={temp_input}°C:")
print(f"Cold = {μ_cold:.3f}")
print(f"Warm = {μ_warm:.3f}")
print(f"Hot = {μ_hot:.3f}")

# Rule Application
```

```
# IF Cold → High
rule1 = np.fmin(μ_cold, heater_high)
# IF Warm → Medium
rule2 = np.fmin(μ_warm, heater_medium)
# IF Hot → Low
rule3 = np.fmin(μ_hot, heater_low)

# Combine all rules (OR = max)
aggregated = np.fmax(rule1, np.fmax(rule2, rule3))

# Defuzzify using centroid
crisp_output = fuzz.defuzz(heater, aggregated, 'centroid')
μ_output = fuzz.interp_membership(heater, aggregated, crisp_output)

print("\nAggregated Fuzzy Output Membership Degrees:")
print(f"High: {μ_cold:.3f}, Medium: {μ_warm:.3f}, Low: {μ_hot:.3f}")

print(f"\nCrisp Heater Power Output = {crisp_output:.2f}%")

# Plotting
plt.figure(figsize=(10, 5))
plt.plot(heater, heater_low, 'b', linestyle='--', label='Low')
plt.plot(heater, heater_medium, 'g', linestyle='--', label='Medium')
plt.plot(heater, heater_high, 'r', linestyle='--', label='High')
plt.fill_between(heater, aggregated, color='orange', alpha=0.5)
plt.plot([crisp_output, crisp_output], [0, μ_output], 'k', linewidth=2, label='Crisp Output')
plt.title(f'Heater Power for Temperature = {temp_input}°C')
plt.xlabel('Heater Power (%)')
plt.ylabel('Membership Degree')
plt.legend()
plt.show()
```

Output:



Experiment 5

Aim: Train a single perceptron using the Sigmoid activation and MSE loss to update parameters via gradient descent.
Given:
X = [0.6, 0.9], W = [0.4, 0.3], b = 0.2, t = 1, η = 0.1, Activation: Sigmoid, Loss: MSE
Tasks:
A) Compute updated weights and bias after one training step.
B) Write Python code to train for multiple epochs until convergence, showing loss and final parameters.

Theory: A single-layer perceptron is the simplest form of an artificial neural network. It takes an input vector X, computes a weighted sum using W, adds a bias b, and applies an activation function to produce an output. The sigmoid activation function introduces nonlinearity and squashes the output into the range (0,1), suitable for binary classification tasks.

The mean squared error (MSE) loss quantifies the difference between the target output and the actual output. The model learns by minimizing this loss through gradient descent, updating weights and bias in the direction that reduces the error:

$$W_{new} = W - \eta \frac{\partial E}{\partial W}, b_{new} = b - \eta \frac{\partial E}{\partial b}$$

where $\eta$is the learning rate. This iterative process continues until the model converges (loss becomes minimal), resulting in optimized parameters that best fit the data.

Code:

```
import numpy as np

# ========================
# PART A — ONE TRAINING STEP
# ========================

# Given values
X = np.array([0.6, 0.9])
W = np.array([0.4, 0.3])
b = 0.2
t = 1
eta = 0.1

def sigmoid(z):
    return 1 / (1 + np.exp(-z))

# ---- Forward Pass ----
z = np.dot(W, X) + b
y = sigmoid(z)

# ---- Loss (MSE) ----
```

```
E = 0.5 * (t - y)**2

# ---- Backpropagation ----
dE_dy = -(t - y)
dy_dz = y * (1 - y)
dE_dz = dE_dy * dy_dz

# Gradients for weights and bias
dE_dW = dE_dz * X
dE_db = dE_dz

# ---- Parameter Update ----
W_new = W - eta * dE_dW
b_new = b - eta * dE_db

print("=== PART A: One Training Step ===")
print(f"Output y = {y:.4f}")
print(f"Loss = {E:.6f}")
print(f"Updated Weights = {W_new}")
print(f"Updated Bias = {b_new:.4f}")
print("\n")

# ========================
# PART B — MULTIPLE EPOCHS (TRAINING UNTIL CONVERGENCE)
# ========================

# Reset initial parameters
W = np.array([0.4, 0.3])
b = 0.2

def mse(t, y):
    return 0.5 * (t - y)**2

print("=== PART B: Training Over Multiple Epochs ===")

# Train for multiple epochs
for epoch in range(1, 101):
    # Forward pass
    z = np.dot(W, X) + b
    y = sigmoid(z)

    # Compute loss
    loss = mse(t, y)

    # Backpropagation
    dE_dy = -(t - y)
    dy_dz = y * (1 - y)
    dE_dz = dE_dy * dy_dz

    # Gradients
    dE_dW = dE_dz * X
    dE_db = dE_dz

    # Update parameters
    W -= eta * dE_dW
```

```
    b -= eta * dE_db

    # Show progress every 10 epochs
    if epoch % 10 == 0:
        print(f"Epoch {epoch}: Loss={loss:.6f}, y={y:.4f}, W={W}, b={b:.4f}")

print("\nFinal parameters after training:")
print("W =", W)
print("b =", b)
```

Output:

```
=== PART A: One Training Step ===
Output y = 0.6704
Loss = 0.054318
Updated Weights = [0.40436976 0.30655464]
Updated Bias = 0.2073


=== PART B: Training Over Multiple Epochs ===
Epoch 10: Loss=0.045232, y=0.6992, W=[0.44074015 0.36111023], b=0.2679
Epoch 20: Loss=0.037663, y=0.7255, W=[0.47575911 0.41363866], b=0.3263
Epoch 30: Loss=0.031954, y=0.7472, W=[0.50619947 0.4592992 ], b=0.3770
Epoch 40: Loss=0.027553, y=0.7653, W=[0.53295187 0.49942781], b=0.4216
Epoch 50: Loss=0.024090, y=0.7805, W=[0.55670338 0.53505508], b=0.4612
Epoch 60: Loss=0.021314, y=0.7935, W=[0.57798504 0.56697756], b=0.4966
Epoch 70: Loss=0.019051, y=0.8048, W=[0.59721044 0.59581566], b=0.5287
Epoch 80: Loss=0.017179, y=0.8146, W=[0.614705  0.6220575], b=0.5578
Epoch 90: Loss=0.015611, y=0.8233, W=[0.63072765 0.64609147], b=0.5845
Epoch 100: Loss=0.014281, y=0.8310, W=[0.6454867  0.66823005], b=0.6091

Final parameters after training:
W = [0.6454867  0.66823005]
b = 0.6091444957384374
```

Experiment 6

Aim: Train a 2-layer neural network (1 hidden layer + 1 output layer) with the following parameters:
• Input: X = [1, 0]
• Hidden layer: 2 neurons
• Output layer: 1 neuron
• Activation: Sigmoid
• Learning rate: $\eta = 0.5$
• Target: t = 1
A) Compute updated parameters after one training step.
B) Implement the training in Python for multiple epochs until convergence.

Theory: A two-layer neural network consists of one hidden layer and one output layer. The input passes through the hidden layer where weights and biases are applied, and the sigmoid activation introduces nonlinearity. The output is compared with the target to compute the error. Using backpropagation and gradient descent, the weights and biases are updated to minimize the mean squared error (MSE). This iterative process continues until the network output converges close to the target value.

Code:

```
import numpy as np

# Sigmoid activation and derivative
def sigmoid(x):
    return 1 / (1 + np.exp(-x))

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

# Given values
X = np.array([[1, 0]])  # input
t = np.array([[1]])     # target
eta = 0.5               # learning rate

# Initialize weights and biases
np.random.seed(42)
W1 = np.random.rand(2, 2)
b1 = np.random.rand(1, 2)
W2 = np.random.rand(2, 1)
b2 = np.random.rand(1, 1)

print("Initial Parameters:")
print("W1:", W1)
print("b1:", b1)
print("W2:", W2)
print("b2:", b2)
print("\n================ PART A ================\n")

# ------------------------------
# PART A: One training step
```

```
# --------------------------------

# Forward pass
hidden_input = np.dot(X, W1) + b1
hidden_output = sigmoid(hidden_input)
final_input = np.dot(hidden_output, W2) + b2
output = sigmoid(final_input)

print("Forward Pass Values:")
print("Hidden Input:", hidden_input)
print("Hidden Output:", hidden_output)
print("Final Input:", final_input)
print("Output:", output)

# Compute error
error = t - output
print("Error:", error)

# Backpropagation
d_output = error * sigmoid_derivative(output)
d_hidden = d_output.dot(W2.T) * sigmoid_derivative(hidden_output)

# Update weights and biases
W2 += eta * hidden_output.T.dot(d_output)
b2 += eta * d_output
W1 += eta * X.T.dot(d_hidden)
b1 += eta * d_hidden

print("\nUpdated Parameters After One Training Step:")
print("Updated W1:", W1)
print("Updated b1:", b1)
print("Updated W2:", W2)
print("Updated b2:", b2)
print("\n================ PART B ================\n")

# --------------------------------
# PART B: Multiple epochs till convergence
# --------------------------------

# Reinitialize weights for fresh training
np.random.seed(42)
W1 = np.random.rand(2, 2)
b1 = np.random.rand(1, 2)
W2 = np.random.rand(2, 1)
b2 = np.random.rand(1, 1)

epochs = 10000
for epoch in range(epochs):
    # Forward pass
    hidden_input = np.dot(X, W1) + b1
    hidden_output = sigmoid(hidden_input)
    final_input = np.dot(hidden_output, W2) + b2
    output = sigmoid(final_input)

    # Error
```

```
    error = t - output

    # Backpropagation
    d_output = error * sigmoid_derivative(output)
    d_hidden = d_output.dot(W2.T) * sigmoid_derivative(hidden_output)

    # Update weights
    W2 += eta * hidden_output.T.dot(d_output)
    b2 += eta * d_output
    W1 += eta * X.T.dot(d_hidden)
    b1 += eta * d_hidden

    # Print progress every 1000 epochs
    if (epoch + 1) % 1000 == 0:
        print(f"Epoch {epoch+1}: Output = {output}, Error = {np.mean(np.abs(error))}")

    # Stop early if converged
    if np.mean(np.abs(error)) < 0.001:
        print(f"\n Converged at epoch {epoch+1}")
        break

print("\nFinal Output After Training:")
print("Output:", output)
print("Final Error:", np.mean(np.abs(error)))
print("\nFinal Parameters:")
print("W1:", W1)
print("b1:", b1)
print("W2:", W2)
print("b2:", b2)
```

Output:

```
Initial Parameters:
W1: [[0.37454012 0.95071431]
 [0.73199394 0.59865848]]
b1: [[0.15601864 0.15599452]]
W2: [[0.05808361]
 [0.86617615]]
b2: [[0.60111501]]

=============== PART A ===============

Forward Pass Values:
Hidden Input: [[0.53055876 1.10670883]]
Hidden Output: [[0.62961342 0.75151503]]
Final Input: [[1.28862962]]
Output: [[0.78391515]]
Error: [[0.21608485]]

Updated Parameters After One Training Step:
Updated W1: [[0.37478802 0.95367458]
 [0.73199394 0.59865848]]
Updated b1: [[0.15626654 0.15895479]]
Updated W2: [[0.06960651]
 [0.87993003]]
Updated b2: [[0.61941655]]

=============== PART B ===============

Epoch 1000: Output = [[0.97822185]], Error = 0.02177814667434408
Epoch 2000: Output = [[0.98492154]], Error = 0.015078462065135523
Epoch 3000: Output = [[0.98782622]], Error = 0.012173779530098683
Epoch 4000: Output = [[0.98953577]], Error = 0.010464231827594372
Epoch 5000: Output = [[0.99069205]], Error = 0.009307954441523436
Epoch 6000: Output = [[0.9915398]], Error = 0.0084602022237108
Epoch 7000: Output = [[0.9921951]], Error = 0.007804900621973254
Epoch 8000: Output = [[0.99272095]], Error = 0.00727905122174588
Epoch 9000: Output = [[0.99315485]], Error = 0.006845147339855750 5
Epoch 10000: Output = [[0.99352071]], Error = 0.006479294022393001

Final Output After Training:
Output: [[0.99352071]]
Final Error: 0.006479294022393001

Final Parameters:
W1: [[0.59126136 1.32774182]
 [0.73199394 0.59865848]]
b1: [[0.37273989 0.53302204]]
W2: [[1.1667568 ]
 [2.21597392]]
b2: [[2.27038192]]
```

Experiment 7

Aim: Implement and demonstrate various crossover operations in Python for genetic algorithms.

Theory: Crossover is a key genetic algorithm operation that combines genetic information from two parent solutions to produce offspring. Single-point crossover exchanges genes after a randomly chosen point, multi-point crossover swaps segments at multiple points, and uniform crossover swaps each gene independently with a fixed probability. These operations introduce variation into the population, promoting exploration of the solution space while preserving useful traits from the parents.

Code:

```
import random

def single_point_crossover(parent1, parent2):
    point = random.randint(1, len(parent1)-1)
    child1 = []
    child2 = []
    for i in range(len(parent1)):
        if i < point:
            child1.append(parent1[i])
            child2.append(parent2[i])
        else:
            child1.append(parent2[i])
            child2.append(parent1[i])
    return child1, child2

def multi_point_crossover(parent1, parent2, num_points=2):
    points = sorted(random.sample(range(1, len(parent1)), num_points))
    child1 = []
    child2 = []
    swap = False
    idx = 0
    for i in range(len(parent1)):
        if idx < len(points) and i == points[idx]:
            swap = not swap
            idx += 1
        if not swap:
            child1.append(parent1[i])
            child2.append(parent2[i])
        else:
            child1.append(parent2[i])
            child2.append(parent1[i])
    return child1, child2

def uniform_crossover(parent1, parent2, swap_prob=0.5):
    child1 = []
    child2 = []
    for i in range(len(parent1)):
        if random.random() < swap_prob:
            child1.append(parent2[i])
```

```
            child2.append(parent1[i])
        else:
            child1.append(parent1[i])
            child2.append(parent2[i])
    return child1, child2

# Example parents
parent1 = [1, 0, 1, 1, 0, 0, 1, 0]
parent2 = [0, 1, 0, 0, 1, 1, 0, 1]

print("Parent 1:", parent1)
print("Parent 2:", parent2)

c1, c2 = single_point_crossover(parent1, parent2)
print("\nSingle Point Crossover:")
print("Child 1:", c1)
print("Child 2:", c2)

c1, c2 = multi_point_crossover(parent1, parent2, num_points=3)
print("\nMulti Point Crossover:")
print("Child 1:", c1)
print("Child 2:", c2)

c1, c2 = uniform_crossover(parent1, parent2)
print("\nUniform Crossover:")
print("Child 1:", c1)
print("Child 2:", c2)
```

Output:

```
Parent 1: [1, 0, 1, 1, 0, 0, 1, 0]
Parent 2: [0, 1, 0, 0, 1, 1, 0, 1]

Single Point Crossover:
Child 1: [1, 0, 1, 1, 0, 0, 1, 1]
Child 2: [0, 1, 0, 0, 1, 1, 0, 0]

Multi Point Crossover:
Child 1: [1, 1, 0, 0, 0, 1, 0, 1]
Child 2: [0, 0, 1, 1, 1, 0, 1, 0]

Uniform Crossover:
Child 1: [0, 0, 1, 0, 1, 0, 1, 1]
Child 2: [1, 1, 0, 1, 0, 1, 0, 0]
```

Experiment 8

Aim: Implement and demonstrate various mutation operations in Python for genetic algorithms.

Theory: Mutation introduces random changes to chromosomes in a genetic algorithm to maintain diversity and prevent premature convergence. Bit-flip mutation inverts a gene, random resetting assigns a random value, swap mutation exchanges positions of two genes, scramble mutation shuffles a sub-segment, and inversion mutation reverses a sub-segment. These variations help explore new solutions while preserving genetic material from previous generations.

Code:

```
import random

def bit_flip_mutation(chromosome):
    idx = random.randint(0, len(chromosome) - 1)
    chromosome[idx] = 1 - chromosome[idx]
    return chromosome

def random_resetting_mutation(chromosome):
    idx = random.randint(0, len(chromosome) - 1)
    chromosome[idx] = random.choice([0, 1])
    return chromosome

def swap_mutation(chromosome):
    idx1, idx2 = random.sample(range(len(chromosome)), 2)
    chromosome[idx1], chromosome[idx2] = chromosome[idx2], chromosome[idx1]
    return chromosome

def scramble_mutation(chromosome):
    start = random.randint(0, len(chromosome) - 2)
    end = random.randint(start + 1, len(chromosome) - 1)
    sub_segment = chromosome[start:end+1]
    random.shuffle(sub_segment)
    chromosome[start:end+1] = sub_segment
    return chromosome

def inversion_mutation(chromosome):
    start = random.randint(0, len(chromosome) - 2)
    end = random.randint(start + 1, len(chromosome) - 1)
    chromosome[start:end+1] = chromosome[start:end+1][::-1]
    return chromosome

chromosome = [1, 0, 1, 1, 0, 0, 1, 0]
print("Original:", chromosome)

print("Bit Flip:", bit_flip_mutation(chromosome[:]))
print("Random Reset:", random_resetting_mutation(chromosome[:]))
print("Swap:", swap_mutation(chromosome[:]))
print("Scramble:", scramble_mutation(chromosome[:]))
print("Inversion:", inversion_mutation(chromosome[:]))
```

Output:

```
Original: [1, 0, 1, 1, 0, 0, 1, 0]
Bit Flip: [1, 0, 1, 1, 0, 0, 0, 0]
Random Reset: [1, 0, 1, 1, 0, 0, 1, 0]
Swap: [0, 0, 1, 1, 1, 0, 1, 0]
Scramble: [1, 0, 1, 1, 0, 0, 1, 0]
Inversion: [1, 0, 1, 1, 0, 0, 1, 0]
```

Experiment 9

Aim: Implement a GA from scratch that takes the objective function and iteratively solves it, returning the best solutions

Theory: A Genetic Algorithm (GA) is a search heuristic inspired by natural evolution. It iteratively evolves a population of candidate solutions to optimize an objective function. The process involves:

1. Initialization: Generate a random population of chromosomes representing possible solutions.
2. Evaluation: Compute the fitness of each chromosome based on the objective function.
3. Selection: Choose parents probabilistically based on fitness (e.g., tournament selection).
4. Crossover: Combine parts of two parents to produce offspring, promoting inheritance of good traits.
5. Mutation: Introduce random changes to maintain diversity and explore new solutions.
6. Iteration: Repeat evaluation, selection, crossover, and mutation for multiple generations until convergence.

This process converges towards the optimal solution while balancing exploration and exploitation in the search space.

Code:

```
import random
import numpy as np

# ------------------------------
# Problem Definition
# Equation: a + 2b + 3c + 4d = 30
# ------------------------------

target = 30
gene_length = 4  # [a,b,c,d]
pop_size = 6
generations = 10
gene_bounds = [(0,30)]*gene_length  # reasonable range
crossover_rate = 0.8
mutation_rate = 0.2

# Objective function: difference from target
def objective(chrom):
    a,b,c,d = chrom
    return abs((a + 2*b + 3*c + 4*d) - target)

# Initialize population (Step 1)
population = [ [random.randint(0,30) for _ in range(gene_length)] for _ in range(pop_size) ]

print("=== Step 1: Initialization ===")
for i, chrom in enumerate(population,1):
    print(f"Chromosome[{i}] = {chrom}")

# ------------------------------
# Run iterations
```

```
# ----------------------------
for gen in range(1, generations+1):
    # Step 2: Evaluation
    fitness = [objective(chrom) for chrom in population]

    print(f"\n--- Generation {gen} ---")
    for i,(chrom,f) in enumerate(zip(population,fitness),1):
        print(f"Chromosome[{i}] = {chrom}, F_obj = {f}")

    # Step 3: Selection (Tournament selection)
    def tournament(pop, fitness):
        i,j = random.sample(range(len(pop)),2)
        return pop[i] if fitness[i]<fitness[j] else pop[j]

    new_population = []

    # Step 4: Crossover and Step 5: Mutation
    while len(new_population) < pop_size:
        # Selection
        parent1 = tournament(population, fitness)
        parent2 = tournament(population, fitness)

        # Crossover
        if random.random() < crossover_rate:
            point = random.randint(1,gene_length-1)
            child1 = parent1[:point] + parent2[point:]
            child2 = parent2[:point] + parent1[point:]
        else:
            child1, child2 = parent1[:], parent2[:]

        # Mutation
        def mutate(chrom):
            for i in range(len(chrom)):
                if random.random() < mutation_rate:
                    low, high = gene_bounds[i]
                    chrom[i] = random.randint(low, high)
            return chrom

        child1 = mutate(child1)
        child2 = mutate(child2)

        new_population.extend([child1, child2])

    population = new_population[:pop_size]

# Final evaluation
fitness = [objective(chrom) for chrom in population]
best_idx = np.argmin(fitness)
best_chrom = population[best_idx]
print("\n=== Final Best Solution ===")
print(f"Chromosome = {best_chrom}, F_obj = {fitness[best_idx]}")
```

Output:

```
=== Step 1: Initialization ===
Chromosome[1] = [9, 18, 20, 5]
Chromosome[2] = [12, 5, 8, 0]
Chromosome[3] = [0, 28, 30, 8]
Chromosome[4] = [8, 0, 18, 7]
Chromosome[5] = [19, 24, 26, 21]
Chromosome[6] = [26, 26, 18, 26]

--- Generation 1 ---
Chromosome[1] = [9, 18, 20, 5], F_obj = 95
Chromosome[2] = [12, 5, 8, 0], F_obj = 16
Chromosome[3] = [0, 28, 30, 8], F_obj = 148
Chromosome[4] = [8, 0, 18, 7], F_obj = 60
Chromosome[5] = [19, 24, 26, 21], F_obj = 199
Chromosome[6] = [26, 26, 18, 26], F_obj = 206

--- Generation 2 ---
Chromosome[1] = [1, 0, 18, 5], F_obj = 45
Chromosome[2] = [9, 18, 20, 7], F_obj = 103
Chromosome[3] = [12, 5, 20, 5], F_obj = 72
Chromosome[4] = [9, 18, 8, 0], F_obj = 39
Chromosome[5] = [12, 18, 20, 5], F_obj = 98
Chromosome[6] = [9, 5, 8, 0], F_obj = 13

--- Generation 3 ---
Chromosome[1] = [9, 0, 8, 0], F_obj = 3
Chromosome[2] = [12, 5, 20, 5], F_obj = 72
Chromosome[3] = [9, 9, 8, 0], F_obj = 21
Chromosome[4] = [9, 5, 8, 0], F_obj = 13
Chromosome[5] = [8, 0, 18, 0], F_obj = 32
Chromosome[6] = [9, 18, 8, 5], F_obj = 59

--- Generation 4 ---
Chromosome[1] = [9, 9, 2, 29], F_obj = 119
Chromosome[2] = [9, 0, 8, 0], F_obj = 3
Chromosome[3] = [1, 18, 7, 1], F_obj = 32
Chromosome[4] = [9, 5, 8, 5], F_obj = 33
Chromosome[5] = [9, 9, 15, 0], F_obj = 42
Chromosome[6] = [9, 0, 8, 0], F_obj = 3

--- Generation 5 ---
Chromosome[1] = [9, 25, 7, 1], F_obj = 54
Chromosome[2] = [2, 18, 8, 0], F_obj = 32
Chromosome[3] = [9, 0, 8, 0], F_obj = 3
Chromosome[4] = [3, 0, 8, 17], F_obj = 65
Chromosome[5] = [9, 18, 7, 1], F_obj = 40
Chromosome[6] = [1, 9, 15, 0], F_obj = 34

--- Generation 6 ---
Chromosome[1] = [1, 0, 8, 6], F_obj = 19
Chromosome[2] = [9, 9, 15, 0], F_obj = 42
Chromosome[3] = [27, 12, 15, 0], F_obj = 66
Chromosome[4] = [1, 9, 8, 3], F_obj = 25
Chromosome[5] = [9, 0, 8, 23], F_obj = 95
Chromosome[6] = [2, 7, 8, 0], F_obj = 10

--- Generation 7 ---
Chromosome[1] = [1, 0, 8, 3], F_obj = 7
Chromosome[2] = [1, 9, 8, 6], F_obj = 37
Chromosome[3] = [2, 9, 5, 0], F_obj = 5
Chromosome[4] = [9, 7, 8, 30], F_obj = 137
Chromosome[5] = [9, 9, 8, 0], F_obj = 21
Chromosome[6] = [2, 7, 15, 5], F_obj = 51

--- Generation 8 ---
Chromosome[1] = [22, 7, 5, 0], F_obj = 21
Chromosome[2] = [2, 9, 14, 5], F_obj = 52
Chromosome[3] = [1, 0, 8, 3], F_obj = 7
Chromosome[4] = [9, 12, 1, 8], F_obj = 38
Chromosome[5] = [1, 4, 5, 0], F_obj = 6
Chromosome[6] = [2, 0, 8, 3], F_obj = 8

--- Generation 9 ---
Chromosome[1] = [2, 0, 11, 3], F_obj = 17
Chromosome[2] = [2, 0, 3, 3], F_obj = 7
Chromosome[3] = [1, 4, 5, 0], F_obj = 6
Chromosome[4] = [23, 4, 9, 0], F_obj = 28
Chromosome[5] = [6, 0, 8, 8], F_obj = 32
Chromosome[6] = [20, 4, 5, 3], F_obj = 25

--- Generation 10 ---
Chromosome[1] = [2, 0, 11, 3], F_obj = 17
Chromosome[2] = [20, 4, 5, 3], F_obj = 25
Chromosome[3] = [23, 4, 5, 3], F_obj = 28
Chromosome[4] = [20, 4, 15, 0], F_obj = 43
Chromosome[5] = [1, 0, 3, 3], F_obj = 8
Chromosome[6] = [2, 10, 14, 0], F_obj = 34

=== Final Best Solution ===
Chromosome = [2, 0, 1, 3], F_obj = 13
```