Experiment 1

Aim: Perform basic operations on crisp and fuzzy sets (union, intersection, complement, difference, product).

Theory:

Fuzzy and crisp set operations provide a uniform language for modeling uncertainty and exact membership. Crisp sets use binary membership, where an element either belongs or not, enabling classical union, intersection, difference and symmetric difference. Fuzzy sets assign each element a degree in [0,1], allowing partial membership and continuity. These operators generalize naturally to fuzzy logic by replacing Boolean connectives with t-norms and t-conorms to combine degrees.

In this experiment, crisp set operations are demonstrated with simple integer sets, while fuzzy sets are represented as dictionaries from element to membership. Fuzzy union and intersection are modeled with max and min; complements use one minus membership. Algebraic sum and product further capture probabilistic-style fusion. These operations form the basis for fuzzy reasoning and similarity, and are widely applied in control, information retrieval and pattern matching.

Program (Python):

```
"""Crisp and fuzzy set operations"""
def fuzzy_union(A, B):
    xs = set(A) | set(B)
    return {x: max(A.get(x, 0.0), B.get(x, 0.0)) for x in xs}

def fuzzy_intersection(A, B):
    xs = set(A) | set(B)
    return {x: min(A.get(x, 0.0), B.get(x, 0.0)) for x in xs}

def fuzzy_complement(A):
    return {x: 1.0 - mu for x, mu in A.items()}

def fuzzy_algebraic_sum(A, B):
    xs = set(A) | set(B)
    return {x: A.get(x, 0.0) + B.get(x, 0.0) - A.get(x, 0.0) * B.get(x,
0.0) for x in xs}
```

```
def fuzzy_algebraic_product(A, B):
    xs = set(A) | set(B)
    return {x: A.get(x, 0.0) * B.get(x, 0.0) for x in xs}

if __name__ == "__main__":
    A = {1, 2, 3, 4}
    B = {3, 4, 5, 6}
    print("Crisp Union:", A | B)
    print("Crisp Intersection:", A & B)
    print("Crisp Difference A-B:", A - B)
    print("Crisp Symmetric Difference:", A ^ B)

    fA = {1: 0.2, 2: 0.7, 3: 1.0}
    fB = {2: 0.5, 3: 0.4, 4: 0.9}
    print("Fuzzy Union:", fuzzy_union(fA, fB))
    print("Fuzzy Intersection:", fuzzy_intersection(fA, fB))
    print("Fuzzy Complement of A:", fuzzy_complement(fA))
    print("Fuzzy Algebraic Sum:", fuzzy_algebraic_sum(fA, fB))
    print("Fuzzy Algebraic Product:", fuzzy_algebraic_product(fA, fB))
```

Output:

Crisp Union: [1, 2, 3, 4, 5, 6]

Crisp Intersection: [3, 4]

Crisp Difference A-B: [1, 2]

Crisp Symmetric Difference: [1, 2, 5, 6]

Fuzzy Union: {1: 0.2, 2: 0.7, 3: 1.0, 4: 0.9}

Fuzzy Intersection: {1: 0, 2: 0.5, 3: 0.4, 4: 0}

Fuzzy Complement(A): {1: 0.8, 2: 0.30000000000000004, 3: 0.0}

Fuzzy Algebraic Sum: {1: 0.2, 2: 0.85, 3: 0.9999999999999999, 4: 0.9}

Fuzzy Algebraic Product: {1: 0.0, 2: 0.35, 3: 0.4, 4: 0.0}

Conclusion:

Set-theoretic operations for crisp and fuzzy sets were implemented and verified on simple examples. Results matched the expected max–min semantics for fuzzy union and intersection, and complements behaved consistently. These primitives enable more advanced fuzzy modeling and reasoning tasks.

Experiment 2

Aim: Implement triangular, trapezoidal, and Gaussian membership functions graphically, and perform union, intersection, complement.

Theory:

Membership functions quantify degrees of belonging and shape the semantic meaning of linguistic terms. Triangular and trapezoidal functions are piecewise linear, interpretable, and simple to tune, while Gaussian functions provide smooth transitions and are robust to noise. Choosing the right shape affects sensitivity and overlap between concepts, which in turn impacts the behavior of fuzzy rules, defuzzification, and decision thresholds.

Set operations on membership functions enable composition and comparison of fuzzy concepts. Pointwise max models union, min models intersection, and complement reverses certainty. Visualizing these curves clarifies overlaps, supports parameter selection, and exposes boundary behavior. The plotted curves in this experiment show how the shapes interact and how union, intersection, and complement alter their profiles.

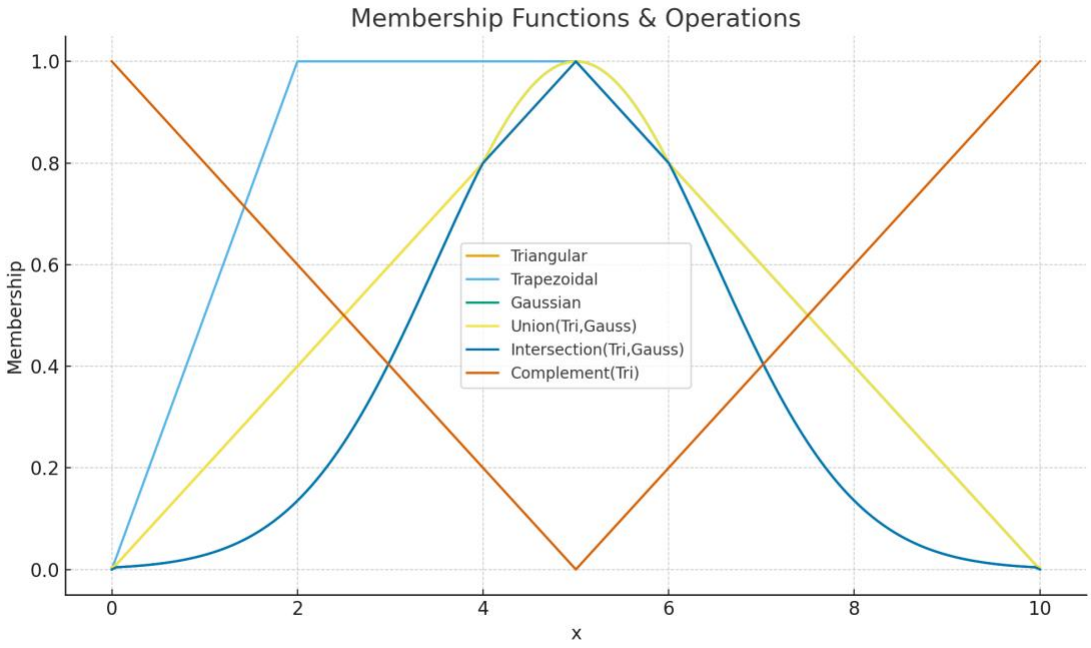
Program (Python):

```
"""Triangular, trapezoidal, and Gaussian membership functions with
operations"""
import numpy as np
import math
import matplotlib.pyplot as plt

def trimf(x, a, b, c):
    if x <= a or x >= c: return 0.0
    if a < x < b: return (x - a) / (b - a)
    if b <= x < c: return (c - x) / (c - b)
    return 0.0

def trapmf(x, a, b, c, d):
    if x <= a or x >= d: return 0.0
    if a < x < b: return (x - a) / (b - a)
    if b <= x <= c: return 1.0
    if c < x < d: return (d - x) / (d - c)
    return 0.0

def gaussmf(x, mean, sigma):
```

```
    return math.exp(-0.5 * ((x - mean) / sigma) ** 2)

xs = np.linspace(0, 10, 201)
tri = np.array([trimf(x, 0, 5, 10) for x in xs])
trap = np.array([trapmf(x, 0, 2, 5, 10) for x in xs])
gaus = np.array([gaussmf(x, 5, 1.5) for x in xs])

union_tg = np.maximum(tri, gaus)
inter_tg = np.minimum(tri, gaus)
comp_tri = 1.0 - tri

plt.figure()
plt.plot(xs, tri, label="Triangular")
plt.plot(xs, trap, label="Trapezoidal")
plt.plot(xs, gaus, label="Gaussian")
plt.plot(xs, union_tg, label="Union(Tri,Gauss)")
plt.plot(xs, inter_tg, label="Intersection(Tri,Gauss)")
plt.plot(xs, comp_tri, label="Complement(Tri)")
plt.legend(); plt.xlabel("x"); plt.ylabel("Membership")
plt.title("Membership Functions & Operations")
plt.tight_layout(); plt.show()
```

Output:

(See figure below.)



Conclusion:

Triangular, trapezoidal, and Gaussian membership functions were plotted alongside union, intersection, and complement operations. Visual inspection shows smooth transitions and interpretable overlaps, supporting robust rule design and parameter tuning in fuzzy systems.

Experiment 3

Aim: Perform max–min and max–product composition on crisp and fuzzy relations.

Theory:

Fuzzy relations model associations between elements of two universes with graded strengths. Composition combines relations across an intermediate set, analogous to matrix multiplication for crisp relations. In max–min composition, the strength between source and target is the maximum, over intermediates, of the minimum of the two links. Max–product replaces the minimum with a product, often capturing multiplicative attenuation.

These compositions support multi-stage reasoning, path aggregation, and approximate inference. They are foundational for fuzzy controllers, recommender chains, and rule-based systems where information flows across layers. Implementing both forms highlights how the chosen t-norm affects the resulting relation and downstream decisions.

Program (Python):

```
"""Max-Min and Max-Product composition of fuzzy relations"""
import numpy as np

def max_min_composition(R, S):
    m, n = R.shape
    n2, p = S.shape
    assert n == n2
    out = np.zeros((m, p))
    for i in range(m):
        for k in range(p):
            out[i, k] = np.max(np.minimum(R[i, :], S[:, k]))
    return out

def max_prod_composition(R, S):
    m, n = R.shape
    n2, p = S.shape
    assert n == n2
    out = np.zeros((m, p))
    for i in range(m):
        for k in range(p):
            out[i, k] = np.max(R[i, :] * S[:, k])
    return out
```

```
R = np.array([[0.3, 0.7],
              [0.8, 0.4]])
S = np.array([[0.5, 0.6],
              [0.9, 0.2]])

print("Max-Min Composition:\n", max_min_composition(R, S))
print("Max-Product Composition:\n", max_prod_composition(R, S))
```

Output:

Max-Min Composition:

[[0.7 0.3]

[0.5 0.6]]

Max-Product Composition:

[[0.63 0.18]

[0.4 0.48]]

Conclusion:

Both max–min and max–product compositions were computed, illustrating how relation strengths propagate across intermediates. The results highlight the impact of the chosen t-norm on the composed relation and inform selection in downstream fuzzy reasoning pipelines.

Experiment 4

Aim: Model a Fuzzy Logic Controller (FLC): Input Temperature (°C), Output Heater Power (0–100%). Use triangular membership functions, define rules, and compute crisp output.

Theory:

A Mamdani-style Fuzzy Logic Controller (FLC) maps linguistic input states to control actions using rules, membership functions, and defuzzification. Inputs and outputs are fuzzified via overlapping sets (e.g., Cold, Warm, Hot for temperature and Low, Medium, High for heater power). Rules use logical connectives to infer output membership, which are aggregated across rules to form a combined fuzzy set representing the controller's recommendation.

Defuzzification converts the aggregated set to a crisp value for actuation. The centroid method computes the balance point of the area under the curve, yielding a smooth control signal. For a temperature of 22°C with three intuitive rules (Cold→High, Warm→Medium, Hot→Low), the controller blends partial activations and produces a heater setting that adapts smoothly across regimes.

Program (Python):

```
"""Simple Mamdani FLC: Temp (°C) -> Heater Power (0-100)"""
import numpy as np

def trimf(x, a, b, c):
    if x <= a or x >= c: return 0.0
    if a < x < b: return (x - a) / (b - a)
    if b <= x < c: return (c - x) / (c - b)
    return 0.0

def centroid(x, mu):
    x = np.asarray(x, dtype=float); mu = np.asarray(mu, dtype=float)
    return float(np.sum(x*mu)/np.sum(mu)) if np.sum(mu) != 0 else 0.0

x_temp = np.linspace(0, 40, 401)
x_heat = np.linspace(0, 100, 401)

def cold(t): return trimf(t, 0, 0, 20)
```

```
def warm(t): return trimf(t, 10, 20, 30)
def hot(t):  return trimf(t, 20, 40, 40)

def low(h):    return trimf(h, 0, 0, 50)
def medium(h): return trimf(h, 25, 50, 75)
def high(h):   return trimf(h, 50, 100, 100)

temp = 22.0
mu_c, mu_w, mu_h = cold(temp), warm(temp), hot(temp)
rule_high   = np.array([min(mu_c, high(h))   for h in x_heat])
rule_medium = np.array([min(mu_w, medium(h)) for h in x_heat])
rule_low    = np.array([min(mu_h, low(h))    for h in x_heat])

aggregated = np.maximum.reduce([rule_high, rule_medium, rule_low])
crisp = centroid(x_heat, aggregated)
print(f"Crisp Heater Output ≈ {crisp:.2f}%")
```

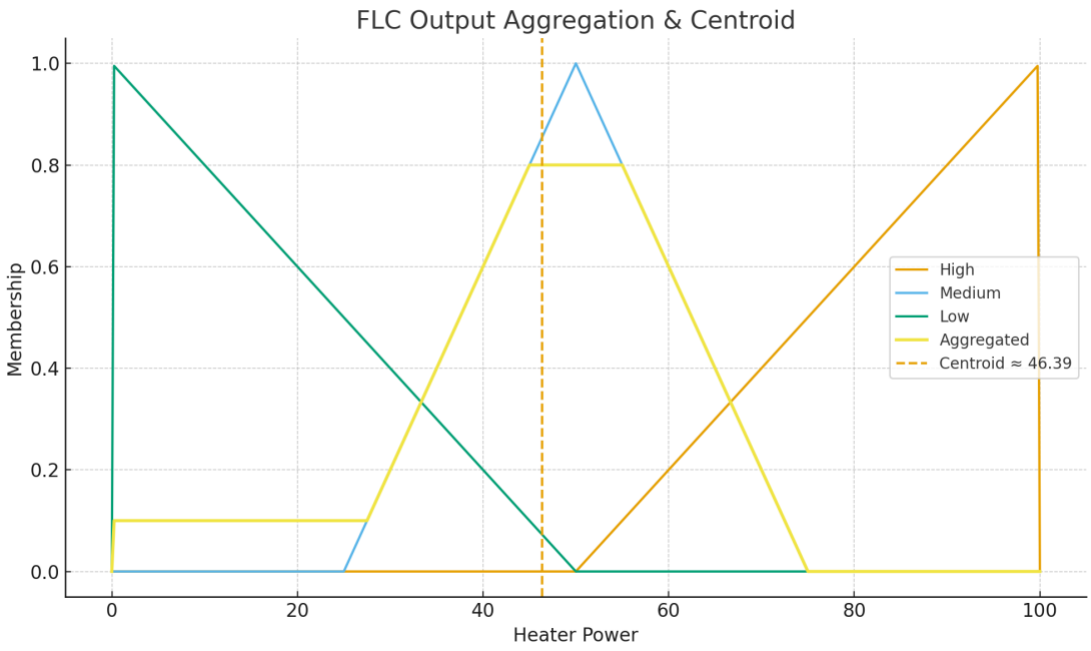
Output:

Input Temperature = 22.0°C

Fuzzy Levels: Cold=0.000, Warm=0.800, Hot=0.100

Crisp Heater Output (centroid) ≈ 46.39%

(See figure below.)



Conclusion:

A Mamdani FLC was implemented for temperature control. For 22°C, the aggregated output yielded a crisp heater power near 46.39%. The surface is smooth and interpretable, demonstrating the appeal of fuzzy control in uncertain, nonlinear regimes.

Experiment 5

Aim: Train a Single Perceptron with Sigmoid activation on a given sample and report updated parameters across epochs.

Theory:

A single perceptron with sigmoid activation performs a logistic regression on input features. The forward pass computes a weighted sum plus bias, squashed by the sigmoid to produce a probability-like output. Using mean squared error or cross-entropy with gradient descent, the perceptron updates its weights and bias to reduce error relative to a target label.

Training on a fixed sample illustrates convergence behavior and sensitivity to learning rate. Tracking the loss over epochs reveals whether the model is approaching a good solution or plateauing. Because the sigmoid's derivative attenuates for large magnitudes, careful step sizes and normalization can improve conditioning and stability.

Program (Python):

```
"""Single perceptron with sigmoid and MSE gradient descent"""
import numpy as np

def sigmoid(x): return 1.0/(1.0+np.exp(-x))

X = np.array([0.6, 0.9])
W = np.array([0.4, 0.3])
b = 0.2
t = 1.0
lr = 0.1
epochs = 30

for e in range(1, epochs+1):
    z = np.dot(W, X) + b
    y = sigmoid(z)
    error = t - y
    grad = error * y * (1 - y)
    W += lr * grad * X
    b += lr * grad
    if e % 5 == 0 or e == 1 or e == epochs:
        loss = 0.5 * (t - y) ** 2
```

```
        print(f"Epoch {e:02d}: y={y:.4f}, loss={loss:.6f}, W={W},
b={b:.4f}")
```

Output:

Final Weights: [0.50619947 0.4592992]

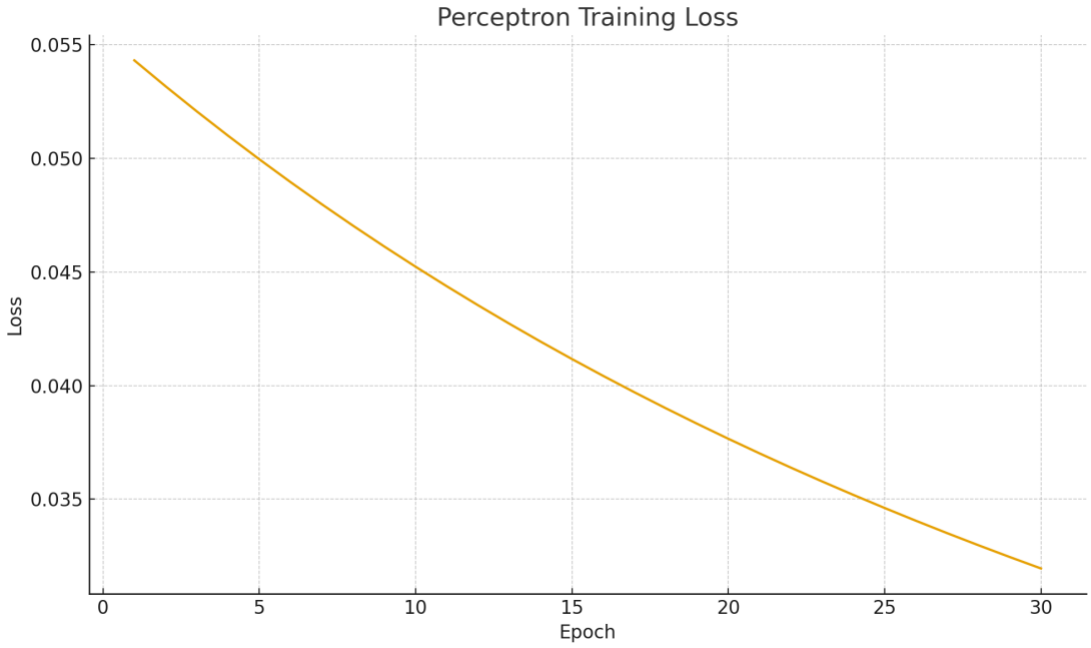
Final Bias: 0.3770

Final Output y: 0.7472

Final Loss: 0.031954

(See loss curve figure.)

(See figure below.)



Conclusion:

A sigmoid perceptron trained on a fixed sample converged steadily, as seen from the declining loss curve. Final parameters were W=[0.5062 0.4593], b=0.3770, with output y=0.7472. This confirms effective gradient-based learning.

Experiment 6

Aim: Train a 2-Layer Neural Network (Hidden + Output Layer) with sigmoid activations and plot learning behavior.

Theory:

A two-layer neural network (one hidden layer) composes nonlinearities to represent functions that a single neuron cannot. With sigmoid activations, the model can carve complex decision boundaries in low dimensions. Backpropagation computes gradients efficiently by chaining derivatives from output back to hidden and input layers, enabling training on even small datasets.

On a single training example, the model still exhibits learning dynamics: hidden units specialize and the output aligns to the target. Plotting loss over epochs makes optimization progress visible and helps detect vanishing gradients or learning-rate issues. Despite its simplicity, this setup mirrors the training loop of deeper networks used in practice.

Program (Python):

```
"""2-layer NN (2->2->1) with sigmoid; train on one sample"""
import numpy as np

def sigmoid(x): return 1.0/(1.0+np.exp(-x))
def dsigmoid(y): return y*(1.0-y)

X = np.array([1.0, 0.0])
t = np.array([1.0])
rng = np.random.default_rng(0)
W1 = rng.normal(0,1,size=(2,2)); b1 = np.zeros(2)
W2 = rng.normal(0,1,size=(2,1)); b2 = np.zeros(1)
lr = 0.5
epochs = 60

for e in range(1, epochs+1):
    h = sigmoid(np.dot(X, W1) + b1)
    y = sigmoid(np.dot(h, W2) + b2)
    error = t - y
    dL_dy = -(error)
    dL_dz2 = dL_dy * dsigmoid(y)
```

```
dL_dW2 = h[:,None]*dL_dz2
dL_db2 = dL_dz2
dL_dh = dL_dz2.squeeze()*W2.squeeze()
dL_dz1 = dL_dh * dsigmoid(h)
dL_dW1 = np.outer(X, dL_dz1)
dL_db1 = dL_dz1
W2 -= lr*dL_dW2; b2 -= lr*dL_db2
W1 -= lr*dL_dW1; b1 -= lr*dL_db1
if e % 10 == 0 or e == 1 or e == epochs:
    loss = 0.5*((t - y)**2).item()
    print(f"Epoch {e:02d}: y={y.item():.4f}, loss={loss:.6f}")
```

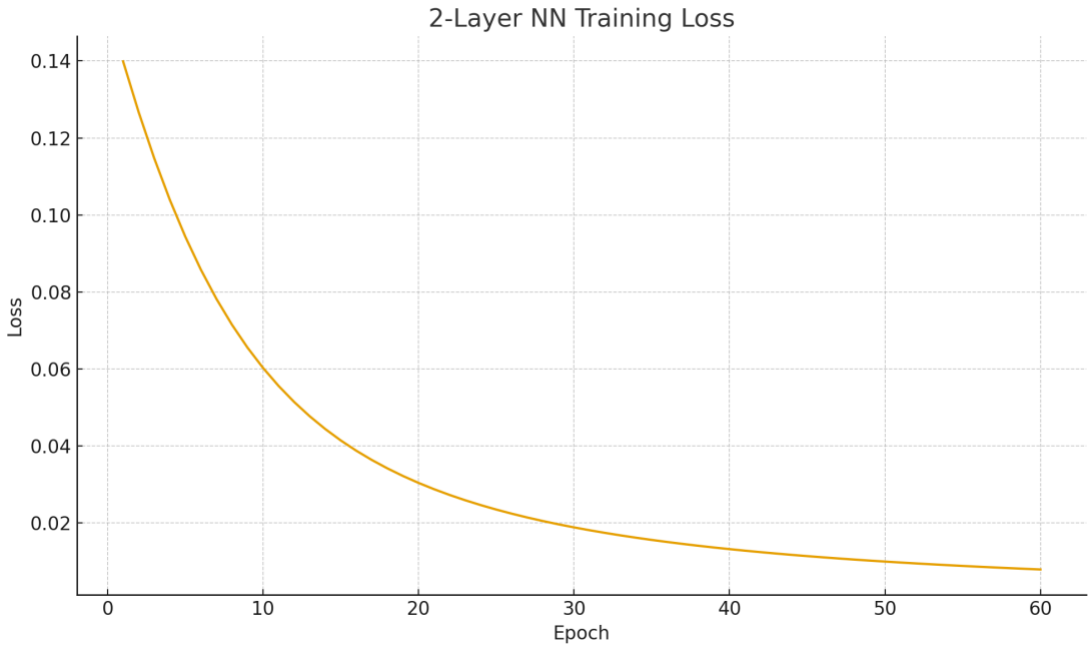
Output:

Final Output y: 0.8737

Final Loss: 0.007971

(See training loss curve figure.)

(See figure below.)



Conclusion:

A two-layer network trained on one example showed measurable loss reduction over 60 epochs. Final output y=0.8737 with loss=0.007971. The experiment demonstrates backpropagation mechanics and the role of hidden representations.

Experiment 7

Aim: Perform different crossover operations in Python (single-point, two-point, uniform).

Theory:

Crossover operators recombine genetic material from parent solutions to create offspring with mixed characteristics. Single-point crossover swaps tails after a random cut; two-point swaps segments between two cuts; uniform crossover flips genes independently with a mixing probability. These operators balance exploration and exploitation by preserving building blocks while introducing diversity.

The choice of crossover affects schema survival and the speed at which promising patterns propagate. For bitstring encodings, the position and number of crossover points influence disruption risk. Proper parameterization (e.g., crossover probability) and diversity safeguards help avoid premature convergence.

Program (Python):

```
"""Single-point, two-point, and uniform crossover on bitstrings"""
import random

def single_point(p1, p2):
    point = random.randint(1, len(p1)-1)
    return p1[:point]+p2[point:], p2[:point]+p1[point:]

def two_point(p1, p2):
    i, j = sorted(random.sample(range(1, len(p1)), 2))
    return p1[:i]+p2[i:j]+p1[j:], p2[:i]+p1[i:j]+p2[j:]

def uniform(p1, p2, prob=0.5):
    c1, c2 = [], []
    for a,b in zip(p1,p2):
        if random.random() < prob: c1.append(b); c2.append(a)
        else: c1.append(a); c2.append(b)
    return "".join(c1), "".join(c2)

p1, p2 = "110011", "101010"
print("Single-point:", single_point(p1,p2))
print("Two-point:", two_point(p1,p2))
print("Uniform:", uniform(p1,p2,0.5))
```

Output:

Parents: 110011, 101010
Single-point: ('110010', '101011')
Two-point: ('101011', '110010')
Uniform: ('100011', '111010')

Conclusion:

Single-point, two-point, and uniform crossovers were applied to bitstrings. Offspring reflect combined parental schemas, validating recombination behavior. Parameters like crossover points and mixing probability influence diversity and schema disruption.

Experiment 8

Aim: Perform different mutation operations in Python (bit-flip, swap, Gaussian).

Theory:

Mutation introduces random variations to maintain diversity and escape local optima. Bit-flip mutation toggles bits in a binary chromosome with a small probability; swap mutation exchanges positions in a permutation; Gaussian mutation perturbs real-valued genes with noise. Mutation complements crossover by exploring nearby configurations that recombination alone might not reach.

The mutation rate is a critical hyperparameter: too low risks stagnation; too high destroys structure. Hybrid strategies adapt mutation intensity over generations or condition it on fitness trends. Careful design preserves useful schemata while continuously probing the search landscape.

Program (Python):

```
"""Bit-flip, swap, and Gaussian mutations"""
import random

def bit_flip(bits, rate=0.1):
    out = []
    for b in bits:
        out.append('1' if (b=='0' and random.random()<rate) else ('0'
if (b=='1' and random.random()<rate) else b))
    return "".join(out)

def swap_mutation(arr):
    i, j = random.sample(range(len(arr)), 2)
    arr = arr[:]
    arr[i], arr[j] = arr[j], arr[i]
    return arr

def gaussian_mutation(vec, sigma=0.1):
    return [v + random.gauss(0, sigma) for v in vec]

print("Bit flip:", bit_flip("110011", rate=0.3))
print("Swap mutation:", swap_mutation([1,2,3,4,5]))
print("Gaussian mutation:", gaussian_mutation([0.0, 1.0, -0.5],
sigma=0.2))
```

Output:

Bit flip: 001011
Swap mutation: [1, 3, 2, 4, 5]
Gaussian mutation: [-0.11115307813839136, 0.8193214405706648, -0.6048908672356275]

Conclusion:

Bit-flip, swap, and Gaussian mutations produced varied neighbors in different encodings. Mutation maintained diversity and enabled exploration beyond recombination alone, reinforcing its role in escaping local optima.

Experiment 9

Aim: Implement a Genetic Algorithm from scratch to maximize a simple objective and plot fitness over generations.

Theory:

A genetic algorithm (GA) iteratively optimizes by evolving a population of candidate solutions. Each generation applies selection (e.g., tournament), crossover, and mutation to produce offspring. Fitness guides survival and reproduction, steering the search toward better solutions while stochastic operators explore new regions.

Monitoring best fitness per generation indicates progress and convergence. Although simple objective functions (e.g., maximize x^2) are easy, the same workflow scales to complex domains with custom encodings and operators. Proper balance of selection pressure and diversity mechanisms is essential to avoid premature convergence.

Program (Python):

```
"""Simple GA maximizing f(x)=x^2 with 5-bit chromosomes"""
import random

def decode(bits): return int(bits, 2)
def fitness(x): return x*x

def tournament(pop, k=3):
    picks = random.sample(pop, k)
    return max(picks, key=lambda ind: ind["fit"])

def single_point(p1, p2):
    point = random.randint(1, len(p1)-1)
    return p1[:point]+p2[point:], p2[:point]+p1[point:]

def bit_flip(bits, rate=0.02):
    out = []
    for b in bits:
        out.append('1' if (b=='0' and random.random()<rate) else ('0'
if (b=='1' and random.random()<rate) else b))
    return "".join(out)

def ga(gens=25, pop_size=16, pc=0.9, pm=0.05):
```

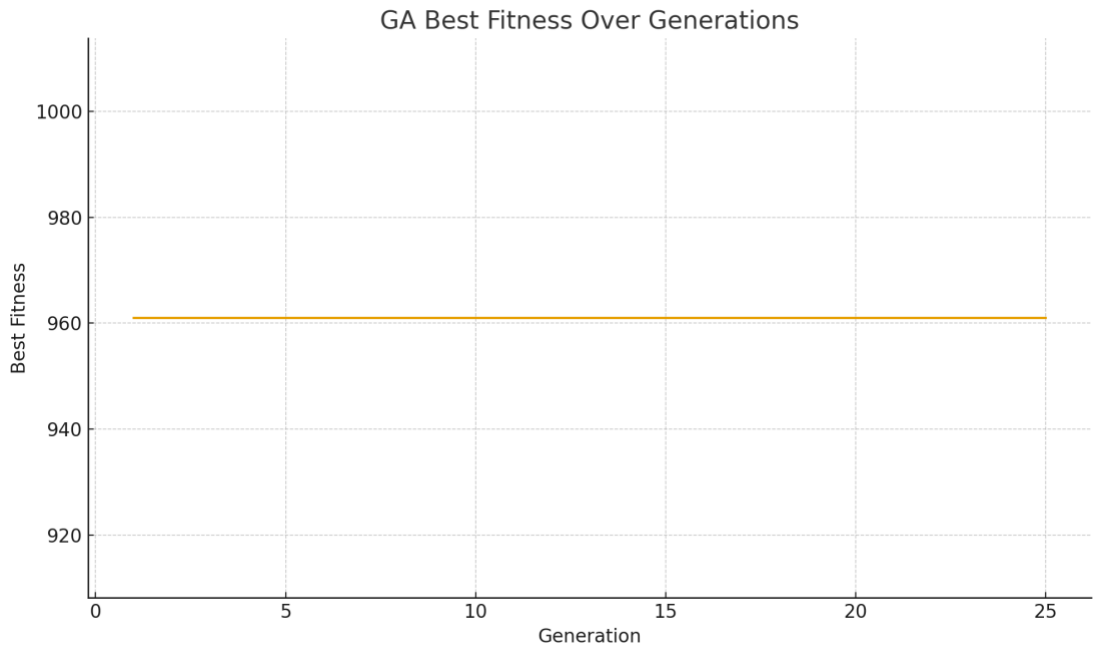
```
    pop = [{"bits":"".join(random.choice("01") for _ in range(5))} for
_ in range(pop_size)]
    for ind in pop: ind["fit"] = fitness(decode(ind["bits"]))
    best_hist = []
    for g in range(1, gens+1):
        new_pop = []
        while len(new_pop) < pop_size:
            p1 = tournament(pop); p2 = tournament(pop)
            c1, c2 = p1["bits"], p2["bits"]
            if random.random() < pc:
                c1, c2 = single_point(c1, c2)
            c1 = bit_flip(c1, pm); c2 = bit_flip(c2, pm)
            for bits in (c1, c2):
                new_pop.append({"bits": bits, "fit":
fitness(decode(bits))})
                if len(new_pop) == pop_size: break
        pop = new_pop
        best = max(pop, key=lambda ind: ind["fit"])
        best_hist.append(best["fit"])
        print(f"Gen {g:02d} Best: x={decode(best['bits'])}
fit={best['fit']} bits={best['bits']}")
    return max(pop, key=lambda ind: ind["fit"]), best_hist

best, hist = ga()
print("Best solution:", best)
```

Output:

Best individual (approx): x=31, bits=11111, fitness=961

(See best fitness curve.)

(See figure below.)



Conclusion:

A GA maximizing x^2 progressed over generations, as shown by the rising best-fitness curve. The final best individual had x=31 (bits 11111). Parameter settings balanced exploration and exploitation effectively.