INSTITUTE OF TECHNOLOGY & MANAGEMENT
www.itmgoi.in

श्रेष्ठ इंडस्ट्री इन्टरफेस के लिए
CMAI, AICTE & RGPV
द्वारा पुरस्कृत

INSTITUTE OF TECHNOLOGY
& MANAGEMENT
GWALIOR • MP • INDIA

Laboratory Manual

SOFT COMPUTING
(IT-701)

For

Fourth Year Student
Department: Information Technology

Department of Information Technology

Vision of IT Department

The Department of Information Technology envisions preparing technically competent problem solvers, researchers, innovators, entrepreneurs, and skilled IT professionals for the development of rural and backward areas of the country for the modern computing challenges.

Mission of the IT Department

- To offer valuable education through an effective pedagogical teaching-learning process.
- To shape technologically strong students for industry, research & higher studies.
- To stimulate the young brain entrenched with ethical values and professional behaviors for the progress of society.

Program Educational Objectives

Graduates will be able to

- Our graduates will show management skills and teamwork to attain employers' objectives in their careers.
- Our graduates will explore the opportunities to succeed in research and/or higher studies.
- Our graduates will apply technical knowledge of Information Technology for innovation and entrepreneurship.
- Our graduates will evolve ethical and professional practices for the betterment of society.

Program Outcomes (POs)

Engineering Graduates will be able to:

1. **Engineering knowledge**: Apply the knowledge of mathematics, science, engineering Fundamentals, and an engineering specialization to the solution of complex engineering problems.
2. **Problem analysis**: Identify, formulate, review research literature, and analyze complex engineering problems reaching substantiated conclusions using first principles of mathematics, natural sciences, and engineering sciences.
3. **Design/development of solutions**: Design solutions for complex engineering problemsand design system components or processes that meet the specified needs with appropriate consideration for the public health and safety, and the cultural, societal, andenvironmental considerations.
4. **Conduct investigations of complex problems**: Use research-based knowledge and research methods including design of experiments, analysis and interpretation of data,and synthesis of the information to provide valid conclusions.
5. **Modern tool usage**: Create, select, and apply appropriate techniques, resources, and modern engineering and IT tools including prediction and modeling to complex engineering activities with an understanding of the limitations.
6. **The engineer and society**: Apply reasoning informed by the contextual knowledge to assess societal, health, safety, legal and cultural issues and the consequent responsibilities relevant to the professional engineering practice.
7. **Environment and sustainability**: Understand the impact of the professional engineering solutions in societal and environmental contexts, and demonstrate the knowledge of, and need for sustainable development.
8. **Ethics**: Apply ethical principles and commit to professional ethics and responsibilitiesand norms of the engineering practice.
9. **Individual and team work**: Function effectively as an individual, and as a member or leader in diverse teams, and in multidisciplinary settings.
10. **Communication**: Communicate effectively on complex engineering activities with the engineering community and with society at large, such as, being able to comprehend and write effective reports and design documentation, make effective presentations, andgive and receive clear instructions.
11. **Project management and finance**: Demonstrate knowledge and understanding of the engineering and management principles and apply these to one’s own work, as a member and leader in a team, to manage projects and in multidisciplinary environments.
12. **Life-long learning**: Recognize the need for, and have the preparation and ability to engage in independent and life-long learning in the broadest context of technological change.

Course Outcomes

Soft Computing (IT 701)

CO1:	Understand and apply the basic of Soft computing techniques Like Artificial Neural Network (ANN), Genetic Algorithms.
CO2:	Understand the needs of Neural Networks and its types
CO3:	Implement and evaluate the Fuzzy logic.
CO4:	Understand the basic concepts of genetic algorithm LIKE Genetic algorithm and search space
CO5:	Apply the hybrid soft computing techniques and its type in real life applications.

Course	Course Outcomes	CO Attainment	P01	P02	P03	P04	P05	P06	P07	P08	P09	P010	P011	P012	PSO1	PSO2	PSO3
IT 701.1	Understand concept of ANN and explain the XOR problem.		1	1	0	0	0	0	0	0	0	0	0	0	1	0	0
IT 701.2	Use supervised neural networks to classify given inputs.		0	0	1	0	1	0	0	0	0	0	0	0	0	0	1
IT 701.3	Understand unsupervised neural networks for clustering data.		0	1	1	0	0	0	0	0	0	0	0	0	1	0	0
IT 701.4	Build Fuzzy inference system using concepts of fuzzy logic		0	1	1	1	0	0	0	0	0	0	0	0	0	0	1
IT 701.5	Obtain an optimized solution to a given problem using genetic algorithm		0	1	1	0	0	0	0	0	0	0	0	0	0	0	1
Average			**0.2**	**0.8**	**0.8**	**0.2**	**0.2**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0.4**	**0**	**0.6**

List of Programs

S. No.	**List**	**Course Outcome**	**Page No.**
	INTRODUCTION TO SOFT COMPUTING		1
1	Form a perceptron net for basic logic gates with binary input and output	CO2	2-3
2	Calculation of new weights for a Back propagation network, given the values of input pattern, output pattern, target output, learning rate and activation function.	CO2	4-4
3	Implement Travelling salesman problem using Genetic Algorithm.	CO5	5-7
4	Optimisation of problem like Job shop scheduling using Genetic algorithm.	CO5	8-10
5	Using Adaline net, generate XOR function with bipolar inputs and targets.	CO1	11-12
6	Design fuzzy inference system for a given problem.	CO4	13-14
7	Maximize the function y =3x2 + 2 for some given values of x using Genetic algorithm.	CO5	15-16

INTRODUCTION TO SOFT COMPUTING

Soft computing is a branch of computer science and computational intelligence that deals with approximate reasoning, uncertainty, and imprecision to handle complex real-world problems. Unlike traditional computing techniques, which rely on precise mathematical models and algorithms, soft computing techniques employ heuristic methods to approximate solutions in situations where crisp mathematical models may be insufficient.

Soft computing encompasses several subfields, including fuzzy logic, neural networks, evolutionary computation, and probabilistic reasoning, each of which addresses different aspects of imprecision and uncertainty in problem-solving. Here's a detailed introduction to each of these components:

1. Fuzzy Logic: Fuzzy logic provides a framework for dealing with uncertainty by allowing for degrees of truth instead of the usual binary true/false (1/0) values. It employs linguistic variables, which are described by fuzzy sets that represent imprecise concepts. Fuzzy logic enables the modeling of human reasoning processes by incorporating linguistic terms such as "very hot" or "slightly cold," allowing for more flexible and human-like decision-making systems.
2. Neural Networks: Neural networks are computational models inspired by the structure and functioning of biological neural networks in the human brain. They consist of interconnected nodes (neurons) organized into layers, with each neuron processing information and transmitting signals to other neurons. Neural networks learn from data through a process called training, where they adjust their internal parameters to minimize errors and improve performance on specific tasks, such as pattern recognition, classification, and prediction.
3. Evolutionary Computation: Evolutionary computation algorithms are inspired by principles of natural selection and genetics. They include genetic algorithms, evolutionary strategies, and genetic programming, among others. These algorithms iteratively generate and evaluate a population of candidate solutions to a problem, selecting the fittest individuals for reproduction and applying genetic operators (mutation, crossover) to produce offspring with potentially improved characteristics. Over successive generations, evolutionary computation techniques converge towards optimal or near-optimal solutions in complex search spaces.
4. Probabilistic Reasoning: Probabilistic reasoning involves reasoning under uncertainty using probability theory. It provides a formal framework for representing and updating beliefs about uncertain events based on available evidence. Techniques such as Bayesian networks, Markov models, and probabilistic graphical models are commonly used in soft computing to model uncertain and dynamic systems, make predictions, and perform decision-making under uncertain

1. Form a perceptron net for basic logic gates with binary input and output

A perceptron is a simple neural network model typically used for binary classification tasks. While a single perceptron can only solve linearly separable problems, we can combine multiple perceptron to form networks capable of implementing more complex functions, including basic logic gates like AND, OR, and NOT gates. Here, I'll show you how to form perceptron networks for these basic logic gates with binary input and output.

AND Gate: The AND gate outputs 1 (or True) only if both inputs are 1; otherwise, it outputs 0 (or False). To implement the AND gate using a perceptron, we need to set appropriate weights and bias.

Input 1 (x1)	Input 2 (x2)	Bias (b)	Output (y)
0	0	-1	0
0	1	-1	0
1	0	-1	0
1	1	-1	1

Here, the weights are both 1, and the bias is -1. The perceptron's output is 1 only when both inputs are 1.

OR Gate: The OR gate outputs 1 if at least one of the inputs is 1. We can implement the OR gate using a perceptron with appropriate weights and bias.

Input 1 (x1)	Input 2 (x2)	Bias (b)	Output (y)
0	0	0	0
0	1	1	1
1	0	1	1
1	1	1	1

Here, both weights are 1, and the bias is 0. The perceptron's output is 1 if either or both inputs are 1.

NOT Gate: The NOT gate outputs the opposite of its input. We can implement the NOT gate using a single-input perceptron.

Input (x)	Bias (b)	Output (y)
0	1	1
1	-1	0

Here, the weight is -1, and the bias is 1. The perceptron's output is the opposite of the input.

These perceptron networks demonstrate how simple neural networks can represent basic logic gates using binary inputs and outputs. By combining multiple perceptrons or using more complex neural network architectures, we can build networks capable of implementing more intricate logical functions and solving diverse computational tasks.

2. Calculation of new weights for a Back propagation network, given the values of input pattern, output pattern, target output, learning rate and activation function.

To calculate new weights for a back propagation neural network, you need to follow these steps:
1. Forward Propagation: Compute the network's output using the current weights.
2. Calculate Error: Find the error between the computed output and the target output.
3. Back propagation: Propagate the error backward through the network to update the weights.
4. Update Weights: Use the error and learning rate to adjust the weights.

Let's break down each step:

1. Forward Propagation: Compute the output of the neural network for a given input pattern using the current weights. This involves passing the input through the network's layers and applying the activation function at each neuron to get the output.
2. Calculate Error: Find the error between the computed output and the target output. This could be calculated using a suitable error function, such as mean squared error (MSE) or cross-entropy loss, depending on the problem.
3. Back propagation: Back propagation involves propagating the error backward through the network to compute the gradients of the error with respect to the weights. This is done by applying the chain rule of calculus to update the weights of each layer based on the error contributions from subsequent layers.
4. Update Weights: Finally, update the weights of the network using the calculated gradients and the learning rate. This step ensures that the network learns from its mistakes and adjusts the weights to minimize the error.

Here's a simplified formula to update the weights in a back propagation network:

New Weight=Old Weight+Learning Rate×Error×Input×Activation Function′(Weighted Sum)New Weight=Old Weight+Learning Rate×Error×Input×Activation Function′(Weighted Sum)
Where:
• Old Weight Old Weight is the weight before the update.
• Learning Rate Learning Rate is a hyper parameter controlling the size of the weight updates.
• Error Error is the difference between the target output and the computed output.
• Input Input is the input value associated with the weight.
• Activation Function′(Weighted Sum)Activation Function′(Weighted Sum) is the derivative of the activation function evaluated at the weighted sum of inputs to the neuron.

This formula represents how each weight is updated based on the error signal propagated backward through the network during back propagation. The learning rate controls the step size of the weight updates, preventing the network from overshooting the optimal

3. Implement Travelling salesman problem using Genetic Algorithm

Implementing the Traveling Salesman Problem (TSP) using a Genetic Algorithm (GA) involves representing candidate solutions as permutations of cities, creating a population of such solutions, and evolving them over generations to find an optimal or near-optimal solution. Here's a basic outline of how you can implement it:

1. Initialize Population: Create an initial population of candidate solutions (individuals), where each individual represents a possible ordering of cities (a tour).
2. Fitness Calculation: Evaluate the fitness of each individual in the population based on the total distance traveled in the corresponding tour.
3. Selection: Select individuals from the population to create the next generation, favoring individuals with higher fitness values.
4. Crossover: Create new individuals (offspring) by combining genetic material (city orderings) from selected parent individuals.
5. Mutation: Introduce random changes (mutations) to some individuals in the population to maintain genetic diversity.
6. Replacement: Replace the current population with the new generation of individuals.
7. Repeat: Repeat steps 2-6 for a fixed number of generations or until a termination condition is met.

Here's a Python implementation of the above steps:

```
import numpy as np
import random

# Define cities and their coordinates
cities = {
    'A': (0, 0),
    'B': (1, 3),
    'C': (2, 5),
    'D': (3, 2),
    'E': (5, 0)
}

# Calculate distance between two cities

def distance(city1, city2):
    x1, y1 = cities[city1]
    x2, y2 = cities[city2]
    return np.sqrt((x2 - x1)**2 + (y2 - y1)**2)

# Calculate total distance of a tour

def total_distance(tour):
    return sum(distance(tour[i], tour[i+1]) for i in range(len(tour) - 1)) + distance(tour[-1],
tour[0])

# Generate initial population

def generate_population(size):
    return [random.sample(cities.keys(), len(cities)) for _ in range(size)]

# Tournament selection

def select_parents(population, num_parents):
    return random.sample(population, num_parents)

# Order 1 crossover
def crossover(parent1, parent2):
    start, end = sorted(random.sample(range(len(parent1)), 2))
    offspring = parent1[start:end]
    missing_cities = [city for city in parent2 if city not in offspring]
    return offspring + missing_cities

# Swap mutation
def mutate(individual, mutation_rate):
    if random.random() < mutation_rate:
        idx1, idx2 = random.sample(range(len(individual)), 2)
        individual[idx1], individual[idx2] = individual[idx2], individual[idx1]
    return individual

# Genetic Algorithm
def genetic_algorithm(num_generations, population_size, num_parents, mutation_rate):

population = generate_population(population_size)
    for generation in range(num_generations):
        fitness_scores = [1 / total_distance(individual) for individual in population]
        parents = select_parents(population, num_parents)
        offspring = []
        while len(offspring) < population_size:
            parent1, parent2 = random.sample(parents, 2)
            child = crossover(parent1, parent2)
  child = mutate(child, mutation_rate)
            offspring.append(child)
        population = offspring
    best_tour = min(population, key=total_distance)
    return best_tour, total_distance(best_tour)
 Example usage
 best_tour, min_distance = genetic_algorithm(num_generations=100, population_size=100,
num_parents=50, mutation_rate=0.1)
print("Best tour:", best_tour)
print("Minimum distance:", min_distance)
```

This implementation uses a simple genetic algorithm with tournament selection, order 1 crossover, and swap mutation. You may need to adjust parameters such as the population size, number of generations, and mutation rate to find a good solution for your specific TSP instance. Additionally, you can experiment with other genetic operators and strategies to improve the performance of the algorithm.

4. Optimization of problem like Job shop scheduling using Genetic algorithm

Job shop scheduling is a classic optimization problem where a set of jobs need to be processed on a set of machines, each with its own processing time. The objective is to find an optimal schedule that minimizes a certain criterion, such as the total completion time (makespan), total tardiness, or maximum lateness.

Here's how you can approach job shop scheduling using a Genetic Algorithm (GA):

1. **Representation:** Represent each solution (schedule) as a permutation of jobs, where each job represents the order in which the jobs are processed on the machines.

2. **Initialization:** Generate an initial population of schedules randomly or using heuristic methods.

3. **Fitness Evaluation:** Evaluate the fitness of each schedule based on the chosen objective function (e.g., makespan, total tardiness).

4. **Selection:** Select individuals (schedules) from the population to serve as parents for the next generation. Common selection methods include roulette wheel selection, tournament selection, or rank-based selection.

5. **Crossover:** Create new individuals (offspring) by combining genetic material (scheduling sequences) from selected parent individuals. Various crossover techniques like Partially Mapped Crossover (PMX), Order Crossover (OX), or Position-Based Crossover (PBX) can be employed.

6. **Mutation:** Introduce random changes (mutations) to some individuals in the population to maintain genetic diversity. Mutations may involve swapping jobs, changing the order of jobs, or altering machine assignments.

7. **Replacement:** Replace the current population with the new generation of individuals, potentially employing elitism to retain the best solutions.

8. **Termination:** Repeat steps 3-7 for a fixed number of generations or until a termination condition is met, such as reaching a maximum number of iterations or finding a satisfactory solution.

Here's a high-level Python-like pseudocode for implementing GA for job shop scheduling:

```
def initialize_population(population_size):
# Generate random initial population of schedules
pass

def evaluate_fitness(schedule):
# Calculate fitness of a schedule (e.g., makespan, total tardiness)
  pass

def selection(population, fitness_values, num_parents):


 # Select individuals from the population for reproduction
 pass

def crossover(parent1, parent2):
 # Apply crossover to produce offspring
pass

def mutation(individual, mutation_rate):
# Apply mutation to an individual
 pass

def genetic_algorithm(num_generations, population_size, num_parents, mutation_rate):
 population = initialize_population(population_size)
 for generation in range(num_generations):
  fitness_values = [evaluate_fitness(schedule) for schedule in population]
  parents = selection(population, fitness_values, num_parents)
  offspring = []
  while len(offspring) < population_size:
   parent1, parent2 = random.sample(parents, 2)
   child = crossover(parent1, parent2)
   child = mutation(child, mutation_rate)
   offspring.append(child)
  population = offspring
  best_schedule = min(population, key=evaluate_fitness)
  return best_schedule, evaluate_fitness(best_schedule)

# Example usage
best_solution, best_fitness = genetic_algorithm(num_generations=100, population_size=100,
num_parents=50, mutation_rate=0.1)
print("Best solution:", best_solution)
print("Best fitness:", best_fitness)
```

This pseudocode provides a framework for implementing a Genetic Algorithm for job shop scheduling. You need to replace placeholder functions like initialize_population, evaluate_fitness, selection, crossover, and mutation with actual implementations tailored to your specific problem instance and objective function. Additionally, you may need to fine-tune parameters like population size, number of generations, number of parents, and mutation rate to achieve optimal performance.

5. Using Ada line net, generate XOR function with bipolar inputs and targets.

To implement the XOR function using an Ada line neural network with bipolar inputs and targets, we need to follow these steps:
1. Define the Ada line neural network architecture.
2. Initialize the network's weights and bias.
3. Define the activation function (linear activation in the case of Ada line).
4. Train the network using the XOR dataset.
5. Test the trained network.

Here's a Python implementation using the Ada line neural network:

```
import numpy as np

class Adaline:
    def __init__(self, input_size):
   self.input_size = input_size
   self.weights = np.random.randn(input_size)
   self.bias = np.random.randn()
   self.learning_rate = 0.1

    def activation_function(self, x):
      return x  # Linear activation for Adaline

    def predict(self, inputs):

    net_input = np.dot(inputs, self.weights) + self.bias
   return self.activation_function(net_input)

    def train(self, inputs, targets, epochs):
    for epoch in range(epochs):
    for input_vector, target in zip(inputs, targets):
    prediction = self.predict(input_vector)
    error = target – prediction
    self.weights += self.learning_rate * error * input_vector
   self.bias += self.learning_rate * error

    # XOR dataset with bipolar inputs and targets
    inputs = np.array([[-1, -1], [-1, 1], [1, -1], [1, 1]])
    targets = np.array([-1, 1, 1, -1])

    # Create and train Adaline network
    adaline = Adaline(input_size=2)
    adaline.train(inputs, targets, epochs=100)
```

```
# Test the trained network
for input_vector, target in zip(inputs, targets):
    prediction = adaline.predict(input_vector)
    print(f"Input: {input_vector}, Target: {target}, Prediction: {prediction}")
```

• We define the Adaline class representing the Adaline neural network.
• The activation function is linear (activation_function method).
• We train the Adaline network using the XOR dataset with bipolar inputs and targets.
• After training, we test the network's predictions on the same dataset.

Note: Adaline is a linear classifier, so it can only approximate linearly separable functions. The XOR function is not linearly separable, so Adaline may not achieve perfect accuracy on this problem. However, with enough epochs and proper initialization, it should converge to a reasonable solution.

6. Design fuzzy inference system for a given problem.

To design a fuzzy inference system (FIS) for a given problem, we need to follow these steps:

1. Define the problem and its variables.
2. Linguistic variable definition: Identify linguistic terms (fuzzy sets) for each variable.
3. Membership function design: Define membership functions for each linguistic term.
4. Fuzzy rule base creation: Establish fuzzy rules based on expert knowledge or data.
5. Fuzzy inference mechanism: Implement the fuzzy inference process using appropriate inference methods (e.g., Mamdani or Sugeno).
6. Defuzzification: Convert fuzzy output into crisp values.

Let's design a fuzzy inference system for a simple problem: determining the amount of water to be poured into a plant based on the soil moisture level and the temperature.

1. **Define the problem and its variables:**
   •Input variables:
   - Soil Moisture Level (Low, Medium, High)
   - Temperature (Cold, Moderate, Hot)
   - Output variable:
   - Water Amount (Low, Medium, High)

2. **Linguistic variable definition:**
   - Soil Moisture Level: Low, Medium, High
   - Temperature: Cold, Moderate, Hot
   - Water Amount: Low, Medium, High

3. **Membership function design:**
   - Define triangular or trapezoidal membership functions for each linguistic term of the variables. For example, for the "Soil Moisture Level" variable, you might have triangular membership functions for Low, Medium, and High.

4. **Fuzzy rule base creation:**
   - Establish fuzzy rules that relate input variables to output variable. For example:
   - If Soil Moisture Level is Low OR Temperature is Hot, then Water Amount is High.
   - If Soil Moisture Level is High AND Temperature is Cold, then Water Amount is Low.
   - And so on, based on expert knowledge or data.

5. **Fuzzy inference mechanism:**
   - Implement Mamdani or Sugeno fuzzy inference methods to determine the fuzzy output based on fuzzy rules and input values.

6. **Defuzzification:**
   - Convert fuzzy output into crisp values using methods such as centroid, mean of

maximum (MOM), or weighted average.
Here's a Python-like pseudocode implementation:

```
# Define membership functions for each linguistic term

# Define fuzzy rules based on expert knowledge or data

# Implement fuzzy inference mechanism (Mamdani or Sugeno)

# Apply defuzzification to convert fuzzy output into crisp values
```

You would need to replace the placeholder code with actual implementations suitable for your specific problem and use appropriate libraries or tools for fuzzy logic computations if available (e.g., scikit-fuzzy in Python).

7. Maximize the function y =3x2 + 2 for some given values of x using Genetic algorithm.

To maximize the function �=3�2+2y=3x2+2 using a Genetic Algorithm (GA), we need to follow these steps:

1. Define the problem and its representation: In this case, we want to find the maximum value of �y for given values of �x.
2. Define the chromosome representation: We'll represent potential solutions (individuals) as values of �x within a specified range.
3. Define the fitness function: The fitness function will evaluate how close a given solution is to the maximum value of �y.
4. Implement genetic operators: We'll define selection, crossover, and mutation operations to evolve the population.
5. Implement the genetic algorithm: We'll use the genetic operators to iteratively evolve the population and find the optimal solution.

Here's a Python implementation:

```
import numpy as np

# Define the fitness function
def fitness_function(x):
    return 3 * x**2 + 2

# Define genetic operators

def selection(population, fitness_scores, num_parents):
    parents_indices = np.argsort(fitness_scores)[-num_parents:]
    return [population[idx] for idx in parents_indices]

def crossover(parent1, parent2):
    crossover_point = np.random.randint(1, len(parent1))
    child1 = np.concatenate((parent1[:crossover_point], parent2[crossover_point:]))
    child2 = np.concatenate((parent2[:crossover_point], parent1[crossover_point:]))
    return child1, child2

def mutation(individual, mutation_rate):
    mutated_individual = individual.copy()
    for i in range(len(individual)):
   if np.random.rand() < mutation_rate:
  mutated_individual[i] += np.random.uniform(-0.5, 0.5)  # Mutation with random small step
    return mutated_individual

# Genetic Algorithm

def genetic_algorithm(num_generations, population_size, num_parents, mutation_rate,
x_range):
 population = np.random.uniform(x_range[0], x_range[1], size=(population_size,))
 for generation in range(num_generations):
 fitness_scores = fitness_function(population)
 parents = selection(population, fitness_scores, num_parents)
 offspring = []
  while len(offspring) < population_size:
  parent1, parent2 = np.random.choice(parents, size=2, replace=False)
  child1, child2 = crossover(parent1, parent2)
   child1 = mutation(child1, mutation_rate)
   child2 = mutation(child2, mutation_rate)
   offspring.extend([child1, child2])
   population = np.array(offspring)
   best_solution = population[np.argmax(fitness_function(population))]
   return best_solution, fitness_function(best_solution)

  # Example usage
  best_solution, max_value = genetic_algorithm(num_generations=100, population_size=100,
  num_parents=50, mutation_rate=0.1, x_range=(-10, 10))
  print("Best solution:", best_solution)
  print("Maximum value of y:", max_value)
```

In this implementation:

- We define the fitness function as �=3�2+2y=3x2+2.
- We use selection, crossover, and mutation genetic operators to evolve the population.
- We apply the genetic algorithm to find the �x value that corresponds to the maximum value of �y.
- We define the range of �x values as (-10, 10) for this example, but you can adjust it as needed.