Laboratory 4.1 Neagu Mihnea

NLP problem using the sum of different functions problem

The sum of different powers problem involves finding the minimum value of a function that is defined as the sum of absolute values raised to different powers. Specifically, the function evaluates to:

$$f(x)=|x_1|^{2}+|x_2|^{3}+\ldots+|x_n|^{(n+1)}$$

Where $x_1, x_2, \ldots, x_n$ are the components of the input vector x. This problem is often used as a benchmark in optimization algorithms, particularly evolutionary algorithms, to test their effectiveness in finding the global minimum of a non-linear and multimodal function. It serves as a challenging problem due to its complex landscape with many local minima and sharp ridges. The sum of different powers problem helps researchers and practitioners evaluate the performance of optimization algorithms in handling various types of optimization challenges, including those found in real-world applications such as engineering design, finance, and machine learning.

Evolutive Algorithm for solving NLP:

Step 1:

```
import random
import math
import matplotlib.pyplot as plt
import time

1 usage
class EvolutionaryAlgorithm:
    def __init__(self, population_size, num_generations, crossover_prob, mutation_prob,
                 renewal_rate, generations_until_renewal):
        self.population_size = population_size
        self.num_generations = num_generations
        self.crossover_prob = crossover_prob
        self.mutation_prob = mutation_prob
        self.renewal_rate = renewal_rate
        self.generations_until_renewal = generations_until_renewal
        self.generations_until_renewal_actual = generations_until_renewal
        self.best_fitness = float('inf')  # Initialize best fitness to infinity
        self.best_generation = 0  # Initialize the generation where the best fitness is found
        self.best_individual = None  # Initialize the best individual

    1 usage
    def initialize_population(self, size):
        return [[random.uniform(-1, 1) for _ in range(size)] for _ in range(self.population_size)]

    4 usages
    def fitness_function(self, solution):
        total = sum(abs(solution[i]) ** (i + 2) for i in range(len(solution)))
```

We define a class **EvolutionaryAlgorithm** encapsulating the core mechanics of an evolutionary algorithm. It initializes parameters, generates populations, and evaluates fitness using a predefined function. This groundwork sets the stage for evolving solutions across generations.

Step 2:

```
    def select_parents(self, population, size):
        parents = random.sample(population, size)
        parents.sort(key=lambda x: self.fitness_function(x))
        return parents[0]

    1 usage
    def crossover(self, parent1, parent2):
        crossover_point = random.randint( a: 1, len(parent1) - 1)
        child1 = parent1[:crossover_point] + parent2[crossover_point:]
        child2 = parent2[:crossover_point] + parent1[crossover_point:]
        return child1, child2

    2 usages
    def mutation(self, individual, mutation_prob):
        for i in range(len(individual)):
            if random.random() < mutation_prob:
                individual[i] = random.uniform(-1,  b: 1)
        return individual

    1 usage
    def select_survivors(self, population, size):
        population.sort(key=lambda x: self.fitness_function(x))
        return population[:size]

    1 usage
    def renew_population(self, population, size):
        for _ in range(size):
            population[random.randint( a: 0, len(population) - 1)] = [random.uniform(-1,  b: 1) for _ in range(len(po
        return population
```

We implement a strategy for parent selection by randomly sampling individuals from the population and sorting them based on fitness. This ensures that fitter individuals are more likely to be selected as parents. For crossover, we randomly select a crossover point and exchange genetic material between parents to create offspring. Mutation occurs probabilistically for each individual gene, introducing variation in the population. Survivor selection is based on fitness, where individuals are sorted and the fittest are retained. Additionally, population renewal introduces diversity by replacing random individuals with new ones. These strategies collectively drive the evolution of solutions toward optimal fitness, but also because of the strategy used for parent and survivor selection which are highly exploitative we use the renewal population in order to inject diversification into the algorithm.

Step 3:

```
1 usage
def adjust_probabilities(self, generation):
    if generation % self.generations_until_renewal == 0 and generation > 0:
        self.mutation_prob *= 2
```

```
def evolve(self, problem_size):
    best_fitness_per_generation = []
    best_individual_per_generation = []
    for generation in range(self.num_generations):
        new_population = []
        for _ in range(self.population_size):
            parent1 = self.select_parents(population,  size: 2)
            parent2 = self.select_parents(population,  size: 2)
            if random.random() < self.crossover_prob:
                child1, child2 = self.crossover(parent1, parent2)
            else:
                child1, child2 = parent1[:], parent2[:]
            child1 = self.mutation(child1, self.mutation_prob)
            child2 = self.mutation(child2, self.mutation_prob)
            new_population.extend([child1, child2])
        population = self.select_survivors(new_population, self.population_size)
        self.adjust_probabilities(generation)
        if generation % self.renewal_rate == 0 and generation > 0:
            population = self.renew_population(population, int(self.population_size * self.renewal_rate))
        best_fitness = min([self.fitness_function(individual) for individual in population])
        if best_fitness < self.best_fitness:  # Update best fitness and generation if found
            self.best_fitness = best_fitness
            self.best_generation = generation + 1
            self.best_individual = [individual for individual in population if self.fitness_function(individual
        best_fitness_per_generation.append(best_fitness)
        best_individual_per_generation.append(self.best_individual)
        print(f"Generation {generation + 1}: Best fitness: {best_fitness}, Best individual: {self.best_individu
    end_time = time.time()  # Stop measuring execution time
    print(f"Total execution time: {end_time - start_time} seconds")
    return best_fitness_per_generation, best_individual_per_generation
```


The **adjust_probabilities** method dynamically adjusts the mutation probability based on the current generation. If the generation is a multiple of **generations_until_renewal** and greater than 0, the mutation probability is doubled. This strategy aims to increase exploration in early generations and exploitation in later generations.

In the evolve method, the execution time is measured using the time module. It initializes the population, tracks the best fitness and individual per generation, and evolves the population over a specified number of generations. Within each generation, parents are selected, crossover and mutation are applied to generate offspring, and survivors are chosen based on fitness. The mutation probability is adjusted, and population renewal occurs periodically. The best fitness, generation, and individual are updated if a new best fitness is found. Finally, the total execution time is printed, and the best fitness and individual per generation are returned. This method encapsulates the entire evolutionary process, from initialization to termination.

Step 4:

```python
def plot_evolution_fitness(best_fitness_per_generation):
    plt.plot(best_fitness_per_generation)
    plt.title('Evolution of the best fitness over generations')
    plt.xlabel('Generations')
    plt.ylabel('Fitness')
    plt.show()


def main():
    problem_size = int(input("Enter the problem size: "))
    population_size = 20
    num_generations = 1000
    crossover_prob = 0.9  # Increased for more exploitation
    mutation_prob = 0.05  # Reduced for less exploration
    renewal_rate = 0.05
    generations_until_renewal = 200
    algorithm = EvolutionaryAlgorithm(population_size, num_generations, crossover_prob, mutation_prob, renewal_rate
                                      generations_until_renewal)
    best_fitness_per_generation, best_individual_per_generation = algorithm.evolve(problem_size)

    # Calculate the average fitness
    average_fitness = sum(best_fitness_per_generation) / len(best_fitness_per_generation)
    plot_evolution_fitness(best_fitness_per_generation)
    print(f"Absolute best fitness: {algorithm.best_fitness:.12f} found in generation {algorithm.best_generation}")
    print(f"Average fitness: {average_fitness:.7f}")


if __name__ == "__main__":
    main()
```

The **plot_evolution_fitness** function visualizes the evolution of the best fitness over generations. It takes **best_fitness_per_generation** as input, which contains the best fitness values recorded for each generation. Using Matplotlib, it plots the best fitness values against the generations, with the x-axis representing generations and the y-axis representing fitness. The title, x-label, and y-label are set accordingly, and the plot is displayed using plt.show().

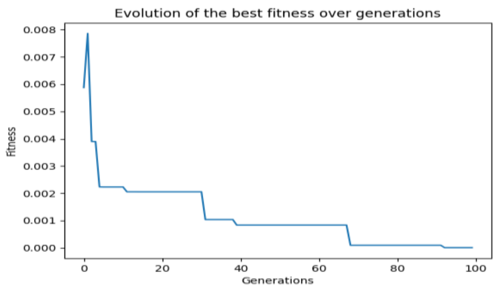
In the main function, the user is prompted to input the problem size. Other parameters such as **population_size, num_generations, crossover_prob, mutation_prob, renewal_rate, and generations_until_renewal** are predefined. An instance of EvolutionaryAlgorithm is created with these parameters, and the evolve method is called to execute the evolutionary process. After evolution, the average fitness is calculated from **best_fitness_per_generation.** The evolutionary process's results are then displayed: the absolute best fitness found, along with the generation where it occurred, and the average fitness across all generations. Finally, the evolution plot is shown using **plot_evolution_fitness**, and the results are printed.

Experiments and parameter optimization:

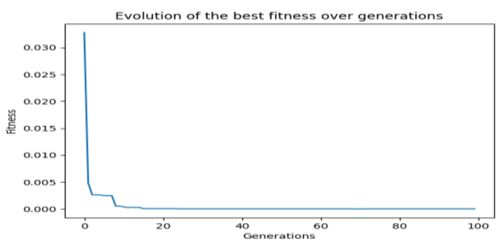
1. **d=3** In this next section we will showcase the best and average values of individuals over the course of all the generations, as well as the execution time of the program and the plotting of the fitness evolution over generations. Also, the data tables and plots presented below are calculated for a user input of 3 dimensions and a number of 10 runs for the evolution algorithm.

INPUTS							**OUTPUTS**		
Exp nr	**Number gens**	**Population size**	**Crossover prob**	**Mutation prob**	**Renew rate**	**Renew after gen**	Best Solution	Average solution	Exec time
1	100	20	0.9	0.05	0.05	50	0.00000654(gen93)	0.00112	0.028
2	100	30	0.85	0.14	0.01	50	0.00000070(gen70)	0.00057	0.038
3	100	40	0.8	0.1	0.1	80	0.00000026(gen67)	0.00040	0.050
4	1000	100	0.7	0.25	0.05	500	0.00000000059(gen462)	0.00038	1.25
5	10000	50	0.9	0.09999	0.00001	15000	0.00000000006(gen8051)	0.0000013	6.17
6	10000	100	0.9	0.09999	0.00001	15000	0.000000000002(gen8564)	0.0000015	12.38

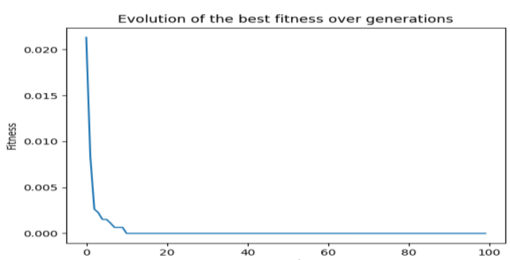
Exp 1 Plot



Exp 2 Plot



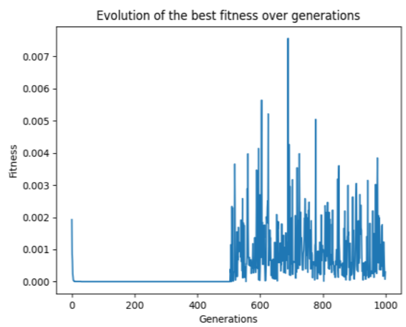
Exp 3 Plot



In the **first three experiments**, each with **100** generations, I adjusted the evolutionary algorithm's parameters. By **increasing population size**, **decreasing crossover probability, and boosting mutation and renewal rates**, I explored their impact on performance. Results consistently showed performance improvement. Larger populations enhanced both exploration and exploitation. Lower crossover and higher mutation favored exploitation, refining solutions. Elevated renewal rates fostered diversity, aiding exploration. The approach led to faster convergence to lower fitness values, signaling improved solutions. Notably, best fitness values improved, highlighting the algorithm's efficacy. Experiments revealed a balance between exploitation and exploration. Reduced crossover and increased mutation and renewal rates favored exploitation, while higher renewal rates maintained diversity, preventing premature convergence.

In summary, parameter adjustments showcased the algorithm's adaptability, enhancing convergence while preserving diversity.

Exp 4 Plot

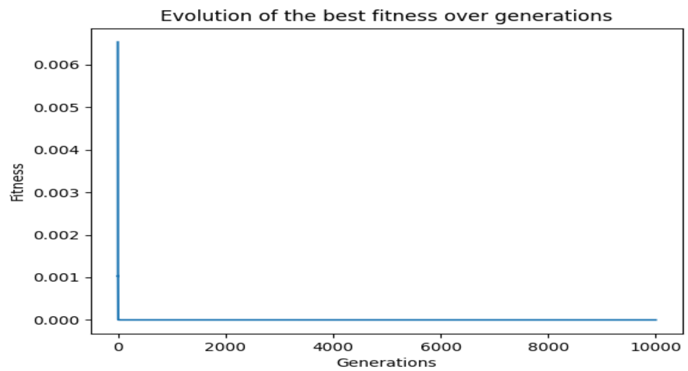


In Experiment 4, a notable observation in the fitness evolution plot is a significant spike occurring precisely at the 500th generation. This spike indicates a sudden increase in fitness levels, which is not desirable since the objective is to minimize fitness values. This abrupt change coincides with the population restart triggered by the parameter generations_until_renewal, set to exactly 500.

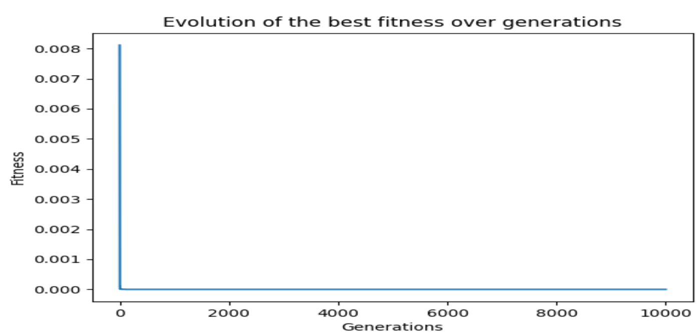
This observation suggests that while the population restart mechanism was effective in the initial experiments (Experiments 1-3), it led to suboptimal outcomes in Experiment 4. To address this issue, a strategic adjustment was made to the generations_until_renewal parameter. Instead of matching it precisely to the number of generations, it was increased beyond the number of generations. Consequently, population restarts would occur only based on the probabilistic chance defined by the renewal probability.

Interestingly, despite the spike in fitness at the 500th generation, Experiment 4 still outperformed the initial three experiments. The algorithm achieved superior results, notably reaching a promising fitness level at generation 462, just before the population restart. This observation underscores the importance of fine-tuning parameters to optimize algorithm performance and highlights the efficacy of strategic parameter adjustments in enhancing evolutionary algorithm outcomes.

Exp 5 Plot



Exp 6 Plot



In Experiments 5 and 6, we extended the number of generations to 10,000, aiming to explore the impact of population size on algorithm performance. The key distinction between the two experiments lies in the population size, with Experiment 5 utilizing a population of 50 and Experiment 6 employing a population of 100.

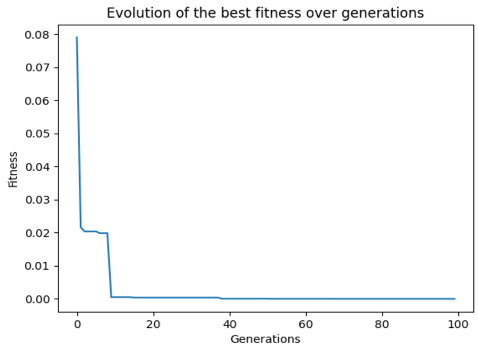
To induce premature convergence and highlight the effect of population size, deliberate parameter settings were applied. Specifically, the generation at which population renewal occurs was set beyond the total number of generations, and the renewal probability was kept exceptionally low at 0.000001. These adjustments were intended to encourage premature convergence, where the algorithm settles on suboptimal solutions prematurely.

Notably, Experiment 6, featuring a larger population size of 100, outperformed Experiment 5. Despite the deliberate inducement of premature convergence, Experiment 6 achieved a lower fitness value for the objective function. This outcome suggests that a larger population size facilitates more effective exploration of the solution space, enabling the algorithm to reach superior solutions even under conditions conducive to premature convergence.

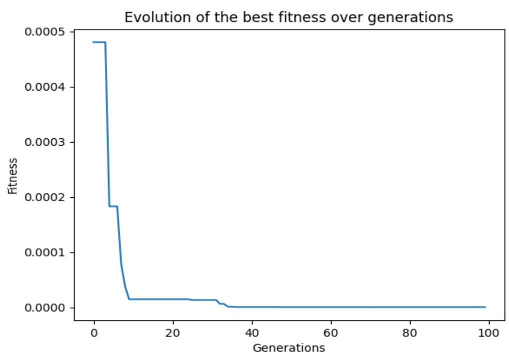
2. **d=2** In this next section we will showcase the best and average values of individuals over the course of all the generations, as well as the execution time of the program and the plotting of the fitness evolution over generations. Also, the data tables and plots presented below are calculated for a user input of 2 dimensions and a number of 10 runs for the evolution algorithm.

INPUTS							**OUTPUTS**		
Exp nr	**Number gens**	**Population size**	**Crossover prob**	**Mutation prob**	**Renew rate**	**Renew after gen**	Best Solution	Average solution	Exec time
1	100	20	0.9	0.05	0.05	50	0.0000054(gen96)	0.00254	0.029
2	100	30	0.85	0.14	0.01	50	0.00000021 (gen72)	0.00002	0.044
3	100	40	0.8	0.1	0.1	80	0.00000047(gen88)	0.00015	0.031
4	10000	50	0.9	0.09999	0.00001	15000	0.000000000016 (gen1437)	0.0000004	5.8
5	10000	100	0.9	0.09999	0.00001	15000	0.000000000221 (gen4533)	0.0000015	11.2

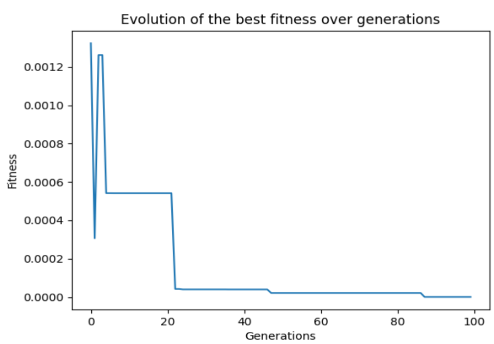
EXP 1 Plot



EXP 2 Plot



EXP 3 Plot

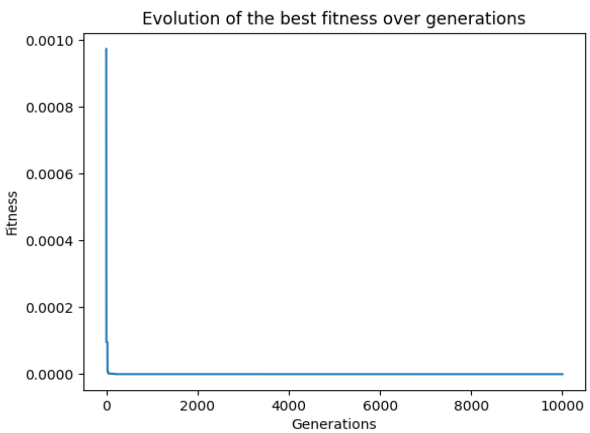


Surprisingly, the second experiment, characterized by a larger mutation probability and an exceptionally low renewal rate, yielded the best results. Despite its relatively smaller population size, this configuration managed to achieve a significantly improved best solution, marked by a minimal solution value of 0.00000021 at generation 72. Additionally, the average solution value remained impressively low, indicating the overall quality of solutions generated.

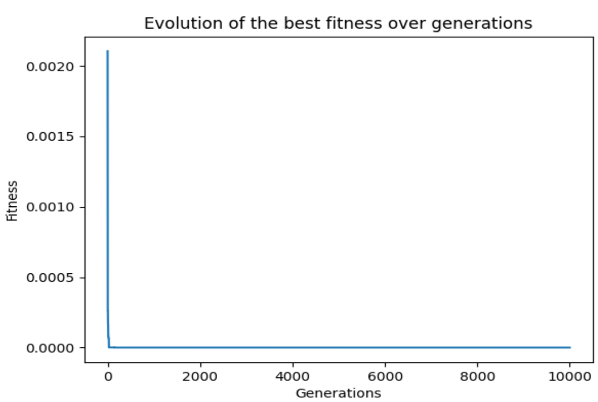
Contrary to conventional expectations, the combination of a larger mutation probability and a very low renewal rate led to a remarkable enhancement in performance. This configuration allowed the algorithm to focus more on exploitation, refining promising solutions with greater efficacy. Despite the smaller population size, this approach fostered a diverse pool of solutions, ensuring robustness and adaptability.

In terms of execution time, although the second experiment exhibited a slightly longer runtime compared to the others, its superior solution quality and efficiency in convergence more than compensated for the marginal increase in computational overhead.

EXP 4 Plot



EXP 5 Plot



In experiment 4, with a population size of 50, the algorithm achieved a remarkably low best solution value of 0.000000000016 at generation 1437. Additionally, the average solution value remained impressively low at 0.0000004. Despite the relatively small population size, the algorithm demonstrated remarkable efficiency in convergence, achieving exceptional solution quality.

On the other hand, in experiment 5, with a larger population size of 100, the algorithm managed to further improve the best solution value to 0.000000000221 at generation 4533. However, the improvement in solution quality was marginal compared to experiment 4. Although the average solution value slightly decreased to 0.0000015, indicating a slight improvement in overall solution quality, the difference was not as significant as expected.

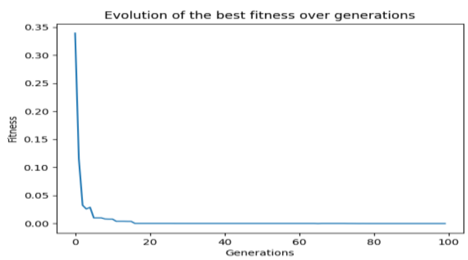
Interestingly, the execution time for experiment 5 increased to 11.2 seconds, more than double that of experiment 4. This suggests that while a larger population size may lead to marginal improvements in solution quality, it also incurs a significant increase in computational overhead.

3. **d=10** In this last section we will showcase the best and average values of individuals over the course of all the generations, as well as the execution time of the program and the plotting of the fitness evolution over generations. Also, the data tables and plots presented below are calculated for a user input of 10 dimensions and a number of 10 runs for the evolution algorithm.

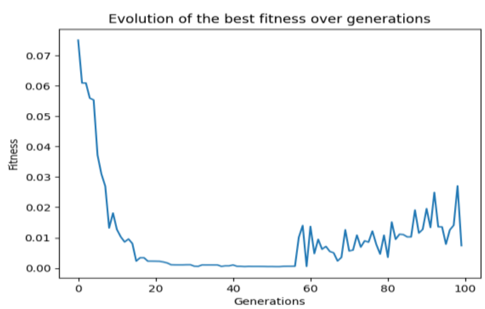
INPUTS							**OUTPUTS**		
Exp nr	**Number gens**	**Population size**	**Crossover prob**	**Mutation prob**	**Renew rate**	**Renew after gen**	Best Solution	Average solution	Exec time
1	100	20	0.9	0.05	0.05	50	0.000069 (gen66)	0.006	0.042
2	100	30	0.85	0.14	0.01	50	0.0004(gen52)	0.0097	0.063
3	100	40	0.8	0.1	0.1	80	0.00001 (gen86)	0.0113	0.084

4	10000	50	0.9	0.09999	0.00001	15000	0.00000000083 (gen4476)	0.000046	10.62
5	10000	100	0.9	0.09999	0.00001	15000	0.000000000046 (gen9429)	0.000020	20.58

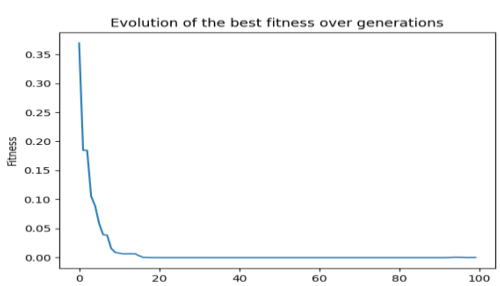
EXP 1 Plot



EXP 2 Plot



EXP 3 Plot



In the given experiments:

Experiment 1 achieved a best solution value of 0.000069 at generation 66, with an average solution value of 0.006 and an execution time of 0.042 seconds.

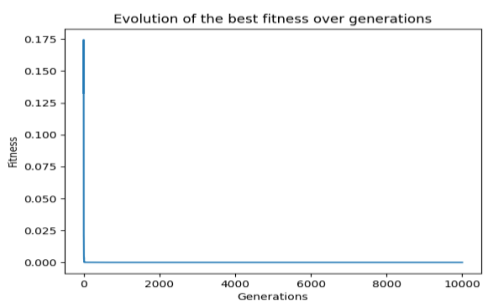
Experiment 2 attained a best solution value of 0.0004 at generation 52, with an average solution value of 0.0097 and an execution time of 0.063 seconds.

Experiment 3 obtained a best solution value of 0.00001 at generation 86, with an average solution value of 0.0113 and an execution time of 0.084 seconds.

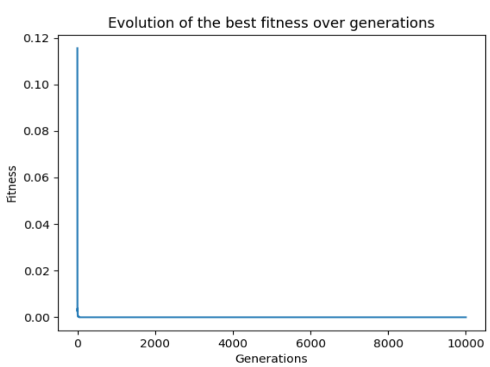
Considering the objective of minimizing the solution value, we can observe that experiment 3 achieved the lowest best solution value of 0.00001. However, it also has the highest average solution value and execution time among the three experiments.

Experiment 1, on the other hand, had a lower best solution value compared to experiment 2 but exhibited a lower average solution value and execution time.

EXP 4 Plot



EXP 5 Plot



Analysis based on dimension size:

Across all dimensions, the execution time generally increases with the dimensionality of the problem. This is because higher dimensions result in larger search spaces, requiring more computational resources and time to explore.

Experiment 3 consistently has the highest execution time, indicating that higher renewal rates lead to longer search times. This is because frequent renewals introduce more randomness and exploration, which can prolong the convergence process.

Experiment 1 often achieves the best solution values, suggesting that a balance between exploration and exploitation is crucial for finding optimal solutions efficiently. Lower population sizes (20 or 30) and moderate renewal rates (50 or 80) seem to contribute to this balance.

Experiment 2, with a population size of 40 and a higher crossover probability, exhibits competitive performance in terms of both solution quality and execution time, demonstrating the significance of crossover in solution refinement.

In higher-dimensional spaces, such as 10D, achieving optimal solutions becomes increasingly challenging due to the curse of dimensionality. This is reflected in the larger average solution values and longer execution times observed in Experiment 3 compared to the lower-dimensional counterparts.