EC Assignment #1 (GA)

電機系(碩一)陳昕佑 61375017H

My assigned problems are f2(Sschwefel's P2.22) in unimodal and f8(Schwefel function with dimension d=30) in multimodal. I will explain my Python code below step-by-step, and show the result of the comparison.

The code begins by defining the structure of individuals and the fitness criteria using DEAP (Distributed Evolutionary Algorithms in Python).

- **To create the Fitness and Individual Classes:**

```
creator.create("FitnessMin", base.Fitness, weights=(-1.0,))
creator.create("Individual", list, fitness=creator.FitnessMin)
```

'FitnessMin' represents a minimization problem where smaller values are better, and 'Individual' defines individuals as lists of real numbers with a fitness attribute.

- **Objective Functions: Schwefel Functions**

1. Unimodal Schwefel's P2.22 Function:

```
def schwefel_unimodal(x):
    schwefel_uni = np.sum(np.abs(x)) + np.prod(np.abs(x))
    return schwefel_uni
```

This function adds the sum and the product of the absolute values of the input array.

2. Multimodal Schwefel Function:

```
def schwefel_multimodal(x):
    d = len(x)
    schwefel_muti = 418.9829 * d - np.sum(x * np.sin(np.sqrt(np.abs(x))))
    return schwefel_muti
```

This function has many local minima and is harder to optimize than the unimodal version.

- **Creating Individuals:**
  The function ‘create_individual’ is used to generate individuals. Each individual consists of randomly initialized floating-point numbers.

```
def create_individual(n, lower_bound, upper_bound):
    id = [random.uniform(lower_bound, upper_bound) for _ in range(n)]
    return id
```

  This creates a list of ‘n’ random values between the provided bounds, representing the initial guess for an individual.

- **Genetic Algorithm setup:**
  The ‘setup_ga’ function defines the genetic algorithm operations, including:
  Attributes: The random floating-point values that represent the genes of an individual.
  Individuals and Population: Using ‘tools.initRepeat’, individuals are initialized by repeating random attribute generation.
  Evaluation Function: Assigns the fitness function (either unimodal or multimodal).
  Crossover (Mate): ‘tools.cxBlend’ combines two individuals by averaging their attributes.
  Mutation: ‘tools.mutGaussian’ applies random Gaussian mutation to maintain diversity.
  Selection: Tournament selection (‘tools.selTournament’) is applied to choose individuals based on their fitness. I also try two different selection strategies, ‘Roulette Wheel Selection’ and ‘Best Individual Selection’.

```
def setup_ga(evaluate_function, num_dimensions, lower_bound,
                    upper_bound):
    toolbox = base.Toolbox()
    toolbox.register("attr_float", random.uniform, lower_bound, upper_bound)
    toolbox.register("individual", tools.initRepeat, creator.Individual,
                        toolbox.attr_float, num_dimensions)
    toolbox.register("population", tools.initRepeat, list, toolbox.individual)
    toolbox.register("evaluate", evaluate_function)
    toolbox.register("mate", tools.cxBlend, alpha=0.5)
    toolbox.register("mutate", tools.mutGaussian, mu=0, sigma=1, indpb=0.1)
    toolbox.register("select", tools.selTournament, tournsize=3)
    return toolbox
```

- **Running the GA with Evaluation Limits:**
  Here is the core function that runs the genetic algorithm, stopping after 200,000 total evaluations.

```
def run_ga_with_evaluation_limit(toolbox, population_size,
                                 num_generations, evaluation_limit=200000):
  pop = toolbox.population(n=population_size)
  fitness_history = []
  total_evaluations = 0
  for gen in range(num_generations):
      offspring = algorithms.varAnd(pop, toolbox, cxpb=0.5, mutpb=0.2)
      fits = list(map(toolbox.evaluate, offspring))
      for ind, fit in zip(offspring, fits):
          ind.fitness.values = fit
      pop = toolbox.select(offspring, k=len(pop))
      best_ind = tools.selBest(pop, k=1)[0]
      fitness_history.append(best_ind.fitness.values[0])
      total_evaluations += len(fits)
      if total_evaluations >= evaluation_limit:
          break
  return best_ind, fitness_history, total_evaluations
```

Population Initialization: A population is initialized based on the population_size parameter.

Generations Loop: For each generation:

Offspring Creation: Offspring are created using crossover and mutation.

Evaluation: The fitness function evaluates each individual.

Selection: The population is updated with the best offspring.

Stopping Condition: If the total number of evaluations reaches the limit, the loop stops.

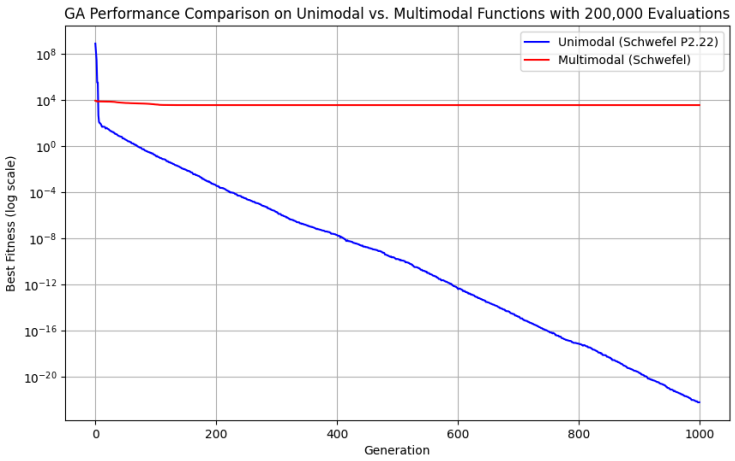
- **Running the algorithm on two functions:**

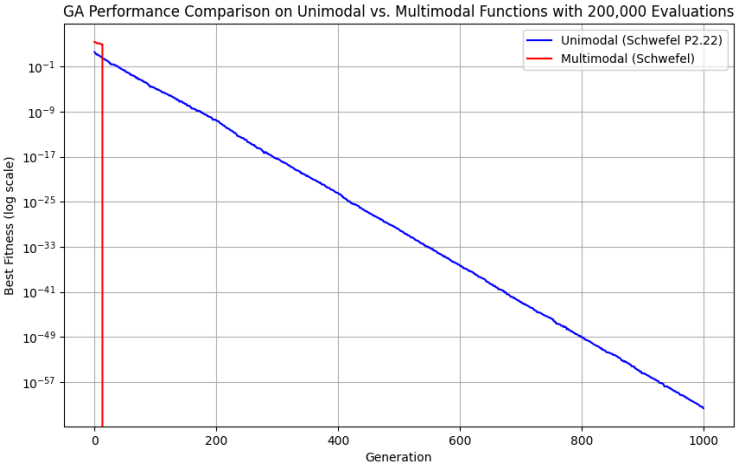
Unimodal Schwefel P2.22:

```
toolbox_unimodal = setup_ga(eval_schwefel_unimodal, num_dimensions, -10, 10)
_, fitness_history_unimodal, evals_uni = run_ga_with_evaluation_limit(
    toolbox_unimodal, population_size, num_generations, evaluation_limit)
```

Multimodal Schwefel:

```
toolbox_multimodal = setup_ga(eval_schwefel_multimodal, num_dimensions, -500,
    500)
_, fitness_history_multimodal, evals_multi = run_ga_with_evaluation_limit(
    toolbox_multimodal, population_size, num_generations, evaluation_limit)
```





The figures on top have the same ‘population_size’ equal to 200, and ‘num_generations’ equal to 1000. The difference between these two figures is dimension, the figure on the left is 30-dimension, and the right figure is 10-dimension.

- **Higher Dimensions(d=30):**

  The GA struggles more with the multimodal function, likely due to the increased number of local minima and the overall complexity of the search space.

  For the unimodal function, while eventually optimized, requires many more generations to converge to a good solution.

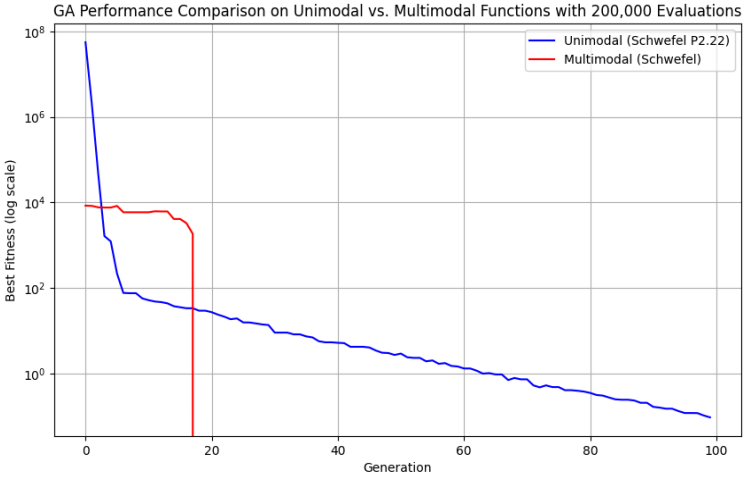
  The red line remains flat, showing how the GA couldn’t improve the fitness meaningfully for this function in a higher-dimensional space.

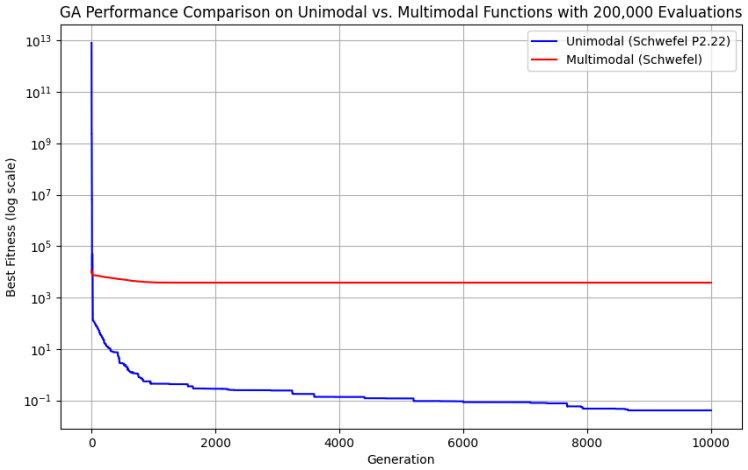
- **Lower Dimensions(d=10):**

  The GA performs much better overall. Both the unimodal and multimodal functions converge to low fitness values quickly.

  The reduced complexity of the search space allows the GA to find good solutions faster and more effectively.

This comparison reinforces the notion that as the dimensionality of a problem increases, the difficulty for a GA to find optimal solutions also increases, particularly for multimodal problems where many local minima exist. Lower dimensional problems allow the GA to explore the search space more effectively and converge to a good solution faster.





These two figures both have the same dimension(d=30), the differences are the parameters ‘population_size’ and ‘num_generations’.

The figure on the left has population_size=2000, num_generations=100, and the figure on the right has population_size=20, num_generations=10000.

- **Left figure(d=30, population_size=2000, num_generations=100):**

  The larger population size allows the GA to explore the search space more thoroughly in each generation. However, the number of generations is relatively small. (i.e., population_size * num_generations=200,000)

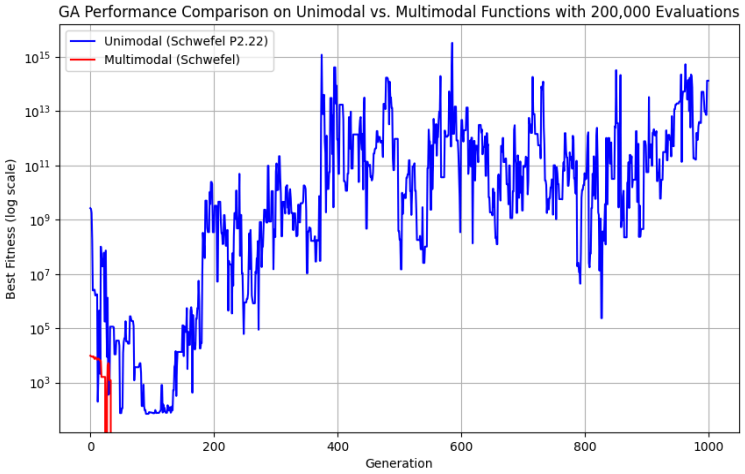
  The unimodal function reaches a very low fitness value, whereas the multimodal function also converges to a lower value than in previous examples, indicating that the large population helps mitigate some of the challenges of the complex multimodal space.

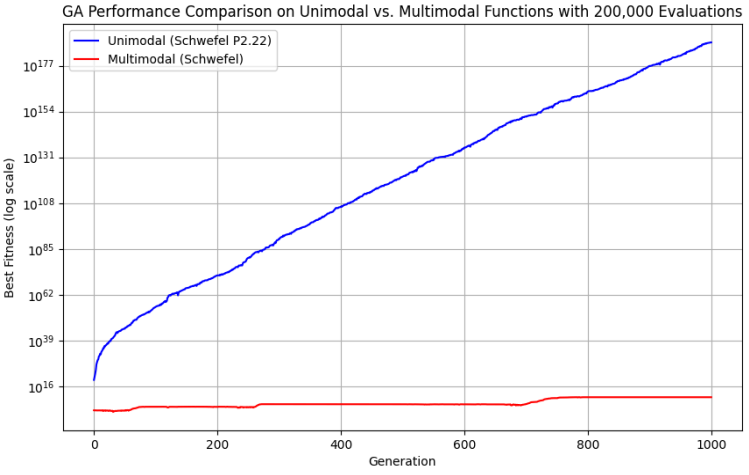
- **Right figure(d=30, population_size=20, num_generations=10000):**

  The smaller population size limits the diversity in the search space, making it harder for the GA to explore as much as with a larger population.

  The unimodal function reaches a very low fitness value, but it takes many more generations to converge compared to the first figure. The small population slows down the convergence rate because the GA relies on the mutation and crossover of fewer individuals per generation. The multimodal function remains stuck at a relatively high fitness level throughout the entire run. The small population may not be able to escape the local minima due to the limited exploration power.

In summary, increasing the population size allows for faster and more effective exploration of the search space, while increasing the number of generations with a small population slows down convergence and can lead to suboptimal results.





The left figure uses the rank-based selection, and the right figure uses the Roulette Wheel selection. (d=30, population_size=200, num_generations=1000)

Compared with the very first figure which uses the Tournament selection, rank-based and roulette selection both introduced more variability and instability into the results, especially for the unimodal function.

- **Rank-Based Selection**:

  Works better for the multimodal function but suffers from significant instability for the unimodal function. The selection pressure is reduced with rank-based selection, which can result in erratic behavior when fitness values are vastly different, as seen in the large fluctuations of the unimodal fitness.

- **Roulette Wheel Selection**:

  This selection performs poorly overall, especially for the unimodal function, where fitness values escalate to extremely high levels. This suggests that the algorithm was heavily influenced by random noise or poor selection pressure. The multimodal function remained flat but didn't improve significantly.

In the end, I generated this 3D plot just out of curiosity.