Optimisation Route for Genetic Algorithms – UFCY3-15-3

Timo Andrei Claudiu – 19000915

1. INTRODUCTION

As stated in the title, I am going to carry on working on the ***Optimisation*** part as my next step after completing worksheet 3, which shows the implementation of a Genetic Algorithm using a maximization function such as *Counting One's* function and two minimisation functions as shown below:

$$f(\mathbf{x}) = (x_1 - 1)^2 + \sum_{i=2}^{d} i\left(2x_i^2 - x_{i-1}\right)^2$$

where $-10 \le x \le 10$, start with $d=20$

Figure 1.1 *Rosenbrock minimisation function*

$$f(x) = -20\exp\left(-0.2\sqrt{\frac{1}{D}\sum_{i=1}^{D} x^2}\right) - \exp\left(\frac{1}{D}\sum_{i=1}^{D} cos2\pi x_i\right)$$

where $-32 \le x \le 32$, start with $D=20$

Figure 1.2 *Ackley minimisation function*

My approach to optimisation is to implement some new operators to compare them with the initial algorithm, and then evaluate which operator is more efficient in terms of competitive performance. The initial selection method required in the worksheet is referred to as *Tournament selection* in my code, and the *Roulette Wheel selection* that I designed is compared to the initial selection method. I ran my code **10 times** to obtain a median fitness and plotted images, which will be shown in the Experimentation section. Furthermore, I have improved the Crossover function to obtain correct answers because the Crossover provided in the worksheet gave me incorrect answers.

In addition, I will investigate the differences between *Tournament* and *Roulette Wheel* selection and investigate what changes in the algorithm implementation will apply to maximisation and minimisation functions, as well as identify and state examples of using AI for a task that has raised ethical issues, all of which will be included in the **background research section**. After completing this assignment, I expect to advance and broaden my knowledge and understanding of how Genetic Algorithms work, as well as the performance of the processes within GAs such as *Selection*, *Crossover*, and *Mutation*.

2. BACKGROUND RESEARCH

2.1 Introduction to Genetic Algorithms

Genetic algorithms are a type of optimisation algorithm that is used to determine the maximum or minimum value of a function (Jenna Carr, 2014). According to this assignment, we will use a Genetic Algorithm to maximise (*Counting One's function*) and minimise (*Rosenbrock and Ackley functions*).

The steps of the Genetic Algorithm are as follows, according to Jadaan, Rajamani, and Rao (2008):

- ***SELECTION***: The first step in GAs is to choose two randomly selected parents for reproduction. That selection method will work differently depending on the optimization function. Individuals with higher fitness, for example, are more likely to be chosen for mating when working with maximisation, while those with lower fitness are more likely to be chosen when working with minimisation.

- ***RECOMBINATION***: Using one point swapping, this step requires the crossover process to create a new better individual's representation of the gene from its' parent's representation.
- ***MUTATION***: After a crossover population is created, the mutation operator is used to change the chromosomes within the individual's representation by bit flipping.
- ***EVALUATION***: Create a rule-base from the genome and apply it to the problem of calculating an individual's fitness using a fitness function.
- ***REPLACEMENT***: When one generation is complete, the parent population is replaced by the offspring population to continue the cycle.

When the population converges on the optimal solution, the Genetic Algorithm comes to a halt.

The following is a visual representation of the Genetic Algorithm process:

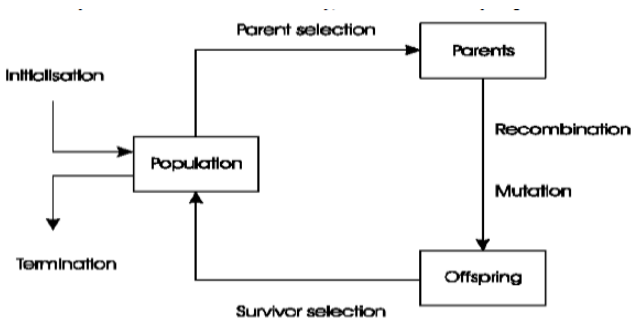


Figure 1.3 *Genetic Algorithm Diagram*

2.2 Selection Methods

In this research paper, I will be discussing the two commonly used selection methods: *Tournament Selection* and *Roulette Wheel Selection.*

Tournament Selection:

In a *Genetic Algorithm*, *Tournament selection* is a method for selecting an individual from a population of individuals. By participating in a series of "Tournaments" amongst several individuals (or "chromosomes") chosen at random from the population. The fittest individual will be picked to represent the next generation. In other words, the Tournament selection is used in a Genetic Algorithm to select the best fittest candidate from the current cycle of generation. In addition, a step-by-step description of the Tournament algorithm is provided below:

1. At random, select *k* (the tournament size) individuals from the population;
2. With probability *p*, select the best individual from the tournament;
3. Select the second best candidate with probability $p\cdot(1-p)$;
4. Select the third best individual with probability $p\cdot(1-p)^2$, and so on.

In my *Tournament Algorithm* implementation, *k* is set to 2, which means that for each of the **N** generations we loop through, we will choose two random parents from the individual's population and compare their fitness, with the one with better fitness being added to the offspring population.

Roulette Wheel Selection

Individuals are selected with a probability that is directly proportional to their fitness values in a proportional roulette wheel, according to Noraini Mohd Razali and John Geraghty, i.e. an individual's selection corresponds to a portion of a roulette wheel. That is, those who are more fit take more proportional slots, while those who are less fit take smaller slots. The number of people chosen is determined by the number of times the wheel is spun. This procedure is repeated until the desired number of people have been selected (Jadaan, Rajamani, & Rao, 2008). Noraini Mohd Razali and John Geraghty also say the following about the probab chose:

$$p_i = \frac{f_i}{\sum_{j=1}^{n} f_j}$$

Figure 1.4 *P_i represents the probability of selection for individual i.*

Where **n** is the population size and f_1, f_2,…,f_n are the fitness of individuals 1,2,…,n.

The following are the steps for selecting a *Roulette Wheel* (A. Shukla, H. M. Pandey, and D. Mehrotra, 2015):

1. Calculate the sum of each individual's fitness values in the population.
2. Divide the fitness value of each individual chromosome by the sum of fitness values for the entire population to

calculate the fitness value of each individual and their probability of selection.

3. Divide the roulette wheel into sectors based on the probabilities calculated in the previous step.
4. Count the number of times the wheel is spinning. When the roulette comes to a stop, the sector on which the pointer is pointing corresponds to the individual being chosen.

2.3 Introduction to ABCs

The artificial bee colony is another algorithm for solving optimization problems. The algorithm functions similarly to a bee colony, with three types of bees: employed bees, onlookers, and scouts (Karaboga, Basturk, 2007). Possible solutions are not represented as individuals with genes in this algorithm, as is the case with genetic algorithms and fitness. The food source represents potential solutions to an optimization problem, while the nectar represents fitness (Karaboga, Basturk, 2007). Using this data, it is clear that the representation of the problem is conceptually similar. Both genetic algorithms and bee algorithms have a problem solution that they represent in their respective ways, and then they have a way of representing the fitness of that problem solution.

2.4 ABC structure

Breaking the process down**, *the initialization phase*** is the first step. This is the stage at which a randomly distributed initial population is generated (Karaboga, Basturk, 2007).

The employed bee phase follows, in which employed bees search for new solutions within the vicinity of the food sources in their memory, and when they find one, they evaluate the fitness of the food source (Karaboga, Gorkemli, Ozturk, 2014). The next step is for an employed bee to share information about the food source with onlooker bees in the hive who are currently dancing in the dancing area after a new food source is produced and greedy selection has been applied between it and its parent (Karaboga, Gorkemli, Ozturk, 2014).

The onlooker phase comes next. Food sources are selected probabilistically by **onlooker bees** based on information received from employed bees during this phase (Karaboga, Gorkemli, Ozturk, 2014). The authors states that a fitness-based selection algorithm, such as roulette wheel selection, can be used. Fitness-based selection is also used in genetic algorithms, but within different ways, such as selecting individuals for crossover rather than using a selected found food source to apply greedy selection between it and a nearby food source after its fitness has been determined (Karaboga, Gorkemli, Ozturk, 2014).

The scout phase is the final phase of the ABC algorithm. Any solutions that cannot be improved after a predetermined number of trials are abandoned, and the employed bees become scouts (Karaboga, Gorkemli, Ozturk, 2014). Scout will then begin to look for new solutions at random (Karaboga, Gorkemli, Ozturk, 2014).

2.5 ABC algorithm parameters

Attempting to compare parameter options for both ***Genetic algorithms***, the basic genetic algorithm can have multiple parameters, in the case of this research paper, including ***Generation***, ***Population***, ***Mutation Rate***, and ***Crossover rate*** compared with ABCs. The basic ***ABC*** used in this study, employs only one control parameter, which is called ***limit*** (Karaboga, Akay, 2009). The study's limit is that if it is exceeded, the food source will no longer be exploited and will be considered abandoned (Karaboga, Akay, 2009).

2.6 Comparison between GA and ABCs

For global optimization, both genetic algorithms and artificial bee colonies can be used. Although genetic algorithms and ABC appear to be very different on the surface, they do share some similarities in the search process. For example, the employed bee phase is comparable to the selection stage in a genetic algorithm, as solutions are selected and their information is saved based on their fitness. Another example could be that the onlooker phase is similar to genetic algorithm mutation, but it is guided rather than random.

2.7 Elitism

An optimisation function that depicts Elitism has been implemented to make the program more efficient. According to Jadaan, Rajamani, and Rao (2008), elitism involves replacing the child population's worst chromosomes with the best members of the parent population. This operator has demonstrated the ability to accelerate GA convergence by retaining the best solution found in each generation. In the Maximisation program, we restore the two worst in the new population with the two best in the old population, and the best fitness is the *maximum* fitness (best fitness). We reverse the approach for the Minimisation one by replacing the two best in the new population with the two worst in the old population; the best fitness is the *minimum* fitness (worst fitness) at specific indices. (For code, see the Appendix).

2.8 Ethical Issues of AI in Medical Applications

One of the related GA applications that I am interested in is Artificial Intelligence via Neural Networks Applied to Medical Applications. Artificial intelligence has the potential to improve the overall health of society (Ternent, James Thompson, Maximilian 2020). It is utilised for diagnostic purposes in health care, such as clinical practice and translational research. Aside from the numerous benefits provided by AI in medical applications, several ethical concerns have been expressed.

First and foremost, we should consider the impact of Ternent, James Thompson, and Maximilian's Expectations of Machine Learning in Medicine. They claim that there are unreasonably high hopes for machine learning in medicine, as well as equally unrealistic fears. They also claim that 63% of the adult population in the UK is distressed to personal data being used to improve healthcare and to artificial intelligence systems replacing doctors and nurses in tasks that they normally perform. However, a large proportion of medical students in Germany believe that machine learning will improve medicine, according to a German survey (Ternent, James Thompson, Maximilian 2020). As a consequence, there may be a difference between a satisfactory explanation for a machine learning expert and a tolerable explanation for a patient, causing distress and possibly harm.

The following section is Data and Privacy, which indicates that the barriers of data sources (such as electronic health records) can introduce bias because they were never intended to be used for anything other than clinical care and billing (in their current state). They also claim that, with bills such as the General Data Protection Regulation (GDPR) generating much-deserved interest in the storage and processing of personal data, people are becoming increasingly extremely sceptical of what personal data is collected and how it is used.

As a result, researchers may be under more pressure to find a compelling reason to include a data stream (Ternent, James Thompson and Maximilian 2020).

Finally, they talk about the lack of an acceptable decision-making explanation. They argue that using a neural network to expose the facets of many issues raises an important ethical question: ***What potential aspects of a decision are appropriate for an ethical explanation, and at what level should unethical aspects be removed or recontextualized?*** Assume that skin tone is an emergent feature in a machine learning model; clearly, this is an unacceptable aspect of a patient to consider. As a result, making medical diagnoses will be extremely difficult.

3. EXPERIMENTATION

From all three worksheets, I implemented a Genetic Algorithm that includes **fitness calculation**, **initialisation**, **tournament selection**, **one-point crossover**, and **bit-flipping mutation**. All processes have a function and can be executed when called. To compare with the *Tournament selection*, I included a new function that demonstrates the *Roulette Wheel Selection* method, so that when I run the two GAs with different selection methods within the program, I can analyse and plot the performance of those using the ***matplotlib.pyplot*** library, and the recorded results will be averages shown over ten runs. Because we are working with both maximisation and minimisation, I created two types of optimisation programs: one for maximisation that uses the *Counting One's* function and another for minimisation that uses the *Rosenbrock* and *Ackley functions* to calculate fitness. We should change the fitness calculation in worksheet 3 to use real numbers rather than the binary representation (0 and 1).

This following piece of code represents my fitness calculation function with real numbers, in which an individual's fitness is equal to the number of 1's in its array of genes (genome), which can easily be extended to the sum of the real-valued genes:

```
# Calculate individual's fitness
def counting_ones(ind):
    fitness = 0
    for i in range(0, N):
        fitness = fitness + ind.gene[i]
    return fitness
```

My minimisation fitness calculation functions with real numbers are shown below:

```
def mini_function(ind):
    fitness = 0
    squared = 0
    first_expression = (ind.gene[0] - 1) ** 2 # (x1-1)^2
    second_expression = 0
    for i in range(1, N):
        # exp = [2 * xi^2 - x(i-1)]^2
        squared = i * ((2*ind.gene[i]*ind.gene[i] - ind.gene[i-1]) ** 2)
        # i * exp
        second_expression += squared
    # (x1-1)^2 + SUM{ i* [2 * xi^2 - x(i-1)]^2 }
    fitness = first_expression + second_expression
    return fitness
```

```
# Calculate individual's fitness
# first minimisation function

def mini_function(ind):
    fitness = 0
    first_exp = 0
    second_exp = 0
    for i in range(0, N):
        first_exp += math.cos(2 * math.pi * ind.gene[i])
        second_exp += ind.gene[i] * ind.gene[i]
    first_exp = math.exp((1/N) * first_exp)
    second_exp = (-20) * math.exp((-0.2) * math.sqrt((1/N) * second_exp))
    fitness = second_exp - first_exp
    return fitness
```

Figure 1.5 1.6 *Rosenbrock fitness calculation function and Ackley fitness calculation function*

I modified my fitness calculation functions by passing the individual's population instance (*ind*) as a parameter, accessing each genome by looping from *i* to the size of the individual's genes (ind.gene[i]), calculating and returning its fitness. In addition, I have changed the ***Crossover algorithm*** to make it work correctly and better because the old crossover returned the wrong individual's gene after swapping, and I believe the reason could be ***temp*** changes when we assign back to offspring[i +1]. I also attempted to use deepcopy, but it did not perform as expected. (See the Appendix for code).

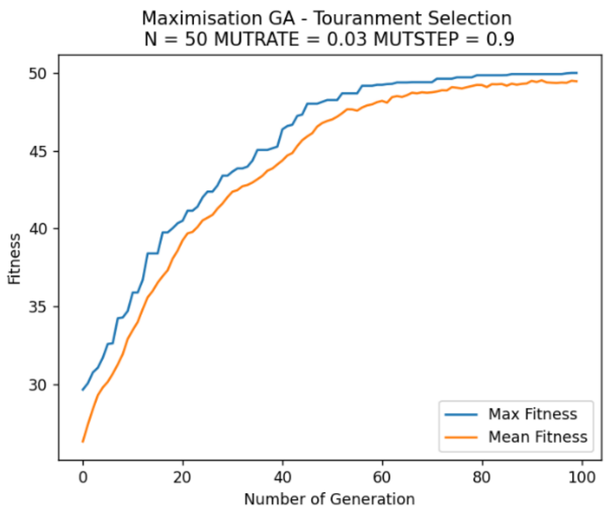
3.1 MAXIMISATION RESULTS

The next two figures below depict *Tournament* and *Roulette Wheel* selection, which I implemented into my maximisation program using the Counting One's function:

```
# Tourament Selection Process
def tourannment_selection(population):
    offspring = []
    # Select two parents and recombine pairs of parents
    for i in range(0, P):
        parent1 = random.randint(0, P - 1)
        off1 = population[parent1]
        parent2 = random.randint(0, P - 1)
        off2 = population[parent2]
        if (off1.fitness > off2.fitness): # if one's fitness higher then add to temp offsptring
            offspring.append(off1)
        else:
            offspring.append(off2)

    return offspring
```

```
# Roulette Wheel Selection Process
def RW_selection(population):
    # total fitness of initial pop
    initial_fits = total_fitness(population)

    offspring = copy.deepcopy(population)
    # Roulette Wheel Selection Process
    # Select two parents and recombine pairs of parents
    for i in range(0, P):
        selection_point = random.uniform(0.0, initial_fits)
        running_total = 0
        j = 0
        while running_total <= selection_point:
            running_total += population[j].fitness
            j += 1
            if(j == P):
                break
        offspring[i] = population[j-1]

    return offspring
```

Figure 1.7 1.8 *Tournament selection Maximisation and Roulette Wheel selection Maximisation*



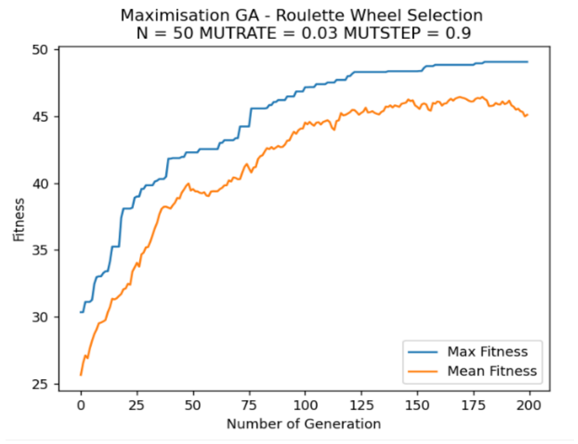
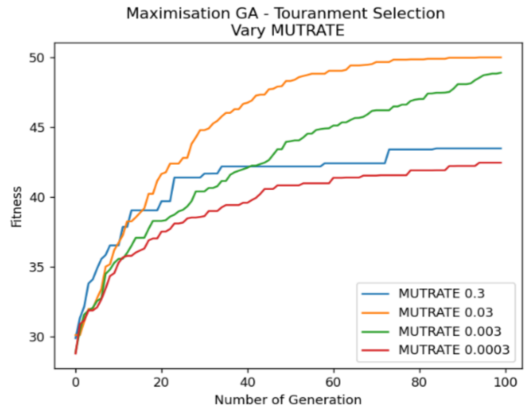


Figure 1.9 2.0 *a single Maximisation GA run - Tournament and Roulette Wheel selection*

Gene length: **N** = 50, Individual population: **P** = 50, Mutation rate: **MUTRATE** = 0.03, Mutation step: **MUTSTEP** = 0.9, Number of Generation: **GENERATIONS** = 100/ 200.

Over the course of ten runs of my maximisation GA with *Tournament selection*, I encountered that the maximum fitness increased and converged into a horizontal line at 50.0, whereas the mean fitness only reached **49.467339709485785**. This plot was carried out over **100 generations**. I plotted the same statistics with my Maximisation GA with *Roulette Wheel selection* over ten runs, and while both **Max** and **Mean** fitness increased, it took over **200 generations** to converge. The maximum fitness was only **49.06776626442139**, while the mean fitness was even lower, around **45.10946136458607**.

After varying the Mutation rate for both maximization GA with *Tournament* and *Roulette Wheel* over four different mutation rates, I discovered that the fitness reached a maximum when **MUTRATE = 0.03** and **MUTSTEP = 0.9**, as shown in the plot below:



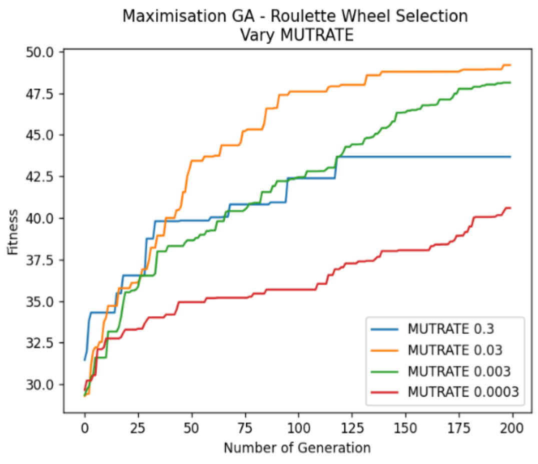
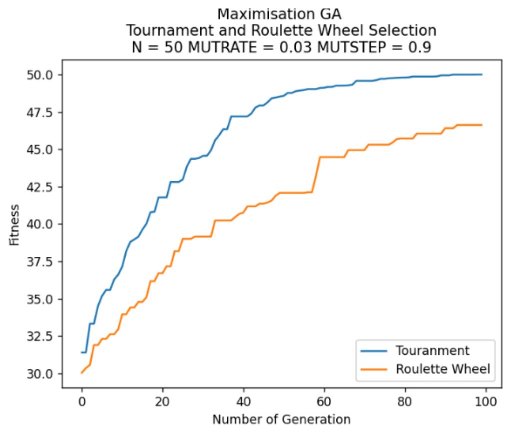


Figure 2.1 *Max fitness at 50.0 - MUTRATE 0.03*
Figure 2.2 *Max fitness at 49.18131827030025 – MUTRATE 0.03*

I also created two plots to demonstrate the clear competitive performance of two selection methods, as shown below:



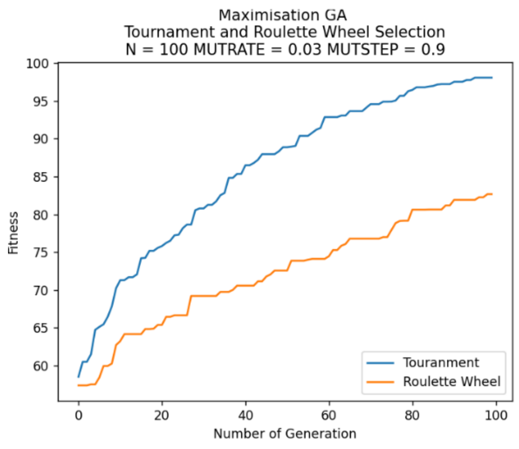


Figure 2.3 2.4 *Contrasts between Tournament and Roulette Wheel Selection*

The first plot has a gene length of 50 and the second has a gene length of 100. You'll notice that for both gene lengths, *Tournament selection* always reaches its maximum fitness before

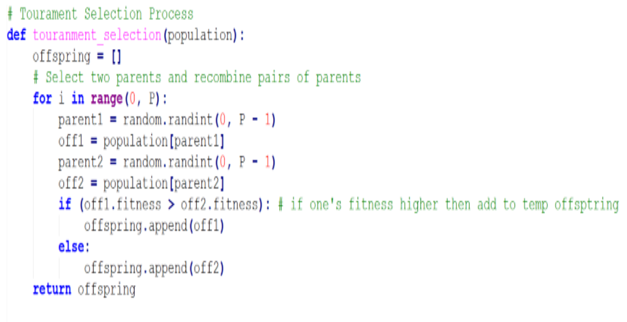
Roulette Wheel selection. Tournament's fitness reached **50.0** in the first plot and **98.60159889718605** in the second, but never reached 100.0 as I predicted. I believe the problem is related to the **number of Generations**; Perhaps if we increase the cycle length, the fitness with 100 genes length will flatten. The Roulette Wheel's fitness peaked at **46.62542552330858** for the first plot and **82.66273008007921** for the second.

3.2 MINIMISATION RESULTS

Because we are working with a minimization program, the best fitness that we obtain should be the smallest, and we should also select individuals with lower fitness from the population. As a result, there will be some alteration in the Selection process while GA is implemented.

3.2.1 Rosenbrock minimisation function

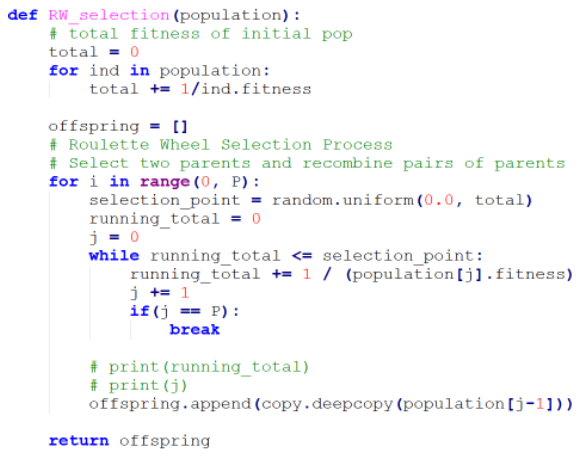
I have changed the *Tournament selection* and *Roulette Wheel selection* from the equation in figure 1.1 - *Rosenbrock minimisation* function to those in figures 1.7 and 1.8.



```python
# Tourament Selection Process
def touranment_selection(population):
    offspring = []
    # Select two parents and recombine pairs of parents
    for i in range(0, P):
        parent1 = random.randint(0, P - 1)
        off1 = population[parent1]
        parent2 = random.randint(0, P - 1)
        off2 = population[parent2]
        if (off1.fitness > off2.fitness): # if one's fitness higher then add to temp offsptring
            offspring.append(off1)
        else:
            offspring.append(off2)
    return offspring
```

Figure 2.5 *Minimisation Tournament - Rosenbrock function*

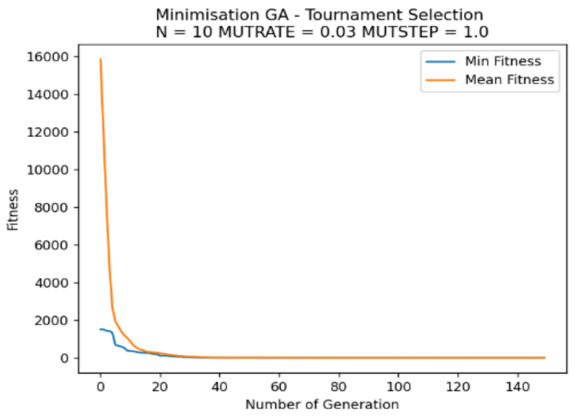
I changed the ***if condition*** in this minimisation Tournament selection so that the individual with the lowest fitness is chosen for the offspring population.



```python
def RW_selection(population):
    # total fitness of initial pop
    total = 0
    for ind in population:
        total += 1/ind.fitness

    offspring = []
    # Roulette Wheel Selection Process
    # Select two parents and recombine pairs of parents
    for i in range(0, P):
        selection_point = random.uniform(0.0, total)
        running_total = 0
        j = 0
        while running_total <= selection_point:
            running_total += 1 / (population[j].fitness)
            j += 1
            if(j == P):
                break

        # print(running_total)
        # print(j)
        offspring.append(copy.deepcopy(population[j-1]))

    return offspring
```

Figure 2.6 *Minimisation Roulette Wheel - Rosenbrock function*

In this minimisation *Roulette Wheel selection*, we wanted to ensure that individuals with low fitness had a higher chance of being chosen, so I inversed the fitness of the individuals and added the sum, which caused the algorithm to pick up the weaker individuals.



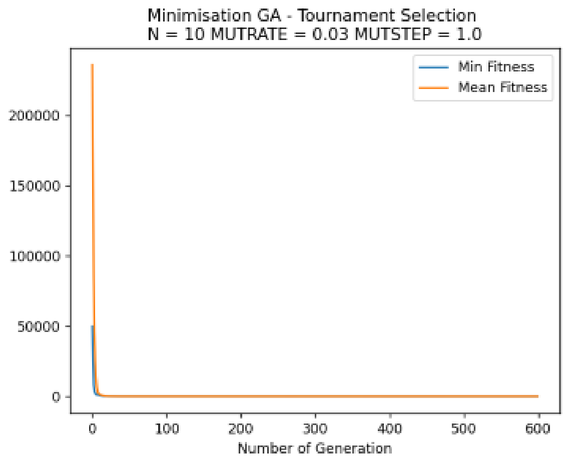


Figure 2.7 2.8 *a single Minimisation GA run. - Tournament and Roulette Wheel selection*

Individual population: **P** = 50, Gene length: **N** = 10. Mutation step: **MUTSTEP** = 1.0, Mutation rate: **MUTRATE** = 0.03, The number of **generations** is 600.

Over the course of ten runs of my minimisation GA with *Tournament selection*, I discovered that the minimum fitness decreased sharply and converged into a horizontal line at **0.011527590079116795**, which is the best fitness that I can get over 10 different runs, but I predicted it to drop to **0.0**, but the more cycles of generations I increased, it still stayed at nearly **0.0**, and I think it will make the algorithm more realistic to have some approximate error, whereas the mean fitness only reduced to **0.7425531247516733**. This plot was carried out over 600 generations. I also plotted a minimisation GA using *Roulette Wheel Selection* using the same statistics, and the lowest possible fitness achieved over 10 runs was **0.03372919809378878**, with a mean fitness of **0.5167875596054593**.

I also experimented with mutation rate and mutation step, and after varying and changing the Mutation step from 0.3 to 0.7 to 1.0, the best one that resulted in a significant decrease in fitness is 1.0, and the best possible mutation rate is 0.03 as always.

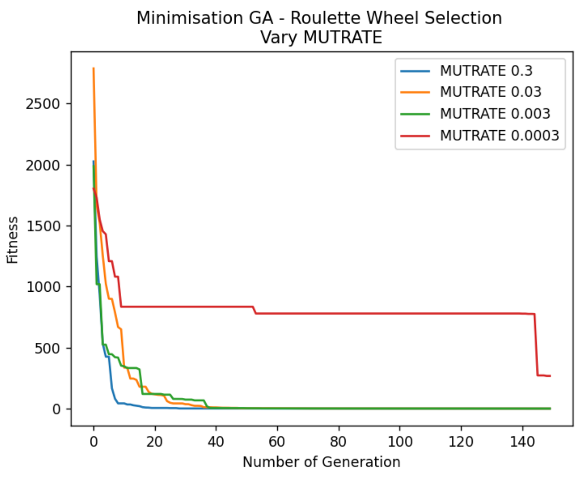


Figure 2.9 *Min fitness at 0.5512263916570717 - MUTRATE 0.3*

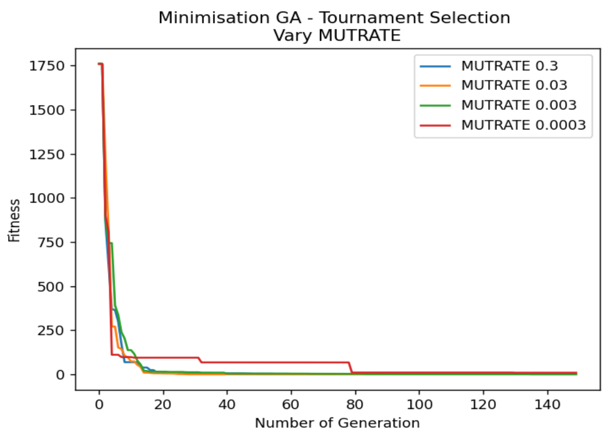
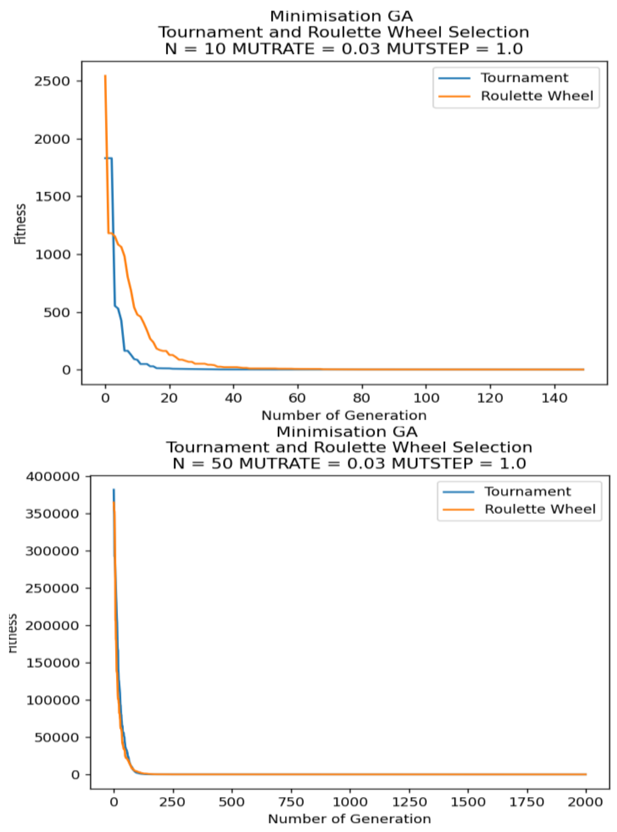


Figure 3.0 *Min fitness at 0.505488373883888 - MUTRATE 0.03*



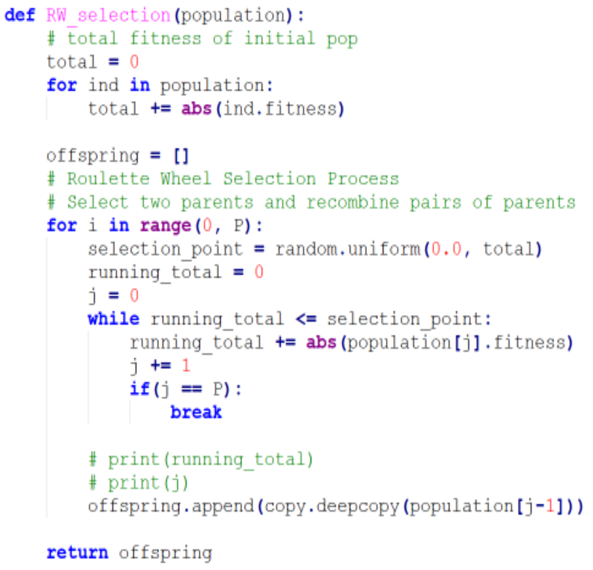
e

Figure 3.1 3.2 *Comparisons between Tournament selection and Roulette Wheel selection*

As we can observe, both approaches perform well for minimising *GA (Rosenbrock function)*, but *Tournament selection* appears to be superior. Tournament has a minimum fitness of **0.13217643429953985**, while *Roulette Wheel* has a minimum fitness of **0.3409070926474461**. Because I wanted to test my GA over a longer gene length of 50, I noticed that they took a little longer to process and required more generations (2000 generations) to converge the horizontal line.

3.2.2 Ackley minimisation function

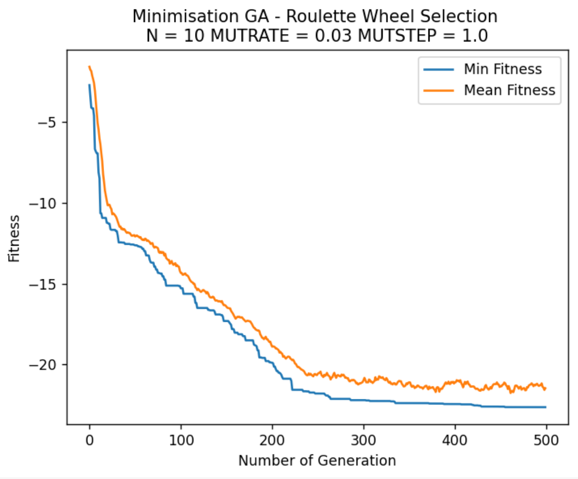
The *Ackley function* from **figure 1.2** is another minimisation function that I need to show the experimental results for. The only thing I needed to change was the Roulette Wheel.



```python
def RW_selection(population):
    # total fitness of initial pop
    total = 0
    for ind in population:
        total += abs(ind.fitness)

    offspring = []
    # Roulette Wheel Selection Process
    # Select two parents and recombine pairs of parents
    for i in range(0, P):
        selection_point = random.uniform(0.0, total)
        running_total = 0
        j = 0
        while running_total <= selection_point:
            running_total += abs(population[j].fitness)
            j += 1
            if(j == P):
                break

        # print(running_total)
        # print(j)
        offspring.append(copy.deepcopy(population[j-1]))

    return offspring
```

Figure 3.3 *Minimization Roulette Wheel selection Ackley function*

Because the *Ackley function* always returns **negative** fitness, I used the math library's *abs()* function to convert all negative numbers to positive ones. As a result, the largest negative number becomes the smallest positive number, and so on. The algorithm will resemble the *Roulette Wheel selection* program for maximisation.



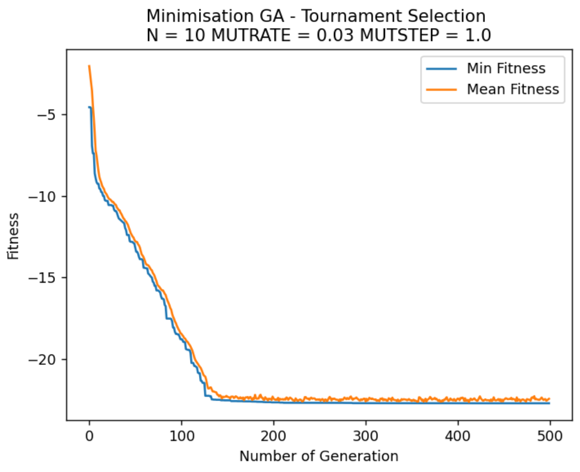


Figure 3.4 3.5 *a single Minimisation GA run (Ackley function). - Tournament and Roulette Wheel selection*

Individual population: **P** = 50, Gene length: **N** = 10. Mutation step: **MUTSTEP** = 1.0, Mutation rate: **MUTRATE** = 0.03, The number of **generations** is 500.

Over the course of ten runs of my minimisation GA (Ackley function) with *Tournament selection*, I observed that the minimum fitness converged into a horizontal line at **-22.698435758025173**, which is the best fitness that I can get over 10 different runs with a mean fitness of **-22.425626443811986**. This plot was carried out over 500 generations. I also plotted a minimisation GA using *Roulette Wheel Selection* using the same statistics, and the lowest possible fitness achieved over 10 runs was **-22.65638596126548**, with a mean fitness of **-21.481933208749926**.

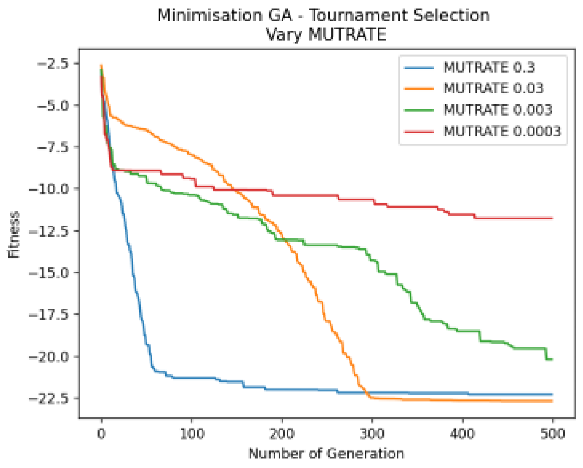


Figure 3.6 *Min fitness at -22.69556884202363 - MUTRATE 0.03*

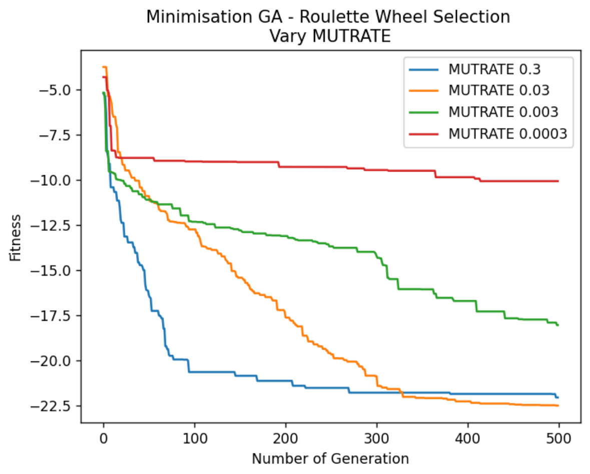


Figure 3.7 *Min fitness at -22.49680201310099 - MUTRATE 0.03*

The following plot compares *Tournament* and *Roulette Wheel selection* over the minimisation GA of the *Ackley function*.

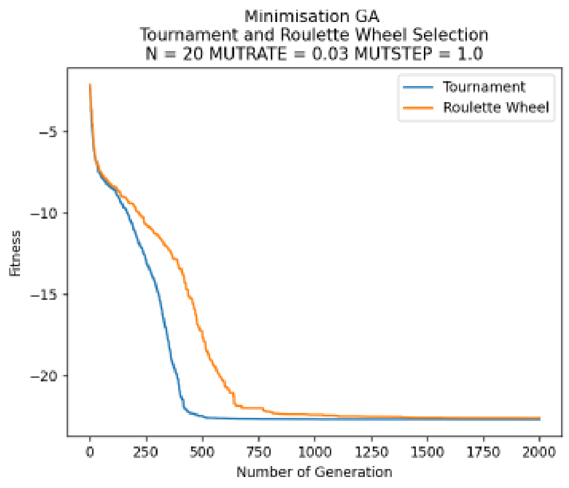


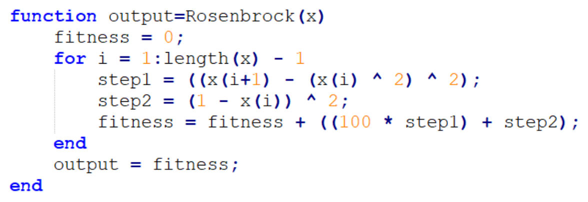
Figure 3.8 *Comparison between Tournament selection and Roulette Wheel selection*

Individual population: **P** = 50, Gene length: **N** = 10. Mutation step: **MUTSTEP** = 1.0, Mutation rate: **MUTRATE** = 0.03, **GENERATIONS** = 2000 is the number of generations.

Each selection method converged at its own line in this plot. Over 2000 generations, the fitness of using *Tournament selection* and *Roulette Wheel selection* converged horizontally at **-22.711649728948863** and **-22.615465913968446**, respectively. Both functions used a **0.03** mutation rate and a **1.0** mutation step. I also discovered that selecting the *Roulette Wheel* took slightly longer to print out the results than selecting the *Tournament*. However, as we can observe, both approaches perform well for minimising *GA (Ackley function)*, but just like the case of *Rosenbrock function*, the *Tournament selection* appears to be superior in this case too.

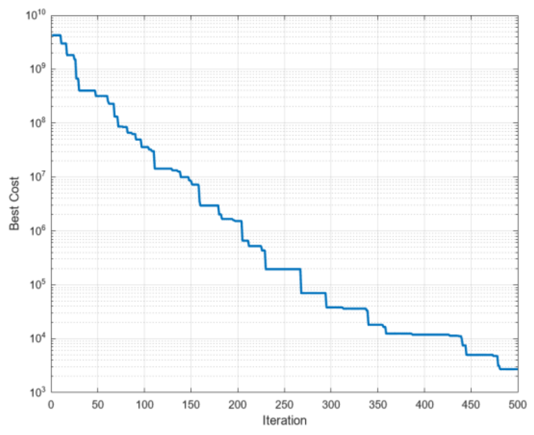
4. Comparing ABC and GA

These are the results of running the artificial bee colony source code written by Yarpiz (2015).

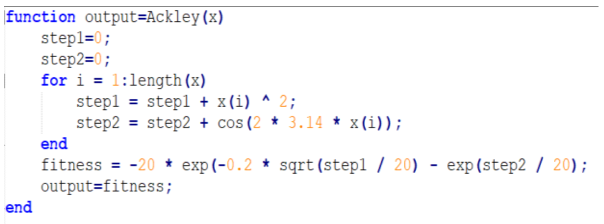


```
function output=Rosenbrock(x)
    fitness = 0;
    for i = 1:length(x) - 1
        step1 = ((x(i+1) - (x(i) ^ 2) ^ 2);
        step2 = (1 - x(i)) ^ 2;
        fitness = fitness + ((100 * step1) + step2);
    end
    output = fitness;
end
```

Figure 3.9 *Rosenbrock function used in Matlab*

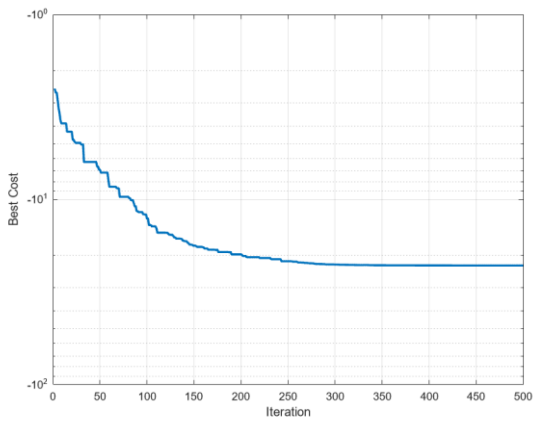


```
Iteration 497: Best Cost = 2713.7585
Iteration 498: Best Cost = 2713.7585
Iteration 499: Best Cost = 2713.7585
Iteration 500: Best Cost = 2713.7585
```



```
function output=Ackley(x)
    step1=0;
    step2=0;
    for i = 1:length(x)
        step1 = step1 + x(i) ^ 2;
        step2 = step2 + cos(2 * 3.14 * x(i));
    end
    fitness = -20 * exp(-0.2 * sqrt(step1 / 20) - exp(step2 / 20);
    output=fitness;
end
```

Figure 4.0 *Ackley function used in Matlab*



```
Iteration 498: Best Cost = -22.7161
Iteration 499: Best Cost = -22.7161
Iteration 500: Best Cost = -22.7161
```

The ABC algorithm chosen, as well as the genetic algorithm used in testing, use the same population count, epoch amount, and number of values in each solution, 300, 500, and 20, respectively.

The ***ABC Rosenbrock method*** results show that it performed significantly worse than the genetic algorithm, and the opposite is true when the Ackley functions are compared.

The ***Roulette Wheel*** is used in this implementation of the Bee algorithm. This was shown to perform poorly in the tested genetic algorithm when compared to other selection operators; the reason for the poor performance of the Rosenbrock function could be related to this issue. In other words, as the number of chromosomes in genes increases, performance suffers as more dimensions are added.

5. CONCLUSIONS

In all of the comparisons between Tournament and Roulette Wheel selection in **Figures 2.3, 2.4 3.1**, **3.2**, and **3.8** respectively, I noticed that the *Tournament* always reached the optimum fitness faster than the *Roulette Wheel*, despite the same statistics. With the same statistics (**mutation rate of 0.03**, **mutation step of 0.9** and **100 generations**), the *Tournament selection* performs better and reaches its maximum fitness while the *Roulette wheel* does not. I also discovered that **0.03 MUTRATE** and **0.9 MUTSTEP** work for both selections; however, I believe that their performances may differ if the statistics are not set equally, which is not equitable to either selection method. In minimisation programs, the mutation rate remains constant at **0.03**, while the mutation step changes to **1.0** for the *Rosenbrock function* and **0.3** for the *Ackley function*.

Based on the results of the tests, I can conclude that a higher **MUTRATE** and **MUTSTEP** do not provide better results because either of these causes too dramatic changes on genes, resulting in dramatically increased or decreased fitness.

Furthermore, some genetic algorithm operators perform better in some situations and worse in others, such as roulette wheel selection, which performs better with fewer chromosomes within genes. If I had the chance to run the tests again, I would test a wider variety of ***GA*** operators and run similar comparisons between ***ABC*** and ***GA***, but test different ***ABC*** algorithms against the ***GA*** algorithm to see how they perform.

The mutation change for each minimisation program had no effect on the overall GAs, particularly the selection methods, because I compared *Tournament* and *Roulette Wheel selection* for each minimisation program. And, once again, the convergence of *Tournament* appears to be better than that of *Roulette Wheel* in all optimization functions, which could be explained by the fact that the *Roulette Wheel* selection is based on probability, so the fitter individuals have a better chance of being chosen, but there is a chance that the weaker individuals are chosen as well, so the algorithm cannot always optimize the highest fitness as it can, whereas the *Tournament selection* is capable of doing so because it always chooses the fittest individuals.

All of these facts lead me to the conclusion that ***Tournament selection*** is a better algorithm for working within GAs.

REFERENCES

- A. Shukla, H. M. Pandey and D. Mehrotra. (2015). “Comparative review of selection techniques in genetic algorithm,”

- Buontempo, F. (2019) Genetic algorithms and machine learning for programmers : create AI models and evolve solutions 1st edition. Raleigh, North Carolina, The Pragmatic Bookshelf.

- Goldberg, D.E. (David E. (1989) Genetic algorithms in search, optimization, and machine learning New Delhi, India, Pearson.

- GeeksforGeeks, Tournament selection. Available URL: https://www.geeksforgeeks.org/tournament-selection-ga/ [Accessed 10 Nov 2022].

- Iliadis, L., Angelov, P.P., Jayne, C. and Pimenidis, E. (2020) Proceedings of the 21st EANN (Engineering Applications of Neural Networks) 2020 Conference Proceedings of the EANN 2020 Iliadis, L., Angelov, P.P., Jayne, C. and Pimenidis, E. (eds.) 1st ed. 2020. [online]. Cham, Springer International Publishing.

- Karaboga, Basturk, D.K, B.B. (2007) A powerful and efficient algorithm for numerical function optimization: artificial bee colony (ABC) algorithm. *Journal of Global Optimization*[online]. Volume 39, Issue 3. [Accessed 17 Nov 2022]

- Karaboga, Gorkemli, Ozturk, D.K, G.B, C.O. (2014) A comprehensive survey: artificial bee colony (ABC) algorithm and applications. *Artificial Intelligence Review* [online]. Volume 42, Issue 3. [Accessed 12 Nov 2022]

- Jenna Carr. (2014). An Introduction to Genetic Algorithms.

- Jadaan, O.A., Rajamani, L., & Rao, C.R. (2008). IMPROVED SELECTION OPERATOR FOR GA 1.

- Razali, Noraini & Geraghty, John. (2011). Genetic Algorithm Performance with Different Selection Strategies in Solving TSP. 2.

- Ternent, J., & Thompson, M. (2020). Ethical Considerations of Artificial Intelligence via Neural Networks Applied to Medical Applications. Worcester Polytechnic Institute.

- Yarpiz (2015) Artificial Bee Colony (ABC) in MATLAB (version 1.0.0.0) [Source Code]. Available from: https://uk.mathworks.com/matlabcentral/fileexchange/52966-artificial-bee-colony-abc-in-matlab [Accessed 15 Nov 2022]

Appendix – Source code

Crossover

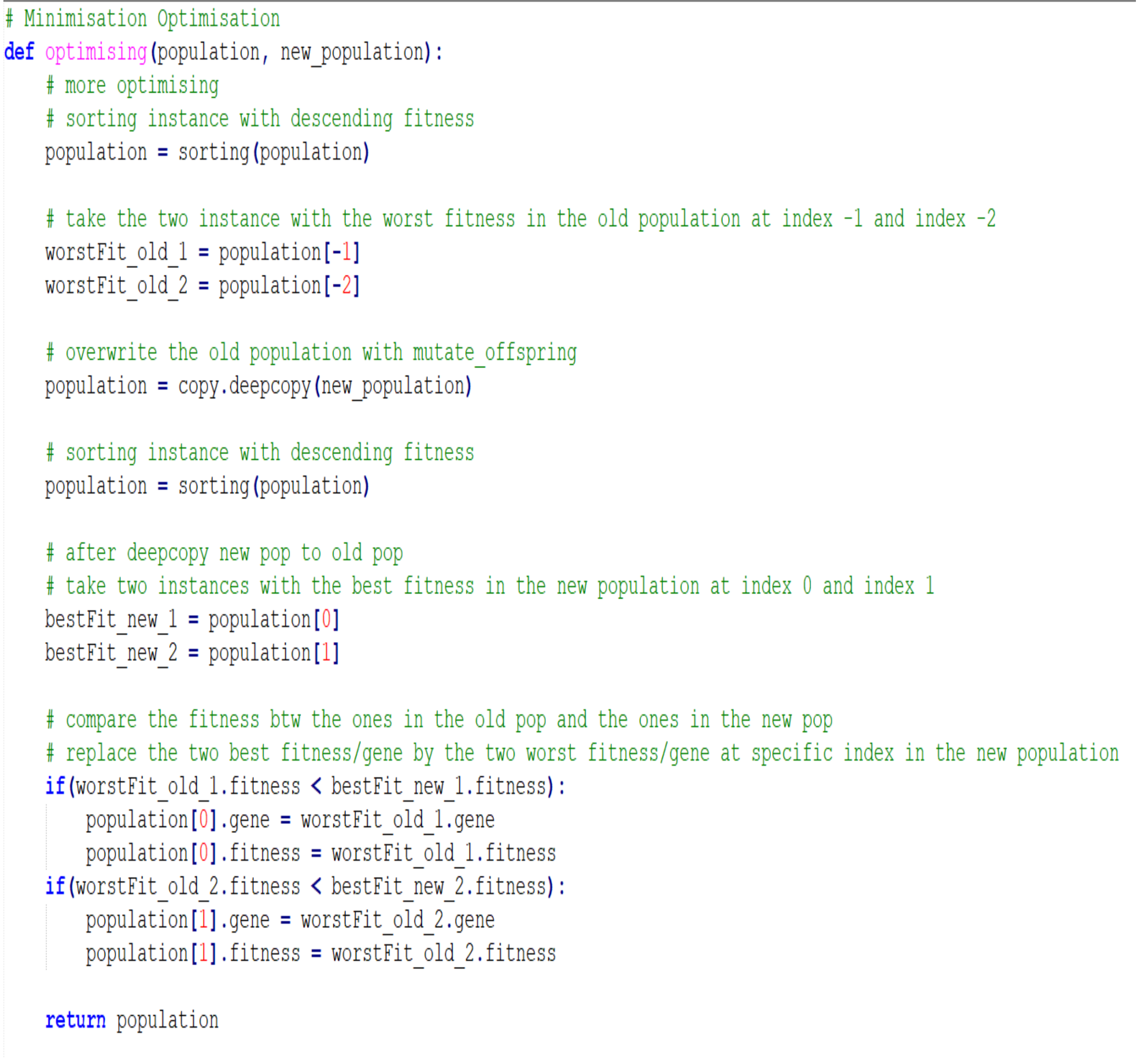
```
# Single-point Crossover process
def crossover(offspring):
    # Recombine pairs of parents from offspring
    crossover_offspring = []
    for i in range(0, P, 2):
        crosspoint = random.randint(0, N - 1) #pick up one random point in the gene length
        # 2 new temporary instances
        temp1 = individual()
        temp2 = individual()
        # 2 heads and 2 tails
        head1 = []
        tail1 = []
        head2 = []
        tail2 = []
        # print(offspring[i].gene, offspring[i+1].gene, crosspoint)
        # 0 to crosspoint adding gene to each head
        for j in range(0, crosspoint):
            head1.append(offspring[i].gene[j])
            head2.append(offspring[i + 1].gene[j])
        # crosspoint to N adding gene to each tail
        for j in range(crosspoint, N):
            tail2.append(offspring[i + 1].gene[j])
            tail1.append(offspring[i].gene[j])
        temp1.gene = head1 + tail2 # add first gene after crossover to temp1
        temp2.gene = head2 + tail1 # add second gene after crossover to temp2
        temp1.fitness = counting_ones(temp1) # call counting_ones to add fitness to temporary indv
        temp2.fitness = counting_ones(temp2)
        # append temp1, temp2 respectively to crosover_offspring_offspring
        crossover_offspring.append(temp1)
        crossover_offspring.append(temp2)

    return crossover_offspring
```

Elitism

- Maximisation

```
# Optimisation
def optimising(population, new_population):
    # more optimising
    # sorting instance with descending fitness
    population = sorting(population)

    # take two instances with the best fitness in the old population at index 0 and index 1
    bestFit_old_1 = population[0]
    bestFit_old_2 = population[1]

    # overwrite the old population with mutate_offspring
    population = copy.deepcopy(new_population)

    # sorting instance with descending fitness
    population = sorting(population)

    # after deepcopy new pop to old pop
    # take the two instance with the worst fitness in the new population at index -1 and index -2
    worstFit_new_1 = population[-1]
    worstFit_new_2 = population[-2]

    # compare the fitness btw the ones in the old pop and the ones in the new pop
    # replace the two worst fitness/gene by the two best fitness/gene at specific index in the new population
    if(bestFit_old_1.fitness > worstFit_new_1.fitness):
        population[-1].fitness = bestFit_old_1.fitness
        population[-1].gene = bestFit_old_1.gene
    if(bestFit_old_2.fitness > worstFit_new_2.fitness):
        population[-2].fitness = bestFit_old_2.fitness
        population[-2].gene = bestFit_old_2.gene

    return population
```

- Minimisation



```
# Minimisation Optimisation
def optimising(population, new_population):
    # more optimising
    # sorting instance with descending fitness
    population = sorting(population)

    # take the two instance with the worst fitness in the old population at index -1 and index -2
    worstFit_old_1 = population[-1]
    worstFit_old_2 = population[-2]

    # overwrite the old population with mutate_offspring
    population = copy.deepcopy(new_population)

    # sorting instance with descending fitness
    population = sorting(population)

    # after deepcopy new pop to old pop
    # take two instances with the best fitness in the new population at index 0 and index 1
    bestFit_new_1 = population[0]
    bestFit_new_2 = population[1]

    # compare the fitness btw the ones in the old pop and the ones in the new pop
    # replace the two best fitness/gene by the two worst fitness/gene at specific index in the new population
    if(worstFit_old_1.fitness < bestFit_new_1.fitness):
        population[0].gene = worstFit_old_1.gene
        population[0].fitness = worstFit_old_1.fitness
    if(worstFit_old_2.fitness < bestFit_new_2.fitness):
        population[1].gene = worstFit_old_2.gene
        population[1].fitness = worstFit_old_2.fitness

    return population
```

Maximisation Code – Counting One’s

```
import random
import copy
import matplotlib.pyplot as plt

# Each individual is represented by a data structure,
# consisting of an array of binary genes and a fitness value
class individual:
    gene = []
    fitness = 0

    def __repr__(self):
        return "Gene string " +  "".join(str(x) for x in self.gene) + " - 
fitness: " + str(self.fitness)

# The initial population array of such individuals,
# and random gene length number of 50 and population of 50
P = 50
N = 50
GENERATIONS = 100 # initialise 100 generations

# random mutation rate and mutation step
MUTRATE = 0.03
MUTSTEP = 0.9

# Calculate individual's fitness
# the individual's fitness is equal to the number of ‘1’s in its array of genes
(genome)
# instance parameter
def counting_ones(ind):
    fitness = 0
    for i in range(0, N):
        # if(ind.gene[i] == 1): # if gene of an individual at index i equals to
1
            fitness = fitness + ind.gene[i]
    return fitness

# Calculate population's fitness
# list parameter
def total_fitness(population):
    totalfit = 0
    for ind in population:
        totalfit += ind.fitness
    return totalfit

# Initialise original population
def initialise_population():
    population = []
    # Initialise population with random candidate solutions
    # Generate random genes and append to a temporary gene list
    # Assign fitness and gene to individual and append one to population
    for x in range(0, P):
        tempgene = []
        for x in range(0, N):
            tempgene.append(random.uniform(0.0, 1.0)) # a random gene between
0.0 and 1.0(inclusive)
        # print(tempgene)
        newindi = individual() # initialise new instance
        newindi.gene = tempgene.copy() # copy the gene from tempgene and assign
to gene of individual
```

```
        newindi.fitness = counting_ones(newindi) # initialise instance's fitness
        population.append(newindi)
    return population

# Tourament Selection Process
def touranment_selection(population):
    offspring = []
    # Select two parents and recombine pairs of parents
    for i in range(0, P):
        parent1 = random.randint(0, P - 1)
        off1 = population[parent1]
        parent2 = random.randint(0, P - 1)
        off2 = population[parent2]
        if (off1.fitness > off2.fitness): # if one's fitness higher then add to
temp offsptring
            offspring.append(off1)
        else:
            offspring.append(off2)

    return offspring

# Roulette Wheel Selection Process
def RW_selection(population):
    # total fitness of initial pop
    initial_fits = total_fitness(population)

    offspring = copy.deepcopy(population)
    # Roulette Wheel Selection Process
    # Select two parents and recombine pairs of parents
    for i in range(0, P):
        selection_point = random.uniform(0.0, initial_fits)
        running_total = 0
        j = 0
        while running_total <= selection_point:
            running_total += population[j].fitness
            j += 1
            if(j == P):
                break
        offspring[i] = population[j-1]

    return offspring

# Single-point Crossover process
def crossover(offspring):
    # Recombine pairs of parents from offspring
    crossover_offspring = []
    for i in range(0, P, 2):
        crosspoint = random.randint(0, N - 1) #pick up one random point in the
gene length
        # 2 new temporary instances
        temp1 = individual()
        temp2 = individual()
        # 2 heads and 2 tails
        head1 = []
        tail1 = []
        head2 = []
        tail2 = []
        # print(offspring[i].gene, offspring[i+1].gene, crosspoint)
        # 0 to crosspoint adding gene to each head
        for j in range(0, crosspoint):
            head1.append(offspring[i].gene[j])
            head2.append(offspring[i + 1].gene[j])
        # crosspoint to N adding gene to each tail
        for j in range(crosspoint, N):
            tail2.append(offspring[i + 1].gene[j])
            tail1.append(offspring[i].gene[j])
        # print("head1 + tail2")
        # print(head1, tail2)
        # print("head2 + tail1")
        # print(head2, tail1)
        temp1.gene = head1 + tail2 # add first gene after crossover to temp1
        temp2.gene = head2 + tail1 # add second gene after crossover to temp2
        temp1.fitness = counting_ones(temp1) # call counting_ones to add fitness
to temporary indv
        temp2.fitness = counting_ones(temp2)
        # append temp1, temp2 respectively to crosover_offspring_offspring
        crossover_offspring.append(temp1)
        crossover_offspring.append(temp2)
        # print(crosover_offspring_offspring[i].gene,
crosover_offspring_offspring[i+1].gene)

    return crossover_offspring

#Bit-wise Mutation
def mutation(crosover_offspring, MUTRATE, MUTSTEP):
    # Mutate the result of new_offspring
    mutate_offspring = []
    for i in range(0, P):
        new_indi = individual()
        new_indi.gene = []
        for j in range(0, N):
            gene = crosover_offspring[i].gene[j]
            ALTER = random.uniform(0.0, MUTSTEP)
            MUTPROB = random.uniform(0.0, 100.0)
            if (MUTPROB < (100*MUTRATE)):
                if(random.randint(0, 1) == 1): # if random num is 1, add ALTER
                    gene += ALTER
                else: # if random num is 0, minus ALTER
                    gene -= ALTER
                if(gene > 1.0): # if gene value is larger than 1.0, reset it to
1.0
                    gene = 1.0
                if(gene < 0.0): # if gene value is smaller than 0.0, reset it to
0.0
                    gene = 0.0
            new_indi.gene.append(gene) # add gene to instance
        new_indi.fitness = counting_ones(new_indi) # add fitness to instance by
calling counting_ones
        mutate_offspring.append(new_indi)

    return mutate_offspring

# Descending sorting
def sorting(population):
    #  descending sorting based on  individual's fitness
    population.sort(key=lambda individual:individual.fitness, reverse=True)

    return population

# Optimisation
def optimising(population, new_population):
    # more optimising
    # sorting instance with descending fitness
    population = sorting(population)

    # take two instances with the best fitness in the old population at index 0
and index 1
    bestFit_old_1 = population[0]
    bestFit_old_2 = population[1]

    # overwrite the old population with mutate_offspring
    population = copy.deepcopy(new_population)

    # sorting instance with descending fitness
    population = sorting(population)

    # after deepcopy new pop to old pop
    # take the two instance with the worst fitness in the new population at
index -1 and index -2
    worstFit_new_1 = population[-1]
    worstFit_new_2 = population[-2]

    # compare the fitness btw the ones in the old pop and the ones in the new
pop
    # replace the two worst fitness/gene by the two best fitness/gene at
specific index in the new population
    if(bestFit_old_1.fitness > worstFit_new_1.fitness):
        population[-1].fitness = bestFit_old_1.fitness
        population[-1].gene = bestFit_old_1.gene
    if(bestFit_old_2.fitness > worstFit_new_2.fitness):
        population[-2].fitness = bestFit_old_2.fitness
        population[-2].gene = bestFit_old_2.gene

    return population

def GA(population, Selection, MUTRATE, MUTSTEP):
    # storing data to plot
    meanFit_values = []
    maxFit_values = []
    # ===========GENETIC ALGORITHM===============

    for gen in range(0, GENERATIONS):
        # # touranment/ RW selection process
        offspring = Selection(population)

        # crossover process
        crossover_offspring = crossover(offspring)
        # mutation process
        mutate_offspring = mutation(crossover_offspring, MUTRATE, MUTSTEP)
        # optimising
        population = optimising(population, mutate_offspring)

        # calculate Max and Mean Fitness
        # storing fitness in a list
        Fit = []
        for ind in population:
            Fit.append(counting_ones(ind))
        # print(Fit)

        maxFit = max(Fit) # take out the max fitness among fitnesses in Fit
        meanFit = sum(Fit)/ P # sum all the fitness and divide by Population
size
```

```
        # append maxFit and meanFit respectively to MaxFit_values and
MeanFit_values
        maxFit_values.append(maxFit)
        meanFit_values.append(meanFit)

        # display
        # print("GENERATION " + str(gen + 1))
        # print("Mean Fitness: " + str(meanFit))
        # print("Max Fitness: " + str(maxFit) + "\n")
    print("Max Fitness: " + str(maxFit) + "\n")
    print("Mean Fitness: " + str(meanFit) + "\n")

    return maxFit_values, meanFit_values

# plotting
plt.ylabel("Fitness")
plt.xlabel("Number of Generation")

#  Storing
maxFit_data1 = []
maxFit_data2 = []
maxFit_data3 = []
maxFit_data4 = []

meanFit_data1 = []
meanFit_data2 = []
meanFit_data3 = []
meanFit_data4 = []


# EXPERIMENT

# ==============================================================
# TOURANMENT vs ROULETTE WHEEL SELECTION COMPARISON
# ==============================================================

# [----------------- UNCOMMENT THIS AND ALTER N TO TEST -----------------]
#N = 50
plt.title("Maximisation GA \n Tournament and Roulette Wheel Selection \n"
            + "N = " + str(N) + " MUTRATE = " + str(MUTRATE) + " MUTSTEP = " +
str(MUTSTEP))

# initialise original population
population = initialise_population()

maxFit_data1, meanFit_data1 = GA(population, touranment_selection, 0.03, 0.9)
maxFit_data2, meanFit_data2 = GA(population, RW_selection, 0.03, 0.9)

plt.plot(maxFit_data1, label="Touranment")
plt.plot(maxFit_data2, label="Roulette Wheel")
# [----------------- UNCOMMENT THIS AND ALTER N TO TEST -----------------]

# N = 50
# MUTRATE = 0.03
# MUTSTEP = 0.9
# Max Fitness: 50.0 - TS
# Max Fitness: 46.62542552330858 - RW



# N = 100
# MUTRATE = 0.03
# MUTSTEP = 0.9
# Max Fitness: 98.06259568588314 - TS
# Max Fitness: 82.66273008007921 - RW



# ============================================================
# TOURANMENT SELECTION
# ============================================================


 #Best Fitness and Mean Fitness of TS
 #[----------------- UNCOMMENT THIS TO TEST -----------------]
# plt.title("Maximisation GA - Touranment Selection \n"
#             + "N = " + str(N) + " MUTRATE = " + str(MUTRATE) + " MUTSTEP = " +
str(MUTSTEP))

# # initialise original population
# population = initialise_population()

# maxFit_data1, meanFit_data1 = GA(population, touranment_selection, 0.03, 0.9)

# plt.plot(maxFit_data1, label="Max Fitness")
# plt.plot(meanFit_data1, label="Mean Fitness")
# [----------------- UNCOMMENT THIS TO TEST -----------------]
# N = 50
# MUTRATE = 0.03
# MUTSTEP = 0.9
# Max Fitness: 50.0
# Mean Fitness: 49.467339709485785



# Vary MUTRATE
# [----------------- UNCOMMENT THIS TO TEST -----------------]
# plt.title("Maximisation GA - Touranment Selection \n"
#             + "Vary MUTRATE")

# # initialise original population
# population = initialise_population()

# maxFit_data1, meanFit_data1 = GA(population, touranment_selection, 0.3, 0.9)
# maxFit_data2, meanFit_data2 = GA(population, touranment_selection, 0.03, 0.9)
# maxFit_data3, meanFit_data3 = GA(population, touranment_selection, 0.003, 0.9)
# maxFit_data4, meanFit_data4 = GA(population, touranment_selection, 0.0003,
0.9)

# plt.plot(maxFit_data1, label="MUTRATE 0.3")
# plt.plot(maxFit_data2, label="MUTRATE 0.03")
# plt.plot(maxFit_data3, label="MUTRATE 0.003")
# plt.plot(maxFit_data4, label="MUTRATE 0.0003")
# [----------------- UNCOMMENT THIS TO TEST -----------------]
# N = 50
# MUTSTEP = 0.9
# Max Fitness: 43.471365866108314 - MUTRATE 0.3
# Max Fitness: 49.987334217326755 - MUTRATE 0.03
# Max Fitness: 48.887789618353786 - MUTRATE 0.003
# Max Fitness: 42.4534292474593   - MUTRATE 0.0003
# ==============================================================
# ROULETTE WHEEL SELECTION
# ==============================================================


# Best Fitness and Mean Fitness of RW
# [------------------ UNCOMMENT THIS TO TEST ------------------]
# GENERATIONS = 200
# plt.title("Maximisation GA - Roulette Wheel Selection \n"
#            + "N = " + str(N) + " MUTRATE = " + str(MUTRATE) + " MUTSTEP = " +
str(MUTSTEP))

# # initialise original population
# population = initialise_population()

# maxFit_data1, meanFit_data1 = GA(population, RW_selection, 0.03, 0.9)

# plt.plot(maxFit_data1, label="Max Fitness")
# plt.plot(meanFit_data1, label="Mean Fitness")
# [------------------ UNCOMMENT THIS TO TEST ------------------]
# N = 50
# MUTRATE = 0.03
# MUTSTEP = 0.9
# Max Fitness: 49.06776626442139
# Mean Fitness: 45.10946136458607


# Vary MUTRATE
# [------------------ UNCOMMENT THIS TO TEST ------------------]
# GENERATIONS = 200
# plt.title("Maximisation GA - Roulette Wheel Selection \n"
#             + "Vary MUTRATE")

# # initialise original population
# population = initialise_population()

# maxFit_data1, meanFit_data1 = GA(population, RW_selection, 0.3, 0.9)
# maxFit_data2, meanFit_data2 = GA(population, RW_selection, 0.03, 0.9)
# maxFit_data3, meanFit_data3 = GA(population, RW_selection, 0.003, 0.9)
# maxFit_data4, meanFit_data4 = GA(population, RW_selection, 0.0003, 0.9)

# plt.plot(maxFit_data1, label="MUTRATE 0.3")
# plt.plot(maxFit_data2, label="MUTRATE 0.03")
# plt.plot(maxFit_data3, label="MUTRATE 0.003")
# plt.plot(maxFit_data4, label="MUTRATE 0.0003")
# [------------------ UNCOMMENT THIS TO TEST ------------------]
# N = 50
# MUTSTEP = 0.9
# Max Fitness: 43.67294750948452 - MUTRATE 0.3
# Max Fitness: 49.18131827030025 - MUTRATE 0.03
# Max Fitness: 48.12886400473725 - MUTRATE 0.003
# Max Fitness: 40.591039436185326 - MUTRATE 0.0003

# DISPLAY PLOT
plt.legend(loc = "lower right")
plt.show()
```

Minimisation Code

1. Rosenbrock Function

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

# Each individual is represented by a data structure,
# consisting of an array of binary genes and a fitness value


class individual:
    gene = []
    fitness = 0

    def __repr__(self):
        return "Gene string " + "".join(str(x) for x in self.gene) + " - 
fitness: " + str(self.fitness)


# The initial population array of such individuals,
# and random gene length number of 10 and population of 50
P = 50
N = 10
GENERATIONS = 600  # initialise 600 generations

# random mutation rate and mutation step
MUTRATE = 0.03
MUTSTEP = 1.0

# Calculate individual's fitness


def mini_function(ind):
    fitness = 0
    squared = 0
    first_expression = (ind.gene[0] - 1) ** 2 # (x1-1)^2
    second_expression = 0
    for i in range(1, N):
        # exp = [2 * xi^2 - x(i-1)]^2
        squared = i * ((2*ind.gene[i]*ind.gene[i] - ind.gene[i-1]) ** 2)
        # i * exp
        second_expression += squared
    # (x1-1)^2 + SUM{ i* [2 * xi^2 - x(i-1)]^2 }
    fitness = first_expression + second_expression
    return fitness

# Calculate population's fitness
# list parameter


def total_fitness(population):
    totalfit = 0
    for ind in population:
        totalfit += ind.fitness
    return totalfit

# Initialise original population
```

```
def initialise_population():
    population = []
    # Initialise population with random candidate solutions
    # Generate random genes and append to a temporary gene list
    # Assign fitness and gene to individual and append one to population
    for x in range(0, P):
        tempgene = []
        for x in range(0, N):
            # a random gene between -10 and 10 (inclusive)
            tempgene.append(random.uniform(-10, 10))
        # print(tempgene)
        newindi = individual()  # initialise new instance
        # copy the gene from tempgene and assign to gene of individual
        newindi.gene = tempgene.copy()
        # initialise instance's fitness
        newindi.fitness = mini_function(newindi)
        population.append(newindi)
    return population

# Tourament Selection Process


def touranment_selection(population):
    offspring = []
    # Select two parents and recombine pairs of parents
    for i in range(0, P):
        parent1 = random.randint(0, P - 1)
        off1 = population[parent1]
        parent2 = random.randint(0, P - 1)
        off2 = population[parent2]
        # if one's fitness higher then add the smaller one to temp offsptring
        if (off1.fitness > off2.fitness):
            offspring.append(off2)
        else:
            offspring.append(off1)

    return offspring

# Roulette Wheel Selection Process


def RW_selection(population):
    # total fitness of initial pop
    total = 0
    for ind in population:
        total += 1/ind.fitness

    offspring = []
    # Roulette Wheel Selection Process
    # Select two parents and recombine pairs of parents
    for i in range(0, P):
        selection_point = random.uniform(0.0, total)
        running_total = 0
        j = 0
        while running_total <= selection_point:
            running_total += 1 / (population[j].fitness)
            j += 1
            if(j == P):
                break
        # print(running_total)
        # print(j)
        offspring.append(copy.deepcopy(population[j-1]))

    return offspring

# Single-point Crossover process


def crossover(offspring):
    # Recombine pairs of parents from offspring
    crossover_offspring = []
    for i in range(0, P, 2):
        # pick up one random point in the gene length
        crosspoint = random.randint(1, N - 1)
        # 2 new temporary instances
        temp1 = individual()
        temp2 = individual()
        # 2 heads and 2 tails
        head1 = []
        tail1 = []
        head2 = []
        tail2 = []
        # print(offspring[i].gene, offspring[i+1].gene, crosspoint)
        # 0 to crosspoint adding gene to each head
        for j in range(0, crosspoint):
            head1.append(offspring[i].gene[j])
            head2.append(offspring[i + 1].gene[j])
        # crosspoint to N adding gene to each tail
        for j in range(crosspoint, N):
            tail2.append(offspring[i + 1].gene[j])
            tail1.append(offspring[i].gene[j])
        # print("head1 + tail2")
        # print(head1, tail2)
        # print("head2 + tail1")
        # print(head2, tail1)
        temp1.gene = head1 + tail2  # add first gene after crossover to temp1
        temp2.gene = head2 + tail1  # add second gene after crossover to temp2
        # call counting_ones to add fitness to temporary indv
        temp1.fitness = mini_function(temp1)
        temp2.fitness = mini_function(temp2)
        # append temp1, temp2 respectively to crosover_offspring_offspring
        crossover_offspring.append(temp1)
        crossover_offspring.append(temp2)
        # print(crosover_offspring_offspring[i].gene,
crosover_offspring_offspring[i+1].gene)

    return crossover_offspring

# Bit-wise Mutation


def mutation(crossover_offspring, MUTRATE, MUTSTEP):
    # Mutate the result of new_offspring
    # Bit-wise Mutation
    mutate_offspring = []
    for i in range(0, P):
        new_indi = individual()
        new_indi.gene = []
        for j in range(0, N):
            gene = crossover_offspring[i].gene[j]
            ALTER = random.uniform(0.0, MUTSTEP)
            MUTPROB = random.uniform(0.0, 100.0)
            if (MUTPROB < (100*MUTRATE)):
                if(random.randint(0, 1) == 1):  # if random num is 1, add ALTER
                    gene += ALTER
                else:  # if random num is 0, minus ALTER
                    gene -= ALTER
                if(gene > 5.12):  # if gene value is larger than 5.12, reset it
to 5.12
                    gene = 5.12
                if(gene < -5.12):  # if gene value is smaller than -5.12, reset
it to -5.12
                    gene = -5.12
            new_indi.gene.append(gene)
        # add fitness to instance by calling mini_function
        new_indi.fitness = mini_function(new_indi)
        mutate_offspring.append(new_indi)

    return mutate_offspring

# Descending sorting


def sorting(population):
    #  descending sorting based on individual's fitness
    population.sort(key=lambda individual: individual.fitness, reverse=True)

    return population

# Minimisation Optimisation


def optimising(population, new_population):
    # more optimising
    # sorting instance with descending fitness
    population = sorting(population)

    # take the two instance with the worst fitness in the old population at
index -1 and index -2
    worstFit_old_1 = population[-1]
    worstFit_old_2 = population[-2]

    # overwrite the old population with mutate_offspring
    population = copy.deepcopy(new_population)

    # sorting instance with descending fitness
    population = sorting(population)

    # after deepcopy new pop to old pop
    # take two instances with the best fitness in the new population at index 0
and index 1
    bestFit_new_1 = population[0]
    bestFit_new_2 = population[1]

    # compare the fitness btw the ones in the old pop and the ones in the new
pop
    # replace the two best fitness/gene by the two worst fitness/gene at
specific index in the new population
    if(worstFit_old_1.fitness < bestFit_new_1.fitness):
        population[0].gene = worstFit_old_1.gene
        population[0].fitness = worstFit_old_1.fitness
    if(worstFit_old_2.fitness < bestFit_new_2.fitness):
        population[1].gene = worstFit_old_2.gene
        population[1].fitness = worstFit_old_2.fitness

    return population


def GA(population, Selection, MUTRATE, MUTSTEP):
    # ===========GENETIC ALGORITHM===============
    # storing data to plot
    meanFit_values = []
    minFit_values = []

    for gen in range(0, GENERATIONS):
        # touranment selection process / RW selection process
        offspring = Selection(population)
        # crossover process
        crossover_offspring = crossover(offspring)
        # mutation process
        mutate_offspring = mutation(crossover_offspring, MUTRATE, MUTSTEP)
        # optimising
        population = optimising(population, mutate_offspring)

        # calculate Max and Mean Fitness
        # storing fitness in a list
        Fit = []
        for ind in population:
            Fit.append(mini_function(ind))
        # print(Fit)

        minFit = min(Fit)  # take out the min fitness among fitnesses in Fit
        meanFit = sum(Fit) / P  # sum all the fitness and divide by P size

        # append minFit and meanFit respectively to MinFit_values and
MeanFit_values
        minFit_values.append(minFit)
        meanFit_values.append(meanFit)

        # display
        # print("GENERATION " + str(gen + 1))
    print("Min Fitness: " + str(minFit) + "\n")
    print("Mean Fitness: " + str(meanFit) + "\n")

    return minFit_values, meanFit_values


# plotting
plt.ylabel("Fitness")
plt.xlabel("Number of Generation")

# Storing
minFit_data1 = []
minFit_data2 = []
minFit_data3 = []
minFit_data4 = []

meanFit_data1 = []
meanFit_data2 = []
meanFit_data3 = []
meanFit_data4 = []
```

```
# EXPERIMENT

# =============================================================
# TOURANMENT vs ROULETTE WHEEL SELECTION COMPARISON
# =============================================================

# [----------------- UNCOMMENT THIS AND ALTER N TO TEST -----------------]
# N = 10
# GENERATIONS = 600
plt.title("Minimisation GA \n Tournament and Roulette Wheel Selection \n"
        + "N = " + str(N) + " MUTRATE = " + str(MUTRATE) + " MUTSTEP = " +
str(MUTSTEP))

# initialise original population
population = initialise_population()

minFit_data1, meanFit_data1 = GA(population, touranment_selection, 0.03, 1.0)
minFit_data2, meanFit_data2 = GA(population, RW_selection, 0.03, 1.0)

plt.plot(minFit_data1, label="Tournament")
plt.plot(minFit_data2, label="Roulette Wheel")
# [----------------- UNCOMMENT THIS AND ALTER N TO TEST -----------------]

# N = 10
# GENERATIONS = 600
# MUTRATE = 0.03
# MUTSTEP = 1.0
# Min Fitness: 0.07047215690374192 - TS
# Min Fitness: 0.0040744373402650386 - RW


# N = 10
# GENERATIONS = 2000
# MUTRATE = 0.03
# MUTSTEP = 1.0
# Min Fitness: 0.5000076286981088 - TS
# Min Fitness: 0.5000168729605512 - RW

# N = 50
# GENERATIONS = 4000
# MUTRATE = 0.03
# MUTSTEP = 1.0
# Min Fitness: 8.730959720472542 - TS
# Min Fitness: 13.398260123920641 - RW


# =============================================================
# TOURANMENT SELECTION
# =============================================================


# Best Fitness and Mean Fitness of TS
# [----------------- UNCOMMENT THIS TO TEST -----------------]
# plt.title("Minimisation GA - Tournament Selection \n"
#              + "N = " + str(N) + " MUTRATE = " + str(MUTRATE) + " MUTSTEP = " +
str(MUTSTEP))

# # initialise original population
# population = initialise_population()
# minFit_data1, meanFit_data1 = GA(population, touranment_selection, 0.03, 1.0)

# plt.plot(minFit_data1, label="Min Fitness")
# plt.plot(meanFit_data1, label="Mean Fitness")

# [------------------ UNCOMMENT THIS TO TEST ------------------]
# N = 10
# MUTRATE = 0.03
# MUTSTEP = 1.0
# Min Fitness: 0.011527590079116795
# Mean Fitness: 0.7425531247516733


# Vary MUTRATE
# [------------------ UNCOMMENT THIS TO TEST ------------------]
# plt.title("Minimisation GA - Tournament Selection \n"
#             + "Vary MUTRATE")

# # initialise original population
# population = initialise_population()

# minFit_data1, meanFit_data1 = GA(population, touranment_selection, 0.3, 1.0)
# minFit_data2, meanFit_data2 = GA(population, touranment_selection, 0.03, 1.0)
# minFit_data3, meanFit_data3 = GA(population, touranment_selection, 0.003, 1.0)
# minFit_data4, meanFit_data4 = GA(population, touranment_selection, 0.0003,
1.0)

# plt.plot(minFit_data1, label="MUTRATE 0.3")
# plt.plot(minFit_data2, label="MUTRATE 0.03")
# plt.plot(minFit_data3, label="MUTRATE 0.003")
# plt.plot(minFit_data4, label="MUTRATE 0.0003")
# [------------------ UNCOMMENT THIS TO TEST ------------------]
# N = 10
# MUTSTEP = 1.0
# Min Fitness: 1.2020694193533954   - MUTRATE 0.3
# Min Fitness: 0.505488373883888 - MUTRATE 0.03
# Min Fitness: 1.9845575126780761   - MUTRATE 0.003
# Min Fitness: 9.81276143781134   - MUTRATE 0.0003


# ============================================================
# ROULETTE WHEEL SELECTION
# ============================================================


# Best Fitness and Mean Fitness of RW
# [------------------ UNCOMMENT THIS TO TEST ------------------]
# plt.title("Minimisation GA - Roulette Wheel Selection \n"
#             + "N = " + str(N) + " MUTRATE = " + str(MUTRATE) + " MUTSTEP = "
+ str(MUTSTEP))

# # initialise original population
# population = initialise_population()

# minFit_data1, meanFit_data1 = GA(population, RW_selection, 0.03, 1.0)

# plt.plot(minFit_data1, label="Min Fitness")
# plt.plot(meanFit_data1, label="Mean Fitness")
# [------------------ UNCOMMENT THIS TO TEST ------------------]
# N = 10
# MUTRATE = 0.03
# MUTSTEP = 1.0
# Min Fitness: 0.03372919809378878
# Mean Fitness: 0.5167875596054593


# Vary MUTRATE
# [------------------ UNCOMMENT THIS TO TEST ------------------]
# plt.title("Minimisation GA - Roulette Wheel Selection \n"
#             + "Vary MUTRATE")

# # initialise original population
# population = initialise_population()

# minFit_data1, meanFit_data1 = GA(population, RW_selection, 0.3, 1.0)
# minFit_data2, meanFit_data2 = GA(population, RW_selection, 0.03, 1.0)
# minFit_data3, meanFit_data3 = GA(population, RW_selection, 0.003, 1.0)
# minFit_data4, meanFit_data4 = GA(population, RW_selection, 0.0003, 1.0)

# plt.plot(minFit_data1, label="MUTRATE 0.3")
# plt.plot(minFit_data2, label="MUTRATE 0.03")
# plt.plot(minFit_data3, label="MUTRATE 0.003")
# plt.plot(minFit_data4, label="MUTRATE 0.0003")
# [------------------ UNCOMMENT THIS TO TEST ------------------]
# N = 10
# MUTSTEP = 1.0
# Min Fitness: 0.5512263916570717 -  MUTRATE 0.3
# Min Fitness: 0.6362177568173231  - MUTRATE 0.03
# Min Fitness: 0.8762496808258389 - MUTRATE 0.003
# Min Fitness: 268.515837998209 - MUTRATE 0.0003


# DISPLAY PLOT
plt.legend(loc="upper right")
plt.show()
```

2. Ackley Function

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

# Each individual is represented by a data structure,
# consisting of an array of binary genes and a fitness value


class individual:
    gene = []
    fitness = 0

    def __repr__(self):
        return "Gene string " + "".join(str(x) for x in self.gene) + " - 
fitness: " + str(self.fitness)


# The initial population array of such individuals,
# and random gene length number of 10 and population of 50
P = 50
```

```python
N = 10
GENERATIONS = 500  # initialise 500 generations
# random mutation rate and mutation step
MUTRATE = 0.03
MUTSTEP = 1.0

# Calculate individual's fitness
# first minimisation function


def mini_function(ind):
    fitness = 0
    first_exp = 0
    second_exp = 0
    for i in range(0, N):
        first_exp += math.cos(2 * math.pi * ind.gene[i])
        second_exp += ind.gene[i] * ind.gene[i]
    first_exp = math.exp((1/N) * first_exp)
    second_exp = (-20) * math.exp((-0.2) * math.sqrt((1/N) * second_exp))
    fitness = second_exp - first_exp
    return fitness

# Calculate population's fitness
# list parameter


def total_fitness(population):
    totalfit = 0
    for ind in population:
        totalfit += ind.fitness
    return totalfit

# Initialise original population


def initialise_population():
    population = []
    # Initialise population with random candidate solutions
    # Generate random genes and append to a temporary gene list
    # Assign fitness and gene to individual and append one to population
    for x in range(0, P):
        tempgene = []
        for x in range(0, N):
            # a random gene between -5.12 and 5.12(inclusive)
            tempgene.append(random.uniform(-32.0, 32.0))
        # print(tempgene)
        newindi = individual()  # initialise new instance
        # copy the gene from tempgene and assign to gene of individual
        newindi.gene = tempgene.copy()
        # initialise instance's fitness
        newindi.fitness = mini_function(newindi)
        population.append(newindi)

    return population

# Tourament Selection Process


def tourament_selection(population):
    offspring = []
    # Select two parents and recombine pairs of parents
    for i in range(0, P):
        parent1 = random.randint(0, P - 1)
        off1 = population[parent1]
        parent2 = random.randint(0, P - 1)
        off2 = population[parent2]
        # if one's fitness higher then add the smaller one to temp offsptring
        if (off1.fitness > off2.fitness):
            offspring.append(off2)
        else:
            offspring.append(off1)

    return offspring

# Roulette Wheel Selection Process


def RW_selection(population):
    # total fitness of initial pop
    total = 0
    for ind in population:
        total += abs(ind.fitness)

    offspring = []
    # Roulette Wheel Selection Process
    # Select two parents and recombine pairs of parents
    for i in range(0, P):
        selection_point = random.uniform(0.0, total)
        running_total = 0
        j = 0
        while running_total <= selection_point:
            running_total += abs(population[j].fitness)
            j += 1
            if(j == P):
                break

        # print(running_total)
        # print(j)
        offspring.append(copy.deepcopy(population[j-1]))

    return offspring

# Single-point Crossover process


def crossover(offspring):
    # Recombine pairs of parents from offspring
    crossover_offspring = []
    for i in range(0, P, 2):
        # pick up one random point in the gene length
        crosspoint = random.randint(1, N - 1)
        # 2 new temporary instances
        temp1 = individual()
        temp2 = individual()
        # 2 heads and 2 tails
        head1 = []
        tail1 = []
        head2 = []
        tail2 = []
        # print(offspring[i].gene, offspring[i+1].gene, crosspoint)
        # 0 to crosspoint adding gene to each head
        for j in range(0, crosspoint):
            head1.append(offspring[i].gene[j])
            head2.append(offspring[i + 1].gene[j])
        # crosspoint to N adding gene to each tail
        for j in range(crosspoint, N):
            tail2.append(offspring[i + 1].gene[j])
            tail1.append(offspring[i].gene[j])
        # print("head1 + tail2")
        # print(head1, tail2)
        # print("head2 + tail1")
        # print(head2, tail1)
        temp1.gene = head1 + tail2  # add first gene after crossover to temp1
        temp2.gene = head2 + tail1  # add second gene after crossover to temp2
        # call counting_ones to add fitness to temporary indv
        temp1.fitness = mini_function(temp1)
        temp2.fitness = mini_function(temp2)
        # append temp1, temp2 respectively to crosover_offspring_offspring
        crossover_offspring.append(temp1)
        crossover_offspring.append(temp2)
        # print(crosover_offspring_offspring[i].gene,
crosover_offspring_offspring[i+1].gene)

    return crossover_offspring

# Bit-wise Mutation


def mutation(crossover_offspring, MUTRATE, MUTSTEP):
    # Mutate the result of new_offspring
    # Bit-wise Mutation
    mutate_offspring = []

    for i in range(0, P):
        new_indi = individual()
        new_indi.gene = []
        for j in range(0, N):
            gene = crossover_offspring[i].gene[j]
            ALTER = random.uniform(0.0, MUTSTEP)
            MUTPROB = random.uniform(0.0, 100.0)
            if (MUTPROB < (100*MUTRATE)):
                if(random.randint(0, 1) == 1):  # if random num is 1, add ALTER
                    gene += ALTER
                else:  # if random num is 0, minus ALTER
                    gene -= ALTER
                if(gene > 32.0):  # if gene value is larger than 32.0, reset it
to 32.0
                    gene = 32.0
                if(gene < -32.0):  # if gene value is smaller than -32.0, reset
it to -32.0
                    gene = -32.0
            new_indi.gene.append(gene)
        # add fitness to instance by calling mini_function
        new_indi.fitness = mini_function(new_indi)
        mutate_offspring.append(new_indi)

    return mutate_offspring

# Descending sorting


def sorting(population):
    #  descending sorting based on individual's fitness
    population.sort(key=lambda individual: individual.fitness, reverse=True)

    return population

# Minimisation Optimisation


def optimising(population, new_population):
    # more optimising
    # sorting instance with descending fitness
    population = sorting(population)

    # take the two instance with the worst fitness in the old population at 
index -1 and index -2
    worstFit_old_1 = population[-1]
    worstFit_old_2 = population[-2]

    # overwrite the old population with mutate_offspring
    population = copy.deepcopy(new_population)

    # sorting instance with descending fitness
    population = sorting(population)

    # after deepcopy new pop to old pop
    # take two instances with the best fitness in the new population at index 0 
and index 1
    bestFit_new_1 = population[0]
    bestFit_new_2 = population[1]

    # compare the fitness btw the ones in the old pop and the ones in the new 
pop
    # replace the two best fitness/gene by the two worst fitness/gene at 
specific index in the new population
    if(worstFit_old_1.fitness < bestFit_new_1.fitness):
        population[0].gene = worstFit_old_1.gene
        population[0].fitness = worstFit_old_1.fitness
    if(worstFit_old_2.fitness < bestFit_new_2.fitness):
        population[1].gene = worstFit_old_2.gene
        population[1].fitness = worstFit_old_2.fitness

    return population


def GA(population, Selection, MUTRATE, MUTSTEP):
    # ===========GENETIC ALGORITHM===============
    # storing data to plot
    meanFit_values = []
    minFit_values = []

    for gen in range(0, GENERATIONS):
        # touranment selection process
        offspring = Selection(population)
        # crossover process
        crossover_offspring = crossover(offspring)
        # mutation process
        mutate_offspring = mutation(crossover_offspring, MUTRATE, MUTSTEP)
        # optimising
        population = optimising(population, mutate_offspring)

        # calculate Max and Mean Fitness
        # storing fitness in a list
        Fit = []
        for ind in population:
            Fit.append(mini_function(ind))
        # print(Fit)

        minFit = min(Fit)  # take out the min fitness among fitnesses in Fit
        meanFit = sum(Fit) / P  # sum all the fitness and divide by P size

        # append minFit and meanFit respectively to MinFit_values and
MeanFit_values
        minFit_values.append(minFit)
        meanFit_values.append(meanFit)

        # # display
        # print("GENERATION " + str(gen + 1))
    print("Min Fitness: " + str(minFit) + "\n")
    print("Mean Fitness: " + str(meanFit) + "\n")

    return minFit_values, meanFit_values


# plotting
plt.ylabel("Fitness")
plt.xlabel("Number of Generation")

# Storing
minFit_data1 = []
minFit_data2 = []
minFit_data3 = []
minFit_data4 = []

meanFit_data1 = []
meanFit_data2 = []
meanFit_data3 = []
meanFit_data4 = []


# EXPERIMENT

# ==============================================================
# TOURANMENT vs ROULETTE WHEEL SELECTION COMPARISON
# ==============================================================

# [----------------- UNCOMMENT THIS AND ALTER N TO TEST -----------------]
N = 10
GENERATIONS = 500
plt.title("Minimisation GA \n Tournament and Roulette Wheel Selection \n"
          + "N = " + str(N) + " MUTRATE = " + str(MUTRATE) + " MUTSTEP = " +
str(MUTSTEP))

# initialise original population
population = initialise_population()

minFit_data1, meanFit_data1 = GA(population, touranment_selection, 0.03, 1.0)
minFit_data2, meanFit_data2 = GA(population, RW_selection, 0.03, 1.0)

plt.plot(minFit_data1, label="Tournament")
plt.plot(minFit_data2, label="Roulette Wheel")

# [----------------- UNCOMMENT THIS AND ALTER N TO FOR TEST PURPOSES------------
-----]
```

```
# N = 10
# MUTRATE = 0.03
# MUTSTEP = 1.0
# GENERATIONS = 500
# Min Fitness: -22.71253771450701 - TS
# Min Fitness: -22.518396025009025 - RW


# N = 20
# MUTRATE = 0.03
# MUTSTEP = 1.0
# GENERATIONS = 2000
# Min Fitness: -22.711649728948863 - TS
# Min Fitness: -22.615465913968446 - RW


# ==============================================================
# TOURANMENT SELECTION
# ==============================================================


# Best Fitness and Mean Fitness of TS
# [------------------ UNCOMMENT THIS TO TEST ------------------]
# plt.title("Minimisation GA - Tournament Selection \n"
#           + "N = " + str(N) + " MUTRATE = " + str(MUTRATE) + " MUTSTEP = " +
str(MUTSTEP))

# # initialise original population
# population = initialise_population()

# minFit_data1, meanFit_data1 = GA(population, touranment_selection, 0.03, 1.0)

# plt.plot(minFit_data1, label="Min Fitness")
# plt.plot(meanFit_data1, label="Mean Fitness")
# [------------------ UNCOMMENT THIS TO TEST ------------------]
# N = 10
# MUTRATE = 0.03
# MUTSTEP = 1.0
# Min Fitness: -22.698435758025173
# Mean Fitness: -22.425626443811986


# Vary MUTRATE
# [------------------ UNCOMMENT THIS TO TEST ------------------]
# plt.title("Minimisation GA - Tournament Selection \n"
#           + "Vary MUTRATE")

# # initialise original population
# population = initialise_population()

# minFit_data1, meanFit_data1 = GA(population, touranment_selection, 0.3, 1.0)
# minFit_data2, meanFit_data2 = GA(population, touranment_selection, 0.03, 1.0)
# minFit_data3, meanFit_data3 = GA(population, touranment_selection, 0.003, 1.0)
# minFit_data4, meanFit_data4 = GA(population, touranment_selection, 0.0003,
1.0)

# plt.plot(minFit_data1, label="MUTRATE 0.3")
# plt.plot(minFit_data2, label="MUTRATE 0.03")
# plt.plot(minFit_data3, label="MUTRATE 0.003")
# plt.plot(minFit_data4, label="MUTRATE 0.0003")
# [------------------ UNCOMMENT THIS TO TEST ------------------]
# N = 10
# MUTSTEP = 1.0
# Min Fitness: -22.30083345122437 - MUTRATE 0.3
# Min Fitness: -22.69556884202363 - MUTRATE 0.03
# Min Fitness: -20.207411550234102 - MUTRATE 0.003
# Min Fitness: -11.77237542572478 - MUTRATE 0.0003


# ===============================================================
# ROULETTE WHEEL SELECTION
# ===============================================================


# Best Fitness and Mean Fitness of RW
# [------------------ UNCOMMENT THIS TO TEST ------------------]
# plt.title("Minimisation GA - Roulette Wheel Selection \n"
#             + "N = " + str(N) + " MUTRATE = " + str(MUTRATE) + " MUTSTEP = " +
str(MUTSTEP))

# # initialise original population
# population = initialise_population()

# minFit_data1, meanFit_data1 = GA(population, RW_selection, 0.03, 1.0)

# plt.plot(minFit_data1, label="Min Fitness")
# plt.plot(meanFit_data1, label="Mean Fitness")
# [------------------ UNCOMMENT THIS TO TEST ------------------]
# N = 10
# MUTRATE = 0.03
# MUTSTEP = 1.0
# Min Fitness: -22.65638596126548
# Mean Fitness: -21.481933208749926


# Vary MUTRATE
# [------------------ UNCOMMENT THIS TO TEST ------------------]
# plt.title("Minimisation GA - Roulette Wheel Selection \n"
#            + "Vary MUTRATE")
# # initialise original population
# population = initialise_population()

# minFit_data1, meanFit_data1 = GA(population, RW_selection, 0.3, 1.0)
# minFit_data2, meanFit_data2 = GA(population, RW_selection, 0.03, 1.0)
# minFit_data3, meanFit_data3 = GA(population, RW_selection, 0.003, 1.0)
# minFit_data4, meanFit_data4 = GA(population, RW_selection, 0.0003, 1.0)

# plt.plot(minFit_data1, label="MUTRATE 0.3")
# plt.plot(minFit_data2, label="MUTRATE 0.03")
# plt.plot(minFit_data3, label="MUTRATE 0.003")
# plt.plot(minFit_data4, label="MUTRATE 0.0003")
# [------------------ UNCOMMENT THIS TO TEST ------------------]
# N = 10
# MUTSTEP = 0.3
# Min Fitness: -22.050692254005554 - MUTRATE 0.3
# Min Fitness: -22.49680201310099  - MUTRATE 0.03
# Min Fitness: -18.045801695277014 - MUTRATE 0.003
# Min Fitness: -10.085628544590312 - MUTRATE 0.0003


# DISPLAY PLOT
```

```
plt.legend(loc="upper right")
plt.show()
```