Biocomputation - Optimisation

Vince Verdadero 19009246.

1 INTRODUCTION

I will be doing the optimisation route for this assignment, which comprises two minimisation functions. This is also a continuation of Worksheet 3 and depicts the development of a Genetic Algorithm that uses a maximisation function like the Counting One's function.

The two minimisation functions are:

$$f(\boldsymbol{x}) = \sum_{i=1}^{n-1} \left[100\left(x_{i+1} - x_i^2\right)^2 + \left(1 - x_i\right)^2 \right]$$

where $-100 \le x \le 100$, start with n=20

Figure 1 - Rosenbrock function, Image is from the optimisation assignment sheet.

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

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

Figure 2 - Ackley function, Image is from the optimisation assignment sheet.

I will try to figure out which operator outperforms competitive performance throughout this project. All of this comes from the initial algorithm and by creating new operators. This will be applied to the two minimisation functions and apply a GA on the maximisation function too.

Furthermore, I will compare Tournament Selection to the Roulette Wheel that I will be implementing. In addition, I will run it and produce 10 different tests on each optimisation to conclude which is better for performance. This will also be reported in the Experimentation part.

Moreover, I will research Genetic Algorithms, common selections in GA, elitism, and nature-inspired optimisation algorithm. Also, discover what occurs by experimenting with my minimisation and maximisation functions.

Throughout this assignment, I aim to obtain a better grasp of how Genetic Algorithms work and learn which performance is better.

2 BACKGROUND RESEARCH

2.1 What is a Genetic Algorithm?

A Genetic Algorithm (GA), as described by (Raynor, 2021), is one of the oldest and most successful optimisation approaches based on the Nature of Evolution. It was first introduced by John Holland to explore the process of evolution and adaptation occurring in nature. It was influenced by Charles Darwin too.

The procedure of GA is conducted in phases. The order will start with the original population and then the 2nd step is determining the fitness. Followed by the Selection. The recombination phase also consists of crossover. Afterwards, mutation and finally will end in termination. These are described from (Mallawaarachchi, 2017) and further explained below:

1. **INITIAL POPULATION:** This begins a set of individuals referred to as a Population or original population. An individual is characterised by several genes. A Chromosome is made up of genes connected and encoded like a string and is represented in binary values.
2. **FITNESS FUNCTION:** This will determine how the fitness of an individual is when it is competing against others. Individuals will also be chosen for reproduction depending on their fitness overall.
3. **SELECTION:** In GAs, it chooses two random parents for reproduction based on their fitness scores. The selection procedure will vary on the optimisation function. For instance, the individuals with low fitness will go the minimisation. However, if it's with high fitness it will go to maximisation. These are all decided by the fitness to be selected for reproduction.
4. **RECOMBINATION:** This involves the crossover process, which contains a cross point and is selected at random to determine all sets of parents to be paired. This is also to build a better new individual's representation of a gene from its parent's representation, and this will create offspring.
5. **MUTATION:** Mutation represents a change in the gene by bit flipping and runs in the background. This is always to ensure that the search algorithm does not become stuck within a set of applicant results. This is all established after the crossover from the offspring population.
6. **TERMINATION:** This occurs that the population has stabled to a point, therefore, it has ended and shows that the Genetic Algorithm has been completed.

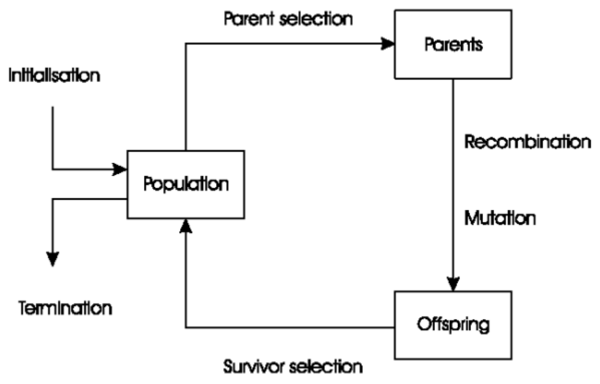


Figure 3 – diagram to show the flow of how the GA works– image is from week 1 Biocomputation, source by Bull., L (2021)

2.2 Selections in GA

The two commonly used selections in the GA are Roulette Wheel (RW) and Tournament and this will be detailed further below.

2.2.1 Roulette Wheel

The roulette wheel is divided into many individuals in the population in a pie according to from (TutorialPoint, no date). This is all determined by their fitness level. The wheel is spun when a fixed point on the pie is picked. As the parent region, it will pick the part of the wheel that is closest to the fixed point and cycle through again for the second parent stated from (TutorialPoint, no date).

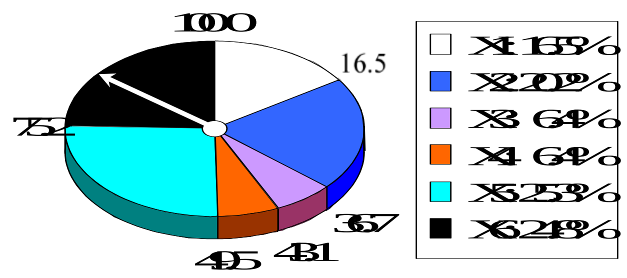
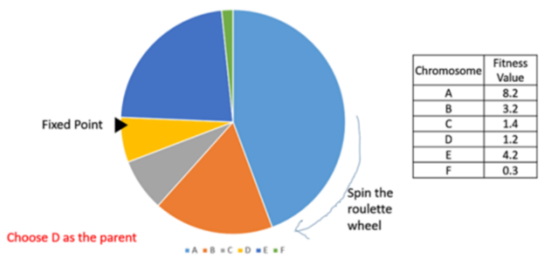


Figure 4 - Image is from the IS lecture slides

Figure 4, for example, shows there are six segments in the pie which represent 6 chromosomes. The roulette wheel would be spun six times to maintain the same population in the next generation.



Chromosome	Fitness Value
A	8.2
B	3.2
C	1.4
D	1.2
E	4.2
F	0.3

Figure 5 - Roulette Wheel image from TutorialPoint

To explain, even more, consider the following example from (TutorialPoint, no date), the bigger the pie of the individual fitness, there will be a high chance of it getting picked when it gets spun. As a result, the likelihood of selecting an individual varies to its fitness.

According to (TutorialPoint, no date), the subsequent stages of RW is:

1. The sum of fitness is the same as calculating 'S'.

2. Random digit will need to be selected among 0 – S.

3. Add the finesses to the partial sum P, starting at the top of the population, until P is smaller than S.

4. When P is greater than S, it will be used as the individual's preferred choice.

2.2.2 Tournament

(GeeksforGeeks, 2018) describes tournament selection as a method of selecting individuals from a population. By organising numerous "Tournaments" across a select group of chromosomes are occurred in an irregular pattern from the population. The following generation will then be picked from the best fitness individual from the current generation cycle. They also provide a step-by-step explanation of how the Tournament algorithm works listed below:

1. For this stage, the population and tournament between them are executed once they select k individuals.
2. The population will determine when they select the size
3. The likelihood of using a chance of p, they will pick the top individual.
4. Using Step 3, this will carry on for p*(1-p) and this will determine the 2nd top individual.
5. This will be repeated until they stop and reach the quantity of population they want.

In terms of my coding for the tournament selection, I am initialising k as 2 and I will contrast from their fitness to see which is top when I put it on the offspring. Therefore, I will choose two random parents when it loops through for each of the Number of generations.

2.3 How to make the optimisation efficient?

Elitism can help me with this assignment by making this programme more efficient for my optimisation functions. Exchanging the worst chromosomes in the child population with the best members of the parent population is what Elitism is according to (Du et al., 2018) Also, (Du et al., 2018) states that increasing GA convergence speed by preserving the best solution discovered in each iteration is used throughout Elitism. For instance, for my minimisation function, I have swapped the best from the new population to the worst, therefore concluding the best fitness will be the worst (minimum) fitness. As well it will work inversely for the maximisation function too at specific indices.

2.4 Nature-inspired bat algorithm

According to (Zebari et al., 2020), many nature-inspired optimisation algorithms are typically based on swarm intelligence. All these algorithms have different characteristics. Some examples include bee colonies, bat swarm, ant colonies, cuckoo search algorithm, etc.

One interesting nature optimisation algorithm is the Bat algorithm. This algorithm was developed in 2010 by Xin-She Yang. This algorithm uses artificial bats as search agents imitating the natural pulse noise and emission rate of real bats. Throughout the method of this algorithm, this suggests high echolocation, which means it's a method of discovering an object by reflected sound by bats since they produce a high sound. Therefore, the bats can sense how far the prey is and the gap among them and their surroundings to hunt for food. Also, the objective is to find the prey at a minimum distance. This method has now evolved and expanded to other applications based on this which shows that this algorithm is efficient. (Zebari et al.,2020).

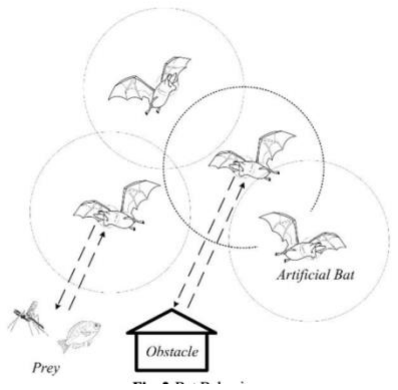


Figure 6 - Bat algorithm (showing the behaviour) Image is from (Zebari et al., 2020)

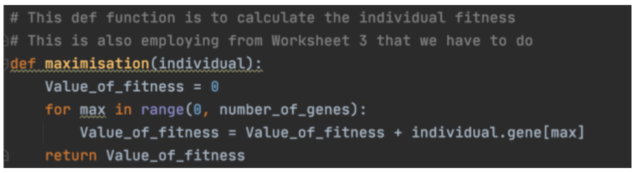
3. EXPERIMENTATION

In this experiment, I have developed GA in all three worksheets. For instance, applying crossover, implementing a population and tournament. These are implemented and executed throughout the code.

Throughout the experiment, I have used a library called matplotlib.pylot to display my graphs and these graphs will show the results when I implement a test on 10 different findings. As I discussed earlier, this will also determine the results and record what the average/ mean fitness is. Also, I will conduct a comparison between the two that I will create and compare the performance of those 2 optimisation programs and mention the maximisation too.

Throughout worksheet 3, binary numbers had to be exchanged to real numbers to initialise and determine the fitness.

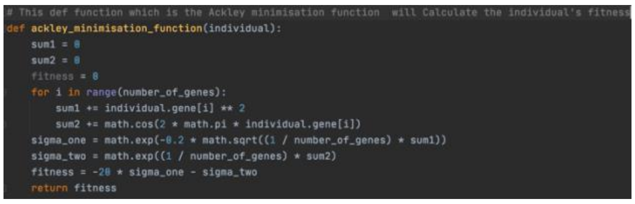
Here is my maximisation (Counting Ones) function. This function will be simply drawn-out to the amount of the real-valued genes due to consuming the fitness of an individual match to the number of '1's.



```python
# This def function is to calculate the individual fitness
# This is also employing from Worksheet 3 that we have to do
def maximisation(individual):
    Value_of_fitness = 0
    for max in range(0, number_of_genes):
        Value_of_fitness = Value_of_fitness + individual.gene[max]
    return Value_of_fitness
```

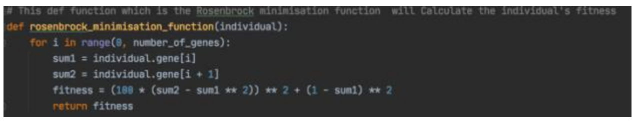
Figure 7 - Maximisation function (Counting Ones)

As well, here are my two minimisation functions: Ackley and the Rosenbrock function.



```python
# This def function which is the Ackley minimisation function  will Calculate the individual's fitness
def ackley_minimisation_function(individual):
    sum1 = 0
    sum2 = 0
    fitness = 0
    for i in range(number_of_genes):
        sum1 += individual.gene[i] ** 2
        sum2 += math.cos(2 * math.pi * individual.gene[i])
    sigma_one = math.exp(-0.2 * math.sqrt((1 / number_of_genes) * sum1))
    sigma_two = math.exp((1 / number_of_genes) * sum2)
    fitness = -20 * sigma_one - sigma_two
    return fitness
```

Figure 8 - Ackley minimisation



```python
# This def function which is the Rosenbrock minimisation function  will Calculate the individual's fitness
def rosenbrock_minimisation_function(individual):
    for i in range(0, number_of_genes):
        sum1 = individual.gene[i]
        sum2 = individual.gene[i + 1]
        fitness = (100 * (sum2 - sum1 ** 2)) ** 2 + (1 - sum1) ** 2
        return fitness
```

Figure 9 - Rosenbrock minimisation

Throughout, my minimisation function, the parameter I used was the instance of the individual population. Then for each gene, I used a for loop function to calculate from i in range to the number of genes which then comes back to its fitness.

Moreover, in the process of my experiment, I have used the deep copy from worksheet 2 that we had to implement and expanded by adding Roulette Wheel and Elitism. However, I tried using the deep copy on the crossover, but it didn't work as I anticipated.

3.1 Results for maximisation optimisation

For this assignment, I have added the roulette wheel and elitism to the experiment and will be sourced in the appendix. This maximisation part is expanded and extended from Worksheet 3. That is why I have added these maximisation results for this assignment.

As shown in Figure 10, I have experimented with the difference of tournament and RW. This can identify which performance is better.

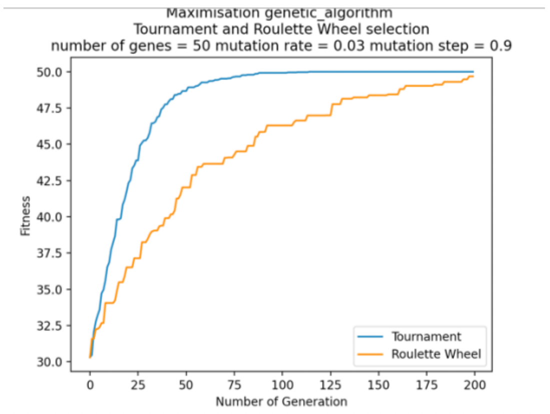


Figure 10 - Tournament and RW with the number of genes of 50

For this test, I have made two plots to represent which selection is shown from the plot labels. I have used 50 for the number of genes. The results also show that Tournament Selection has a higher maximisation fitness compared to RW. The results that I have found for Tournament Selection maximisation fitness is 50.0 and for the Roulette wheel is 49.68540419269904.

As well, I have also experimented with this comparison, but the generation is 500 and genes is 200 as shown in Figure 11.

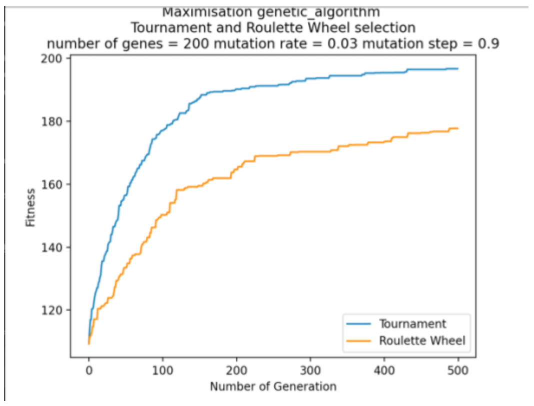


Figure 11 – Both Selections with the genes 200 and generations of 500

As shown from Figure 11, the tournament selection is a clear indicator that it is a better fit than RW. Here are my results for the tournament: 196.66654480202897 and Roulette Wheel is 177.7189295321322.

I will be conducting a test on the tournament selection and finding the best fitness and mean fitness as shown in Figure 12. For this test, I will be using the same parameters on my code such as the number of genes is 50, population size is 50, generations are 200, mutation rate to 0.03 and mutation step to 0.9. As Figure 12 shows, the result of maximum fitness is: 50.0 and the average fitness is: 49.56545039526755. This also shows in figure 13 that it reaches the maximum fitness as well when I change the number of genes to 200, and the number of generations to 500.

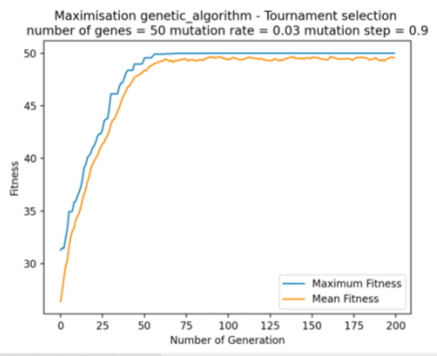


Figure 12 - Tournament to represent the mean and maximum fitness

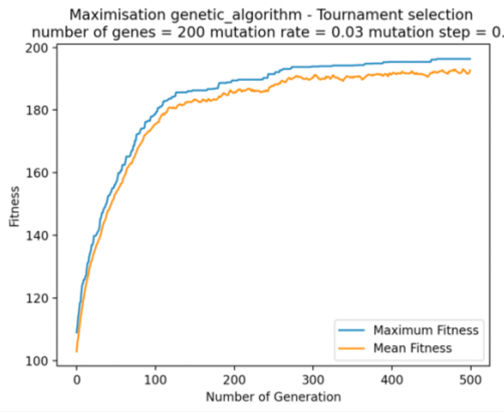


Figure 13 - Another TS mean and max fitness but 200 genes and 500 generations

For the varied mutation rate, for both selections, I have conducted that the best mutation rate is 0.03 as shown in Figure 14 and Figure 15.

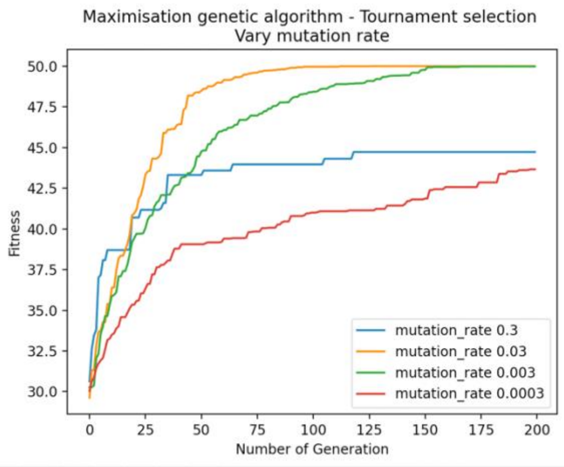


Figure 14 - Tournament Selection vary mutation rate

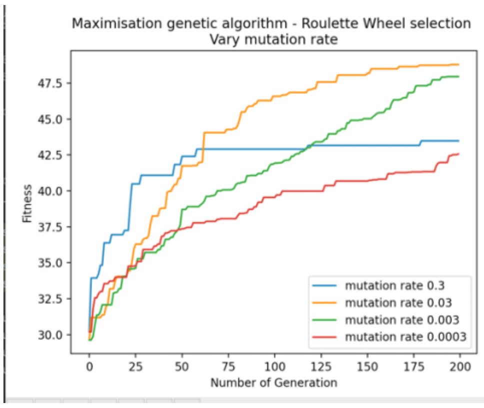


Figure 15 - Roulette Wheel vary mutation rate

In terms of the mutation step for both selection methods, 0.9 is the best for the mutation step. As Figure 16, the maximum fitness reaches 50 on all of them on the Tournament Selection, however on the Roulette Wheel (Figure 17) it shows that 0.9 is the best. As well RW, shows that 0.3 is 48.275278639417465, 0.9 is 49.86481084718186, 0.6 is 49.4756549983502 and 1.0 is 49.7304374102101.

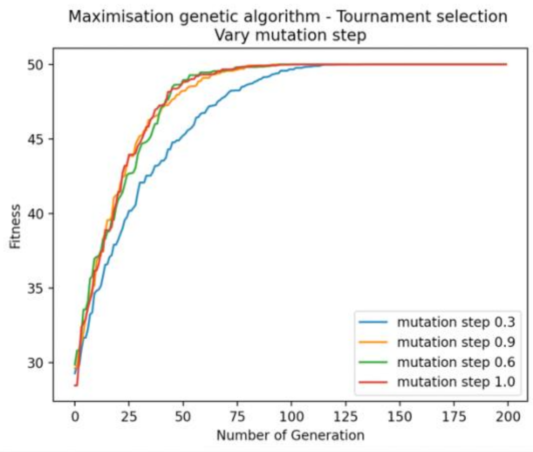


Figure 16 - Tournament Selection vary mutation step

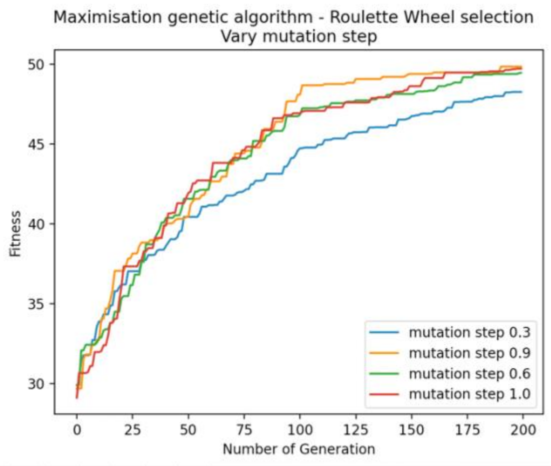


Figure 17 - Roulette Wheel mutation step

For my Roulette Wheel, I have conducted the same way as the Tournament too. However, when I increase the number of genes and number of generations for instance in Figure 18. The maximum fitness is not close to the fitness compared to the Tournament Selection as shown from Figure 12 and Figure 13. As results show for Figure 19, the maximum fitness is 177.68517124750394 and the mean is 167.3043662854009.

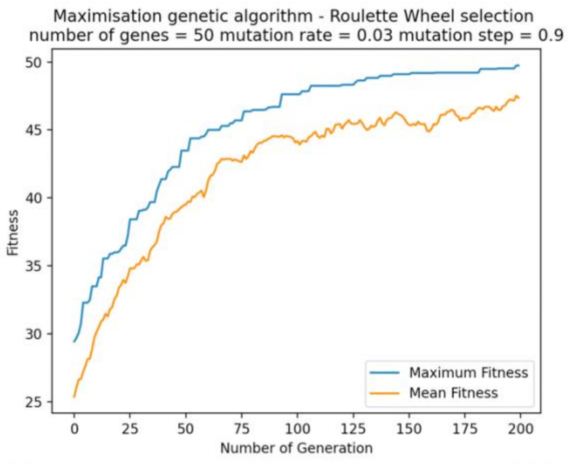


Figure 18 - Roulette Wheel max and mean 200 generations and 50 genes.

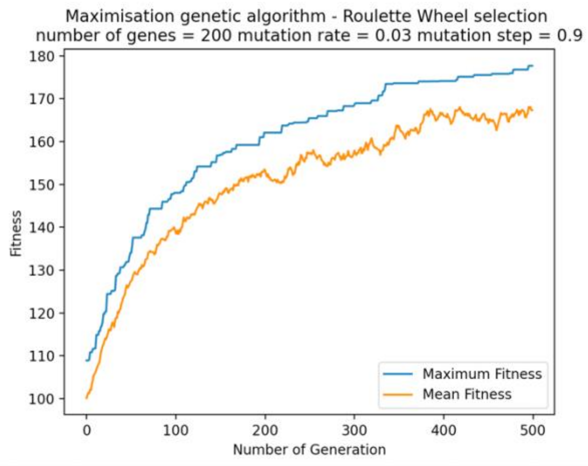


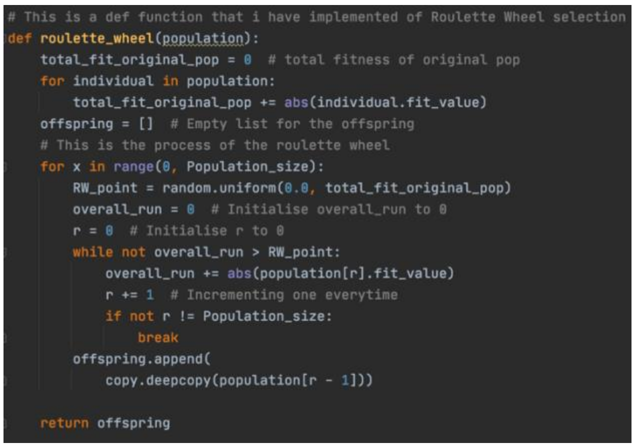
Figure 19 - Roulette wheel max and mean for 500 generations and 200 genes

3.2 Results for minimisation optimisation

For this minimisation experiment, the best fitness should be the lowest and individuals with lesser fitness in the population should be picked.

3.2.1 Ackley function

During the process of this experiment, I had to make some changes for this minimisation compared to the maximisation. These include the two selection methods, mutation, and the original / initialised population. The mutation and the initialise population are just changing the parameters since we must include – 32 and 32. As shown in Figure 20, this is the Roulette wheel for this minimisation. This is because this function will convert all the negative numbers into positives from the use of a math library. This is all from using abs ().



```
# This is a def function that i have implemented of Roulette Wheel selection
def roulette_wheel(population):
    total_fit_original_pop = 0  # total fitness of original pop
    for individual in population:
        total_fit_original_pop += abs(individual.fit_value)
    offspring = []  # Empty list for the offspring
    # This is the process of the roulette wheel
    for x in range(0, Population_size):
        RW_point = random.uniform(0.0, total_fit_original_pop)
        overall_run = 0  # Initialise overall_run to 0
        r = 0  # Initialise r to 0
        while not overall_run > RW_point:
            overall_run += abs(population[r].fit_value)
            r += 1  # Incrementing one everytime
            if not r != Population_size:
                break
        offspring.append(
            copy.deepcopy(population[r - 1]))

    return offspring
```

Figure 20 – RW of Ackley

Figure 21 demonstrates that the Tournament Selection is better. The results show that the minimum fitness for Tournament Selection is -22.704611477446004 and the Roulette Wheel is -19.177750463050234.

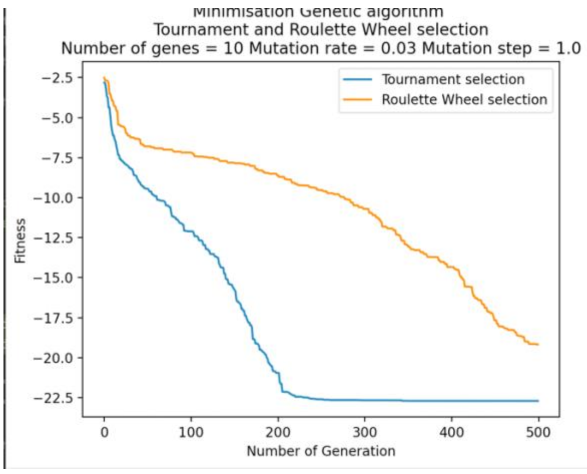


Figure 21 – Both Selection methods with 500 generation and genes as 10

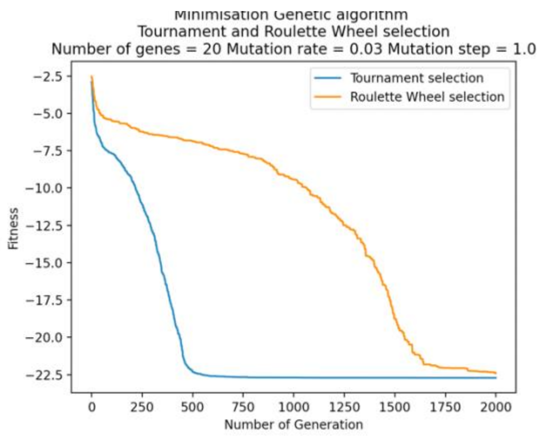


Figure 22 – Both selection methods but with 2000 generations and 20 genes

For Figures 23 and 24, I will be testing the minimisation and mean fitness for the Tournament Selection. I will conduct different parameters for the number of genes and the number of generations. For instance, Figure 24, has 200 genes and 2000 generations. The image below shows that the mean fitness is near the minimum fitness. The results for Figure 23, show that the minimum fitness is -22.706646033560325 and the mean fitness is -22.429084244001093. As well for Figure 24, these results show for the minimum fitness is -22.712028010145485 and the mean fitness is -22.23368346736088.

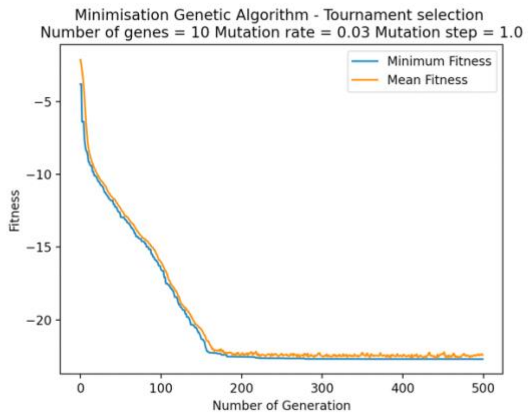


Figure 23 - Tournament Selection for min and mean, generation 500 and 10 genes

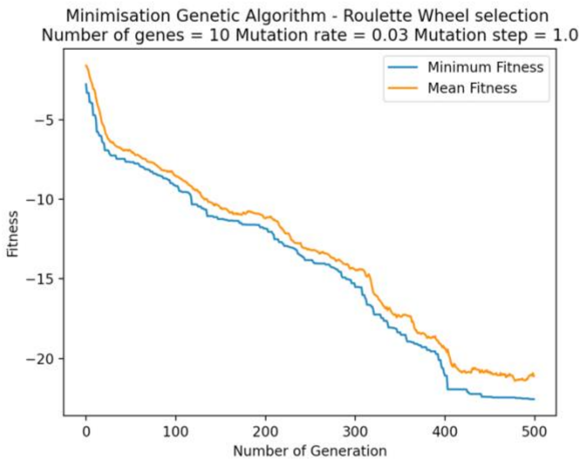


Figure 25 - Roulette Wheel min and mean fitness with 500 generation and 10 as genes.

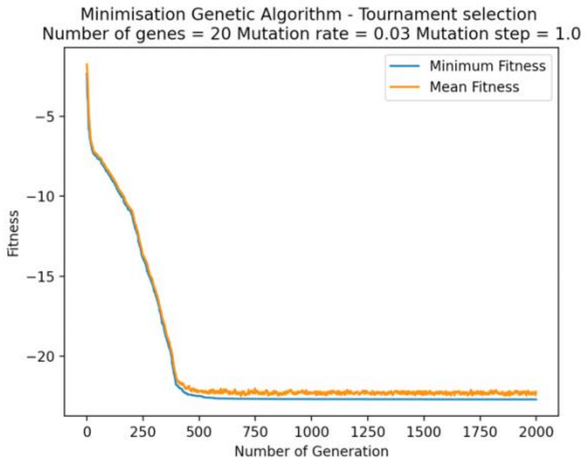


Figure 24 - Tournament Selection for min and mean for 2000 as number of generation and 20 genes

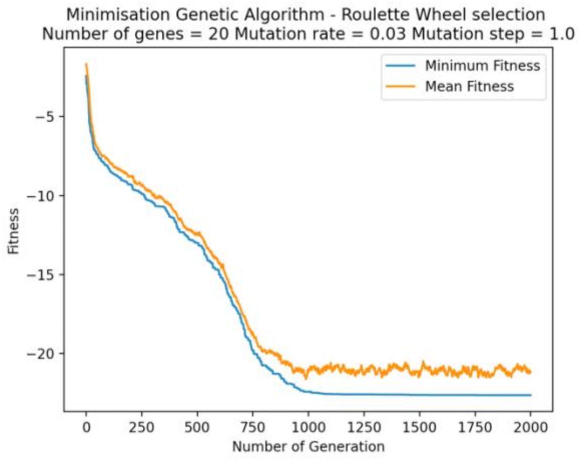


Figure 26 - RW for min and mean with 2000 generations and 20 genes

In comparison, I have also made results for the Roulette Wheel for the minimum and mean fitness. For figure 25, the minimum fitness for this is -22.55854919885521 and the mean fitness is -21.11099840829077. As well for figure 26, I have done 20 genes and 2000 generations, the minimum fitness is -22.639323282334917 and the mean fitness is -21.160239266094578.

To conclude for the minimum and mean fitness of Tournament and Roulette Wheel Selection, Tournament Selection is better because for example the number of genes = 10 and number of generations = 500 on both these selections, Tournament Selection has -22.706646033560325 whilst Roulette Wheel has -22.55854919885521 for minimum fitness.

Furthermore, I have done a mutation rate for the Tournament Selection. Here are my results. So, for the minimum fitness for 0.3, it is

-22.255183183228812, minimum fitness for 0.03 is -22.693948962173167, minimum fitness for 0.003 is -19.09860948384937 and for 0.0003 is -12.76190919476929. This suggests that 0.03 is the best for decreasing fitness.

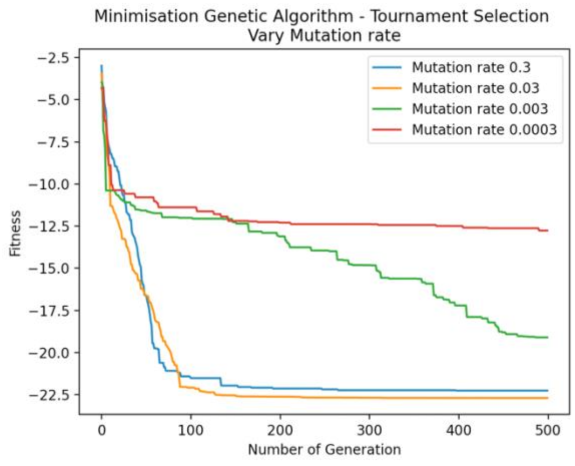


Figure 27 - Tournament Selection - vary Mutation rate

As well, I have done a mutation rate for my Roulette Wheel. As figure 28 shows that 0.03 is the best for decreasing fitness too.

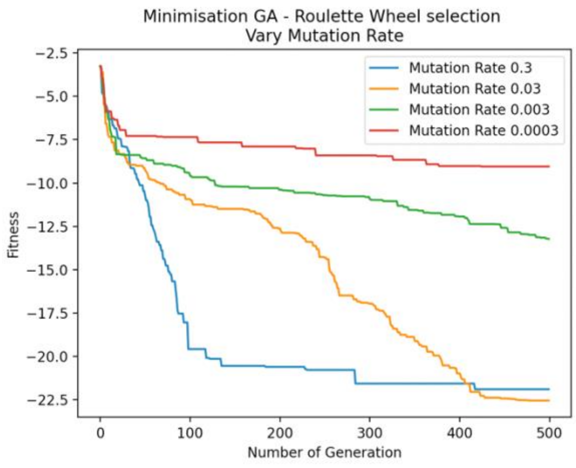


Figure 28 - Roulette Wheel vary mutation rate

Additionally, I have done the mutation step for Tournament and Roulette Wheel Selection as shown from Figure 29 and Figure 30.

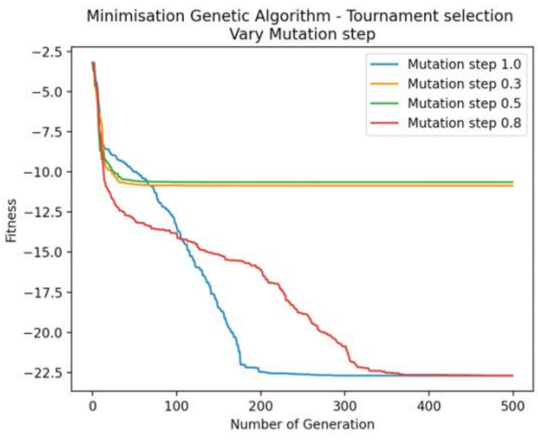


Figure 29 - Tournament Selection, vary Mutation Step

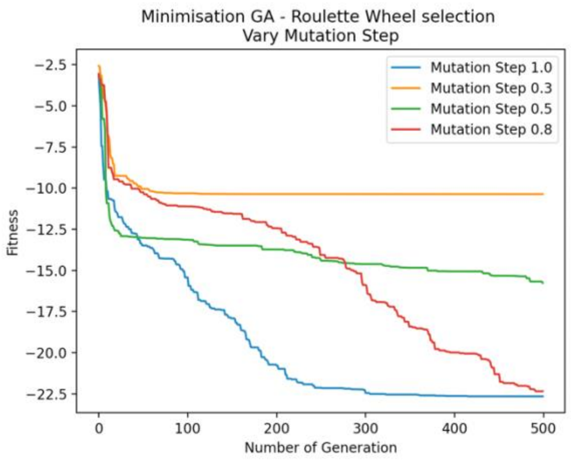


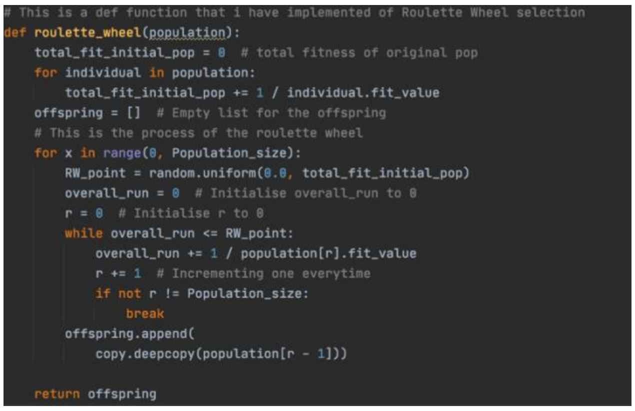
Figure 30 - Roulette Wheel, vary Mutation step

RW indicates that the Mutation step for 1.0 is the best for decreasing fitness. As well on the Tournament Selection, the results show that the mutation step for 1.0 is the best as well. This is because the results for the Tournament Selection for 1.0 is -22.702125925118224 and for 0.8 it is -22.68909013072996 since the graph in Figure 29 doesn't indicate which is the best out of the two of them.

3.2.2 Rosenbrock function

Throughout the process of this experiment, I had to make some changes for this minimisation too. This included the minimisation itself as shown in Figure 9 and some changes on mutation, the original population and RW. The original population and mutation have been changed since we must include – 100 and 100.

As shown in Figure 31 for the Roulette Wheel, I inversed the fitness of the individuals and took the sum to enable the algorithm to decrease by choosing up the weaker individuals. This will ensure that those with a low level of fitness have a chance to be chosen.



```python
# This is a def function that i have implemented of Roulette Wheel selection
def roulette_wheel(population):
    total_fit_initial_pop = 0  # total fitness of original pop
    for individual in population:
        total_fit_initial_pop += 1 / individual.fit_value
    offspring = []  # Empty list for the offspring
    # This is the process of the roulette wheel
    for x in range(0, Population_size):
        RW_point = random.uniform(0.0, total_fit_initial_pop)
        overall_run = 0  # Initialise overall_run to 0
        r = 0  # Initialise r to 0
        while overall_run <= RW_point:
            overall_run += 1 / population[r].fit_value
            r += 1  # Incrementing one everytime
            if not r != Population_size:
                break
        offspring.append(
            copy.deepcopy(population[r - 1]))

    return offspring
```

Figure 31 - Roulette Wheel for Rosenbrock function

During my experiment, I have noticed that most of the graphs I will be showing converged in a horizontal line at around 0. Throughout this experiment, I conducted several tests to show the performance for the two selections and determine which is closest to 0.

Figures 32 and 33 will show a contrast between the tournament and RW. In Figure 32 and Figure 33, the Tournament Selection is better than RW for the lowest minimum fitness. The results show that in Figure 32, the minimum fitness of Tournament Selection is 0.04675685781301373 and the minimum fitness for RW is 3.0513237789938987. Furthermore, I also conducted another test with 20 as the number of genes and 2000 generations as shown from Figure 33. For this test, the results show that the minimum fitness for Tournament Selection is 0.6675311201741557 and the minimum fitness for Roulette Wheel is 0.7652588513390385. The results can show that the more generations there are, there will be a chance of the selection to get closer to 0.

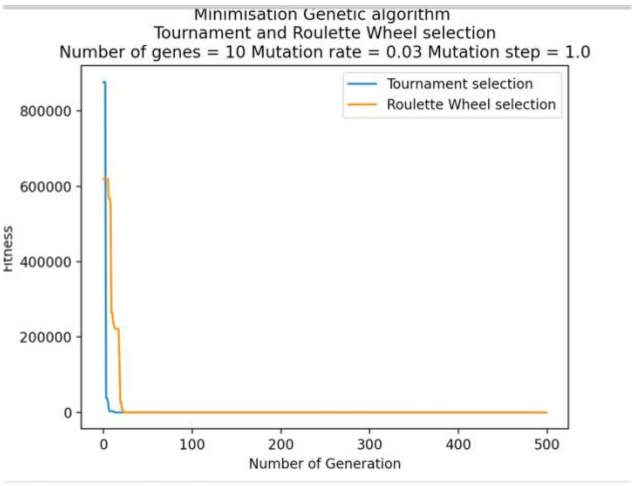


Figure 32 – Both Selections with 500 generations and 10 as the number of genes

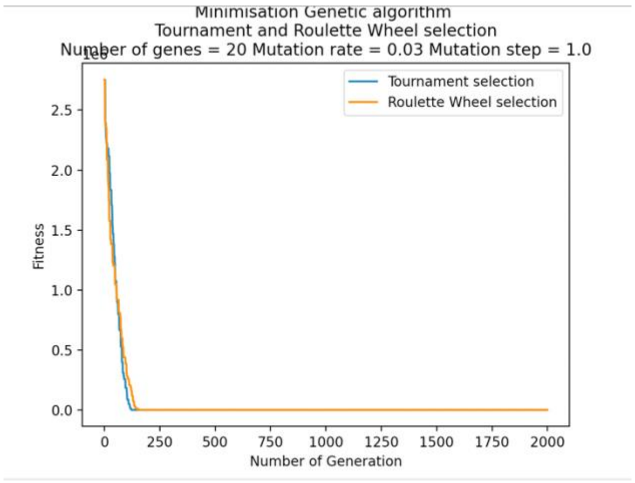


Figure 33 – Both Selections with 2000 generations and 20 genes

Moreover, I will be conducting a test on the tournament selection and finding the lowest fitness and mean. Also, be conducting a different number of genes and generations too. For instance, in figure 35, I will be doing

20 genes and 2000 generations compared to Figure 34, I did 10 as the number of genes and 500 generations. The results also show that the more generations it has, it will likely be closer to 0 too. For instance, in Figure 35 the minimum fitness is 0.2908852330233489 and the mean fitness is 0.2908852330233492. This can be compared to Figure 34, that the minimum fitness is 0.47955751554643433 and 0.47955751554643455 for the mean.

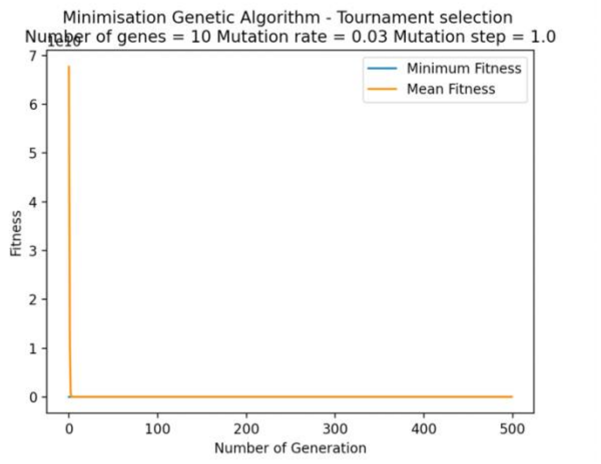


Figure 34 - Tournament Selection with 500 generations and 10 as number of genes

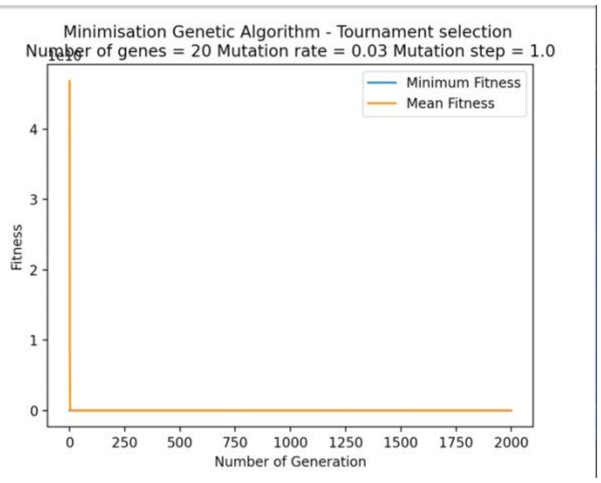


Figure 35 - Tournament Selection with 2000 generations and genes as 20

In comparison, I will be conducting tests with the same parameters from the Tournament and inputted for the RW as stated in Figure 36 and Figure 37. The results show that in Figure 36 that the minimum fitness is 0.5152251383208061 and the mean fitness is 0.5152251383208057. However, in Figure 37, the minimum fitness is 0.8612633528856898 and the mean fitness is 0.8612633528856907. As more generations have been added, it gets further away from 0.

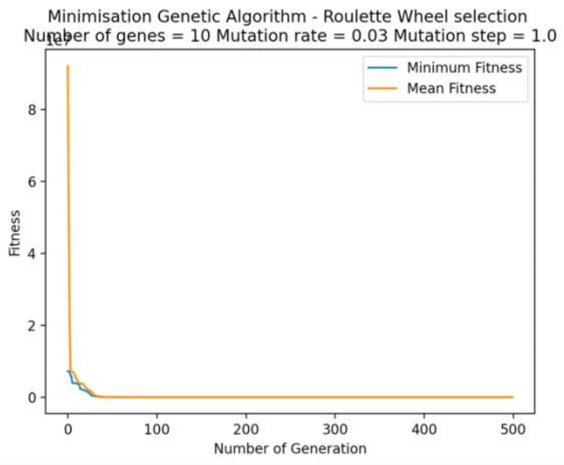


Figure 36 - Roulette Wheel for 500 generations and genes as 10

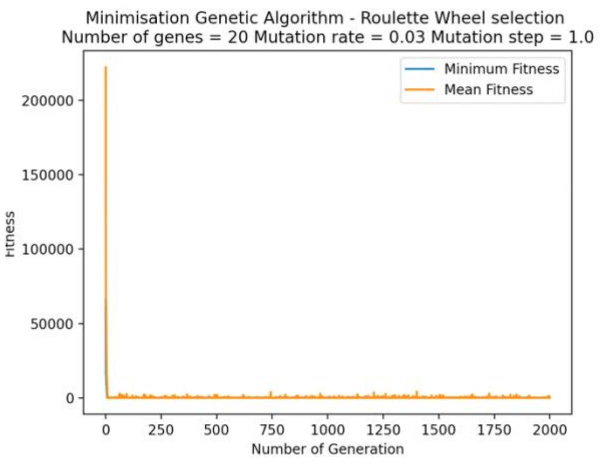


Figure 37 - Roulette Wheel for number of 2000 generations and number of genes 20

Additionally, I have done tests on a varied mutation rate for the two. The graphs below will show which mutation rate is better to use for those Selections and I will compare the results between them.

For tournament Selection, as shown from Figure 38, it clearly shows that the best mutation rate is 0.03. My results for this test show that the mutation rate for 0.3 is 7.45778406198979, 0.03 is 3.7670108478633475, 0.003 is 1542911.9711373826 and 0.0003 158772.34586930278. The result for 0.003 and 0.0003 is high because of the fitness 1*10^7 or 1e7.

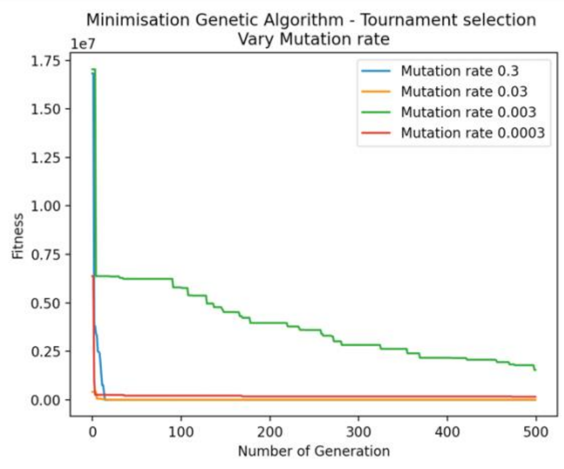


Figure 38 - Tournament Selection for vary mutation rate

I also found out that the best mutation rate is 0.03 given from the results for RW as shown in Figure 39. So, for my results my mutation rate for 0.3 is 11.51301614919247, 0.03 is 11.43437092882608, 0.003 is 14.186172901198361 and 0.0003 is 410.56423037045363.

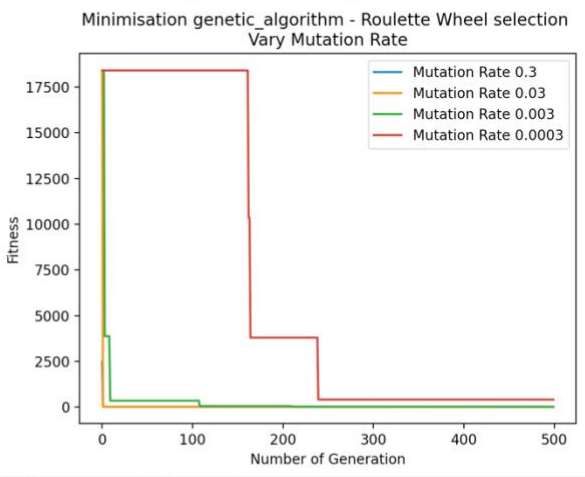


Figure 39 - Vary mutation rate for Roulette Wheel

Finally, the 2 graphs below show the various mutation step for the two selection methods.

So, for the Tournament Selection, I have conducted different mutation steps to see which one is better for performance. For my results, I have found out that 1.0 is the best for the mutation step. Here are my results, so for 1.0 it is 0.7783590526885409, 0.3 is 66.88658099218682, 0.5 is 66.88848858492034 and lastly 0.8 is 66.88755895279033. This indicates in Figure 40, that the best mutation step is 1.0.

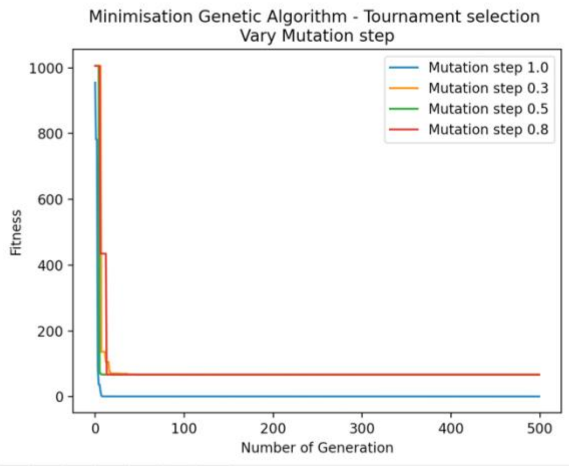


Figure 40 - vary Mutation Step for Tournament Selection

Furthermore, my Roulette wheel also shows that the best mutation step is 1.0 too from the

Results. My results for 1.0 is 1.0390480354380542, 0.3 is 1448542.1358728495, 0.5 is 97382.48375883694 and finally 0.8 is 2.173148964654722. These results are shown from Figure 41 and the fitness results are high because 1 is times to ten to the power of 6 or 1e6.

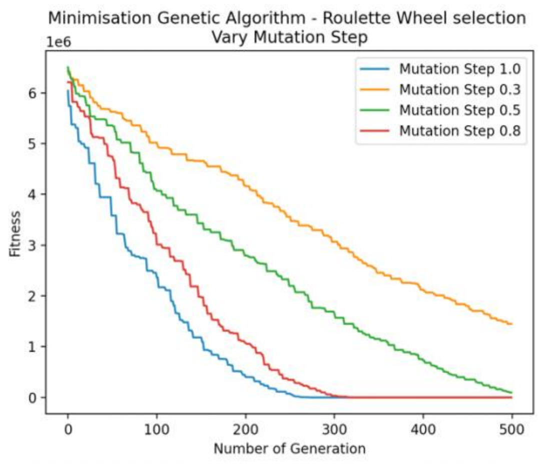


Figure 41 - Vary mutation step for Roulette Wheel

4. CONCLUSION

In conclusion for these optimisation results. I have compared the performance for the two selections. As several figures show, I have concluded that the Tournament Selection is the better performance. This is with the same parameters on each selection method so it will be fair. Furthermore, I have concluded that the minimisation mutation step is 1.0 and the mutation rate for my maximisation and minimisation show that 0.03 is the best.

Moreover, as shown from the graphs, tournament selection is the best for performance because Roulette Wheel is based on chance. Besides, RW might pick the weaker individuals instead of the best so the algorithm cannot always optimise the best it can, for instance in Figure 36, the more generations it has for the Roulette Wheel, the more likely it will pick the weaker individual.

To further explore this topic, I could implement this with the nature bat Algorithm, this is because the nature algorithm has been proven and this possibly can be adapted to find something useful for different findings. A sample could be that the tournament selection may be adapted to compare the number of frequencies due to the bats high sound and determine whether this can be effective for the bat's surroundings.

REFERENCES

Bull. L. (2021) Lecture slides week 1- *Genetic Algorithms*. PowerPoint. Biocomputation [online] Available from: Blackboard [Accessed 16 November 2021]

Du, H., Wang, Z., Zhan, W. and Guo, J. (2018) Elitism and Distance Strategy For Selection of Evolutionary Algorithms. *IEEE Access* [online]. 6, pp. 44531-44541. [Accessed 27 November 2021].

GeeksforGeeks (2018). *Tournament Section (GA)*. Available from: https://www.geeksforgeeks.org/tournament-selection-ga/ [Accessed 26 November 2021]

Mallawaarachchi. V. (2017) *Introduction to Genetic Algorithms – Including Example Code*. Available from: https://towardsdatascience.com/introduction-to-genetic-algorithms-including-example-code-e396e98d8bf3 [Accessed 20 November 2021]

Raynor. P. (2021). Lecture slides – *Evolutionary Computation: Genetic Algorithms*. PowerPoint. Intelligent Systems [online] Available from: Blackboard [Accessed 16 November 2021]

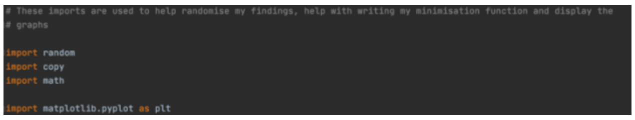
TutorialPoint (no date), *Genetic Algorithms – Parent Selection*. Available from: https://www.tutorialspoint.com/genetic_algorithms/genetic_algorithms_parent_selection.htm [Accessed 21 November 2021]

Zebari, A.Y., Almufti, S.M. and Abdulrahman, C.M. (2020) Bat Algorithm (Ba): Review, Applications and Modifications. *International Journal of Scientific World* [online]. 8 (1), pp. 1-7. [Accessed 01 December 2021]

SOURCE CODE AS AN APPENDIX

Throughout, this source code, I will be going from the start to the end as what it would look like in my Python code, I have done 3 files, however further down below since my 2 minimisations are both nearly identical, I will state which parts have changed on them.

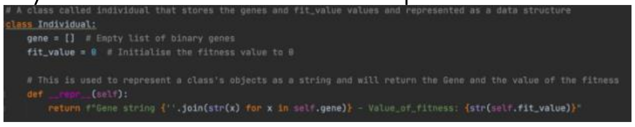
Here are the imports that I have used for my minimum and maximum optimisations:



```python
# These imports are used to help randomise my findings, help with writing my minimisation function and display the
# graphs

import random
import copy
import math

import matplotlib.pyplot as plt
```

Figure 42 - My imports

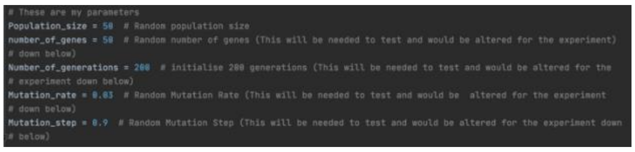
My class function for all the optimisations too.



```python
# A class called individual that stores the genes and fit_value values and represented as a data structure
class Individual:
    gene = []  # Empty list of binary genes
    fit_value = 0  # Initialise the fitness value to 0

    # This is used to represent a class's objects as a string and will return the Gene and the value of the fitness
    def __repr__(self):
        return f"Gene string {''.join(str(x) for x in self.gene)} - Value_of_fitness: {str(self.fit_value)}"
```

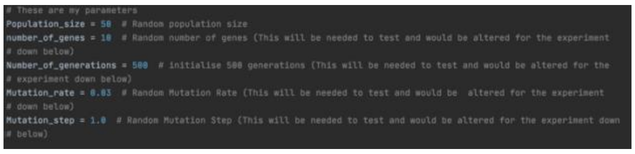
Figure 43 - A class function called individual

My Parameters for maximisation and minimisation:



```python
# These are my parameters
Population_size = 50  # Random population size
number_of_genes = 50  # Random number of genes (This will be needed to test and would be altered for the experiment)
# down below)
Number_of_generations = 200  # initialise 200 generations (This will be needed to test and would be altered for the
# experiment down below)
Mutation_rate = 0.03  # Random Mutation Rate (This will be needed to test and would be  altered for the experiment
# down below)
Mutation_step = 0.9  # Random Mutation Step (This will be needed to test and would be altered for the experiment down
# below)
```

Figure 44 - Parameters for Maximisation

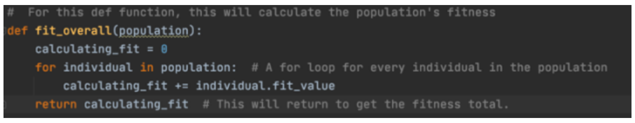


```python
# These are my parameters
Population_size = 50  # Random population size
number_of_genes = 10  # Random number of genes (This will be needed to test and would be altered for the experiment
# down below)
Number_of_generations = 500  # initialise 500 generations (This will be needed to test and would be altered for the
# experiment down below)
Mutation_rate = 0.03  # Random Mutation Rate (This will be needed to test and would be  altered for the experiment
# down below)
Mutation_step = 1.0  # Random Mutation Step (This will be needed to test and would be altered for the experiment down
# below)
```

Figure 45 - Parameters for Minimisation for both Ackley and Rosenbrock

After these parameters, I have added the functions for the optimisations. These have already been stated in Figures 7,8 and 9.

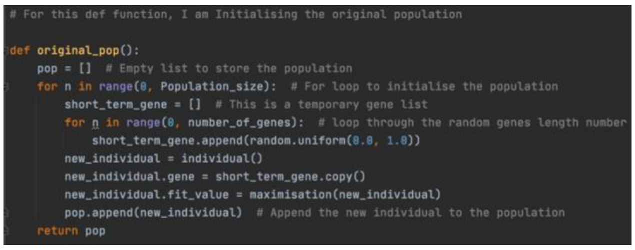
Figure 46, This def function, calculate the population's fitness and is used throughout all the optimisations.



```python
#  For this def function, this will calculate the population's fitness
def fit_overall(population):
    calculating_fit = 0
    for individual in population:  # A for loop for every individual in the population
        calculating_fit += individual.fit_value
    return calculating_fit  # This will return to get the fitness total.
```

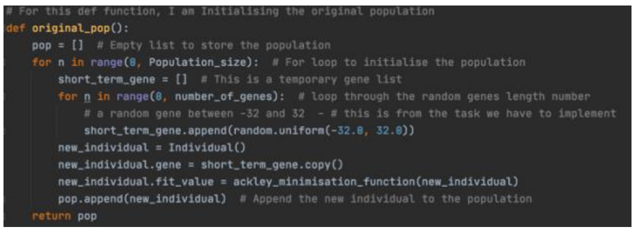
Figure 46 - Initialising the population

For my def function, I am initialising the original population. The only difference is on the 6th line for all of them.



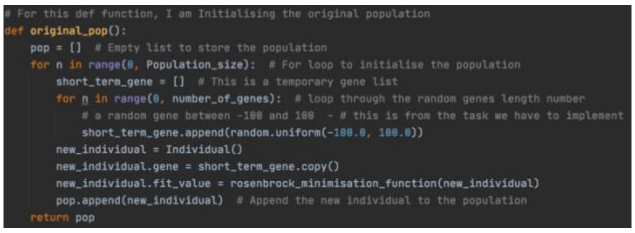
```python
# For this def function, I am Initialising the original population

def original_pop():
    pop = []  # Empty list to store the population
    for n in range(0, Population_size):  # For loop to initialise the population
        short_term_gene = []  # This is a temporary gene list
        for n in range(0, number_of_genes):  # loop through the random genes length number
            short_term_gene.append(random.uniform(0.0, 1.0))
        new_individual = individual()
        new_individual.gene = short_term_gene.copy()
        new_individual.fit_value = maximisation(new_individual)
        pop.append(new_individual)  # Append the new individual to the population
    return pop
```

Figure 47 - Maximisation 0.0, 1.0



```python
# For this def function, I am Initialising the original population
def original_pop():
    pop = []  # Empty list to store the population
    for n in range(0, Population_size):  # For loop to initialise the population
        short_term_gene = []  # This is a temporary gene list
        for n in range(0, number_of_genes):  # loop through the random genes length number
            # a random gene between -32 and 32  - # this is from the task we have to implement
            short_term_gene.append(random.uniform(-32.0, 32.0))
        new_individual = Individual()
        new_individual.gene = short_term_gene.copy()
        new_individual.fit_value = ackley_minimisation_function(new_individual)
        pop.append(new_individual)  # Append the new individual to the population
    return pop
```

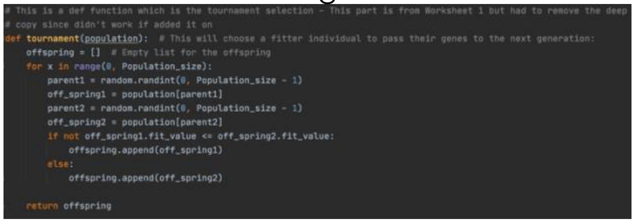
Figure 48 - Ackley function -32.0, 32.0



```python
# For this def function, I am Initialising the original population
def original_pop():
    pop = []  # Empty list to store the population
    for n in range(0, Population_size):  # For loop to initialise the population
        short_term_gene = []  # This is a temporary gene list
        for n in range(0, number_of_genes):  # loop through the random genes length number
            # a random gene between -100 and 100  - # this is from the task we have to implement
            short_term_gene.append(random.uniform(-100.0, 100.0))
        new_individual = Individual()
        new_individual.gene = short_term_gene.copy()
        new_individual.fit_value = rosenbrock_minimisation_function(new_individual)
        pop.append(new_individual)  # Append the new individual to the population
    return pop
```

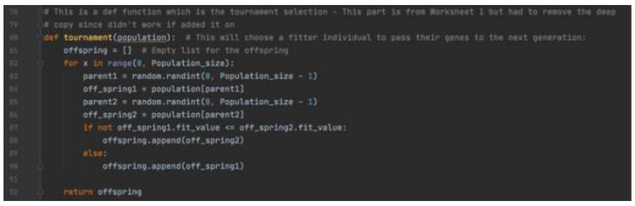
Figure 49 - Rosenbrock function -100.0, 100.0

Then it goes to the Tournament selection, the difference is the ending on the If statements.



```python
# This is a def function which is the tournament selection - This part is from Worksheet 1 but had to remove the deep
# copy since didn't work if added it on
def tournament(population):  # This will choose a fitter individual to pass their genes to the next generation:
    offspring = []  # Empty list for the offspring
    for x in range(0, Population_size):
        parent1 = random.randint(0, Population_size - 1)
        off_spring1 = population[parent1]
        parent2 = random.randint(0, Population_size - 1)
        off_spring2 = population[parent2]
        if not off_spring1.fit_value <= off_spring2.fit_value:
            offspring.append(off_spring1)
        else:
            offspring.append(off_spring2)

    return offspring
```

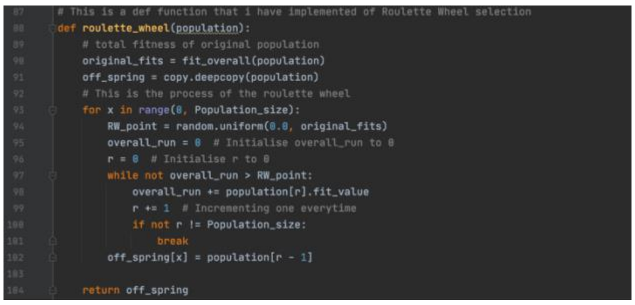
Figure 50 - Tournament Selection for the maximisation



```python
# This is a def function which is the tournament selection - This part is from Worksheet 1 but had to remove the deep
# copy since didn't work if added it on
def tournament(population):  # This will choose a fitter individual to pass their genes to the next generation:
    offspring = []  # Empty list for the offspring
    for x in range(0, Population_size):
        parent1 = random.randint(0, Population_size - 1)
        off_spring1 = population[parent1]
        parent2 = random.randint(0, Population_size - 1)
        off_spring2 = population[parent2]
        if not off_spring1.fit_value <= off_spring2.fit_value:
            offspring.append(off_spring2)
        else:
            offspring.append(off_spring1)

    return offspring
```

Figure 51 - Tournament Selection for the minimisation

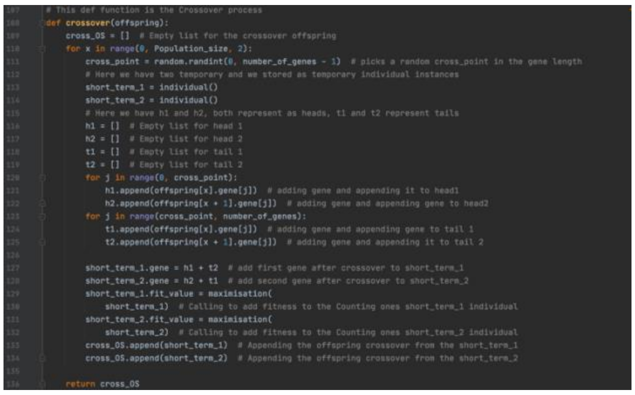
Then it goes to RW. I have stated my Roulette wheel for my minimisations in Figures 20 and 31. Here is my maximisation:



```python
# This is a def function that i have implemented of Roulette Wheel selection
def roulette_wheel(population):
    # total fitness of original population
    original_fits = fit_overall(population)
    off_spring = copy.deepcopy(population)
    # This is the process of the roulette wheel
    for x in range(0, Population_size):
        RW_point = random.uniform(0.0, original_fits)
        overall_run = 0  # Initialise overall_run to 0
        r = 0  # Initialise r to 0
        while not overall_run > RW_point:
            overall_run += population[r].fit_value
            r += 1  # Incrementing one everytime
            if not r != Population_size:
                break
        off_spring[x] = population[r - 1]

    return off_spring
```

Figure 52 - Maximisation for Roulette Wheel

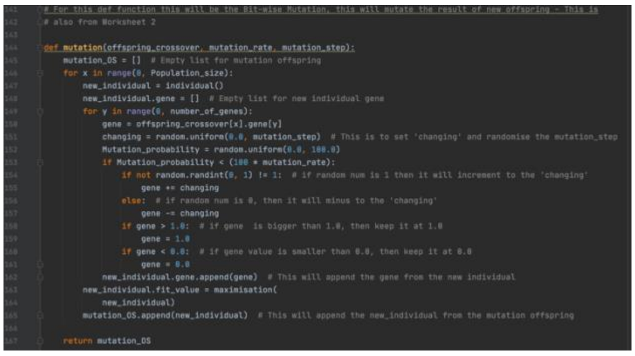
Here is my crossover function for all optimisations.



```python
# This def function is the Crossover process
def crossover(offspring):
    cross_OS = []  # Empty list for the crossover offspring
    for x in range(0, Population_size, 2):
        cross_point = random.randint(0, number_of_genes - 1)  # picks a random cross_point in the gene length
        # Here we have two temporary and we stored as temporary individual instances
        short_term_1 = individual()
        short_term_2 = individual()
        # Here we have h1 and h2, both represent as heads, t1 and t2 represent tails
        h1 = []  # Empty list for head 1
        h2 = []  # Empty list for head 2
        t1 = []  # Empty list for tail 1
        t2 = []  # Empty list for tail 2
        for j in range(0, cross_point):
            h1.append(offspring[x].gene[j])  # adding gene and appending it to head1
            h2.append(offspring[x + 1].gene[j])  # adding gene and appending gene to head2
        for j in range(cross_point, number_of_genes):
            t1.append(offspring[x].gene[j])  # adding gene and appending gene to tail 1
            t2.append(offspring[x + 1].gene[j])  # adding gene and appending it to tail 2

        short_term_1.gene = h1 + t2  # add first gene after crossover to short_term_1
        short_term_2.gene = h2 + t1  # add second gene after crossover to short_term_2
        short_term_1.fit_value = maximisation(
            short_term_1)  # Calling to add fitness to the Counting ones short_term_1 individual
        short_term_2.fit_value = maximisation(
            short_term_2)  # Calling to add fitness to the Counting ones short_term_2 individual
        cross_OS.append(short_term_1)  # Appending the offspring crossover from the short_term_1
        cross_OS.append(short_term_2)  # Appending the offspring crossover from the short_term_2

    return cross_OS
```

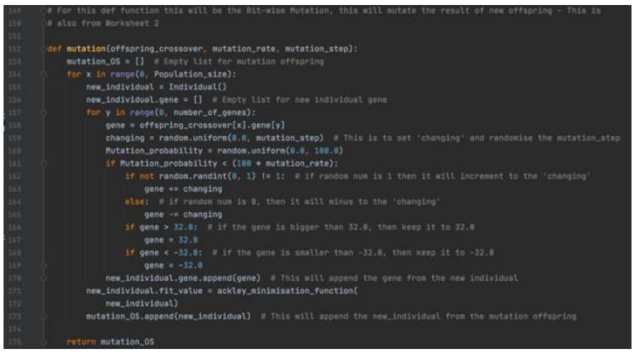
Figure 53 - Crossover for all optimisations

Here is my mutation function. The only difference is that the parameters have been changed.



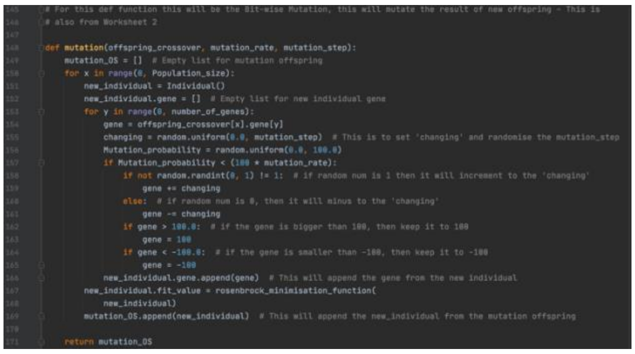
```python
# For this def function this will be the Bit-wise Mutation, this will mutate the result of new offspring - This is
# also from Worksheet 2

def mutation(offspring_crossover, mutation_rate, mutation_step):
    mutation_OS = []  # Empty list for mutation offspring
    for x in range(0, Population_size):
        new_individual = individual()
        new_individual.gene = []  # Empty list for new individual gene
        for y in range(0, number_of_genes):
            gene = offspring_crossover[x].gene[y]
            changing = random.uniform(0.0, mutation_step)  # This is to set 'changing' and randomise the mutation_step
            Mutation_probability = random.uniform(0.0, 100.0)
            if Mutation_probability < (100 * mutation_rate):
                if not random.randint(0, 1) != 1:  # if random num is 1 then it will increment to the 'changing'
                    gene += changing
                else:  # if random num is 0, then it will minus to the 'changing'
                    gene -= changing
                if gene > 1.0:  # if gene  is bigger than 1.0, then keep it at 1.0
                    gene = 1.0
                if gene < 0.0:  # if gene value is smaller than 0.0, then keep it at 0.0
                    gene = 0.0
            new_individual.gene.append(gene)  # This will append the gene from the new individual
        new_individual.fit_value = maximisation(
            new_individual)
        mutation_OS.append(new_individual)  # This will append the new_individual from the mutation offspring

    return mutation_OS
```

Figure 54 - Mutation for maximisation



```python
# For this def function this will be the Bit-wise Mutation, this will mutate the result of new offspring - This is
# also from Worksheet 2

def mutation(offspring_crossover, mutation_rate, mutation_step):
    mutation_OS = []  # Empty list for mutation offspring
    for x in range(0, Population_size):
        new_individual = Individual()
        new_individual.gene = []  # Empty list for new individual gene
        for y in range(0, number_of_genes):
            gene = offspring_crossover[x].gene[y]
            changing = random.uniform(0.0, mutation_step)  # This is to set 'changing' and randomise the mutation_step
            Mutation_probability = random.uniform(0.0, 100.0)
            if Mutation_probability < (100 * mutation_rate):
                if not random.randint(0, 1) != 1:  # if random num is 1 then it will increment to the 'changing'
                    gene += changing
                else:  # if random num is 0, then it will minus to the 'changing'
                    gene -= changing
                if gene > 32.0:  # if the gene is bigger than 32.0, then keep it to 32.0
                    gene = 32.0
                if gene < -32.0:  # if the gene is smaller than -32.0, then keep it to -32.0
                    gene = -32.0
            new_individual.gene.append(gene)  # This will append the gene from the new individual
        new_individual.fit_value = ackley_minimisation_function(
            new_individual)
        mutation_OS.append(new_individual)  # This will append the new_individual from the mutation offspring

    return mutation_OS
```

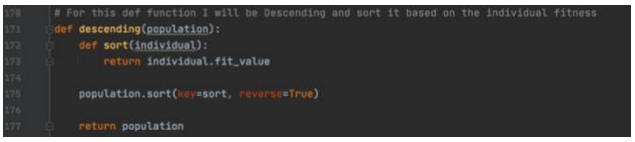
Figure 55 - Mutation for Ackley



```python
# For this def function this will be the Bit-wise Mutation, this will mutate the result of new offspring - This is
# also from Worksheet 2

def mutation(offspring_crossover, mutation_rate, mutation_step):
    mutation_OS = []  # Empty list for mutation offspring
    for x in range(0, Population_size):
        new_individual = Individual()
        new_individual.gene = []  # Empty list for new individual gene
        for y in range(0, number_of_genes):
            gene = offspring_crossover[x].gene[y]
            changing = random.uniform(0.0, mutation_step)  # This is to set 'changing' and randomise the mutation_step
            Mutation_probability = random.uniform(0.0, 100.0)
            if Mutation_probability < (100 * mutation_rate):
                if not random.randint(0, 1) != 1:  # if random num is 1 then it will increment to the 'changing'
                    gene += changing
                else:  # if random num is 0, then it will minus to the 'changing'
                    gene -= changing
                if gene > 100.0:  # if the gene is bigger than 100, then keep it to 100
                    gene = 100
                if gene < -100.0:  # if the gene is smaller than -100, then keep it to -100
                    gene = -100
            new_individual.gene.append(gene)  # This will append the gene from the new individual
        new_individual.fit_value = rosenbrock_minimisation_function(
            new_individual)
        mutation_OS.append(new_individual)  # This will append the new_individual from the mutation offspring

    return mutation_OS
```

Figure 56 - Mutation for Rosenbrock

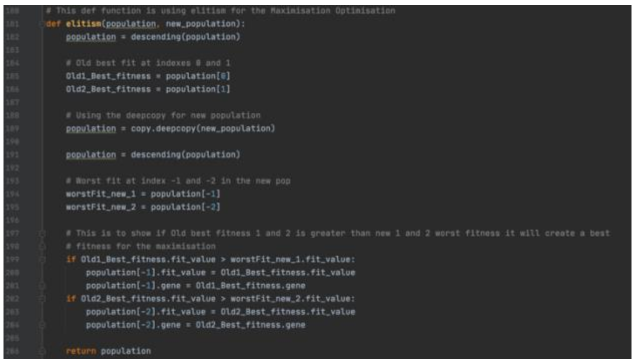
Then I have a def function for descending and sorting out the value of the individual. This is used on all optimisations.



```
# For this def function I will be Descending and sort it based on the individual fitness
def descending(population):
    def sort(individual):
        return individual.fit_value

    population.sort(key=sort, reverse=True)

    return population
```

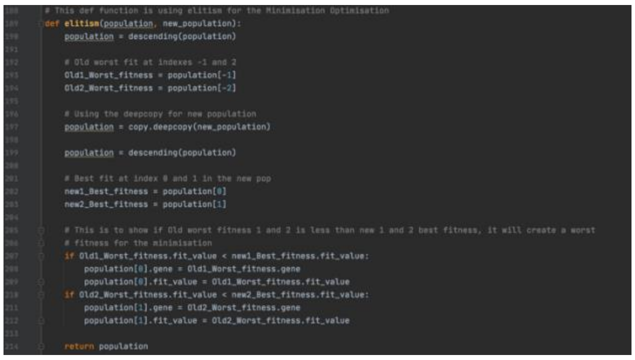
Figure 57 - a def function that descends based on the individual fitness

Afterwards, I have added elitism. As stated in my research background already, the functions are different for minimisation and maximisation.



```
# This def function is using elitism for the Maximisation Optimisation
def elitism(population, new_population):
    population = descending(population)

    # Old best fit at indexes 0 and 1
    Old1_Best_fitness = population[0]
    Old2_Best_fitness = population[1]

    # Using the deepcopy for new population
    population = copy.deepcopy(new_population)

    population = descending(population)

    # Worst fit at index -1 and -2 in the new pop
    worstFit_new_1 = population[-1]
    worstFit_new_2 = population[-2]

    # This is to show if Old best fitness 1 and 2 is greater than new 1 and 2 worst fitness it will create a best
    # fitness for the maximisation
    if Old1_Best_fitness.fit_value > worstFit_new_1.fit_value:
        population[-1].fit_value = Old1_Best_fitness.fit_value
        population[-1].gene = Old1_Best_fitness.gene
    if Old2_Best_fitness.fit_value > worstFit_new_2.fit_value:
        population[-2].fit_value = Old2_Best_fitness.fit_value
        population[-2].gene = Old2_Best_fitness.gene

    return population
```

Figure 58 - Maximisation for Elitism

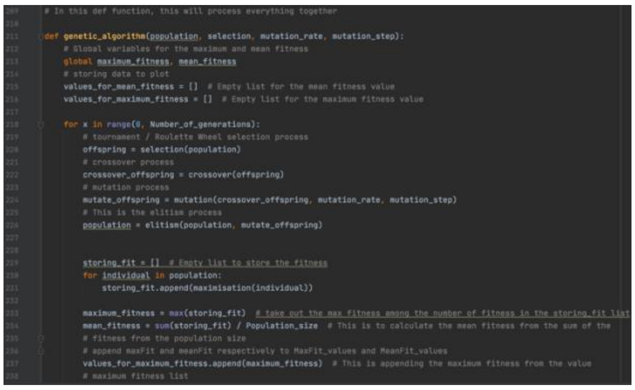


```
# This def function is using elitism for the Minimisation Optimisation
def elitism(population, new_population):
    population = descending(population)

    # Old worst fit at indexes -1 and 2
    Old1_Worst_fitness = population[-1]
    Old2_Worst_fitness = population[-2]

    # Using the deepcopy for new population
    population = copy.deepcopy(new_population)

    population = descending(population)

    # Best fit at index 0 and 1 in the new pop
    new1_Best_fitness = population[0]
    new2_Best_fitness = population[1]

    # This is to show if Old worst fitness 1 and 2 is less than new 1 and 2 best fitness, it will create a worst
    # fitness for the minimisation
    if Old1_Worst_fitness.fit_value < new1_Best_fitness.fit_value:
        population[0].gene = Old1_Worst_fitness.gene
        population[0].fit_value = Old1_Worst_fitness.fit_value
    if Old2_Worst_fitness.fit_value < new2_Best_fitness.fit_value:
        population[1].gene = Old2_Worst_fitness.gene
        population[1].fit_value = Old2_Worst_fitness.fit_value

    return population
```

Figure 59 - Minimisation for Elitism

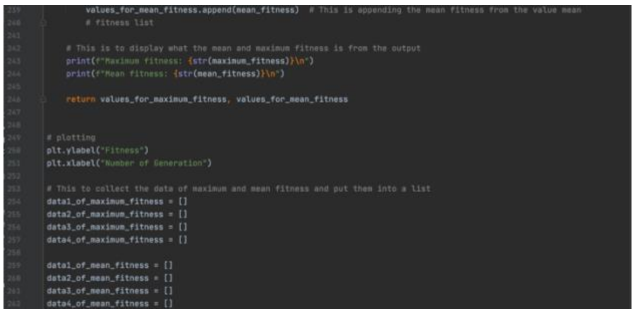
Moreover, I have added a def function called Genetic Algorithm for my maximisation and minimisation.

MAXIMISATION:



```
# In this def function, this will process everything together

def genetic_algorithm(population, selection, mutation_rate, mutation_step):
    # Global variables for the maximum and mean fitness
    global maximum_fitness, mean_fitness
    # storing data to plot
    values_for_mean_fitness = []  # Empty list for the mean fitness value
    values_for_maximum_fitness = []  # Empty list for the maximum fitness value

    for x in range(0, Number_of_generations):
        # tournament / Roulette Wheel selection process
        offspring = selection(population)
        # crossover process
        crossover_offspring = crossover(offspring)
        # mutation process
        mutate_offspring = mutation(crossover_offspring, mutation_rate, mutation_step)
        # This is the elitism process
        population = elitism(population, mutate_offspring)


        storing_fit = []  # Empty list to store the fitness
        for individual in population:
            storing_fit.append(maximisation(individual))

        maximum_fitness = max(storing_fit)  # take out the max fitness among the number of fitness in the storing_fit list
        mean_fitness = sum(storing_fit) / Population_size  # This is to calculate the mean fitness from the sum of the
        # fitness from the population size
        # append maxFit and meanFit respectively to MaxFit_values and MeanFit_values
        values_for_maximum_fitness.append(maximum_fitness)  # This is appending the maximum fitness from the value
        # maximum fitness list
```

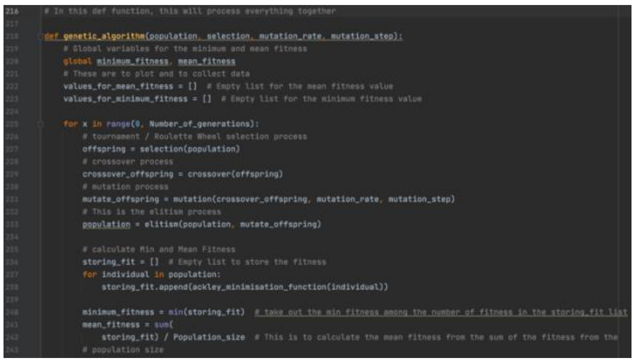
Figure 60 - Genetic Algorithm function for maximisation



```
        values_for_mean_fitness.append(mean_fitness)  # This is appending the mean fitness from the value mean
        # fitness list

    # This is to display what the mean and maximum fitness is from the output
    print(f"Maximum fitness: {str(maximum_fitness)}\n")
    print(f"Mean fitness: {str(mean_fitness)}\n")

    return values_for_maximum_fitness, values_for_mean_fitness


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

# This to collect the data of maximum and mean fitness and put them into a list
data1_of_maximum_fitness = []
data2_of_maximum_fitness = []
data3_of_maximum_fitness = []
data4_of_maximum_fitness = []

data1_of_mean_fitness = []
data2_of_mean_fitness = []
data3_of_mean_fitness = []
data4_of_mean_fitness = []
```

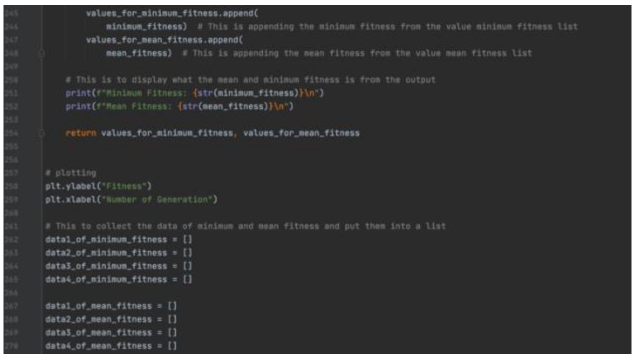
Figure 61 - Cond. of Genetic Algorithm for maximisation

THE 2 MINIMISATION FUNCTIONS ARE THE SAME:



```
# In this def function, this will process everything together

def genetic_algorithm(population, selection, mutation_rate, mutation_step):
    # Global variables for the minimum and mean fitness
    global minimum_fitness, mean_fitness
    # These are to plot and to collect data
    values_for_mean_fitness = []  # Empty list for the mean fitness value
    values_for_minimum_fitness = []  # Empty list for the minimum fitness value

    for x in range(0, Number_of_generations):
        # tournament / Roulette Wheel selection process
        offspring = selection(population)
        # crossover process
        crossover_offspring = crossover(offspring)
        # mutation process
        mutate_offspring = mutation(crossover_offspring, mutation_rate, mutation_step)
        # This is the elitism process
        population = elitism(population, mutate_offspring)

        # calculate Min and Mean Fitness
        storing_fit = []  # Empty list to store the fitness
        for individual in population:
            storing_fit.append(ackley_minimisation_function(individual))

        minimum_fitness = min(storing_fit)  # take out the min fitness among the number of fitness in the storing_fit list
        mean_fitness = sum(
            storing_fit) / Population_size  # This is to calculate the mean fitness from the sum of the fitness from the
        # population size
```

Figure 62 - Minimisation for Genetic Algorithm



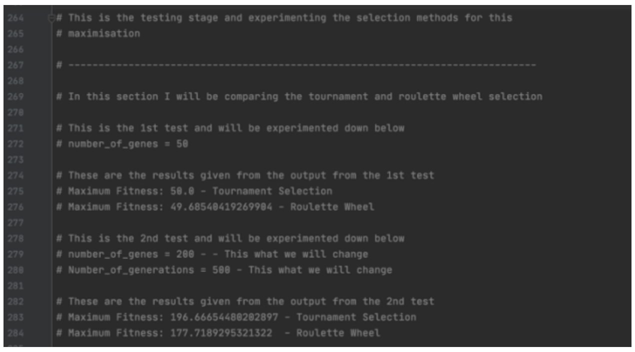
```python
        values_for_minimum_fitness.append(
            minimum_fitness)  # This is appending the minimum fitness from the value minimum fitness list
        values_for_mean_fitness.append(
            mean_fitness)  # This is appending the mean fitness from the value mean fitness list

    # This is to display what the mean and minimum fitness is from the output
    print(f"Minimum Fitness: {str(minimum_fitness)}\n")
    print(f"Mean Fitness: {str(mean_fitness)}\n")

    return values_for_minimum_fitness, values_for_mean_fitness


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

# This to collect the data of minimum and mean fitness and put them into a list
data1_of_minimum_fitness = []
data2_of_minimum_fitness = []
data3_of_minimum_fitness = []
data4_of_minimum_fitness = []

data1_of_mean_fitness = []
data2_of_mean_fitness = []
data3_of_mean_fitness = []
data4_of_mean_fitness = []
```

Figure 63 - Cond. for GA for minimisation

Then it goes to the print statement and experimentation part where I show my results.

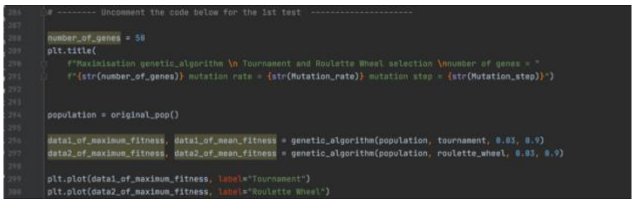
MAXIMISATION:

The 1st test and 2nd test is to compare the two selection methods and is stated in Figure 10 and Figure 11.



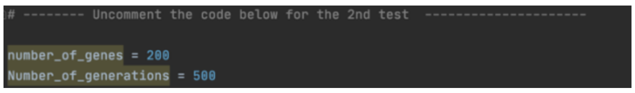
```python
# This is the testing stage and experimenting the selection methods for this
# maximisation

# --------------------------------------------------------------------------------

# In this section I will be comparing the tournament and roulette wheel selection

# This is the 1st test and will be experimented down below
# number_of_genes = 50

# These are the results given from the output from the 1st test
# Maximum Fitness: 50.0 - Tournament Selection
# Maximum Fitness: 49.68540419269904 - Roulette Wheel

# This is the 2nd test and will be experimented down below
# number_of_genes = 200 - - This what we will change
# Number_of_generations = 500 - This what we will change

# These are the results given from the output from the 2nd test
# Maximum Fitness: 196.66654480202897 - Tournament Selection
# Maximum Fitness: 177.7189295321322  - Roulette Wheel
```

Figure 64 - Comments on my experiment and results for 1st and 2nd test



```python
# -------- Uncomment the code below for the 1st test  ---------------------

number_of_genes = 50
plt.title(
    f"Maximisation genetic_algorithm \n Tournament and Roulette Wheel selection \nnumber of genes = "
    f"{str(number_of_genes)} mutation rate = {str(Mutation_rate)} mutation step = {str(Mutation_step)}")


population = original_pop()

data1_of_maximum_fitness, data1_of_mean_fitness = genetic_algorithm(population, tournament, 0.03, 0.9)
data2_of_maximum_fitness, data2_of_mean_fitness = genetic_algorithm(population, roulette_wheel, 0.03, 0.9)

plt.plot(data1_of_maximum_fitness, label="Tournament")
plt.plot(data2_of_maximum_fitness, label="Roulette Wheel")
```

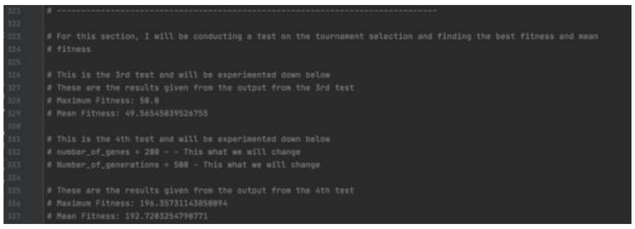
Figure 65 - 1st test



```python
# -------- Uncomment the code below for the 2nd test  ---------------------

number_of_genes = 200
Number_of_generations = 500
```

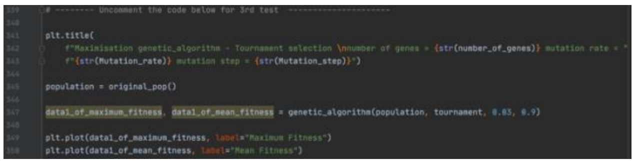
Figure 66 - This is what I changed by adding these two variables onto the same code from 1st test

Then for the 3rd and 4th tests, I have experimented with the max fitness and mean fitness for Tournament Selection. These graphs are stated in Figures 12 and 13.



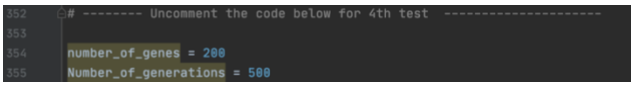
```python
# --------------------------------------------------------------------------------

# For this section, I will be conducting a test on the tournament selection and finding the best fitness and mean
# fitness

# This is the 3rd test and will be experimented down below
# These are the results given from the output from the 3rd test
# Maximum Fitness: 50.0
# Mean Fitness: 49.56545039526755

# This is the 4th test and will be experimented down below
# number_of_genes = 200 - - This what we will change
# Number_of_generations = 500 - This what we will change

# These are the results given from the output from the 4th test
# Maximum Fitness: 196.35731143858894
# Mean Fitness: 192.7283254790771
```

Figure 67 - Comments on my experiment and results for 3rd test and 4th test



```python
# -------- Uncomment the code below for 3rd test  ---------------------

plt.title(
    f"Maximisation genetic_algorithm - Tournament selection \nnumber of genes = {str(number_of_genes)} mutation rate = "
    f"{str(Mutation_rate)} mutation step = {str(Mutation_step)}")

population = original_pop()

data1_of_maximum_fitness, data1_of_mean_fitness = genetic_algorithm(population, tournament, 0.03, 0.9)

plt.plot(data1_of_maximum_fitness, label="Maximum Fitness")
plt.plot(data1_of_mean_fitness, label="Mean Fitness")
```

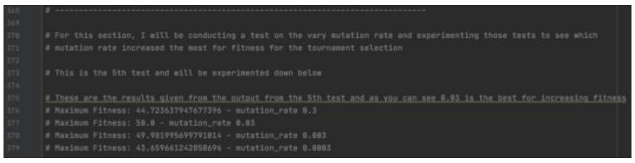
Figure 68 - 3rd test



```python
# -------- Uncomment the code below for 4th test  ---------------------

number_of_genes = 200
Number_of_generations = 500
```

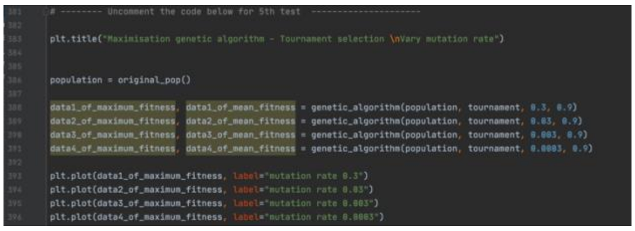
Figure 69 - This is what I changed by adding these two variables onto the same code from 3rd test

For my 5th test, I did a mutation rate for my Tournament Selection and the graph is stated in Figure 14.



```python
# --------------------------------------------------------------------------------

# For this section, I will be conducting a test on the vary mutation rate and experimenting those tests to see which
# mutation rate increased the most for fitness for the tournament selection

# This is the 5th test and will be experimented down below

# These are the results given from the output from the 5th test and as you can see 0.03 is the best for increasing fitness
# Maximum Fitness: 44.723637947677396 - mutation_rate 0.3
# Maximum Fitness: 50.0 - mutation_rate 0.03
# Maximum Fitness: 49.981995699791814 - mutation_rate 0.003
# Maximum Fitness: 43.659661242858696 - mutation_rate 0.0003
```

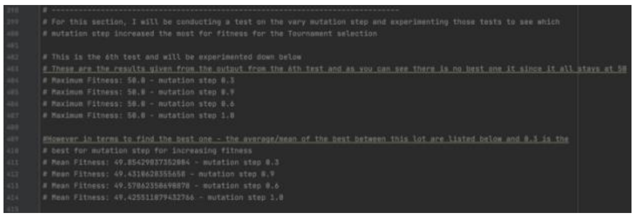
Figure 70 - Comments on my experiment and results for the 5th test



```python
# -------- Uncomment the code below for 5th test  ---------------------

plt.title("Maximisation genetic algorithm - Tournament selection \nVary mutation rate")


population = original_pop()

data1_of_maximum_fitness, data1_of_mean_fitness = genetic_algorithm(population, tournament, 0.3, 0.9)
data2_of_maximum_fitness, data2_of_mean_fitness = genetic_algorithm(population, tournament, 0.03, 0.9)
data3_of_maximum_fitness, data3_of_mean_fitness = genetic_algorithm(population, tournament, 0.003, 0.9)
data4_of_maximum_fitness, data4_of_mean_fitness = genetic_algorithm(population, tournament, 0.0003, 0.9)

plt.plot(data1_of_maximum_fitness, label="mutation rate 0.3")
plt.plot(data2_of_maximum_fitness, label="mutation rate 0.03")
plt.plot(data3_of_maximum_fitness, label="mutation rate 0.003")
plt.plot(data4_of_maximum_fitness, label="mutation rate 0.0003")
```

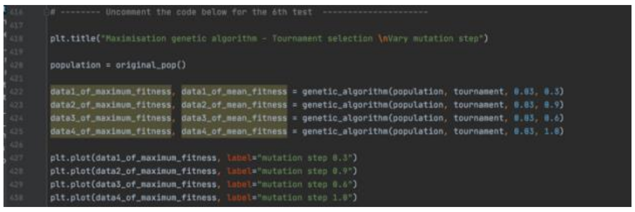
Figure 71 - 5th test

For my 6th test, I did the mutation step. The graph is in Figure 16.



```python
# ---------------------------------------------------------------------------------
# For this section, I will be conducting a test on the vary mutation step and experimenting those tests to see which
# mutation step increased the most for fitness for the Tournament selection

# This is the 6th test and will be experimented down below
# These are the results given from the output from the 6th test and as you can see there is no best one it since it all stays at 50
# Maximum Fitness: 50.0 - mutation step 0.3
# Maximum Fitness: 50.0 - mutation step 0.9
# Maximum Fitness: 50.0 - mutation step 0.6
# Maximum Fitness: 50.0 - mutation step 1.0

#However in terms to find the best one - the average/mean of the best between this lot are listed below and 0.3 is the
# best for mutation step for increasing fitness
# Mean Fitness: 49.85429837352884 - mutation step 0.3
# Mean Fitness: 49.43186283555658 - mutation step 0.9
# Mean Fitness: 49.57862358698878 - mutation step 0.6
# Mean Fitness: 49.42551187943276 6 - mutation step 1.0
```

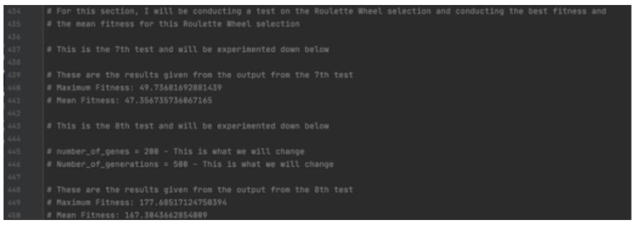
Figure 72 - Comments on my experiment and results for the 6th test



```python
# -------- Uncomment the code below for the 6th test  ---------------------

plt.title("Maximisation genetic algorithm - Tournament selection \nVary mutation step")

population = original_pop()

data1_of_maximum_fitness, data1_of_mean_fitness = genetic_algorithm(population, tournament, 0.03, 0.3)
data2_of_maximum_fitness, data2_of_mean_fitness = genetic_algorithm(population, tournament, 0.03, 0.9)
data3_of_maximum_fitness, data3_of_mean_fitness = genetic_algorithm(population, tournament, 0.03, 0.6)
data4_of_maximum_fitness, data4_of_mean_fitness = genetic_algorithm(population, tournament, 0.03, 1.0)

plt.plot(data1_of_maximum_fitness, label="mutation step 0.3")
plt.plot(data2_of_maximum_fitness, label="mutation step 0.9")
plt.plot(data3_of_maximum_fitness, label="mutation step 0.6")
plt.plot(data4_of_maximum_fitness, label="mutation step 1.0")
```

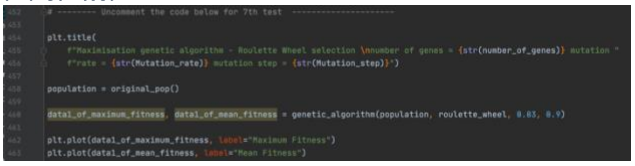
Figure 73 - 6th test

For my 7th and 8th test, I have experimented with the average and maximum fitness for RW. These graphs are stated in Figures 18 and 19.



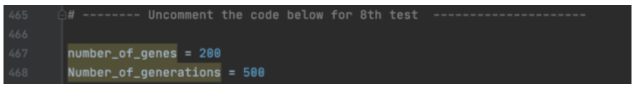
```python
# For this section, I will be conducting a test on the Roulette Wheel selection and conducting the best fitness and
# the mean fitness for this Roulette Wheel selection

# This is the 7th test and will be experimented down below

# These are the results given from the output from the 7th test
# Maximum Fitness: 49.73681692881439
# Mean Fitness: 47.356735736867165

# This is the 8th test and will be experimented down below

# number_of_genes = 200 - This is what we will change
# Number_of_generations = 500 - This is what we will change

# These are the results given from the output from the 8th test
# Maximum Fitness: 177.68517124750394
# Mean Fitness: 167.3043662854089
```

Figure 74 - Comments on my experiment and results for the 7th and 8th test



```python
# -------- Uncomment the code below for 7th test  ---------------------

plt.title(
    f"Maximisation genetic algorithm - Roulette Wheel selection \nnumber of genes = {str(number_of_genes)} mutation "
    f"rate = {str(Mutation_rate)} mutation step = {str(Mutation_step)}")

population = original_pop()

data1_of_maximum_fitness, data1_of_mean_fitness = genetic_algorithm(population, roulette_wheel, 0.03, 0.9)

plt.plot(data1_of_maximum_fitness, label="Maximum Fitness")
plt.plot(data1_of_mean_fitness, label="Mean Fitness")
```

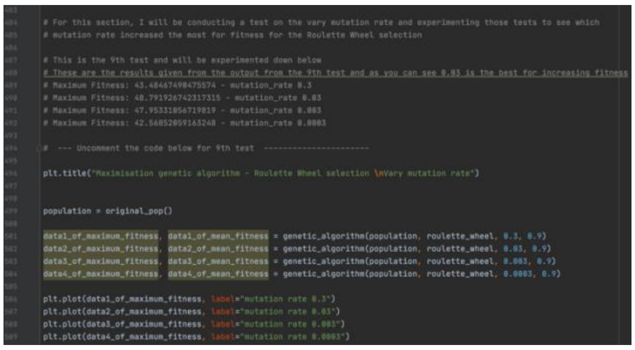
Figure 75 - 7th test



```python
# -------- Uncomment the code below for 8th test  ---------------------

number_of_genes = 200
Number_of_generations = 500
```

Figure 76 - This is what I changed by adding these two variables onto the same code from the 7th test

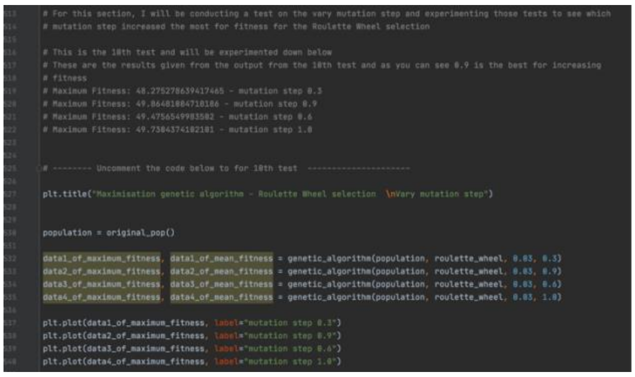
For my 9th test, I did the mutation rate for the Roulette Wheel. These results are in Figure 15.



```python
# For this section, I will be conducting a test on the vary mutation rate and experimenting those tests to see which
# mutation rate increased the most for fitness for the Roulette Wheel selection

# This is the 9th test and will be experimented down below
# These are the results given from the output from the 9th test and as you can see 0.03 is the best for increasing fitness
# Maximum Fitness: 43.48467498475574 - mutation_rate 0.3
# Maximum Fitness: 48.791926742317315 - mutation_rate 0.03
# Maximum Fitness: 47.95331856719819 - mutation_rate 0.003
# Maximum Fitness: 42.56852859163248 - mutation_rate 0.0003

#  --- Uncomment the code below for 9th test  ---------------------

plt.title("Maximisation genetic algorithm - Roulette Wheel selection \nVary mutation rate")


population = original_pop()

data1_of_maximum_fitness, data1_of_mean_fitness = genetic_algorithm(population, roulette_wheel, 0.3, 0.9)
data2_of_maximum_fitness, data2_of_mean_fitness = genetic_algorithm(population, roulette_wheel, 0.03, 0.9)
data3_of_maximum_fitness, data3_of_mean_fitness = genetic_algorithm(population, roulette_wheel, 0.003, 0.9)
data4_of_maximum_fitness, data4_of_mean_fitness = genetic_algorithm(population, roulette_wheel, 0.0003, 0.9)

plt.plot(data1_of_maximum_fitness, label="mutation rate 0.3")
plt.plot(data2_of_maximum_fitness, label="mutation rate 0.03")
plt.plot(data3_of_maximum_fitness, label="mutation rate 0.003")
plt.plot(data4_of_maximum_fitness, label="mutation rate 0.0003")
```

Figure 77 - This is to experiment with the mutation rate of the Roulette Wheel. 9th test

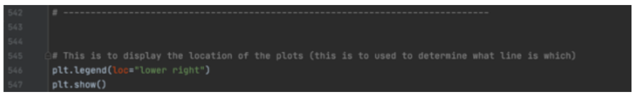
For my last 10 tests, I experimented with the Roulette Wheel for the mutation step. These results are in Figure 17.



```python
# For this section, I will be conducting a test on the vary mutation step and experimenting those tests to see which
# mutation step increased the most for fitness for the Roulette Wheel selection

# This is the 10th test and will be experimented down below
# These are the results given from the output from the 10th test and as you can see 0.9 is the best for increasing
# fitness
# Maximum Fitness: 48.275278639417465 - mutation step 0.3
# Maximum Fitness: 49.86481884718186 - mutation step 0.9
# Maximum Fitness: 49.4756549983582 - mutation step 0.6
# Maximum Fitness: 49.7384374182181 - mutation step 1.0


# -------- Uncomment the code below to for 10th test  ---------------------

plt.title("Maximisation genetic algorithm - Roulette Wheel selection  \nVary mutation step")


population = original_pop()

data1_of_maximum_fitness, data1_of_mean_fitness = genetic_algorithm(population, roulette_wheel, 0.03, 0.3)
data2_of_maximum_fitness, data2_of_mean_fitness = genetic_algorithm(population, roulette_wheel, 0.03, 0.9)
data3_of_maximum_fitness, data3_of_mean_fitness = genetic_algorithm(population, roulette_wheel, 0.03, 0.6)
data4_of_maximum_fitness, data4_of_mean_fitness = genetic_algorithm(population, roulette_wheel, 0.03, 1.0)

plt.plot(data1_of_maximum_fitness, label="mutation step 0.3")
plt.plot(data2_of_maximum_fitness, label="mutation step 0.9")
plt.plot(data3_of_maximum_fitness, label="mutation step 0.6")
plt.plot(data4_of_maximum_fitness, label="mutation step 1.0")
```

Figure 78 - 10th test

Lastly, I have added a plot to display where this is placed. This is displayed on the lower right for the maximisation fitness.

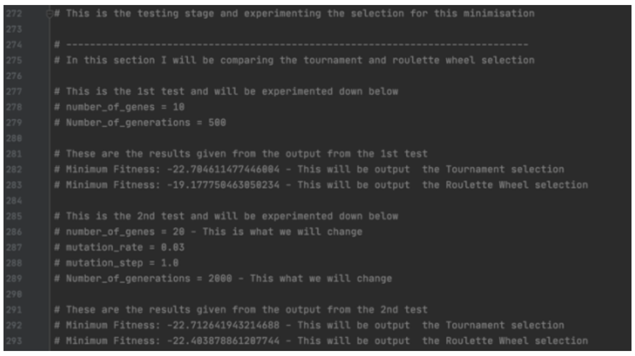


```python
# ---------------------------------------------------------------------------------


# This is to display the location of the plots (this is to used to determine what line is which)
plt.legend(loc="lower right")
plt.show()
```

Figure 79 - Display plot for maximisation

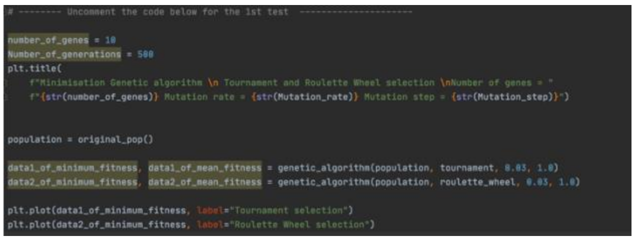
ACKLEY OPTIMISATION:

The 1st test and 2nd test is to compare the two selection methods and is stated in Figure 21 and 22.



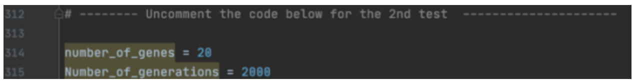
```
# This is the testing stage and experimenting the selection for this minimisation

# ---------------------------------------------------------------------------
# In this section I will be comparing the tournament and roulette wheel selection

# This is the 1st test and will be experimented down below
# number_of_genes = 10
# Number_of_generations = 500

# These are the results given from the output from the 1st test
# Minimum Fitness: -22.704611477446004 - This will be output  the Tournament selection
# Minimum Fitness: -19.177750463050234 - This will be output  the Roulette Wheel selection

# This is the 2nd test and will be experimented down below
# number_of_genes = 20 - This is what we will change
# mutation_rate = 0.03
# mutation_step = 1.0
# Number_of_generations = 2000 - This what we will change

# These are the results given from the output from the 2nd test
# Minimum Fitness: -22.712641943214688 - This will be output  the Tournament selection
# Minimum Fitness: -22.403878861207744 - This will be output  the Roulette Wheel selection
```

Figure 80 - Comments on my experiment and results for 1st and 2nd test



```
# -------- Uncomment the code below for the 1st test  --------------------

number_of_genes = 10
Number_of_generations = 500
plt.title(
    f"Minimisation Genetic algorithm \n Tournament and Roulette Wheel selection \nNumber of genes = "
    f"{str(number_of_genes)} Mutation rate = {str(Mutation_rate)} Mutation step = {str(Mutation_step)}")


population = original_pop()

data1_of_minimum_fitness, data1_of_mean_fitness = genetic_algorithm(population, tournament, 0.03, 1.0)
data2_of_minimum_fitness, data2_of_mean_fitness = genetic_algorithm(population, roulette_wheel, 0.03, 1.0)

plt.plot(data1_of_minimum_fitness, label="Tournament selection")
plt.plot(data2_of_minimum_fitness, label="Roulette Wheel selection")
```

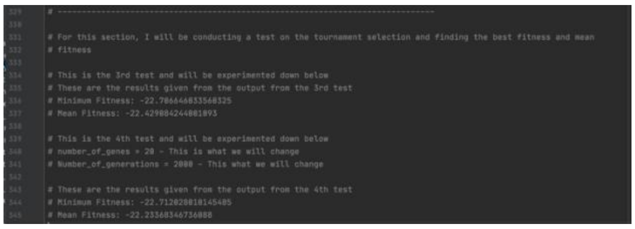
Figure 81 - 1st test



```
# -------- Uncomment the code below for the 2nd test  --------------------

number_of_genes = 20
Number_of_generations = 2000
```

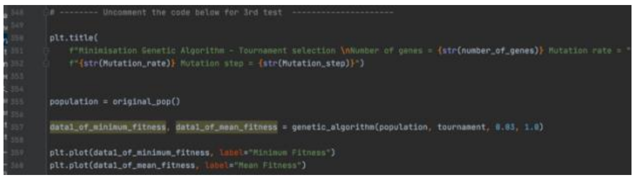
Figure 82 - This is what I changed by adding these two variables onto the same code from 1st test

This is for my 3rd and 4th test. This is to experiment with my Tournament Selection for the mean and minimum fitness. This graph is in Figures 23 and 24.



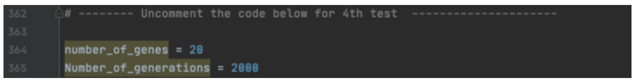
```
# ---------------------------------------------------------------------------

# For this section, I will be conducting a test on the tournament selection and finding the best fitness and mean
# fitness

# This is the 3rd test and will be experimented down below
# These are the results given from the output from the 3rd test
# Minimum Fitness: -22.706646833568325
# Mean Fitness: -22.429884244081893

# This is the 4th test and will be experimented down below
# number_of_genes = 20 - This is what we will change
# Number_of_generations = 2000 - This what we will change

# These are the results given from the output from the 4th test
# Minimum Fitness: -22.712028010145405
# Mean Fitness: -22.23368346736088
```

Figure 83 - Comments on my experiment and results for 3rd and 4th test



```
# -------- Uncomment the code below for 3rd test  --------------------

plt.title(
    f"Minimisation Genetic Algorithm - Tournament selection \nNumber of genes = {str(number_of_genes)} Mutation rate = "
    f"{str(Mutation_rate)} Mutation step = {str(Mutation_step)}")


population = original_pop()

data1_of_minimum_fitness, data1_of_mean_fitness = genetic_algorithm(population, tournament, 0.03, 1.0)

plt.plot(data1_of_minimum_fitness, label="Minimum Fitness")
plt.plot(data1_of_mean_fitness, label="Mean Fitness")
```

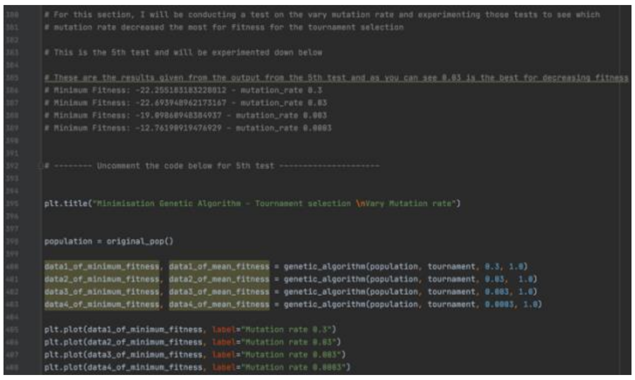
Figure 84 - 3rd test



```
# -------- Uncomment the code below for 4th test  --------------------

number_of_genes = 20
Number_of_generations = 2000
```

Figure 85 - This is what I changed by adding these two variables onto the same code from 3rd test

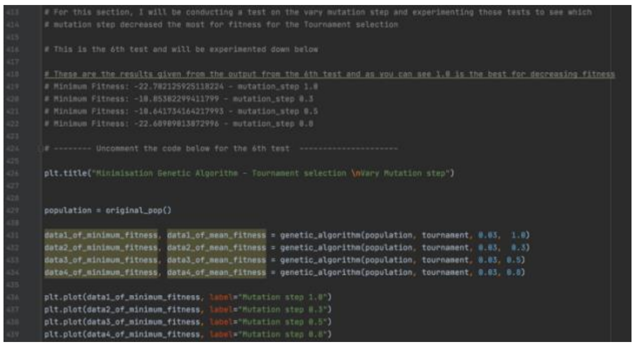
For my 5th test, I did a mutation rate for the Tournament Selection. This shows in Figure 27.



```
# For this section, I will be conducting a test on the vary mutation rate and experimenting those tests to see which
# mutation rate decreased the most for fitness for the tournament selection

# This is the 5th test and will be experimented down below

# These are the results given from the output from the 5th test and as you can see 0.03 is the best for decreasing fitness
# Minimum Fitness: -22.255183183228012 - mutation_rate 0.3
# Minimum Fitness: -22.693948962173167 - mutation_rate 0.03
# Minimum Fitness: -19.09868948384937 - mutation_rate 0.003
# Minimum Fitness: -12.76198919476929 - mutation_rate 0.0003


# -------- Uncomment the code below for 5th test --------------------


plt.title("Minimisation Genetic Algorithm - Tournament selection \nVary Mutation rate")


population = original_pop()

data1_of_minimum_fitness, data1_of_mean_fitness = genetic_algorithm(population, tournament, 0.3, 1.0)
data2_of_minimum_fitness, data2_of_mean_fitness = genetic_algorithm(population, tournament, 0.03,  1.0)
data3_of_minimum_fitness, data3_of_mean_fitness = genetic_algorithm(population, tournament, 0.003, 1.0)
data4_of_minimum_fitness, data4_of_mean_fitness = genetic_algorithm(population, tournament, 0.0003, 1.0)

plt.plot(data1_of_minimum_fitness, label="Mutation rate 0.3")
plt.plot(data2_of_minimum_fitness, label="Mutation rate 0.03")
plt.plot(data3_of_minimum_fitness, label="Mutation rate 0.003")
plt.plot(data4_of_minimum_fitness, label="Mutation rate 0.0003")
```

Figure 86 - Ackley mutation rate Tournament Selection experiment and results

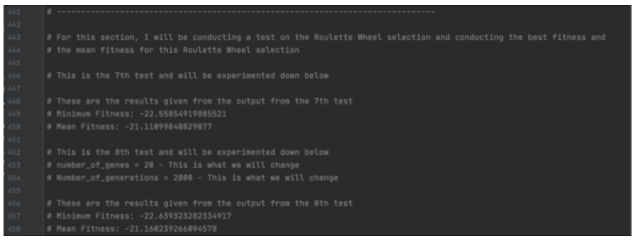
For my 6th test, I also did a mutation step for my Tournament. This shows in Figure 29.



```
# For this section, I will be conducting a test on the vary mutation step and experimenting those tests to see which
# mutation step decreased the most for fitness for the Tournament selection

# This is the 6th test and will be experimented down below

# These are the results given from the output from the 6th test and as you can see 1.0 is the best for decreasing fitness
# Minimum Fitness: -22.782125925118224 - mutation_step 1.0
# Minimum Fitness: -10.85382299411799 - mutation_step 0.3
# Minimum Fitness: -10.641734164217993 - mutation_step 0.5
# Minimum Fitness: -22.68989813872996 - mutation_step 0.8

# -------- Uncomment the code below for the 6th test  --------------------

plt.title("Minimisation Genetic Algorithm - Tournament selection \nVary Mutation step")


population = original_pop()

data1_of_minimum_fitness, data1_of_mean_fitness = genetic_algorithm(population, tournament, 0.03,  1.0)
data2_of_minimum_fitness, data2_of_mean_fitness = genetic_algorithm(population, tournament, 0.03,  0.3)
data3_of_minimum_fitness, data3_of_mean_fitness = genetic_algorithm(population, tournament, 0.03, 0.5)
data4_of_minimum_fitness, data4_of_mean_fitness = genetic_algorithm(population, tournament, 0.03, 0.8)

plt.plot(data1_of_minimum_fitness, label="Mutation step 1.0")
plt.plot(data2_of_minimum_fitness, label="Mutation step 0.3")
plt.plot(data3_of_minimum_fitness, label="Mutation step 0.5")
plt.plot(data4_of_minimum_fitness, label="Mutation step 0.8")
```

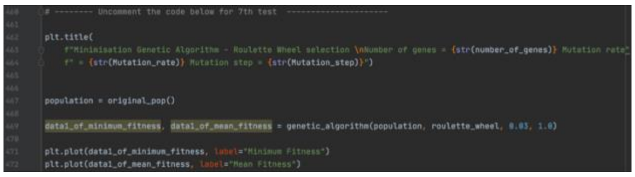
Figure 87 - Ackley mutation step for Tournament selection experiment and results

The 7th and 8th test is finding the best minimum and mean fitness for the Roulette Wheel. These graphs are in Figures 25 and 26.



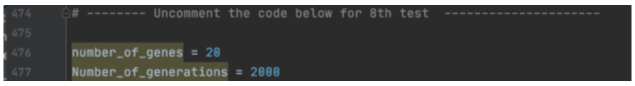
```
# ---------------------------------------------------------------------------

# For this section, I will be conducting a test on the Roulette Wheel selection and conducting the best fitness and
# the mean fitness for this Roulette Wheel selection

# This is the 7th test and will be experimented down below

# These are the results given from the output from the 7th test
# Minimum Fitness: -22.55854919885521
# Mean Fitness: -21.11899840829077

# This is the 8th test and will be experimented down below
# number_of_genes = 20 - This is what we will change
# Number_of_generations = 2000 - This is what we will change

# These are the results given from the output from the 8th test
# Minimum Fitness: -22.639323282334917
# Mean Fitness: -21.160239266894578
```

Figure 88 - Comments on my experiment and results for the 7th and 8th test



```
# -------- Uncomment the code below for 7th test  --------------------

plt.title(
    f"Minimisation Genetic Algorithm - Roulette Wheel selection \nNumber of genes = {str(number_of_genes)} Mutation rate"
    f" = {str(Mutation_rate)} Mutation step = {str(Mutation_step)}")


population = original_pop()

data1_of_minimum_fitness, data1_of_mean_fitness = genetic_algorithm(population, roulette_wheel, 0.03, 1.0)

plt.plot(data1_of_minimum_fitness, label="Minimum Fitness")
plt.plot(data1_of_mean_fitness, label="Mean Fitness")
```

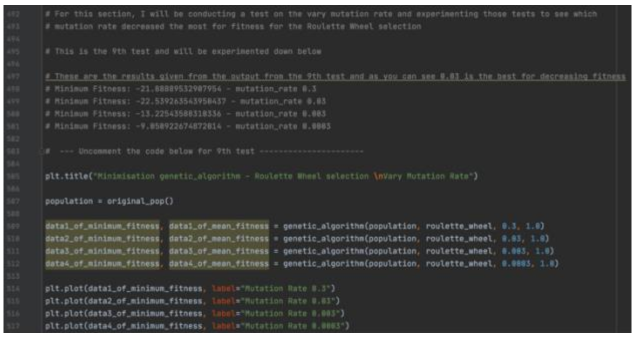
Figure 89 - 7th test



```
# -------- Uncomment the code below for 8th test  --------------------

number_of_genes = 20
Number_of_generations = 2000
```

Figure 90 - This is what I changed by adding these two variables onto the same code the 8th test

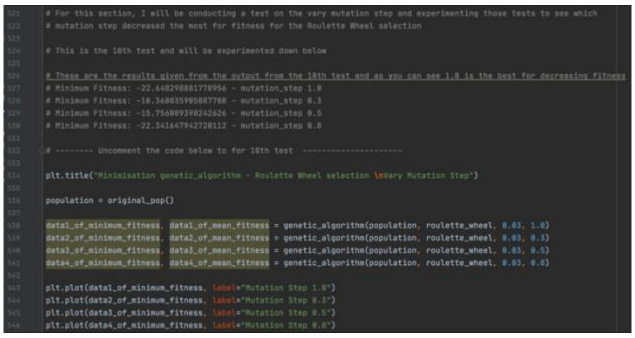
For my 9th Test, I did the mutation rate for the Roulette Wheel. This is a source in Figure 28.



```
# For this section, I will be conducting a test on the vary mutation rate and experimenting those tests to see which
# mutation rate decreased the most for fitness for the Roulette Wheel selection

# This is the 9th test and will be experimented down below

# These are the results given from the output from the 9th test and as you can see 0.03 is the best for decreasing fitness
# Minimum Fitness: -21.88889532907954 - mutation_rate 0.3
# Minimum Fitness: -22.539263543950437 - mutation_rate 0.03
# Minimum Fitness: -13.22543588318336 - mutation_rate 0.003
# Minimum Fitness: -9.858922674872814 - mutation_rate 0.0003

#  --- Uncomment the code below for 9th test ---------------------

plt.title("Minimisation genetic_algorithm - Roulette Wheel selection \nVary Mutation Rate")

population = original_pop()

data1_of_minimum_fitness, data1_of_mean_fitness = genetic_algorithm(population, roulette_wheel, 0.3, 1.0)
data2_of_minimum_fitness, data2_of_mean_fitness = genetic_algorithm(population, roulette_wheel, 0.03, 1.0)
data3_of_minimum_fitness, data3_of_mean_fitness = genetic_algorithm(population, roulette_wheel, 0.003, 1.0)
data4_of_minimum_fitness, data4_of_mean_fitness = genetic_algorithm(population, roulette_wheel, 0.0003, 1.0)

plt.plot(data1_of_minimum_fitness, label="Mutation Rate 0.3")
plt.plot(data2_of_minimum_fitness, label="Mutation Rate 0.03")
plt.plot(data3_of_minimum_fitness, label="Mutation Rate 0.003")
plt.plot(data4_of_minimum_fitness, label="Mutation Rate 0.0003")
```

Figure 91 - Ackley mutation rate Roulette Wheel experiment and results

For my last test. I did the mutation step for the Roulette Wheel. This is a source in Figure 30.



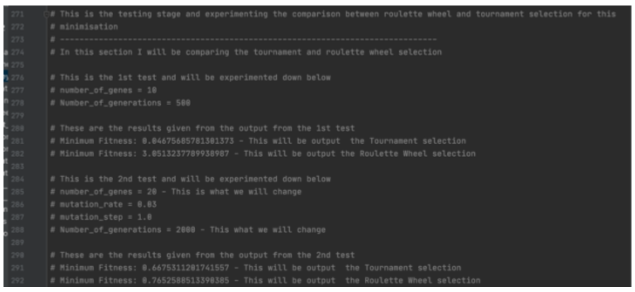
```
# For this section, I will be conducting a test on the vary mutation step and experimenting those tests to see which
# mutation step decreased the most for fitness for the Roulette Wheel selection

# This is the 10th test and will be experimented down below

# These are the results given from the output from the 10th test and as you can see 1.0 is the best for decreasing fitness
# Minimum Fitness: -22.648298881778956 - mutation_step 1.0
# Minimum Fitness: -10.368835985887708 - mutation_step 0.3
# Minimum Fitness: -15.756889398242626 - mutation_step 0.5
# Minimum Fitness: -22.341647942728112 - mutation_step 0.8

# -------- Uncomment the code below to for 10th test  --------------------

plt.title("Minimisation genetic_algorithm - Roulette Wheel selection \nVary Mutation Step")

population = original_pop()

data1_of_minimum_fitness, data1_of_mean_fitness = genetic_algorithm(population, roulette_wheel, 0.03, 1.0)
data2_of_minimum_fitness, data2_of_mean_fitness = genetic_algorithm(population, roulette_wheel, 0.03, 0.3)
data3_of_minimum_fitness, data3_of_mean_fitness = genetic_algorithm(population, roulette_wheel, 0.03, 0.5)
data4_of_minimum_fitness, data4_of_mean_fitness = genetic_algorithm(population, roulette_wheel, 0.03, 0.8)

plt.plot(data1_of_minimum_fitness, label="Mutation Step 1.0")
plt.plot(data2_of_minimum_fitness, label="Mutation Step 0.3")
plt.plot(data3_of_minimum_fitness, label="Mutation Step 0.5")
plt.plot(data4_of_minimum_fitness, label="Mutation Step 0.8")
```

Figure 92 - Ackley mutation step Tournament Selection experiment and results

ROSENBROCK OPTIMISATION:

For my Rosenbrock, since this is based on the same print statements from the Ackley function, I will just be showing my test results.

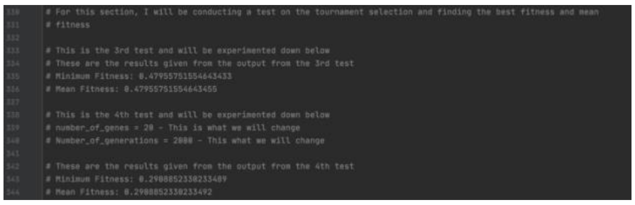
For my 1st and 2nd test, these are displayed in Figures 32 and 33. This is to show the comparison between them.



```
# This is the testing stage and experimenting the comparison between roulette wheel and tournament selection for this
# minimisation
# ---------------------------------------------------------------------------
# In this section I will be comparing the tournament and roulette wheel selection

# This is the 1st test and will be experimented down below
# number_of_genes = 10
# Number_of_generations = 500

# These are the results given from the output from the 1st test
# Minimum Fitness: 0.8467568578130137 3 - This will be output  the Tournament selection
# Minimum Fitness: 3.0513237789938987 - This will be output the Roulette Wheel selection

# This is the 2nd test and will be experimented down below
# number_of_genes = 20 - This is what we will change
# mutation_rate = 0.03
# mutation_step = 1.0
# Number_of_generations = 2000 - This what we will change

# These are the results given from the output from the 2nd test
# Minimum Fitness: 0.6675311201741557 - This will be output  the Tournament selection
# Minimum Fitness: 0.7652588513398385 - This will be output  the Roulette Wheel selection
```

Figure 93 - Comments on my experiment and results for 1st and 2nd test

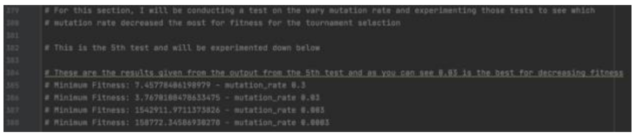
For my 3rd and 4th tests, these are displayed in Figures 34 and 35. This is to show the tournament mean and minimum fitness.



```
# For this section, I will be conducting a test on the tournament selection and finding the best fitness and mean
# fitness

# This is the 3rd test and will be experimented down below
# These are the results given from the output from the 3rd test
# Minimum Fitness: 0.47955751554643433
# Mean Fitness: 0.47955751554643455

# This is the 4th test and will be experimented down below
# number_of_genes = 20 - This is what we will change
# Number_of_generations = 2000 - This what we will change

# These are the results given from the output from the 4th test
# Minimum Fitness: 0.2988852330233489
# Mean Fitness: 0.2988852330233492
```

Figure 94 - Comments on my experiment and results for 3rd and 4th test

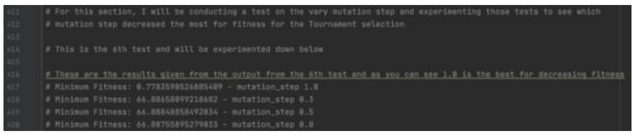
For my 5th test, this is displayed in Figure 38. This is to show the tournament mutation rate.



```
# For this section, I will be conducting a test on the vary mutation rate and experimenting those tests to see which
# mutation rate decreased the most for fitness for the tournament selection

# This is the 5th test and will be experimented down below

# These are the results given from the output from the 5th test and as you can see 0.03 is the best for decreasing fitness
# Minimum Fitness: 7.457784086198979 - mutation_rate 0.3
# Minimum Fitness: 3.7678108478633475 - mutation_rate 0.03
# Minimum Fitness: 1542911.9711373826 - mutation_rate 0.003
# Minimum Fitness: 158772.34586930278 - mutation_rate 0.0003
```

Figure 95 - 5th test of the mutation rate for Tournament selection. These are the experiment and the results

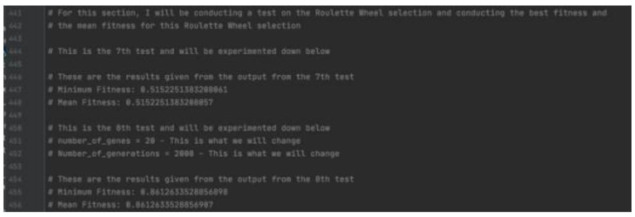
For my 6th test, I did the mutation step for the Tournament, and this is shown in Figure 40.



```
# For this section, I will be conducting a test on the vary mutation step and experimenting those tests to see which
# mutation step decreased the most for fitness for the Tournament selection

# This is the 6th test and will be experimented down below

# These are the results given from the output from the 6th test and as you can see 1.0 is the best for decreasing fitness
# Minimum Fitness: 0.7783599526885609 - mutation_step 1.0
# Minimum Fitness: 66.88658899218682 - mutation_step 0.3
# Minimum Fitness: 66.88848858492034 - mutation_step 0.5
# Minimum Fitness: 66.88755895279833 - mutation_step 0.8
```

Figure 96 - 6th test of the mutation step for Tournament selection. These are the experiment and the results

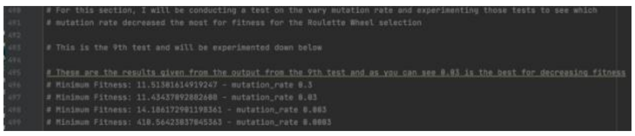
For my 7th and 8th test, I have minimum and mean fitness for the Roulette Wheel. This is shown in Figures 36 and 37.



```
# For this section, I will be conducting a test on the Roulette Wheel selection and conducting the best fitness and
# the mean fitness for this Roulette Wheel selection

# This is the 7th test and will be experimented down below

# These are the results given from the output from the 7th test
# Minimum Fitness: 0.5152251383288061
# Mean Fitness: 0.5152251383288857

# This is the 8th test and will be experimented down below
# number_of_genes = 20 - This is what we will change
# Number_of_generations = 2000 - This is what we will change

# These are the results given from the output from the 8th test
# Minimum Fitness: 0.8612633528856898
# Mean Fitness: 0.8612633528856907
```

Figure 97 - Comments on my experiment and results for the 7th and 8th test

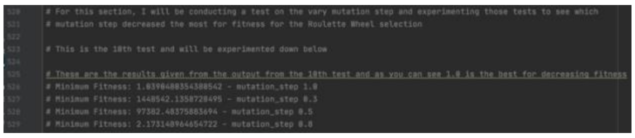
My 9th test demonstrates the mutation rate for RW. The graph is in Figure 39.



```
# For this section, I will be conducting a test on the vary mutation rate and experimenting those tests to see which
# mutation rate decreased the most for fitness for the Roulette Wheel selection

# This is the 9th test and will be experimented down below

# These are the results given from the output from the 9th test and as you can see 0.03 is the best for decreasing fitness
# Minimum Fitness: 11.51381614919247 - mutation_rate 0.3
# Minimum Fitness: 11.43437892882688 - mutation_rate 0.03
# Minimum Fitness: 14.186172981198361 - mutation_rate 0.003
# Minimum Fitness: 418.56423837845363 - mutation_rate 0.0003
```

Figure 98 - Rosenbrock mutation rate Roulette Wheel results

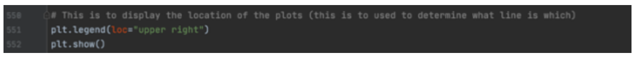
My last test demonstrates the mutation step for the Roulette Wheel. The graph is in Figure 41.



```
# For this section, I will be conducting a test on the vary mutation step and experimenting those tests to see which
# mutation step decreased the most for fitness for the Roulette Wheel selection

# This is the 10th test and will be experimented down below

# These are the results given from the output from the 10th test and as you can see 1.0 is the best for decreasing fitness
# Minimum Fitness: 1.8398488354388542 - mutation_step 1.0
# Minimum Fitness: 1448542.1358728495 - mutation_step 0.3
# Minimum Fitness: 97382.48375883694 - mutation_step 0.5
# Minimum Fitness: 2.173148964654722 - mutation_step 0.8
```

Figure 99 - Rosenbrock mutation step Roulette Wheel results

Lastly, I have added a plot to display where this is placed. This is displayed on the upper right for the minimisation fitness. These are both displayed, on both minimisation functions.



```
# This is to display the location of the plots (this is to used to determine what line is which)
plt.legend(loc="upper right")
plt.show()
```

Figure 100 - Display plot for minimisation