Artificial Intelligence II - Assignment

Arthur Milner 21035478

1 INTRODUCTION

The task I have been presented with is to demonstrate my ability to fine tune AI genetic algorithms in order to maximise their effectiveness, this is to ensure I have a deep understanding of what occurs during the execution of the algorithm and proves I could fine tune algorithms to different data sets effectively. My main approach was to select high medium and low values then slowly change those values run by run in order to get closer to an optimal fitness, all whilst using my knowledge of how the algorithm works to guide my selection of reasonable values.

2 BACKGROUND RESEARCH

The impressive advances of artificial intelligence has brought with it a discussion as unavoidable as it is exhaustive, that is the discussion of the ethics around AI. Perhaps the main basis of these discussions are how we can mitigate the potentially "disruptive" side effects of AI technologies. **(Hagendorff, 2020)** The discussions scope covers many factors, such as, protecting populations from discrimination, respecting the privacy of individuals and even environmental factors such as the pollution high cost AI algorithms might produce. Recently the European Union has involved itself in the discussion of AI ethics regulation, giving further weight to the seriousness the topic must be met with. **(Christoforaki and Beyan, 2022)** Leslie, D. gives what I believe to be a good definition for the goal of AI ethics, which is, "a set of values, principles, and techniques that employ widely accepted standards of right and wrong to guide moral conduct in the development and use of AI technologies". **(Leslie, 2019, cited by Christoforaki and Beyan, 2022)**

It is debated much of the push for ethics within AI is somewhat performative as associations such as "Partnership on AI" (a coalition of companies such as Google and Amazon to promote ethical AI) can act as a sort of scape-goat preventing any "serious commitment" as members will simply point at their involvement in such a cause as a substantial effort to enforce ethical policy. **(Hagendorff, 2020)** The application of ethics in AI has been considered "toothless", it is even argued AI ethics implementation is purposefully manipulated in order to benefit "commercial interests" and allow for further "flexibility" which would be deemed impossible should it be regulated by law. Essentially pushing for commercial success over being morally correct. **(Rességuier and Rodrigues, 2020)**

For a more specific example to showcase the importance of proper implementation of AI ethics you could look at the recruitment tool Amazon tested over a ten year period. A BBC report **(BBC - Amazon Scrapped 'Sexist AI' Tool, 2018)** states much of the data fed to the AI were from male applicants, which is perhaps the main reason the AI was skewed to prefer male candidates, consequently unfairly disregarding female candidates based on gender alone.

As well as looking bad on the company itself the algorithm would be highly immoral to utilise as it is irrefutably discriminatory against women. Whilst the use of AI in recruitment has potential to be effective ensuring the algorithm does not discriminate on areas such as race or gender is vital.

Perhaps to avoid this Amazon could have made sure the data they feed the algorithm was all complete, as some applications may miss information that others possess, in face removing any indication of gender, race, etc. would ensure a fairer analysis of the data. The company could have also used dummy data to test against any biases and identify the discrimination early on. There is also an argument for whether using AI for recruitment is ever a good idea, as the challenge of making it completely

unbiased could prove near impossible, alas you could also make a similar discourse for human judgement.

A further example could be the infamous Microsoft Tay chatbot **(Hunt, E., 2016)**, what was designed to slowly teach itself the art of "playful conversation" through its interactions over twitter, quickly devolved into incredibly offensive and racially insensitive tweets. Perhaps the biggest oversight of the Tay bot experiment is where the bot would get its data from, any user could interact with the bot and consequently people were quick to take advantage. Perhaps they could have marked certain words or sentences to be ignored by the bot and heavily regulated what tweets the bot was making part of its dataset. The bot required a much greater attempt at learning social guidelines and discrimination before being pushed out onto the social platform.

3 EXPERIMENTATION

The evolutionary algorithm I have implemented contains the parameters P to represent the number of individuals, N to represent the length of the gene individuals possess, MIN deciding the lowest value a single part of a gene can be, MAX deciding the highest value a single part of a gene can be, MUTRATE deciding the probability of mutation occurring on a part of a gene, MUTSTEP deciding the amount of change to occur if mutation is to happen (this value can be added or subtracted from the part of the gene) and finally MAXGEN which is the number of generations for a single run of the algorithm.

The fitness calculation of the algorithm is done by running the values of the individuals genes through an equation and returning the overall results per individual as that specific individuals fitness, with the overall fitness being the sum of all individuals fitness.

The algorithm will first populate the population array full of individuals with genes of random values between the MIN and MAX parameters. Once the population array has been filled tournament selection will occur. Tournament selection is the process of taking two random individuals from the population, comparing the fitness of the two, appending the parent with better fitness to a new population array and repeating this process until you are left with a new population of P size.

The next step in the algorithm is crossover, here you take two individuals at a time from your new population and at a randomly generated number in the range of 1 to N, stored as the crosspoint, you swap the values of the genes in individual 1 and 2 with each other in the range of the crosspoint to N. You are essentially randomly splitting and swapping the gene values with one another.

The penultimate step in the algorithm is mutation, here each individual has each part of its gene mutated at random. Here a mutation is changing part of the individuals gene by + or – the MUTSTEP. Essentially the values of each gene are looped through and each time, should a randomly generated number be lower than that of the MUTRATE, a mutation will occur at that part of the gene. Should + or – the MUTSTEP make the part of the genes value go lower or higher than MIN/MAX the gene value is simply set to the MIN or MAX respectively.

The final step is to re-evaluate the fitness of the population via the fitness function, the expected outcome here is for the average fitness of an individual to go down, along with the best fitness of an individual.

As the algorithm is non-deterministic I have set it to run ten times and then plot a graph showing the overall average/best fitness of these ten runs, with the results of each run plotted in transparent lines, this is to have greater confidence in the results over only showing a single run as they can vary greatly. The plots on the 3D scatterplot graphs are also the result of a ten run average.

Rosenbrock Function

To begin experimentation on the Rosenbrock function I have started with the default parameters (P = 50, N = 20, MIN = -100, MAX = 100, MUTRATE = 1/P, MAXGEN = 50, MUTSTEP = 0.1), this yielded a final average fitness of 1360097731.09. This is clearly far from ideal and I must begin to tweak these parameters in order to optimise these

fitness values.

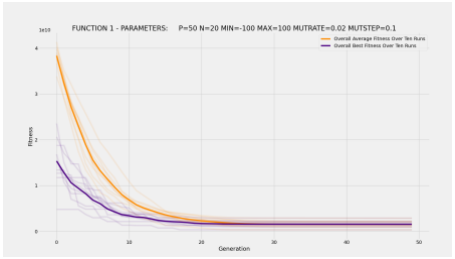

Figure 1 Rosenbrock function using the recommended starting parameters

I decided to begin by methodically modifying the mutation rate and mutation step, I firstly set-up some code which will 3D scatter plot the final average fitness of my code against given MUTRATEs/MUTSTEPs. I have chosen to begin with low medium and high values for each MUTRATE and MUTSTEP, then testing them against each other. I will then move on to ranges of random numbers dependant on the greatest combination of low/medium/high values. My reason for starting with low/medium/high values is because this means I have only 9 overall combinations to test.

I believe a good high MUTRATE value would be 0.4, anything over 0.5 is probably too high as in theory you would be mutating much of an individuals gene each generation, potentially messing up good fitness individuals as much as you improve low fitness ones. A decent medium value for MUTRATE could be 0.2 as it can still be considered a substantial MUTRATE, and finally a good low value could be the generally recommended MUTRATE value of 1/N, which in this case is 0.05. Moving towards MUTSTEP values, a good high value might be 15, this will change genes during mutation a large amount but could prove to be effective, potentially when combined with a low MUTRATE as the high change will seldom occur. A medium MUTSTEP value could be 8, this is a nice middle ground between the high of 15 and the low I will set to 1.

The low MUTSTEP gave very sub-optimal results compared to the results of the medium and high MUTSTEP, this is true for all values of MUTRATE combined with the low MUTSTEP. On the other hand, the medium and high MUTSTEP generally gave pretty even results at all MUTRATE values, the best combinations however were the medium MUTSTEP and high MUTRATE and the high MUTSTEP and medium/low MUTRATE. Perhaps the reason for these results is the fact the MIN/MAX is set to -100 to 100 so a higher MUTSTEP suits a higher MIN/MAX.

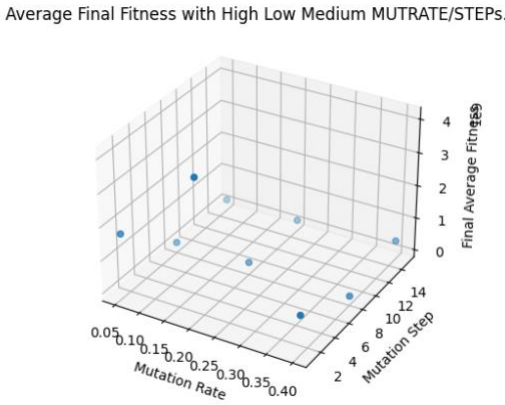

Figure 2 First cycle of high/medium/low values

Following this conclusion I have decided to test low/medium/high values one more time before moving towards a random range, this is because the results between the medium and high MUTSTEP at all MUTRATEs were extremely close. The new high/medium/low values I have decided for MUTSTEP are 22, 14 and 8, whilst keeping the MUTRATE high/medium/low unchanged. I believe these high/medium/low values to be justified as they have been adjusted according to previous results.

Unsurprisingly the high MUTSTEP and high MUTRATE has yielded very sub-optimal results, in fact the high MUTSTEP has given very poor results overall, again this isn't entirely unexpected as 22 is a very large change in a gene for mutation, especially if it is commonly occurring. The low/medium MUTSTEP at a medium MUTRATE yields the most optimal result here, with the low MUTSTEP with medium MUTRATE giving the most optimal fitness, which is somewhat logical as it provides a consistent but not overly drastic change over the entire course of the algorithm. Using this information I will move onto to random ranges, I believe a reasonable range to be 6 to 12 for MUTSTEP and 0.1 to 0.3 for MUTRATE.

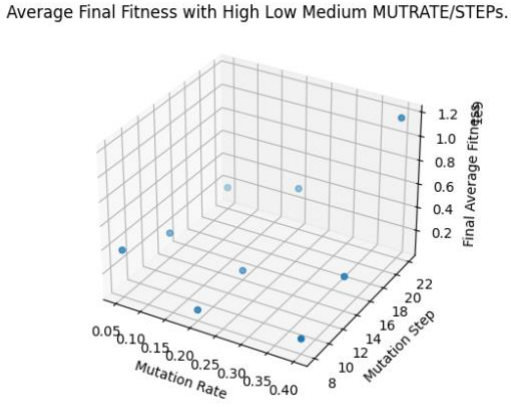

Figure 3 Second cycle of high/medium/low values

The result of this random range was a MUTSTEP of around 9.25 and a MUTRATE of around 0.175, again these values would provide a somewhat consistent change and consequently the good performance makes sense, using these values for MUTRATE and MUTSTEP the average fitness becomes 65466765.37 and the best fitness becomes 21317222.34, this is still very far from ideal, however, a huge improvement is already seen.

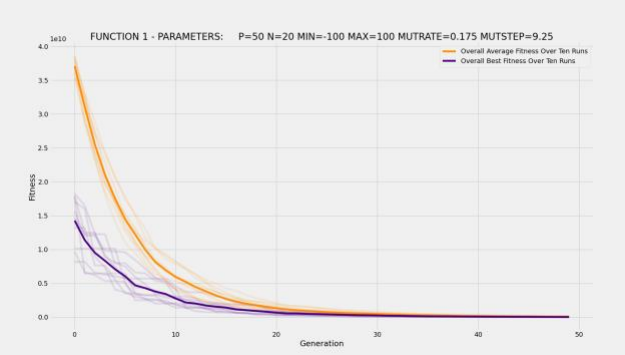

Figure 4 Graph showing the improved performance

Now that I have further optimised the MUTSTEP and MUTRATE I can begin the modification of the other parameters, particularly the population and number of generations. I will be modifying these together as they can be closely related to one another, for example a high population size but a small number of max generations is likely to be sub-optimal as operations such as tournament selection/mutation will not have a suitable amount of runs to become truly effective.

To modify P and MAXGEN I will again use the high/medium/low values in the same manner as with MUTSTEP and MUTRATE, the values for P I will start with will be 150 for high, 100 for medium and 50 for low, I have decided to keep the low at 50 because a very low P gives the algorithm very little data to work effectively and as the P is somewhat randomly generated there would be more potential for the P to be initialised with a majority of poor/good individuals rather than an even spread, perhaps making the inconsistency/efficiency of results greater. I have also chosen a high P of 150 as the higher the population, in theory, the more generations it should take to optimise them all to an effective standard. For MAXGEN I have decided on the same values, my reasoning for this is because I believe it to cover a good range of values whilst maintaining consciousness of the environmental impact of a large number of generations.

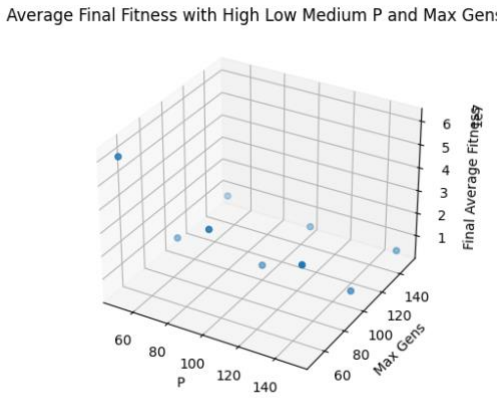

Figure 5 H/M/L P: 150, 100, 50 H/M/L MAXGEN: 150, 100, 50

The results of this testing has shown a higher P to give better results, this is particularly evident in the results of low MAXGEN where the difference is extremely noticeable. Perhaps this is because a higher population means less chance a good mutation/change to an individual will become lost as there are a greater number of other individuals also with the potential for a good/similar mutation. The most effective combinations were all involving the higher MAXGEN, this is perhaps expected as the MUTRATE and MUTSTEP are already quite optimised meaning giving it a longer time to make good changes will show in a better end result, again this is obvious in the change from 50 to 100 generations. The best result was with a high MAXGEN and a medium P, consequently I will use 120, 100 and 80 for the high/medium/low of P and 200, 150 and 130 for the high/medium/low of MAXGEN. My reasoning for including a MAXGEN lower than 150 is to consider whether the added generations are actually worth the increase in performance due to environmental factors.

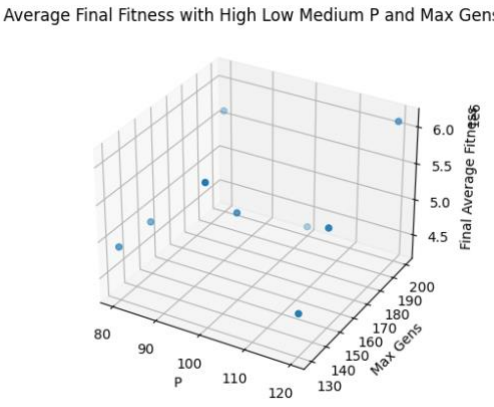

Figure 6 H/M/L P: 120, 100, 80 H/M/L MAXGEN: 200, 150, 130

The results are somewhat interesting as they seem very spread out. The low P outperforms the high and medium P at the low/medium MAXGEN, however, at the high MAXGEN the medium P outperforms all other combinations by a very considerable amount. The combination of medium P and high MAXGEN yields an average fitness of 5290141.28 and a best fitness of 249111.39, this is a huge difference from before we began modifying the P and MAXGEN. It appears that a P of 100 and a MAXGEN of 200 perhaps gives such good results because the P is lower than the MAXGEN but both are still somewhat high, he MAXGEN being higher means there is a good amount of runs given for the genetic algorithms operations to have worked effectively for the given P size. I am happy to consider these values for P and MAXGEN as my final optimal values as the ratio of environmental cost to effectiveness is somewhat worth it.

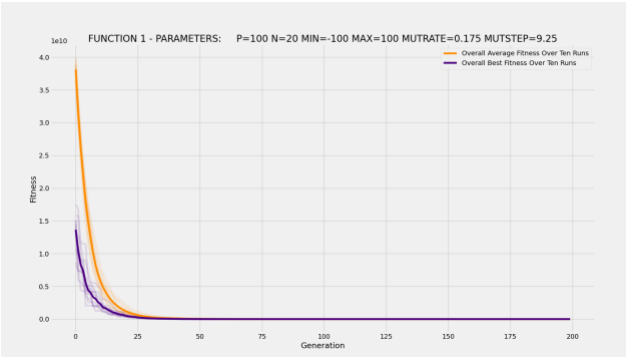

Figure 7 Graph showing these optimal values

If I were to disregard environmental costs I found I could get the fitness to as low as 155340.58 with 500 generations.

The Second Function

The graph below shows the results of the default parameters for function 2 (P = 50, N = 20, MIN = -5, MAX = 10, MUTRATE = 1/P, MAXGEN = 50, MUTSTEP = 0.1). The result is a much smaller value than the one given from the Rosenbrock function, giving a final average fitness of 258.65 and a final best fitness of 194.83. The average fitness also decreases very efficiently, as shown by the steep curve on the graph below.

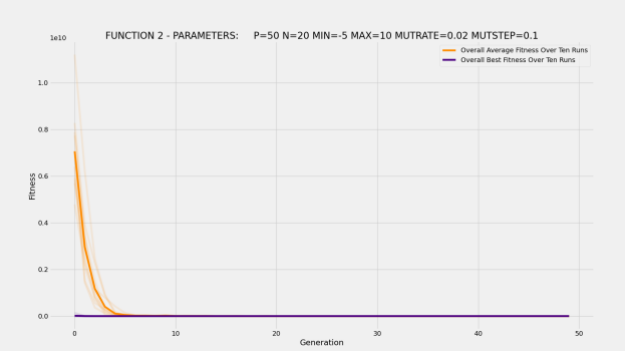

Figure 8 Function 2 with default parameters

I then applied the same methods from the Rosenbrock function to find optimal MUTRATE/STEP values. My starting H/M/L values for MUTRATE remained the same however I opted for smaller MUTSTEP values due to a much smaller MIN/MAX. The results of this is shown below:

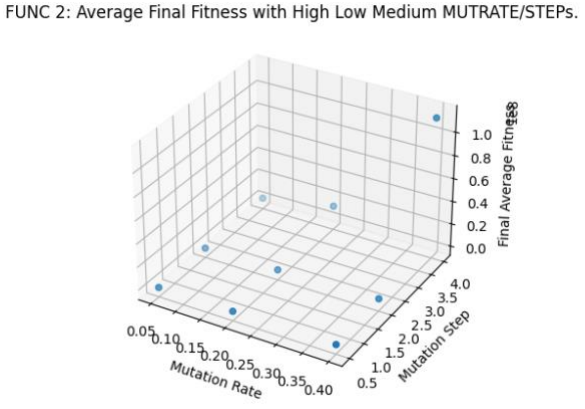

Figure 9 Func 2 MUTSTEP: 4, 2, 0.5 MUTRATE: 0.4, 0.2, 0.05

Unsurprisingly the high MUTRATE/STEP combination is again proved very ineffective, of course this is because the genes values are changing so constantly and so extremely that its always changing the fitness, even of individuals which you might consider good. Following these results I continued refining my H/M/L values until I came to the values of 0.2 for MUTRATE and 0.175 for MUTSTEP to be the most optimal. As shown by graph below this gives the results of 171.15 best fitness.

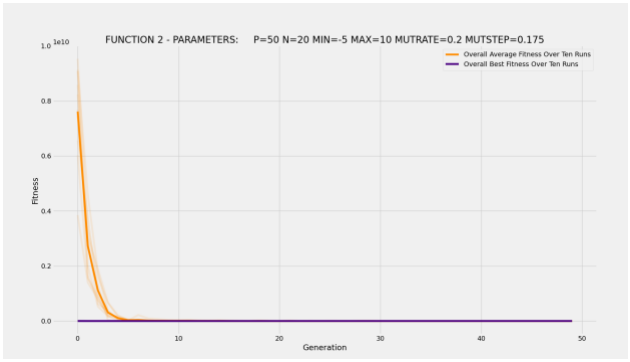

Figure 10 Graph showing optimal MUTRATE/STEP

For modifying the P and MAXGEN I will again use H/M/L values, using this I can get the best fitness down to 28.19 with the P value of 36 and MAXGEN of 500 as shown by the graph below:

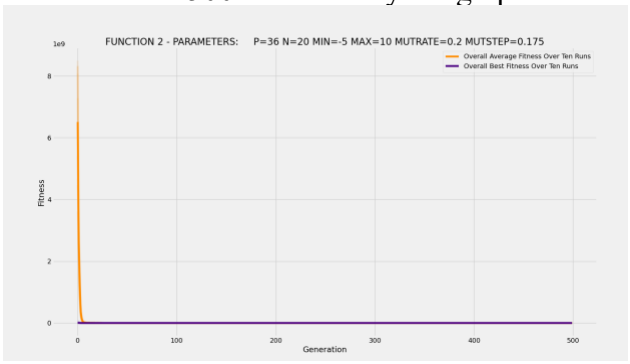

Figure 11 Func 2 optimal values

The combination of low P and high MAXGEN produced great results. The graph shows how quickly this function can reduce fitness, however, those extra generations work great in refining the small numbers you see after a few generations which doesn't show on the graphs. Perhaps the lower population helps as it lets the good individuals traits spread more effectively through the entire population through methods such as crossover.

4 CONCLUSIONS

Through this experimentation I have gotten a deeper understanding of how I might go about modifying parameters to make them optimal. I have gained insight into appropriate MUTRATE/STEPs relative to things such as the MIN/MAX, and the relationship between the MUTRATE/STEP, such as how a high MUTSTEP works better with a low MUTRATE so you aren't constantly making huge jumps in an individual's values. I have also learned how to find the balance between MAXGEN and computing power in order to maximise fitness whilst remaining environmentally conscious.

If I were to do things differently I would perhaps attempt further fitness functions and compare them to each other. I would also like to have compared my GA to the performance you might get from using a different algorithm such as particle swarm optimisation. I could also have changed parts of the algorithm such as the crossover to instead blend the values together and see what results that might yield.

REFERENCES

Amazon scrapped 'sexist AI' tool (2018) BBC News. BBC. Available from: https://www.bbc.co.uk/news/technology-45809919 [Accessed November 30, 2022]

Christoforaki, M. and Beyan, O. (2022) AI Ethics – A Bird's Eye View. *Applied Sciences.* 12(9) 4130. [Accessed 11 November 2022]

Hagendorff, T. (2020) The Ethics of AI Ethics: An Evaluation of Guidelines. *Minds and Machines.* 30, 99-120. [Accessed 10 November 2022]

Hunt, E. (2016) Tay, Microsoft's AI chatbot, gets a crash course in racism from Twitter, The Guardian. Guardian News and Media. Available from: https://www.theguardian.com/technology/2016/mar/24/tay-microsofts-ai-chatbot-gets-a-crash-course-in-racism-from-twitter [Accessed November 30, 2022]

Rességuier, A. and Rodrigues, R. (2020). AI Ethics Should Not Remain Toothless! A Call to Bring Back the Teeth of Ethics. *Big Data & Society*, 7. [Accessed 10 November 2022]

Source Code:

Rastigrin Function:

```
def test_function(ind):
        fitness = 0
        for i in range(N-1):
            x = ind.gene[i]
            x1 = ind.gene[i+1]
            fitness = fitness + (100*(math.pow((x1-math.pow(x, 2)), 2)))+math.pow((1-x),2)
        return fitness
```

Function 2:

```
def test_function(ind):
    fitness = 0
    x0 = 0
    x1 = 0
    x2 = 0
    x3 = 0
    for i in range(N):
        x0 += math.pow(ind.gene[i], 2)
        x1 += (0.5 * i) * ind.gene[i]
    x2 = math.pow(x1, 2)
    x3 = math.pow(x1, 4)
    fitness += (x0 + x2 + x3)
    return fitness
```

Code for making the 2D average graph:

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

plt.style.use("fivethirtyeight")
overallAverageFitness = np.zeros(500) #Change to MAXGEN
overallBestFitness = np.zeros(500)

for run in range(10):
    #P decides the size of population
    P = 36
    #N decides the length of the gene
    N = 20
    #MIN decides the minimum value a part of the gene can be
    MIN = -5
    #MAX decides the max value a part of the gene can be
    MAX = 10
    #MUTRATE is the probability a mutation will occur within part of a gene during the mutation stage
    MUTRATE = 0.2
    #MAXGEN decides the number of generations to be ran
    MAXGEN = 500
    #MUTSTEP decides the amount of change to occur should a mutation occur (e.g. + or - the MUTSTEP from the gene value)
    MUTSTEP = 0.175

    #Initialises the arrays
    population = []
    offspring = []
    genAvgFitness = []
    genBestFitness = []

    #Class to create the individuals of the population and initialise them with an empty gene and 0 fitness
    class individual:
        def __init__(self):
            self.gene = [0]*N
            self.fitness = 0

    #Test function to use function 1 from the assignment
    def test_function(ind):
        fitness = 0
        x0 = 0
        x1 = 0
        x2 = 0
        x3 = 0
```

```
    for i in range(N):
        x0 += math.pow(ind.gene[i], 2)
        x1 += (0.5 * i) * ind.gene[i]
    x2 = math.pow(x1, 2)
    x3 = math.pow(x1, 4)
    fitness += (x0 + x2 + x3)
    return fitness


#Function which simply prints the fitness of the population
def print_pop_fitness(population):
    fitness = 0
    for i in range(0, P):
        fitness += population[i].fitness
    print(fitness)


#Function which fills the population with actual values for their gene and fitness
def fill_population(population):
    for x in range(0, P):
        tempgene = []
        for y in range (0, N):
            tempgene.append(random.uniform(MIN, MAX))
        newind = individual()
        newind.gene = tempgene.copy()
        newind.fitness = test_function(newind)
        population.append(newind)
    return population


#Function in which offspring are selected based on their fitness, simply selects two
random parents and chooses the parent with the higher fitness
#for the offspring
def selection(population):
    offspring = []
    for i in range(0, P):
        parent1 = random.randint(0, P-1)
        off1 = copy.deepcopy(population[parent1])
        parent2 = random.randint(0, P-1)
        off2 = copy.deepcopy(population[parent2])
        if off1.fitness < off2.fitness:
            offspring.append(off1)
        else:
            offspring.append(off2)
    return offspring


#Function in which crossover occurs, selects crosspoint randomly each time in range
of N and applies crossover
#from that point between two individuals in the offspring then returns the offspring
array with the updated values
def crossover(offspring):
    toff1 = individual()
    toff2 = individual()
    temp = individual()
    for i in range(0, P, 2):
        toff1 = copy.deepcopy(offspring[i])
        toff2 = copy.deepcopy(offspring[i+1])
        temp = copy.deepcopy(offspring[i])
        crosspoint = random.randint(1, N)
        for j in range(crosspoint, N):
            toff1.gene[j] = toff2.gene[j]
            toff2.gene[j] = temp.gene[j]
        offspring[i] = copy.deepcopy(toff1)
        offspring[i+1] = copy.deepcopy(toff2)
    return offspring


#Function in which mutation occurs depending on whether or not the randomly
generated mutprob is lower than the MUTRATE,
#if the mutation would take the gene value over the MAX or MIN it simply keeps the
gene at the MAX or MIN value
def mutation(offspring):
    newOff = []
    for i in range(0, P):
```

```
            newind = individual()
            newind.gene = []
            for j in range(0, N):
                gene = offspring[i].gene[j]
                mutprob = random.random()
                if(mutprob < MUTRATE):
                    alter = random.uniform(-MUTSTEP, MUTSTEP)
                    gene = gene + alter
                    if gene > MAX:
                        gene = MAX
                    if gene < MIN:
                        gene = MIN
                newind.gene.append(gene)
            newOff.append(newind)
        return newOff


    #Function which updates the fitness of the new population after selection, crossover
and mutation has been applied
    def evaluate(population):
        for i in range(0, P):
            population[i].fitness = test_function(population[i])
        return population


    #Function to store the current populations fitness and update the best fitness where
appropriate, will be used to plot the graph
    def store_current_fitness(population, genAvgFitness, genBestFitness):
        currentFitness = 0
        if not genBestFitness:
            bestFitness = 1000000000000000000000000000000000000000000000000000000000
        else:
            bestFitness = genBestFitness[-1]

        for i in range(0, P):
            currentFitness += population[i].fitness
            if(population[i].fitness < bestFitness):
                bestFitness = population[i].fitness
        currentFitness = currentFitness / P
        genAvgFitness.append(currentFitness)
        genBestFitness.append(bestFitness)




    #Here the population is filled with its beginning values and the fitness is stored
    population = fill_population(population)
    store_current_fitness(population, genAvgFitness, genBestFitness)
    print_pop_fitness(population)


    #Loop which runs the GA MAXGEN amount of times
    for j in range(MAXGEN-1):
        offspring = selection(population) #Selects offspring
        offspring = crossover(offspring) #Performs crossover on the offspring
        offspring = mutation(offspring) #Mutates offspring and stores mutated values into
population
        population = copy.deepcopy(offspring) #Copies offspring into population
        population = evaluate(population) #Stores fitness of new population
        store_current_fitness(population, genAvgFitness, genBestFitness) #Stores current
gen fitness in an array
        print_pop_fitness(population) #Just prints current pop fitness
    print("Final Population Fitness:")
    print_pop_fitness(population)
    #Plotting each generation as a line on the graph
    plt.plot(genAvgFitness, "darkorange", alpha=0.1)
    plt.plot(genBestFitness, "indigo", alpha=0.1)
    #Adding the values from each generation to the overall arrays
    overallAverageFitness = np.add(overallAverageFitness, genAvgFitness)
    overallBestFitness = np.add(overallBestFitness, genBestFitness)


#Plots the graph using the stored average and best fitnesses
#Dividing by ten as it is over ten runs
overallAverageFitness = np.divide(overallAverageFitness, 10)
```

```
overallBestFitness = np.divide(overallBestFitness, 10)
print("Final Average Fitness Per Individual:")
print(overallAverageFitness[-1])
print("Final Individual Best Fitness:")
print(overallBestFitness[-1])
plt.plot(overallAverageFitness, "darkorange", label="Overall Average Fitness Over Ten
Runs")
plt.plot(overallBestFitness, "indigo", label="Overall Best Fitness Over Ten Runs")
plt.xlabel("Generation")
plt.ylabel("Fitness")
plt.title("FUNCTION 2 - PARAMETERS:\
    P="+str(P)+" N="+str(N)+" MIN="+str(MIN)+" MAX="+str(MAX)+"
MUTRATE="+str(MUTRATE)+" MUTSTEP="+str(MUTSTEP))
plt.legend()
fig = plt.gcf()
fig.set_size_inches(18.5, 10.5)
plt.savefig("C:/Users/Arthur/OneDrive - UWE Bristol/Year 2/Artificial Intelligence
II/Assignment/Graphs/Function2/P="+ str(P) + "N="+ str(N) +"MINandMAX=" +
str(MIN) +"-"+ str(MAX) + "MUTRATE=" + str(MUTRATE) + "MUTSTEP=" + str(MUTSTEP)
+ ".png", dpi=100)
plt.show()
```

Code for creating the 3D scatterplot with random values in a range:

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


def func1(MUTRATE, MUTSTEP):
    #plt.style.use("fivethirtyeight")
    overallAverageFitness = np.zeros(50)
    overallBestFitness = np.zeros(50)


    for run in range(10):
        #P decides the size of population
        P = 50
        #N decides the length of the gene
        N = 20
        #MIN decides the minimum value a part of the gene can be
        MIN = -5
        #MAX decides the max value a part of the gene can be
        MAX = 10
        #MUTRATE is the probability a mutation will occur within part of a gene during the
mutation stage
        #MUTRATE = 1 / P
        #MAXGEN decides the number of generations to be ran
        MAXGEN = 50
        #MUTSTEP decides the amount of change to occur should a mutation occur (e.g. +
or - the MUTSTEP from the gene value)
        #MUTSTEP = 0.1


        #Initialises the arrays
        population = []
        offspring = []
        genAvgFitness = []
        genBestFitness = []


        #Class to create the individuals of the population and initialise them with an empty
gene and 0 fitness
        class individual:
            def __init__(self):
                self.gene = [0]*N
                self.fitness = 0


        #Test function to use function 1 from the assignment
        def test_function(ind):
```

```
        fitness = 0
        x0 = 0
        x1 = 0
        x2 = 0
        x3 = 0
        for i in range(N):
            x0 += math.pow(ind.gene[i], 2)
            x1 += (0.5 * i) * ind.gene[i]
        x2 = math.pow(x1, 2)
        x3 = math.pow(x1, 4)
        fitness += (x0 + x2 + x3)
        return fitness


    #Function which simply prints the fitness of the population
    def print_pop_fitness(population):
        fitness = 0
        for i in range(0, P):
            fitness += population[i].fitness
        print(fitness)


    #Function which fills the population with actual values for their gene and fitness
    def fill_population(population):
        for x in range(0, P):
            tempgene = []
            for y in range (0, N):
                tempgene.append(random.uniform(MIN, MAX))
            newind = individual()
            newind.gene = tempgene.copy()
            newind.fitness = test_function(newind)
            population.append(newind)
        return population


    #Function in which offspring are selected based on their fitness, simply selects two
random parents and chooses the parent with the higher fitness
    #for the offspring
    def selection(population):
        offspring = []
        for i in range(0, P):
            parent1 = random.randint(0, P-1)
            off1 = copy.deepcopy(population[parent1])
            parent2 = random.randint(0, P-1)
            off2 = copy.deepcopy(population[parent2])
            if off1.fitness < off2.fitness:
                offspring.append(off1)
            else:
                offspring.append(off2)
        return offspring


    #Function in which crossover occurs, selects crosspoint randomly each time in
range of N and applies crossover
    #from that point between two individuals in the offspring then returns the offspring
array with the updated values
    def crossover(offspring):
        toff1 = individual()
        toff2 = individual()
        temp = individual()
        for i in range(0, P, 2):
            toff1 = copy.deepcopy(offspring[i])
            toff2 = copy.deepcopy(offspring[i+1])
            temp = copy.deepcopy(offspring[i])
            crosspoint = random.randint(1, N)
            for j in range(crosspoint, N):
                toff1.gene[j] = toff2.gene[j]
                toff2.gene[j] = temp.gene[j]
            offspring[i] = copy.deepcopy(toff1)
            offspring[i+1] = copy.deepcopy(toff2)
        return offspring


    #Function in which mutation occurs depending on whether or not the randomly
generated mutprob is lower than the MUTRATE,
```

```
    #if the mutation would take the gene value over the MAX or MIN it simply keeps
the gene at the MAX or MIN value
    def mutation(offspring):
        newOff = []
        for i in range(0, P):
            newind = individual()
            newind.gene = []
            for j in range(0, N):
                gene = offspring[i].gene[j]
                mutprob = random.random()
                if(mutprob < MUTRATE):
                    alter = random.uniform(-MUTSTEP, MUTSTEP)
                    gene = gene + alter
                    if gene > MAX:
                        gene = MAX
                    if gene < MIN:
                        gene = MIN
                newind.gene.append(gene)
            newOff.append(newind)
        return newOff


    #Function which updates the fitness of the new population after selection,
crossover and mutation has been applied
    def evaluate(population):
        for i in range(0, P):
            population[i].fitness = test_function(population[i])
        return population


    #Function to store the current populations fitness and update the best fitness
where appropriate, will be used to plot the graph
    def store_current_fitness(population, genAvgFitness, genBestFitness):
        currentFitness = 0
        if not genBestFitness:
            bestFitness = 1000000000000000000
        else:
            bestFitness = genBestFitness[-1]


        for i in range(0, P):
            currentFitness += population[i].fitness
            if(population[i].fitness < bestFitness):
                bestFitness = population[i].fitness
        currentFitness = currentFitness / P
        genAvgFitness.append(currentFitness)
        genBestFitness.append(bestFitness)




    #Here the population is filled with its beginning values and the fitness is stored
    population = fill_population(population)
    store_current_fitness(population, genAvgFitness, genBestFitness)
    print_pop_fitness(population)


    #Loop which runs the GA MAXGEN amount of times
    for j in range(MAXGEN-1):
        offspring = selection(population) #Selects offspring
        offspring = crossover(offspring) #Performs crossover on the offspring
        offspring = mutation(offspring) #Mutates offspring and stores mutated values
into population
        population = copy.deepcopy(offspring) #Copies offspring into population
        population = evaluate(population) #Stores fitness of new population
        store_current_fitness(population, genAvgFitness, genBestFitness) #Stores current
gen fitness in an array
        #print_pop_fitness(population) #Just prints current pop fitness
    #print("Final Population Fitness:")
    #print_pop_fitness(population)
    #Plotting each generation as a line on the graph
    #Adding the values from each generation to the overall arrays
    overallAverageFitness = np.add(overallAverageFitness, genAvgFitness)
    overallBestFitness = np.add(overallBestFitness, genBestFitness)
    print("Final Average Fitness Per Individual:")
```

```
        print(overallAverageFitness[-1])
        print(MUTRATE, MUTSTEP)
        return overallAverageFitness[-1]


minMutRate = 0.01
maxMutRate = 0.2
minMutStep = 0.1
maxMutStep = 5
mutRates = []
mutSteps = []
averageFitnessList = []

for i in range(500): #Fill the lists with random values in an appropriate range
    mutRates.append(random.uniform(minMutRate, maxMutRate))
    mutSteps.append(random.uniform(minMutStep, maxMutStep))
    averageFitnessList.append(func1(mutRates[i], mutSteps[i]))

fig = plt.figure()
ax = fig.add_subplot(projection='3d')
ax.scatter(mutRates, mutSteps, averageFitnessList)
ax.set_xlabel("Mutation Rate")
ax.set_ylabel("Mutation Step")
ax.set_zlabel("Final Average Fitness")
fig.set_size_inches(18.5, 10.5)
plt.title("Average Final Fitness with Varying Mutation Rate and Step MR= "+
str(minMutRate) + "-" + str(maxMutRate) + " MS= " + str(minMutStep) + "-" +
str(maxMutStep) +".")

plt.savefig("C:/Users/Arthur/OneDrive - UWE Bristol/Year 2/Artificial Intelligence
II/Assignment/Graphs/Function2/Showing Average Final Fitness with Varying Mutation
Rate and Step MR= "+ str(minMutRate) + " " + str(maxMutRate) + " MS= " +
str(minMutStep) + " " + str(maxMutStep) +".png", dpi=100)
plt.show()
```

Code for 3D scatter plot using H/M/L MUTRATE/MUTSTEP values:

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


def func1(MUTRATE, MUTSTEP):
    #plt.style.use("fivethirtyeight")
    overallAverageFitness = np.zeros(50)
    overallBestFitness = np.zeros(50)

    for run in range(10):
        #P decides the size of population
        P = 50
        #N decides the length of the gene
        N = 20
        #MIN decides the minimum value a part of the gene can be
        MIN = -5
        #MAX decides the max value a part of the gene can be
        MAX = 10
        #MUTRATE is the probability a mutation will occur within part of a gene during the
mutation stage
        #MUTRATE = 1 / P
        #MAXGEN decides the number of generations to be ran
        MAXGEN = 50
        #MUTSTEP decides the amount of change to occur should a mutation occur (e.g. +
or - the MUTSTEP from the gene value)
        #MUTSTEP = 0.1

        #Initialises the arrays
        population = []
        offspring = []
        genAvgFitness = []
        genBestFitness = []
```

```
    #Class to create the individuals of the population and initialise them with an empty
gene and 0 fitness
    class individual:
        def __init__(self):
            self.gene = [0]*N
            self.fitness = 0

    #Test function to use function 1 from the assignment
    def test_function(ind):
        fitness = 0
        x0 = 0
        x1 = 0
        x2 = 0
        x3 = 0
        for i in range(N):
            x0 += math.pow(ind.gene[i], 2)
            x1 += (0.5 * i) * ind.gene[i]
        x2 = math.pow(x1, 2)
        x3 = math.pow(x1, 4)
        fitness += (x0 + x2 + x3)
        return fitness

    #Function which simply prints the fitness of the population
    def print_pop_fitness(population):
        fitness = 0
        for i in range(0, P):
            fitness += population[i].fitness
        print(fitness)

    #Function which fills the population with actual values for their gene and fitness
    def fill_population(population):
        for x in range(0, P):
            tempgene = []
            for y in range (0, N):
                tempgene.append(random.uniform(MIN, MAX))
            newind = individual()
            newind.gene = tempgene.copy()
            newind.fitness = test_function(newind)
            population.append(newind)
        return population

    #Function in which offspring are selected based on their fitness, simply selects two
random parents and chooses the parent with the higher fitness
    #for the offspring
    def selection(population):
        offspring = []
        for i in range(0, P):
            parent1 = random.randint(0, P-1)
            off1 = copy.deepcopy(population[parent1])
            parent2 = random.randint(0, P-1)
            off2 = copy.deepcopy(population[parent2])
            if off1.fitness < off2.fitness:
                offspring.append(off1)
            else:
                offspring.append(off2)
        return offspring

    #Function in which crossover occurs, selects crosspoint randomly each time in
range of N and applies crossover
    #from that point between two individuals in the offspring then returns the offspring
array with the updated values
    def crossover(offspring):
        toff1 = individual()
        toff2 = individual()
        temp = individual()
        for i in range(0, P, 2):
            toff1 = copy.deepcopy(offspring[i])
            toff2 = copy.deepcopy(offspring[i+1])
            temp = copy.deepcopy(offspring[i])
            crosspoint = random.randint(1, N)
```

```
            for j in range(crosspoint, N):
                toff1.gene[j] = toff2.gene[j]
                toff2.gene[j] = temp.gene[j]
            offspring[i] = copy.deepcopy(toff1)
            offspring[i+1] = copy.deepcopy(toff2)
        return offspring


    #Function in which mutation occurs depending on whether or not the randomly generated mutprob is lower than the MUTRATE,
    #if the mutation would take the gene value over the MAX or MIN it simply keeps the gene at the MAX or MIN value
    def mutation(offspring):
        newOff = []
        for i in range(0, P):
            newind = individual()
            newind.gene = []
            for j in range(0, N):
                gene = offspring[i].gene[j]
                mutprob = random.random()
                if(mutprob < MUTRATE):
                    alter = random.uniform(-MUTSTEP, MUTSTEP)
                    gene = gene + alter
                    if gene > MAX:
                        gene = MAX
                    if gene < MIN:
                        gene = MIN
                newind.gene.append(gene)
            newOff.append(newind)
        return newOff


    #Function which updates the fitness of the new population after selection, crossover and mutation has been applied
    def evaluate(population):
        for i in range(0, P):
            population[i].fitness = test_function(population[i])
        return population


    #Function to store the current populations fitness and update the best fitness where appropriate, will be used to plot the graph
    def store_current_fitness(population, genAvgFitness, genBestFitness):
        currentFitness = 0
        if not genBestFitness:
            bestFitness = 1000000000000000000
        else:
            bestFitness = genBestFitness[-1]

        for i in range(0, P):
            currentFitness += population[i].fitness
            if(population[i].fitness < bestFitness):
                bestFitness = population[i].fitness
        currentFitness = currentFitness / P
        genAvgFitness.append(currentFitness)
        genBestFitness.append(bestFitness)



    #Here the population is filled with its beginning values and the fitness is stored
    population = fill_population(population)
    store_current_fitness(population, genAvgFitness, genBestFitness)
    print_pop_fitness(population)

    #Loop which runs the GA MAXGEN amount of times
    for j in range(MAXGEN-1):
        offspring = selection(population) #Selects offspring
        offspring = crossover(offspring) #Performs crossover on the offspring
        offspring = mutation(offspring) #Mutates offspring and stores mutated values into population
        population = copy.deepcopy(offspring) #Copies offspring into population
        population = evaluate(population) #Stores fitness of new population
```

```
            store_current_fitness(population, genAvgFitness, genBestFitness) #Stores current
gen fitness in an array
            print_pop_fitness(population) #Just prints current pop fitness
        print("Final Population Fitness:")
        print_pop_fitness(population)
        #Plotting each generation as a line on the graph
        #Adding the values from each generation to the overall arrays
        overallAverageFitness = np.add(overallAverageFitness, genAvgFitness)
        overallBestFitness = np.add(overallBestFitness, genBestFitness)
        print("Final Average Fitness Per Individual:")
        print(overallAverageFitness[-1])
        return overallAverageFitness[-1]


highMutRate = 0.08
mediumMutRate = 0.05
lowMutRate = 0.01
highMutStep = 1
mediumMutStep = 0.5
lowMutStep = 0.10
mutRates = [highMutRate, highMutRate, highMutRate, mediumMutRate,
mediumMutRate, mediumMutRate, lowMutRate, lowMutRate, lowMutRate]
mutSteps = [highMutStep, mediumMutStep, lowMutStep, highMutStep,
mediumMutStep, lowMutStep, highMutStep, mediumMutStep, lowMutStep]
averageFitnessList = []

for i in range(9):
    currentMutRate = mutRates[i]
    currentMutStep = mutSteps[i]
    averageFitnessList.append(func1(currentMutRate, currentMutStep))

fig = plt.figure()
ax = fig.add_subplot(projection='3d')
ax.scatter(mutRates, mutSteps, averageFitnessList)
ax.set_xlabel("Mutation Rate")
ax.set_ylabel("Mutation Step")
ax.set_zlabel("Final Average Fitness")
plt.title("FUNC 2: Average Final Fitness with High Low Medium MUTRATE/STEPs.")

plt.savefig("C:/Users/Arthur/OneDrive - UWE Bristol/Year 2/Artificial Intelligence
II/Assignment/Graphs/Function2/Average Final Fitness with High Low Medium MUTRATE
MUTSTEPs.png")
plt.show()
```

Code for 3D scatterplot using H/M/L P/MG:

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


def func1(P, MAXGEN):
    #plt.style.use("fivethirtyeight")
    overallAverageFitness = np.zeros(MAXGEN)
    overallBestFitness = np.zeros(MAXGEN)

    for run in range(10):
        #P decides the size of population
        #P = 50
        #N decides the length of the gene
        N = 20
        #MIN decides the minimum value a part of the gene can be
        MIN = -5
        #MAX decides the max value a part of the gene can be
        MAX = 10
        #MUTRATE is the probability a mutation will occur within part of a gene during the
mutation stage
        MUTRATE = 0.2
        #MAXGEN decides the number of generations to be ran
        #MAXGEN = 50
        #MUTSTEP decides the amount of change to occur should a mutation occur (e.g. +
or - the MUTSTEP from the gene value)
```

```
    MUTSTEP = 0.175


    #Initialises the arrays
    population = []
    offspring = []
    genAvgFitness = []
    genBestFitness = []


    #Class to create the individuals of the population and initialise them with an empty
gene and 0 fitness
    class individual:
        def __init__(self):
            self.gene = [0]*N
            self.fitness = 0


    #Test function to use function 1 from the assignment
    def test_function(ind):
        fitness = 0
        x0 = 0
        x1 = 0
        x2 = 0
        x3 = 0
        for i in range(N):
            x0 += math.pow(ind.gene[i], 2)
            x1 += (0.5 * i) * ind.gene[i]
        x2 = math.pow(x1, 2)
        x3 = math.pow(x1, 4)
        fitness += (x0 + x2 + x3)
        return fitness


    #Function which simply prints the fitness of the population
    def print_pop_fitness(population):
        fitness = 0
        for i in range(0, P):
            fitness += population[i].fitness
            print(fitness)


    #Function which fills the population with actual values for their gene and fitness
    def fill_population(population):
        for x in range(0, P):
            tempgene = []
            for y in range (0, N):
                tempgene.append(random.uniform(MIN, MAX))
            newind = individual()
            newind.gene = tempgene.copy()
            newind.fitness = test_function(newind)
            population.append(newind)
        return population


    #Function in which offspring are selected based on their fitness, simply selects two
random parents and chooses the parent with the higher fitness
    #for the offspring
    def selection(population):
        offspring = []
        for i in range(0, P):
            parent1 = random.randint(0, P-1)
            off1 = copy.deepcopy(population[parent1])
            parent2 = random.randint(0, P-1)
            off2 = copy.deepcopy(population[parent2])
            if off1.fitness < off2.fitness:
                offspring.append(off1)
            else:
                offspring.append(off2)
        return offspring


    #Function in which crossover occurs, selects crosspoint randomly each time in
range of N and applies crossover
    #from that point between two individuals in the offspring then returns the offspring
array with the updated values
    def crossover(offspring):
```

```
        toff1 = individual()
        toff2 = individual()
        temp = individual()
        for i in range(0, P, 2):
            toff1 = copy.deepcopy(offspring[i])
            toff2 = copy.deepcopy(offspring[i+1])
            temp = copy.deepcopy(offspring[i])
            crosspoint = random.randint(1, N)
            for j in range(crosspoint, N):
                toff1.gene[j] = toff2.gene[j]
                toff2.gene[j] = temp.gene[j]
            offspring[i] = copy.deepcopy(toff1)
            offspring[i+1] = copy.deepcopy(toff2)
        return offspring


    #Function in which mutation occurs depending on whether or not the randomly
generated mutprob is lower than the MUTRATE,
    #if the mutation would take the gene value over the MAX or MIN it simply keeps
the gene at the MAX or MIN value
    def mutation(offspring):
        newOff = []
        for i in range(0, P):
            newind = individual()
            newind.gene = []
            for j in range(0, N):
                gene = offspring[i].gene[j]
                mutprob = random.random()
                if(mutprob < MUTRATE):
                    alter = random.uniform(-MUTSTEP, MUTSTEP)
                    gene = gene + alter
                    if gene > MAX:
                        gene = MAX
                    if gene < MIN:
                        gene = MIN
                newind.gene.append(gene)
            newOff.append(newind)
        return newOff


    #Function which updates the fitness of the new population after selection,
crossover and mutation has been applied
    def evaluate(population):
        for i in range(0, P):
            population[i].fitness = test_function(population[i])
        return population


    #Function to store the current populations fitness and update the best fitness
where appropriate, will be used to plot the graph
    def store_current_fitness(population, genAvgFitness, genBestFitness):
        currentFitness = 0
        if not genBestFitness:
            bestFitness = 1000000000000000000
        else:
            bestFitness = genBestFitness[-1]

        for i in range(0, P):
            currentFitness += population[i].fitness
            if(population[i].fitness < bestFitness):
                bestFitness = population[i].fitness
        currentFitness = currentFitness / P
        genAvgFitness.append(currentFitness)
        genBestFitness.append(bestFitness)



    #Here the population is filled with its beginning values and the fitness is stored
    population = fill_population(population)
    store_current_fitness(population, genAvgFitness, genBestFitness)
    print_pop_fitness(population)


    #Loop which runs the GA MAXGEN amount of times
```

```
        for j in range(MAXGEN-1):
            offspring = selection(population) #Selects offspring
            offspring = crossover(offspring) #Performs crossover on the offspring
            offspring = mutation(offspring) #Mutates offspring and stores mutated values
into population
            population = copy.deepcopy(offspring) #Copies offspring into population
            population = evaluate(population) #Stores fitness of new population
            store_current_fitness(population, genAvgFitness, genBestFitness) #Stores current
gen fitness in an array
            print_pop_fitness(population) #Just prints current pop fitness
        print("Final Population Fitness:")
        print_pop_fitness(population)
        #Plotting each generation as a line on the graph
        #Adding the values from each generation to the overall arrays
        overallAverageFitness = np.add(overallAverageFitness, genAvgFitness)
        overallBestFitness = np.add(overallBestFitness, genBestFitness)
        print("Final Average Fitness Per Individual:")
        print(overallAverageFitness[-1])
        return overallAverageFitness[-1]

highP = 36
mediumP = 26
lowP = 10
highMaxGen = 500
mediumMaxGen = 100
lowMaxGen = 50
Ps = [highP, highP, highP, mediumP, mediumP, mediumP, lowP, lowP, lowP]
maxGens = [highMaxGen, mediumMaxGen, lowMaxGen, highMaxGen, mediumMaxGen,
lowMaxGen, highMaxGen, mediumMaxGen, lowMaxGen]
averageFitnessList = []

for i in range(9):
    currentP = Ps[i]
    currentMaxGen = maxGens[i]
    averageFitnessList.append(func1(currentP, currentMaxGen))
```

```
print(averageFitnessList)
fig = plt.figure()
ax = fig.add_subplot(projection='3d')
ax.scatter(Ps, maxGens, averageFitnessList)
ax.set_xlabel("P")
ax.set_ylabel("Max Gens")
ax.set_zlabel("Final Average Fitness")
plt.title("Average Final Fitness with High Low Medium P and Max Gens.")

plt.savefig("C:/Users/Arthur/OneDrive - UWE Bristol/Year 2/Artificial Intelligence
II/Assignment/Graphs/Function2/Average Final Fitness with High Low Medium P and
MAXGENs.png")
plt.show()
```