Coursework B: Genetic Algorithms

Introduction:

The principles of genetics and natural selection are the foundation of genetic algorithms (GA), which are search-based optimization approaches. They are frequently utilised to locate ideal or almost ideal answers to challenging issues that would otherwise take a lifetime to solve, and are extensively employed in research and machine learning to address optimization issues. GAs are a subset of a much larger branch of computation known as Evolutionary Computation. The University of Michigan's John Holland and David E. Goldberg, among others, invented GAs, which have since been used to solve a variety of optimization issues with great success.

In GAs, there is a population or pool of potential solutions to the given problem. Then, as in natural genetics, these solutions go through recombination and mutation, creating new offspring, a cycle that is repeated over several generations. Based on its objective function value, each individual (or potential solution) is given a fitness value. The fitter individuals are given a higher opportunity to mate and produce more fitter individuals. This is consistent with the Survival of the Fittest Darwinian theory. In this way, over many generations, we continue to evolve better people or solutions until we hit a stopping point. Though sufficiently random, genetic algorithms outperform random local search (in which we simply try numerous random solutions while keeping track of the best so far) because they also take advantage of past information.

This report seeks to demonstrate the effectiveness of GAs when applied to various issues including, the issues involved when determining a given number, optimising a set of parameters for a fifth-order polynomial, and visualising outcomes. The final exercise involved the use of Holland's Schema Theorem to analyse the results related to the fifth-order polynomial.

Exercise 1:

"Modify the code (or create your own from scratch) to create a simple optimization where the function of the optimizer is to simply find a number specified in the shortest number of iterations."

The lab script's original code produced a population of individuals with an array length of six, where each number in the array was limited within the range of 'i_min' and 'i_max', and each integer in the array represented a particular gene of an individual. The sum of these integers was then calculated, with the fitness being how far the sum was from the target. This exercise required the genetic algorithm to be changed; the integers should not be represented in an array but rather a single integer, meaning that the way fitness, mutation, and crossover parameters are handled must adapt in light of this. Several stages of the algorithm must be understood before making changes to the code.

An **individual** is a single integer between the values of 'i_min' and 'i_max'.

The **population** is an array of individuals of length 'p_count'.

The **target** is a quantity that the genetic algorithm strives to reach, a specific integer value.

Fitness is how close a given solution is to the optimum solution of the desired problem; how close the individual is to the target value.

The **Grading** process calculates the average fitness of the population.

Parent Selection is the process of selecting parents who mate and recombine to create off-springs for the next generation. This involves grading the population and ranking them based on their fitness. The top 20% of the graded population are preserved and used as parents, while the remaining 80% are discarded. 5% of the population is also added to the parents at random, promoting genetic diversity and preventing stagnation. This is known as elitism, as only the top-scoring individuals are retained.

Mutations refer to randomly changing some genes of the chromosome; how often parts of a chromosome will be altered. Mutation provides the opportunity to reach parts of the search space that perhaps cannot be reached by crossover alone, and without mutation, we may get premature convergence to a population of identical clones. Mutation helps the exploration of the whole search space by maintaining genetic diversity in the population. This occurs when a randomly generated value is less than the predefined value 'mutate'.

Crossover is a genetic operator used to combine the genetic information of two parents to generate new offspring. It is one way to stochastically generate new solutions from an existing population [1]. Crossover is made in hope that new chromosomes will contain good parts of old chromosomes and therefore the new chromosomes will be better, as shown in Figure 1.

Chromosome1	11011|00100110110
Chromosome2	11011|11000011110
Offspring1	11011|11000011110
Offspring2	11011|00100110110

Single Point Crossover

Figure 1: One-Point Crossover

Generations are the number of iterations before the termination. Each successive generation is more suited for the environment, and the fittest member of the culminating population is outputted after a predetermined number of generations, or after the termination criteria have been met.

To analyse how successful the algorithm performed, a graph of the fitness after each generation was plotted, with examples shown in Figure 2. In this case, the target number was set to 750, with an individual range between 0 and 100.

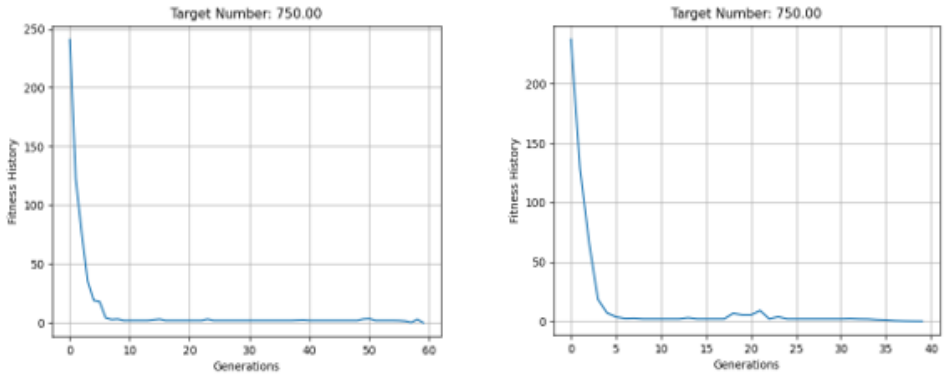


Figure 2: Fitness against Generations for a target number of 750.

As shown in Figure 3, the target value was achieved after 40 iterations, shown by the fitness value of 0. Further changes were made throughout the exercises to optimise functionality and duration.

```
Number of Iterations: 40
Target Number: 750
Population Size: 100
Maximum number of Generations: 1000
Final Fitness Value: 0.0
```

Figure 3: Target Value achieved.

Exercise 2:

"How does the number of generations, mutation, and crossover parameters affect how quickly a solution is found?"

A genetic algorithm (GA) is a method for solving both constrained and unconstrained optimization problems based on a natural selection process that mimics biological evolution. The solution quality is influenced by several important parameters within a GA, those being:

- Generations
- Population Size
- Mutations
- Crossover

Having a sufficient number of generations is necessary as it allows the GA to converge toward the target value. In some cases, hundreds of loops are sufficient, but in other cases, more is required, depending on the problem type and complexity. The sequence of phases is repeated to produce individuals in each new generation who are 'fitter' than the previous generation. As new generations are formed, individuals with the least fitness die, providing space for new offspring. Similarly, the population size is an important parameter that directly influences the ability to search for an optimum

solution in the search space. To an extent, having a larger population leads to increased accuracy in getting an optimal solution.

Crossover is the most significant phase in a genetic algorithm. The likelihood of crossover occurring is determined by the crossover probability. If there is a crossover, offspring are made from parts of both parents' chromosomes. If the crossover probability is 100%, then all offspring are made by crossover. If it is 0%, a whole new generation is made from exact copies of chromosomes from the old population, but this does not mean that the new generation is the same [2]. There are 3 types of crossover, one-point crossover, two-point crossover, and uniform crossover.

In this **One-Point Crossover**, a random crossover point is selected and a new solution is produced by combining the pieces of the original solutions. This has a significant drawback, as the head and the tail of one chromosome cannot be passed together to the offspring. This can be avoided using a **Two-Point Crossover**, which is a generalization of the One-Point Crossover wherein alternating segments are swapped to get new off-springs. **Uniform Crossover** involves treating each gene separately. Each gene in the offspring is created by copying the corresponding gene from one or the other parent, chosen according to a randomly generated binary crossover mask of the same length as the chromosome. Where there is a 1 in the crossover mask the gene is copied from the first parent and where there is a 0 in the mask the gene is copied from the second parent.

Mutation helps to prevent the undesirable effect of stagnation. This is where all the chromosomes in the population have similar genetic information, resulting in the GA becoming stuck in a local maximum. Instead, mutation leads to a population containing a variety of genetic material, an effect known as diversity. This is generally desirable for exploring different parts of the optimization landscape. It must be noted that too much mutation is not ideal, as the GA will then change to a random search.

Normally, mutation takes place after crossover is done. This operator applies the changes randomly to one or more genes to produce a new offspring, so it creates new adaptive solutions that are good to avoid local optima. If there is no mutation, offspring are generated immediately after crossover (or directly copied) without any change. If a mutation is performed, one or more parts of a chromosome are changed. If the mutation probability is 100%, the whole chromosome is changed, if it is 0%, nothing is changed [3].

Generations:

The number of generations represents the duration that the GA will run for, so having the least amount of generations whilst still finding a suitable solution is ideal. To test how the number of generations affects the solution found, the target number was set to 650. P_count was set to 100, whilst i_min was 1 and i_max was 100. 20% of the population was retained, 5% of the population was randomly selected, and mutation probability was set to 1%. The code was then looped 100 times and the average fitness at each generation was calculated. Figure 2 below shows the average fitness at each generation after limiting the code to 10 generations.

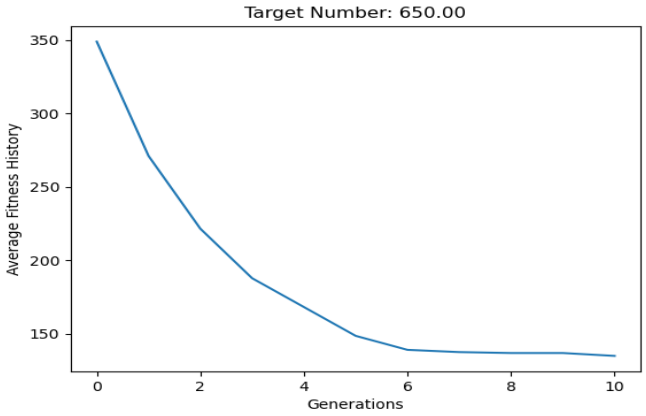


Figure 4: Fitness graph after 10 Generations.

As seen in Figure 5, the fitness was significantly far away from the target value, so it was concluded that 10 generations were not sufficient.

```
Target Number: 650
Population Size: 100
Maximum number of Generations: 10
Final Fitness Value: 135.0
```

Figure 5: Fitness after 10 Generations.

Figure 6 shows the results after 20 generations. The average fitness has still not reached a suitable value.

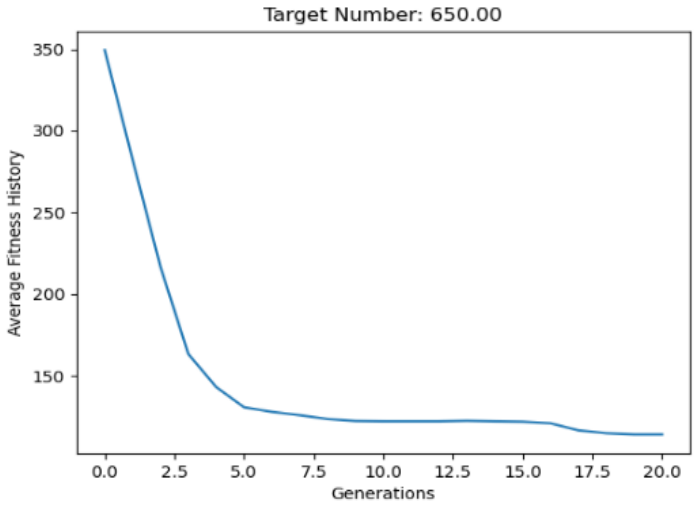


Figure 6: Fitness graph after 20 Generations.

Figure 7 presents the fitness after 20 generations, emphasising that more generations are needed to converge to the target value.

```
Target Number: 650
Population Size: 100
Maximum number of Generations: 20
Final Fitness Value: 114.0
```

Figure 7: Fitness after 20 Generations.

As shown in Figure 8, with a sufficient amount of generations, the target number can be achieved. This follows the theory that with more generations, the target number is easier to reach. However, with a fixed number of generations, such as 250, the code would continue to repeat even if the fitness has reached 0, and become repetitive when the GA is stuck in a local maximum. To prevent this, adding termination criteria is important, this is discussed in exercise 3.

```
Number of Iterations: 138
Target Number: 650
Population Size: 100
Maximum number of Generations: 250
Final Fitness Value: 0.0
```

Figure 8: Fitness after 10 Generations.

Population:

How the population affects the performance of the GA can be interpreted in two different ways. The first relates to the number of individuals, where the population can determine the total number of individuals present. The second refers to the retain percentage, and how much of the population is retained after the grading process. Both factors were tested. The parameters were kept the same as the testing of generations.

To an extent, increasing the population size led to a decrease in the number of generations required. This is because genetic diversity blossoms preventing premature convergence however, a population size too large caused the GA to slow down, whilst a population size too small was insufficient for a good mating pool. In most cases, a suitable solution was found however, when the population size was too small, below 22, the GA failed to find a suitable solution within the allocated number of generations. This is evident in Figure 9, which highlights the total number of generations needed at different population sizes.

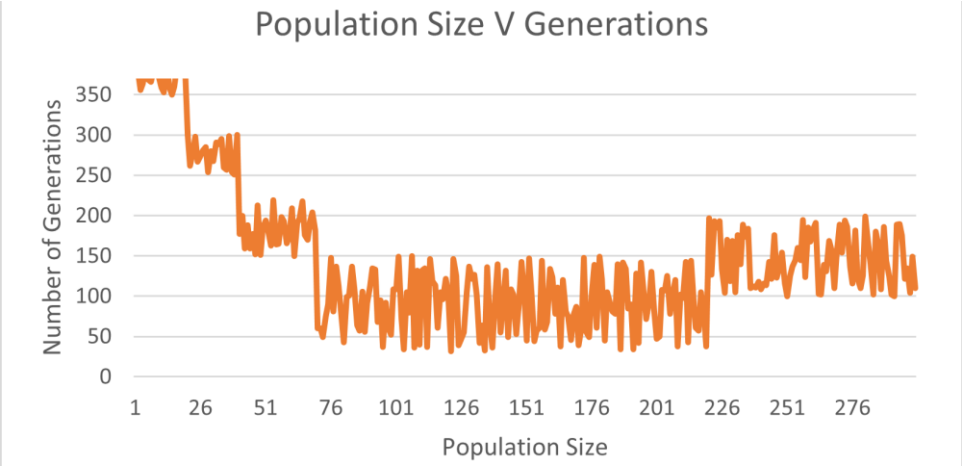


Figure 9: How population size impacts the number of generations needed.

As displayed, at low population sizes the algorithm struggled to find a solution, often reaching the cap of 370 generations before finding a suitable answer. As the population size increased, the efficiency of the GA increased up until a population size of 221, where the efficiency decreased albeit slightly. It can be concluded that for this specific GA, the optimal population size is between 80 and 221, with a population size of 100 producing the fastest time.

Retain:

Changing the number of individuals in the population retained had a different effect than changing the population size. The parameters were altered, using a population size of 100. The target value was set to 250, with the population range set between 0 and 500.

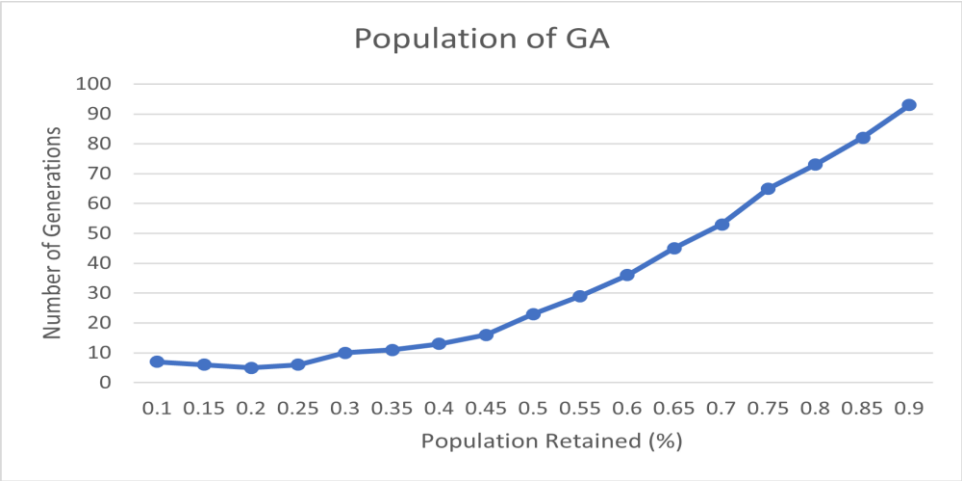


Figure 10: Population retained and the number of generations taken to reach the target.

There was a proportional relationship between the population retained and the number of generations required, as shown in Figure 10; as the population retained increases, the number of generations required to reach the target value increases. This is due to the nature of ranking, if more individuals are present after ranking has occurred, the probability of parents having the desired traits decreases, reducing the impact that ranking has on subsequent generations. Figure 10 shows that the optimum percentage of the population retained for the quickest solution was 20%.

Mutation:

Once the population results were known, the parameters were changed to see how the mutation probability affected how quickly a solution was found. The population retained was set to the optimal value of 0.2%, with a population size of 100. The mutation probability value was changed between a range of 0.05 and 0.9, where a higher value indicates a higher chance of a mutation occurring. The rest of the parameters were kept the same for consistency. Figure 11 presents how the number of generations was affected by different mutation probabilities.

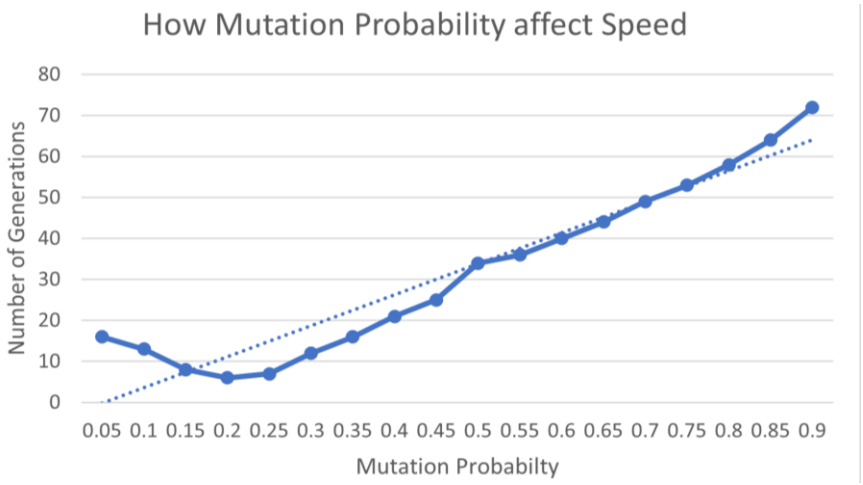


Figure 11: Mutation Probability v Number of Generations.

There was a positive correlation between the mutation probability and the number of generations, highlighted by the positive trendline. Too high of a mutation rate increases the probability of the GA searching more areas in the search space however, prevents the population to converge to the optimal solution, as the GA searches randomly. On the other hand, too small of a mutation rate leads to premature convergence. This is evident in Figure 11, where the optimal mutation probability is 0.2.

Changing the probability additionally had an inverse impact on the success rate of the GA; how often the GA converged to the correct target value. This is evident in Figure 12, showing that as the mutation probability increases, the success rate decreases. This is due to the increase in randomness in the population as mutation occurs more frequently.

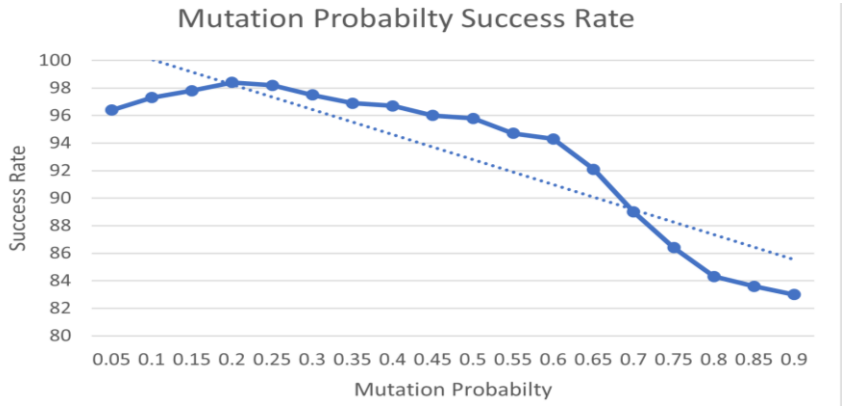


Figure 12: Success rate at different mutation probabilities.

It must be noted that a mutation may not alter the individual, since a 1 may be replaced with a 1, as any random number in the binary string can be transformed to a random integer of either 0 or 1. As a result. This significantly decreases the likelihood of a mutation happening. To see the true impact of mutation, the code should be modified so that whatever random integer is chosen from the binary string must be altered to its opposite counterpart for the mutation to be more successful. For example, a 0 must become a 1, or a 1 must become a 0.

Select Probability:

Varying the random selection probability had a visible effect on the number of generations when the generations were capped at a lower amount, as shown by the trendline in Figure 13, where the generations were capped at 100.

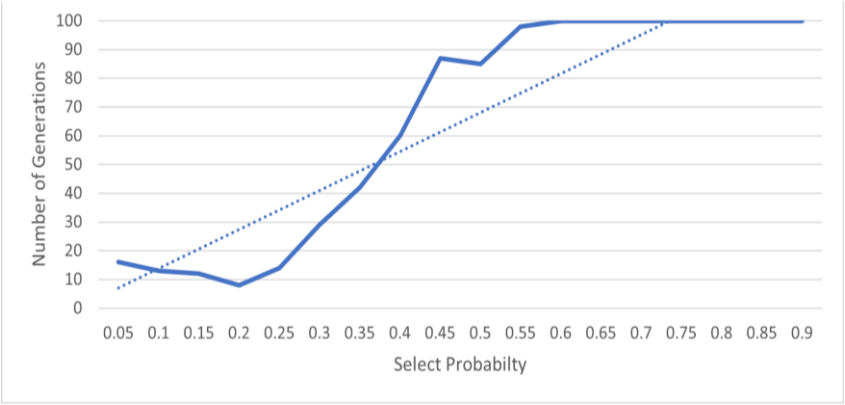


Figure 13: Random Selection Probability.

With a larger generation cap, the correlation between these variables decreased. The reason why the correlation is strong with a smaller cap is due to the amount of variation in the data as the random selection probability increases; the higher the probability, the more individuals within the population, so the higher the variation within the data. Although variation is favoured, too much variation leads to the GA not being able to find a suitable solution within 100 generations as there will be more random individuals. A random select probability of 0.2 produced the fastest results for this GA.

From this data, it is safe to conclude that all variables have a significant impact on how quickly a solution is found. Without termination criteria, having a larger number of generations produced slower solution times, as the code would loop continuously until all generations are completed however, not utilising a sufficient number of generations impacted the quality of the solution; suitable solutions were often not achieved. Concerning population, the larger the percentage of the population retained after grading, the slower the solution is found however, larger population sizes are ideal, with there being a range of suitable population sizes that produce optimal results. The mutation probability must also be kept relatively low to obtain the solution in the fastest manner possible.

Exercise 3:

"How could you stop the algorithm when a suitable solution had been found to avoid excessive computation time?"

There are several different methods in which the algorithm could be terminated when a suitable solution has been obtained.

Method 1: The fitness of the Population reaches 0

The fitness score is the number of values that differ from the values in the target at a particular index, so an individual having a lower fitness value is given more preference. As shown in Figure 13, this code was designed to terminate once the datum in fitness history reaches 0, signifying that the target value has been achieved.

```
breakoutflag = False

for i in range(generations):
    p = evolve(p, target)
    fitness_history.append(grade(p, target))
    for datum in fitness_history:
        if datum == 0:
            breakoutflag = True
            print('Number of Iterations:', len(fitness_history))
            break
    if breakoutflag:
        break
```

Figure 14: Terminate once reaching a fitness of 0.

However, this method has its drawbacks, with the code being unable to stop if the GA tends toward an incorrect value, as the fitness would never reach 0. This would be detrimental if the number of generations is not fixed, as the code would then loop forever.

Method 2: No improvement after a fixed number of iterations

The second method involves stopping the fitness if it has remained the same for a set number of generations as shown in Figure 14. This code is designed to terminate if the last value of fitness is equal to the last 5 values.

```
if len(fitness_history) > 1:
    if fitness_history[-1] == fitness_history[-5:]:
        breakoutflag = True
    break
```

Figure 15: Terminate after repetitive values.

This method prevents the drawback mentioned in method 1 however, it will not instantly terminate if the fitness has reached zero.

Method 3: When the error is below a predetermined value

When some inaccuracy is acceptable or we want to approximate a function rather than find the precise answer, we can utilise this termination condition. In such a situation, the code is terminated and the individual is returned when their fitness is lower than the permitted mistake. For example, with a target of 550, the error could be set to ± 3, meaning that the code will terminate if the output is within this range.

Method 4: Limit Generations

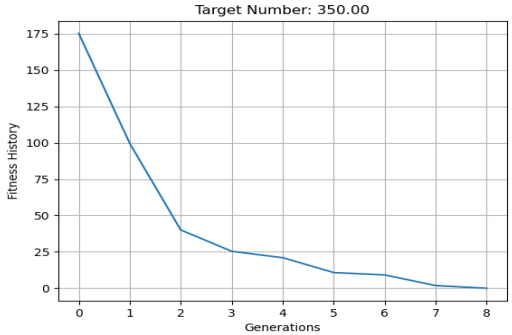
Limiting the number of generations terminates the code within a suitable timeframe, as shown in exercise 2. However, this is highly inefficient as:

- The code will not instantly terminate if a suitable solution has been obtained
- A time-consuming process as the optimal number of generations for different target values would need to be found via trial and error each time, or by significantly altering the code.
- A suitable solution may not always be found within the set number of generations.

Method 5: Combination

This method involved combining methods 1, 2, and 4. This produces the most optimal solution to the question, as the code will now stop if the fitness has reached 0, as well as stop if the fitness has been the same for a set number of generations.

The effectiveness of this method is highlighted in Figure 16. As soon as the fitness reaches 0, the code terminates and the total number of generations taken is displayed.




```
Number of Iterations: 9
Target Number: 350
Population Size: 100
Maximum number of Generations: 1000
Final Fitness Value: 0.0
```

Figure 16: Termination once a Fitness of 0 is reached

Exercise 4:

"Create an optimization using GA to optimize a set of parameters for a curve of a 5th -order polynomial."

The goal was to create an optimization using a genetic algorithm to optimize a set of parameters for a curve of a 5th-order polynomial as shown in Figure 15.

$$y = 25x^5 + 18x^4 + 31x^3 - 14x^2 + 7x - 19$$

Figure 17: Fifth-order polynomial.

There were two ways to approach this problem.

Approach 1:

This approach was similar to exercise 1, each coefficient of the polynomial was set as a target value, and the GA was set to find each value. The targets were set in a list [25, 18, 31, -14, 7, -19] where each gene is treated as a target value. The outputted graph is presented in Figure 16.

Initially, the number of generations was set to 100, with a retain of 0.2, random select of 0.05, and a mutation probability of 0.2. This produced a curve similar to the target curve, however not completely accurate, as shown in Figure 16.

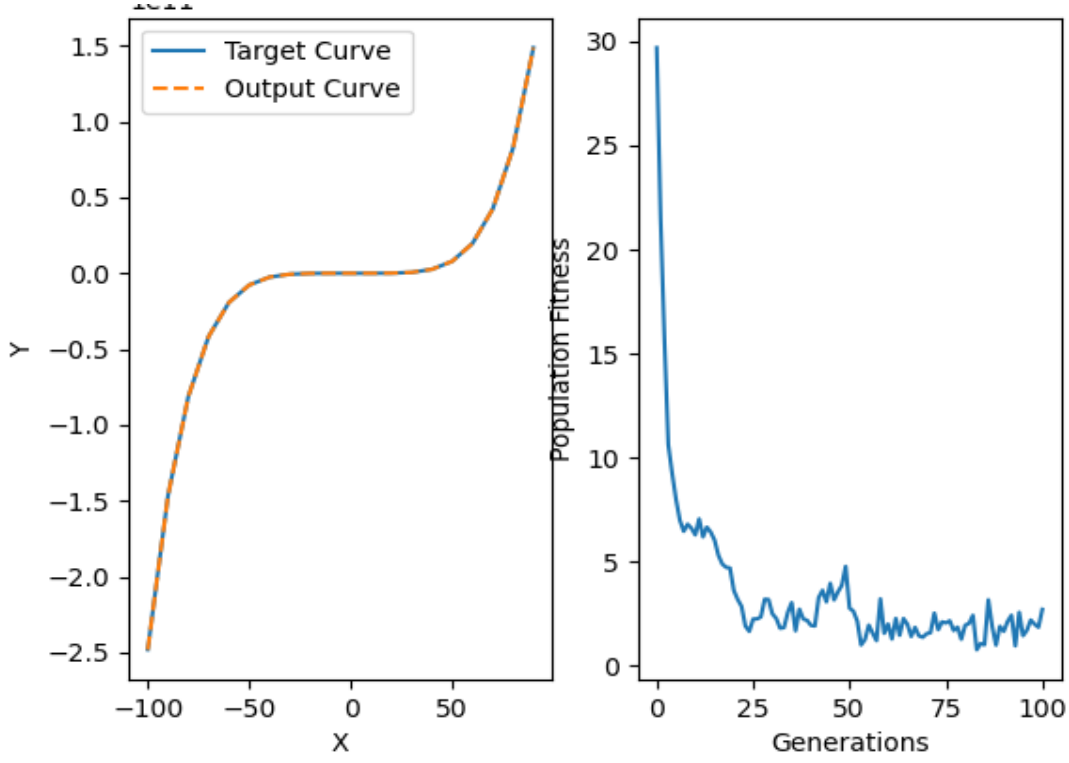


Figure 18: Target, Output Curve, and Fitness.

The coefficients obtained for this curve were [25, 17, 30, -14, 8, -19]. It was necessary to further optimise the parameters to ensure that the correct curve was outputted each time. Retain was changed to 0.15, and the mutation probability was changed to 0.3. The value of i_min was also changed, from 0 to -50, with i_max being 50, to accommodate for the negative numbers present within the polynomial. The outputted graph can be seen in Figure 17.

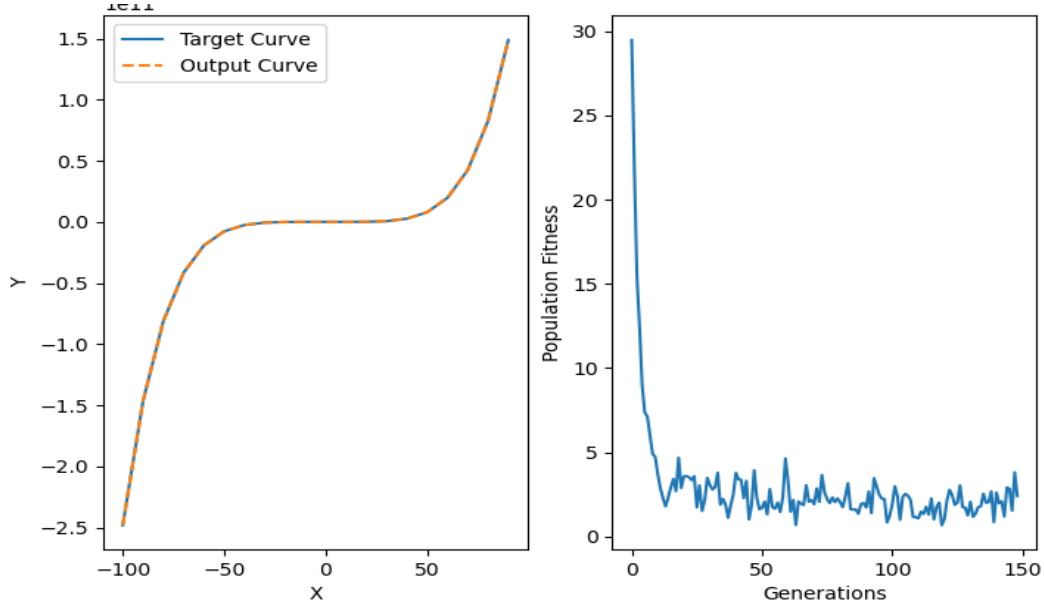


Figure 19: Target, Output, and Population Fitness curve.

The coefficients [25, 18, 31, -14, 7, -19] were found and matched the target curve within 152 generations. This set of parameters had a success rate of 98.4% when tested 1000 times. This is a simple approach that assumes that the values of the parameters are known. Approach 2 differs, with the premise of this process assuming that the parameter values are not known. In this case, a GA is built that finds the best coefficients to fit a polynomial, a process known as curve fitting.

Approach 2:

A genetic method for curve fitting appears more useful because it generates a curve that is similar to or identical to the target polynomial using only a subset of points from the Target Curve. The coefficients are determined using a 5^{th}-order polynomial curve. The outputted curve can be seen in Figure 18.

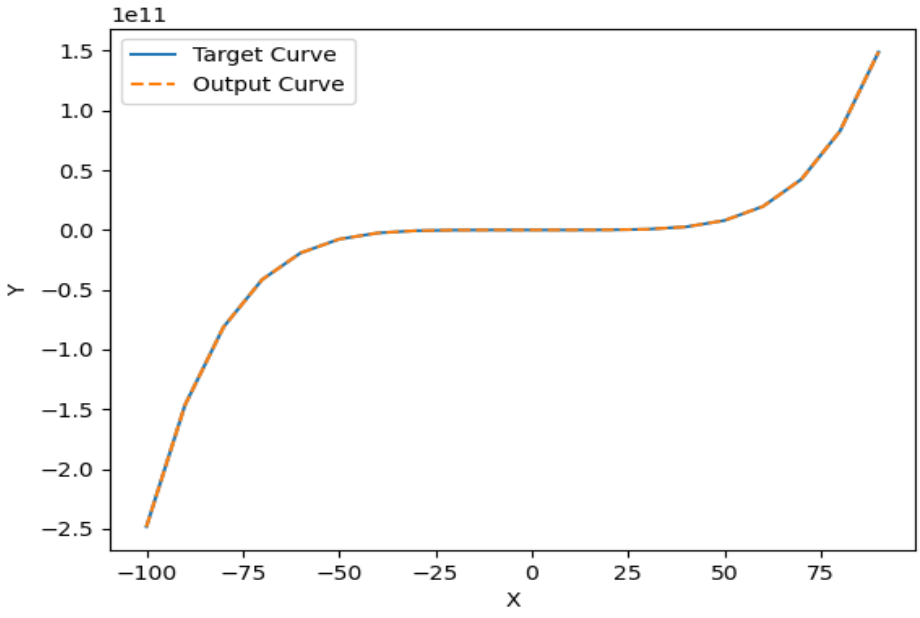


Figure 20: Target, Output, and Population Fitness curve.

The output coefficients produced from this run were [25, 18, 29, -9, 6, -18] as shown in Figure 19. Both the x^5 and x^4 coefficients have been found correctly, with the other coefficients considerably close to the polynomial coefficients. Compared to approach 1, this method took slightly longer, which may not be ideal in practical use.

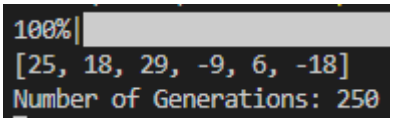


Figure 21: Coefficients obtained.

Analysis:

Both methods replicate the target curve after several iterations. Limiting the number of generations provides different results for each approach. As shown in Figure 18, after 5 generations of using approach 1, the output curve is close to replicating the target curve, obtaining coefficients of [32, 14, 32, -10, 17, -14].

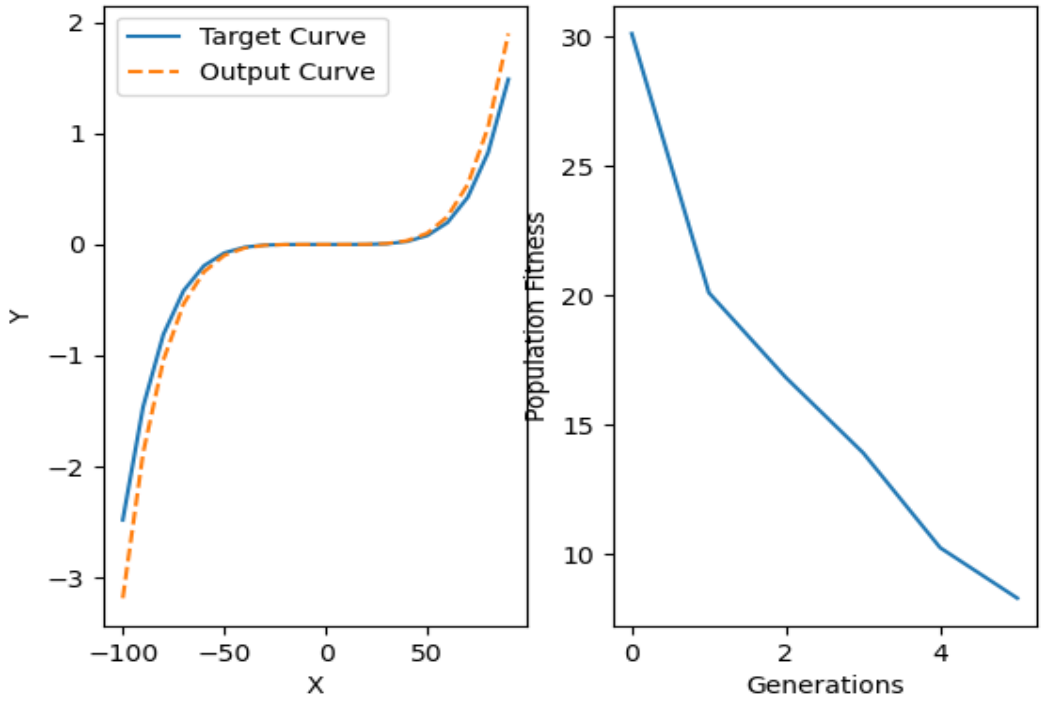


Figure 22: Target, Output, and Population Fitness curve after 5 generations.

When increased to 30 generations the output curve is closer to replicating the target curve, with coefficients of [26, 22, 34, -16, 10, -18]. Figure 19 displays the outputted curve. It can be concluded that the more generations used the closer the GA is to reaching the target array of coefficients.

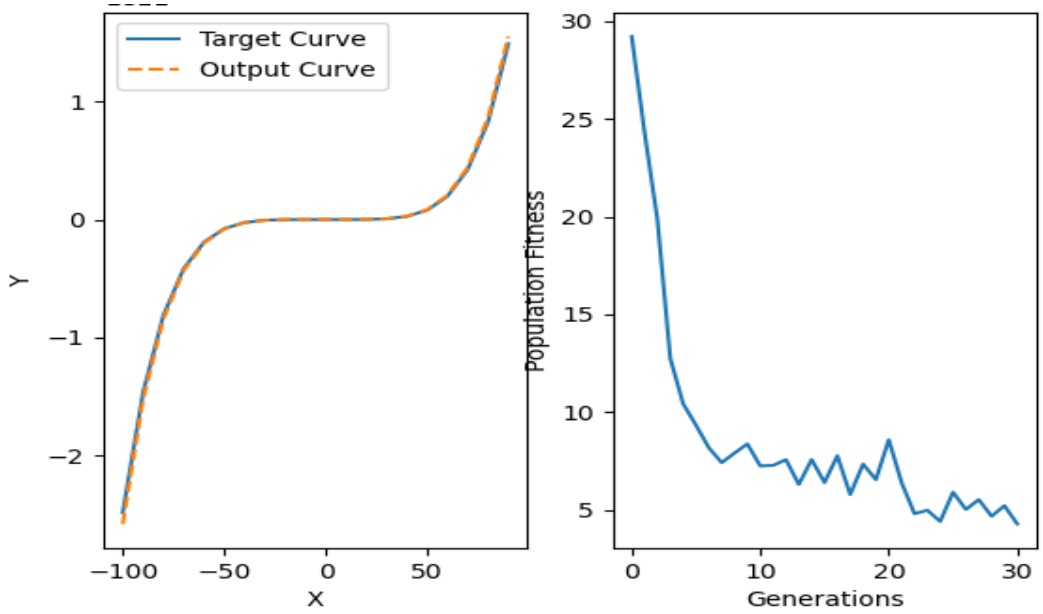


Figure 23: Target, Output, and Population Fitness curve after 30 generations.

Although Approach 2 finds the same correlation as Approach 1 when using various numbers of generations, the variance of the results is lower. This is because Approach 2 finds the x^5 and x^4 coefficients, which are the most significant coefficients when following the curve of the given 5th-order polynomial, very quickly.

A visual showing how the accuracy of the GA changes with larger generation caps is presented in Figure 24. It shows that the GA is more accurate as the number of generations increases.

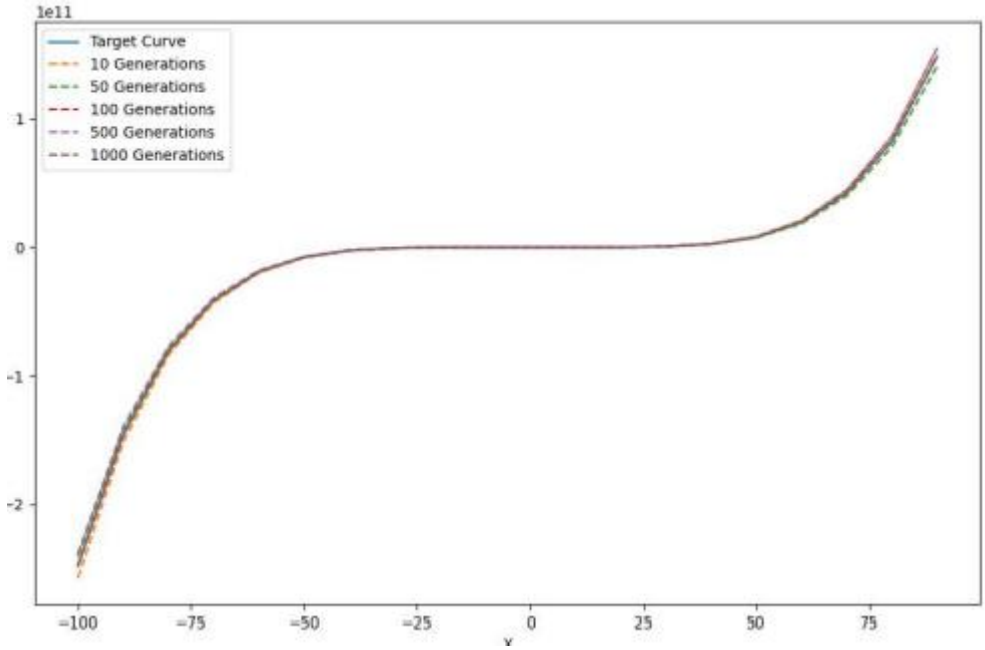


Figure 24: Relationship between generations and the outputted curve.

Exercise 5:

"Show your understanding of Holland's Schema Theorem. Explain Holland's Schema Theorem based on Question 4 using GA with binary encoding."

Holland's schema theorem, also called the fundamental theorem of genetic algorithms, is an inequality that results from coarse-graining an equation for evolutionary dynamics. GAs work because thousands of different schema can be processed with relatively few strings. For example, the number four can be represented by six different schemata. GAs process many more schema than there are strings in the population. The term implicit parallelism describes the GA's ability to evaluate thousands of different schema patterns simultaneously, or in parallel. Through the process of reproduction among the individuals, the fittest schemata will increase or decrease their representations in the populations. Holland discovered that a GA effectively processes n^3 schemata in a population size of n.

A Schema can be defined as a cognitive framework or concept that helps organise and interpret information, in this case, the schema represents a specific gene pattern. There are two assumptions made regarding Holland's Schema Theorem. The first assumption is that genes are represented in their binary form, meaning each gene is represented by a 0 or a 1. The second assumption is roulette wheel selection, a genetic operator used in genetic algorithms for selecting potentially useful solutions for recombination. The Schema Theorem says that short, low-order schemata with above-average fitness increase exponentially in frequency in successive generations. The Schema describes different bit strings in the search space, and they contain the binary alphabet [0,1,*] where the * is a wildcard that represents either a 0 or 1.

For example:

- Schema 11001 matches 1 string: [11001]
- Schema 101*1 matches the 2 strings: [10101, 10111]
- Schema *001* matches the 4 strings: [10010, 10011, 00010, 00011]
- Schema ***** matches any 5-bit string.
- String 10100 is matched by 2^5 schemas:

A string '11' has a length L=2 and belongs to 2^2 different schemas, **, *0, 1*, 10. The length 'L' is the distance between the first and last defined bit and the order 'S' is the number of fixed bits that are not '*'.

For example:

- Schema *1**10*1*, L = 6, S = 4.

The schema of L = 1 with the most significant bit correct is the most likely to be replicated. This schema will then increase in order size tending to the string length in the order of the most significant bits.

The equation for Holland's schema theorem is shown below:

$$E(m(H,k+1) = m(H,k) \cdot \frac{f(H,k)}{\bar{f}} \cdot \left(1 - p_c \cdot \frac{f(H)}{\delta(H) - 1}\right) \cdot (1 - p_m)^{o(H)}$$

Where:

- $E[m(H,k+1]$ is the expected number of instances of schema *H* at next-generation *k+1*
- $m(H,k)$ is the number of instances of schema *H* at generation *k*
- $f(H,k)$ is the average fitness of instances of schema *H* at generation *k*

- $\bar{f}$ is the average fitness of the population
- p_c is the probability of crossover
- ∂H is the defining length of *H*
- p_m is the probability of mutation
- $o(H)$ is the order of *H*

We can analyse Holland's Schema theorem using exercise 4. For example, take a schema [*,*,31,*,*,*] as shown in Figure 25. For simplicity, the numbers are represented differently.

Generation 1		Generation 2		Generation 3	
Population	Schema	Population	Schema	Population	Schema
[*,*,31,*,*,*]	Yes	[*,*,31,*,*,*]	Yes	[*,*,31,*,*,*]	Yes
[*,*,*,*,*,*]	No	[*,*,31,*,*,*]	Yes	[*,*,31,*,*,*]	Yes
[*,*,*,*,*,*]	No	[*,*,*,*,*,*]	No	[*,*,31,*,*,*]	Yes
[*,*,*,*,*,*]	No	[*,*,*,*,*,*]	No	[*,*,31,*,*,*]	Yes
[*,*,*,*,*,*]	No	[*,*,*,*,*,*]	No	[*,*,*,*,*,*]	No
[*,*,*,*,*,*]	No	[*,*,*,*,*,*]	No	[*,*,*,*,*,*]	No
[*,*,*,*,*,*]	No	[*,*,*,*,*,*]	No	[*,*,*,*,*,*]	No
[*,*,*,*,*,*]	No	[*,*,*,*,*,*]	No	[*,*,*,*,*,*]	No
[*,*,*,*,*,*]	No	[*,*,*,*,*,*]	No	[*,*,*,*,*,*]	No
[*,*,*,*,*,*]	No	[*,*,*,*,*,*]	No	[*,*,*,*,*,*]	No

Figure 25: Target, Output, and Population Fitness curve after 30 generations.

This data can then be used to calculate how likely the schema is to appear in the subsequent generations using the equation above:

Generation	No	Expected in Next Generation
1	1	2
2	2	4
3	4	22

Figure 26: Target, Output, and Population Fitness curve after 30 generations.

This highlights the importance of Holland's Schema theorem, indicating that more favoured schemas are likely to appear in future generations. It must be noted that as the number of genes increases, the likelihood of it appearing in the next generation decreases. Take the schemas [*,*,31,*,*,*] and [25,18,*,*,*,*]. As the order and length of schema 2 are larger than that of schema 1, schema 2 is less likely to be replicated when compared to schema 1.

The Schema Theorem as defined by Holland represented a milestone in the development of Genetic Algorithms in particular and later in the development of corresponding theorems for Genetic Programming. However, it has also several significant shortcomings which have led to more modern approaches to theorems for Genetic Algorithms.

- Only the worst-case scenario is considered. No positive effects of the search operators are considered. This has led to the development of Exact Schema Theorems [4].
- The theorem concentrates on the number of schemas surviving not which schema survives. Such considerations have been addressed by the utilization of Markov chains to provide models of behavior associated with specific individuals in the population [5].

References:

[1] Crossover 23/11/2022 [Online] https://en.wikipedia.org/wiki/Crossover_(genetic_algorithm)

[2] Zhang, D. D., 2021. Coursework B Specification, s.l.: Bath University.

[3] Zhang, D. D., 2021. EE40098_10-GAs Part 2, s.l.: Bath University.

[4] "Effective Degrees of Freedom in Genetic Algorithms," C. Stephens, H. Waelbroeck (1998) Physical review: E, 57(3), pp 3251-3264.

[5] "Modeling Genetic Algorithms with Markov Chains, Vose M.D., Nix A. (1992)" Annals of Mathematics and Artificial Intelligence, 5, pp 79-98.

Appendix:

Appendix A: Exercise 1

```
from random import randint, random
import matplotlib.pyplot as plt
import numpy as np

# Create Indiviual
def individual(min, max):
    return [randint(min, max)]

# Create the Population of Individuals
def population(count, min, max):
    return [individual(min, max) for x in range(count)]

# Define the Fitness Function. Returns the absolute value.
def fitness(individual, target):
    Total = 0
    Total = sum(individual)
    return abs(target - Total)

# Grade each target. Break the loop if the fitness has reached 0.
def grade(pop, target, fitness_history):
    sumflag = 0
    breakoutflag2 = 0

    for x in pop:
        if len(fitness_history) > 1:
            if fitness(x, target) == 0:
                breakoutflag2 = 1
                sumflag = np.NaN
        else:
            sumflag = sumflag + fitness(x, target)
    return sumflag / (len(pop) * 1.0), breakoutflag2

# Create offspring. Parameters can be changed to optimise the GA.
# Retain keeps 15% of a population to be parents and breed
# Mutate alters strings in parents
# Random Select adds individuals promoting diversity.
# Crossover breeds parents and creates children.
def evolve(pop, target, retain=0.2, random_select=0.05, mutate=0.2):
    graded = [(fitness(x, target), x) for x in pop]
    graded = [x[1] for x in sorted(graded)]
    retain_length = int(len(graded) * retain)
    parents = graded[:retain_length]

    # Randomly add other individuals to promote gentic Diversity
    for individual in graded[retain_length:]:
```

```
        if random_select > random():
            parents.append(individual)
    # Mutate some Individuals
    for x in parents:
        if mutate > random():
            pos_to_mutate1 = randint(0, len(bin(x[0])[2:]) - 1)

            x = list(bin(x[0])[2:])
            r1 = randint(0, 1)
            x[pos_to_mutate1] = r1

            strings = [str(prnt) for prnt in x]
            x = "".join(strings)
            x = str('0b' + x)
            x = [int(x, 2)]

    # Crossover to Create Children
    parents_length = len(parents)
    desired_length = len(pop) - parents_length
    children = []

    while len(children) < desired_length:
        male = randint(0, parents_length - 1)
        female = randint(0, parents_length - 1)

        if male != female:
            male = parents[male]
            female = parents[female]
            male = bin(male[0])[2:]
            female = bin(female[0])[2:]
            r = randint(0, len(male)-1)
            child = str('0b' + male[:r] + female[r:])
            child = [int(child, 2)]
            children.append(child)

    parents.extend(children)
    return parents


def checklist(L):
    total = 0
    for i in L:
        if isinstance(i, list):
            total += checklist(i)
        else:
            total += i
    return total

target = 450 # Target Number
```

```
p_count = 100 # Total Number of Individuals in Population
i_min = -50 # Minimum Value of Individual
i_max = 1000 # Maximum Value of Individual
generations = 1000 # Maximum Number of Generations
p = population(p_count, i_min, i_max)

print('Initial Population: ', p)
List1 = []
fitness_history, breakoutflag2 = grade(p, target, List1)
fitness_history = [fitness_history, ]
counter = 0


# Loop through Generations and append values.
for i in range(generations):
    p = evolve(p, target)
    error1, breakoutflag2 = grade(p, target, fitness_history)
    fitness_history.append(error1)
    Results = (checklist(p) / len(p))
    print('Average Result = ', Results, ' Average Fitness: ',
"{:.2f}".format(abs(target - Results)))
    counter += 1
    if breakoutflag2 == 1:
        break

# Print Result
if breakoutflag2 == 1:
    print('Number of generations used until achieved: ', counter)

# Plot Graph
plt.plot(fitness_history, )
plt.xlabel("Generations Used")
plt.ylabel("Graded Fitness")
plt.show()
```

Appendix B: Exercise 2

```
from random import randint, random
import matplotlib.pyplot as plt
import numpy as np

# Create Indiviual
def individual(min, max):
    return [randint(min, max)]

# Create the Population of Individuals
def population(count, min, max):
    return [individual(min, max) for x in range(count)]
```

```
# Define the Fitness Function. Returns the absolute value.
def fitness(individual, target):
    Total = 0
    Total = sum(individual)
    return abs(target - Total)

# Grade each target. Break the loop if the fitness has reached 0.
def grade(pop, target, fitness_history):
    sumflag = 0
    breakoutflag2 = 0

    for x in pop:
        if len(fitness_history) > 1:
            if fitness(x, target) == 0:
                breakoutflag2 = 1
                sumflag = np.NaN
        else:
            sumflag = sumflag + fitness(x, target)
    return sumflag / (len(pop) * 1.0), breakoutflag2

# Create offspring. Parameters can be changed to optimise the GA.
# Retain keeps 15% of a population to be parents and breed
# Mutate alters strings in parents
# Random Select adds individuals promoting diversity.
# Crossover breeds parents and creates children.
def evolve(pop, target, retain=0.2, random_select=0.05, mutate=0.2):
    graded = [(fitness(x, target), x) for x in pop]
    graded = [x[1] for x in sorted(graded)]
    retain_length = int(len(graded) * retain)
    parents = graded[:retain_length]

    # Randomly add other individuals to promote gentic Diversity
    for individual in graded[retain_length:]:
        if random_select > random():
            parents.append(individual)
    # Mutate some Individuals
    for x in parents:
        if mutate > random():
            pos_to_mutate1 = randint(0, len(bin(x[0])[2:]) - 1)

            x = list(bin(x[0])[2:])
            r1 = randint(0, 1)
            x[pos_to_mutate1] = r1

            strings = [str(prnt) for prnt in x]
            x = "".join(strings)
            x = str('0b' + x)
            x = [int(x, 2)]
```

```
    # Crossover to Create Children
    parents_length = len(parents)
    desired_length = len(pop) - parents_length
    children = []

    while len(children) < desired_length:
        male = randint(0, parents_length - 1)
        female = randint(0, parents_length - 1)

        if male != female:
            male = parents[male]
            female = parents[female]
            male = bin(male[0])[2:]
            female = bin(female[0])[2:]
            r = randint(0, len(male)-1)
            child = str('0b' + male[:r] + female[r:])
            child = [int(child, 2)]
            children.append(child)

    parents.extend(children)
    return parents


def checklist(L):
    total = 0
    for i in L:
        if isinstance(i, list):
            total += checklist(i)
        else:
            total += i
    return total

target = 450 # Target Number
p_count = 100 # Total Number of Individuals in Population
i_min = -50 # Minimum Value of Individual
i_max = 1000 # Maximum Value of Individual
generations = 1000 # Maximum Number of Generations
p = population(p_count, i_min, i_max)

print('Initial Population: ', p)
List1 = []
fitness_history, breakoutflag2 = grade(p, target, List1)
fitness_history = [fitness_history, ]
counter = 0


# Loop through Generations and append values.
for i in range(generations):
```

```
    p = evolve(p, target)
    error1, breakoutflag2 = grade(p, target, fitness_history)
    fitness_history.append(error1)
    Results = (checklist(p) / len(p))
    print('Average Result = ', Results, ' Average Fitness: ',
"{:.2f}".format(abs(target - Results)))
    counter += 1
    if breakoutflag2 == 1:
        break

# Print Result
if breakoutflag2 == 1:
    print('Number of generations used until achieved: ', counter)

# Plot Graph
plt.plot(fitness_history, )
plt.xlabel("Generations Used")
plt.ylabel("Graded Fitness")
plt.show()
```

Appendic C: Exercise 3

```
import matplotlib.pyplot as plt

from functools import reduce
from operator import add
from random import randint, random

# Individual Created between Min and Max. Called p_count times to create
population.
def individual(length, min, max):
    return [ randint(min,max) for x in range(length) ]

# Create Population of Inidividuals
def population(count, length, min, max):
    return [ individual(length, min, max) for x in range(count) ]

# Calculate Fitness of Individuals
def fitness(individual, target):
    sum = reduce(add, individual, 0)
    return abs(target-sum)

# Ranking each Inidividual
def grade(pop, target):
    summed = reduce(add, (fitness(x, target) for x in pop))
    return summed / (len(pop) * 1.0)

# Create offspring. Parameters can be changed to optimise the GA.
```

```
def evolve(pop, target, retain=0.2, random_select=0.05, mutate=0.01):
    graded = [ (fitness(x, target), x) for x in pop]
    graded = [ x[1] for x in sorted(graded)]
    retain_length = int(len(graded)*retain)
    parents = graded[:retain_length]

    for individual in graded[retain_length:]:
        if random_select > random():
            parents.append(individual)

    for individual in parents:
        if mutate > random():
            pos_to_mutate = randint(0, len(individual)-1)

            individual[pos_to_mutate] = randint(
                min(individual), max(individual))

    parents_length = len(parents)
    desired_length = len(pop) - parents_length
    children = []

    while len(children) < desired_length:
        male = randint(0, parents_length-1)
        female = randint(0, parents_length-1)

        if male != female:
            male = parents[male]
            female = parents[female]
            half = len(male) // 2
            child = male[:half] + female[half:]
            children.append(child)

    parents.extend(children)
    return parents


target = 550 # Target Number
p_count = 100 # Total Number of Individuals in Population
i_length = 6 # Length of Individuals
i_min = 1 # Minimum Value of Individual
i_max = 1000 # Maximum Value of Individual
generations = 1000 # Maximum Number of Generations
p = population( p_count, i_length, i_min, i_max )
fitness_history = [grade(p, target)]

breakoutflag = False

# Loop until Fitness reaches 0, fitness has repeated itself, or Generation cap
is reached.
```

```
for i in range(generations):
    p = evolve(p, target)
    fitness_history.append(grade(p, target))
    for datum in fitness_history:
        if datum == 0:
            breakoutflag = True
            print('Number of Iterations:', len(fitness_history))
            break
        if len(fitness_history) > 1 and datum != 0:
            while fitness_history[-1] == fitness_history[-5:]:
                breakoutflag = True
                break
    if breakoutflag:
        break

print('Target Number:', target)
print('Population Size:', p_count)
print('Maximum Number of Generations:', generations)
print('Final Fitness Value:', fitness_history[-1])
plt.plot(fitness_history)
plt.xlabel("Generations")
plt.ylabel("Fitness History")
plt.title('Target Number: %3.2f' % target)
plt.grid()
plt.show()
```

Appendix D: Exercise 4 Approach 1

```
from random import randint, random
import matplotlib.pyplot as plt
from tqdm import tqdm
import numpy as np

# Define the Target Curve
def targetcurve(x):
    return 25 * x ** 5 + 18 * x ** 4 + 31 * x ** 2 - 14 * x ** 2 + 7 * x -19

# Create Indiviual
def individual(min, max):
    return [randint(min, max) for x in range(length)]

# Create the Population of Individuals
def population(count, min, max):
    return [individual(min, max) for x in range(count)]
```

```
# Define the Fitness Function. Returns the absolute value.
def fitness(individual, target):
    target = np.array(target)
    individual = np.array(individual)
    fit = target - individual
    return np.average(abs(fit))

# Grade each target. Break the loop if the fitness has reached 0.
def grade(pop, target):
    summed = 0
    breakoutflag = 0
    for x in pop:
        if fitness(x, target) == 0:
            breakoutflag = 1
            summed = np.NaN
            break
        else:
            summed = summed + fitness(x, target)
    return summed / (len(pop) * 1.0), breakoutflag

# Create offspring. Parameters can be changed to optimise the GA.
# Retain keeps 15% of a population to be parents and breed
# Mutate alters strings in parents
# Random Select adds individuals promoting diversity.
# Crossover breeds parents and creates children.
def evolve(pop, target, retain=0.15, random_select=0.05, mutate=0.3):
    graded = [(fitness(x, target), x) for x in pop]
    graded = [x[1] for x in sorted(graded)]
    retain_length = int(len(graded) * retain)
    parents = graded[:retain_length]

    for individual in graded[retain_length:]:
        if random_select > random():
            parents.append(individual)

    for individual in parents:
        if mutate > random():
            pos_to_mutate = randint(0, len(individual) - 1)
            individual[pos_to_mutate] = randint(i_min, i_max)

    parents_length = len(parents)
    desired_length = len(pop) - parents_length
    children = []

    while len(children) < desired_length:
        male = randint(0, parents_length - 1)
        female = randint(0, parents_length - 1)
```

```
        if male != female:
            male = parents[male]
            female = parents[female]
            half = len(male) // 2
            child = male[:half] + female[half:]
            children.append(child)

    parents.extend(children)
    return parents


target = [25, 18, 31, -14, 7, -19] # Target List
p_count = 100 # Total Number of Individuals in Population
i_min = -50 # Minimum Value of Individual
i_max = 50 # Maximum Value of Individual
length = 6 # Length of Individuals
generations = 250 # Maximum Number of Generations

p = population(p_count, i_min, i_max)
fitness_history, breakoutflag = grade(p, target)
fitness_history = [fitness_history, ]

list1 = []
list2 = []

# Target Curve within range -100 to 100.
for x in range(-100, 100, 10):
    y = targetcurve(x)
    list1.append(y)
    list2.append(x)

# Loop until Fitness reaches 0 or Generation cap is reached. TQDM shows
progress bar.
for i in tqdm(range(generations)):
    p = evolve(p, target)
    error1, breakoutflag = grade(p, target)
    fitness_history.append(error1)
    if breakoutflag == 1:
        break

graded = [(fitness(x, target), x) for x in p]
graded = [x[1] for x in sorted(graded)]
parents = graded[0]
print(parents)

list3 = []
list4 = []

# Plot GA coefficient polynomial within range of -100 to 100.
```

```
for x in range(-100, 100, 10):
    y = parents[0] * x ** 5 + parents[1] * x ** 4 + parents[2] * x ** 3 +
parents[3] * x ** 2 +  parents[4] * x - parents[5]
    list3.append(x)
    list4.append(y)

# Plot Graph of Results
plt.subplot(121)
plt.plot(list2, list1, label = "Target Curve")
plt.plot(list3, list4, '--', label = "Output Curve")
plt.xlabel("X")
plt.ylabel("Y")
plt.legend()
plt.subplot(122)
plot2 = plt.plot(fitness_history)
plt.xlabel("Generations")
plt.ylabel("Population Fitness")
plt.show()
```

Appendix D: Exercise 4 Approach 2

```
from random import randint, random
import matplotlib.pyplot as plt
from tqdm import tqdm
import numpy as np

# Define the Target Curve
def targetcurve(x):
    return 25 * x ** 5 + 18 * x ** 4 + 31 * x ** 2 - 14 * x ** 2 + 7 * x -19

# Create Indiviual
def individual(min, max):
    return [randint(min, max) for x in range(length)]

# Create the Population of Individuals
def population(count, min, max):
    return [individual(min, max) for x in range(count)]

# Define the Fitness Function. Returns the absolute value.
def fitness(individual, target):
    target = np.array(target)
    individual = np.array(individual)
    fit = target - individual
    return np.average(abs(fit))

# Grade each target. Break the loop if the fitness has reached 0.
def grade(pop, target):
    summed = 0
```

```
    breakoutflag = 0
    for x in pop:
        if fitness(x, target) == 0:
            breakoutflag = 1
            summed = np.NaN
            break
        else:
            summed = summed + fitness(x, target)
    return summed / (len(pop) * 1.0), breakoutflag

# Create offspring. Parameters can be changed to optimise the GA.
def evolve(pop, target, retain=0.15, random_select=0.05, mutate=0.3):
    graded = [(fitness(x, target), x) for x in pop]
    graded = [x[1] for x in sorted(graded)]
    retain_length = int(len(graded) * retain)
    parents = graded[:retain_length]

    for individual in graded[retain_length:]:
        if random_select > random():
            parents.append(individual)

    for individual in parents:
        if mutate > random():
            pos_to_mutate = randint(0, len(individual) - 1)
            individual[pos_to_mutate] = randint(i_min, i_max)

    parents_length = len(parents)
    desired_length = len(pop) - parents_length
    children = []

    while len(children) < desired_length:
        male = randint(0, parents_length - 1)
        female = randint(0, parents_length - 1)

        if male != female:
            male = parents[male]
            female = parents[female]
            half = len(male) // 2
            child = male[:half] + female[half:]
            children.append(child)

    parents.extend(children)
    return parents


target = [25, 18, 31, -14, 7, -19] # Target List
p_count = 100 # Total Number of Individuals in Population
i_min = -50 # Minimum Value of Individual
i_max = 50 # Maximum Value of Individual
```

```
length = 6 # Length of Individuals
generations = 250 # Maximum Number of Generations

p = population(p_count, i_min, i_max)
fitness_history, breakoutflag = grade(p, target)
fitness_history = [fitness_history, ]

list1 = []
list2 = []

# Target Curve within range -100 to 100.
for x in range(-100, 100, 10):
    y = targetcurve(x)
    list1.append(y)
    list2.append(x)

# Loop until Fitness reaches 0 or Generation cap is reached. TQDM shows
progress bar.
for i in tqdm(range(generations)):
    p = evolve(p, target)
    error1, breakoutflag = grade(p, target)
    fitness_history.append(error1)
    if breakoutflag == 1:
        break

graded = [(fitness(x, target), x) for x in p]
graded = [x[1] for x in sorted(graded)]
parents = graded[0]
print(parents)

list3 = []
list4 = []

# Plot GA coefficient polynomial within range of -100 to 100.
for x in range(-100, 100, 10):
    y = parents[0] * x ** 5 + parents[1] * x ** 4 + parents[2] * x ** 3 +
parents[3] * x ** 2 +  parents[4] * x - parents[5]
    list3.append(x)
    list4.append(y)

# Plot Graph of Results
plt.subplot(121)
plt.plot(list2, list1, label = "Target Curve")
plt.plot(list3, list4, '--', label = "Output Curve")
plt.xlabel("X")
plt.ylabel("Y")
plt.legend()
plt.subplot(122)
```

```
plot2 = plt.plot(fitness_history)
plt.xlabel("Generations")
plt.ylabel("Population Fitness")
plt.show()
```