(Zhang, 2021)Coursework B: Genetic Algorithms

Exercise 1-3:

"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 code provided produced a population of individuals each consisting of an array of length 'n'. Each number in the array acted as a specific gene of an individual, confined between a limit between 'i_min' and 'i_max'. The sum of each array is the integer the individual represents.

The modifications made to this genetic algorithm was to remove how the integers are represented as arrays of length 'n', but just to have them as one specific integer. This means how mutation and crossover is handled must change.

The **Individuals** are a specific integer initially confined between the limits 'i_min' and 'i_max'.

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

The **Fitness** is determined by the distance an individual is from the 'target', where the target is desired specific integer output.

The **Parent Selection** is carried out by grading all of the population by ranking them in order of fitness. The top 20% of the graded population are kept and used as parents, when the rest are discarded. This is an example of elitism as only the best graded individuals are kept and unchanged. However, a random 5% of the population is also added to the parents to promote genetic diversity and to reduce the likelihood of the genetic algorithm to tend to a close but incorrect answer.

The **Mutation** occurs when a randomly generated integer is less than a predefined mutation probability 'mutate'. The integer is turned into its binary form and a random number in the binary string is changed to either a 1 or 0.

The **Crossover** is to produce the children that will populate the other 80% of the population. The parents are converted to binary, a random point along the binary string is chosen and at this point the parents are cut and crossed over, as shown in figure 1.

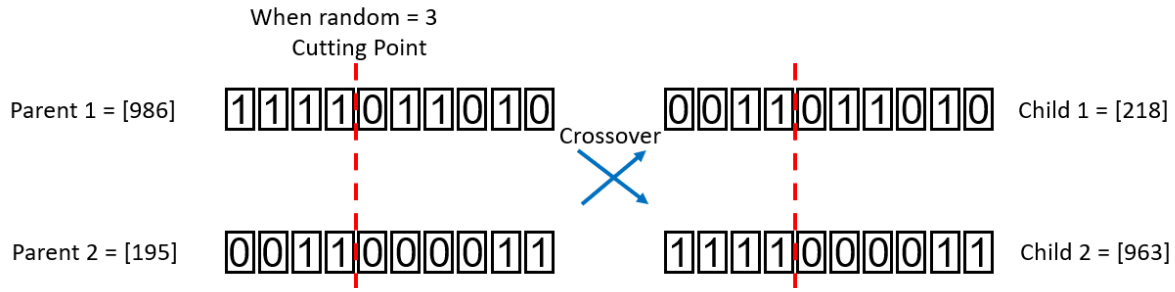

Figure 1. Binary Crossover

After a set number of 'Generations', the fittest individual from the final population is output. This is the final result of the genetic algorithm. The code can be found in *Appendix A, Exercise 1.*

"How do the population size, mutation and crossover parameters (such as mutation probability and crossover probability) affect how quickly a solution is found?"

To test changing the population size, probability of mutation and the retain percentage a set of controlled parameters must be selected. At each parameter change it is tested 1000 times. The rest of the initial parameters are set as follows: p_count = 100, Mutation = 0.01, Retain = 0.2, Random_select = 0.05, i_min = 1, i_max = 100 while guessing a random integer between 1 & 100.

Varying Graded Population Retain Percentage (GPRP)

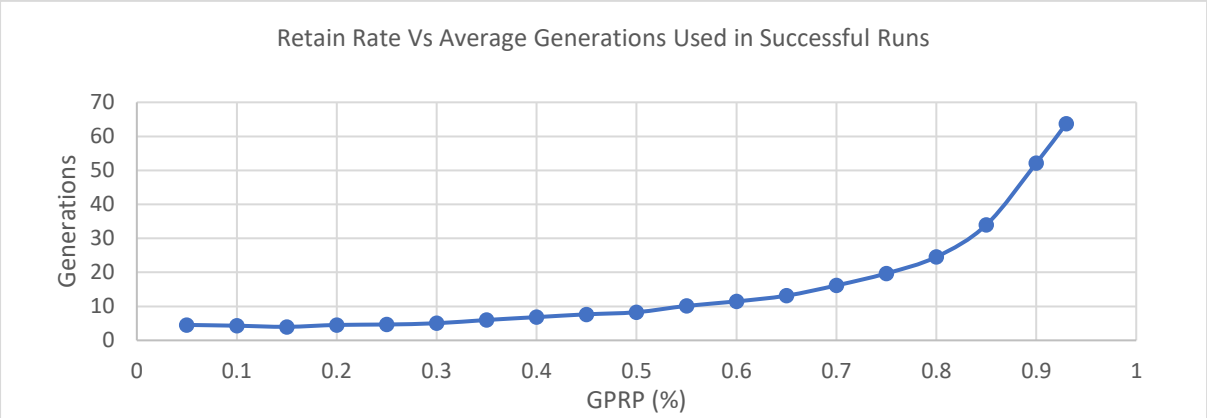

Figure 2. GPRP Vs Average Generations Used in Successful Runs

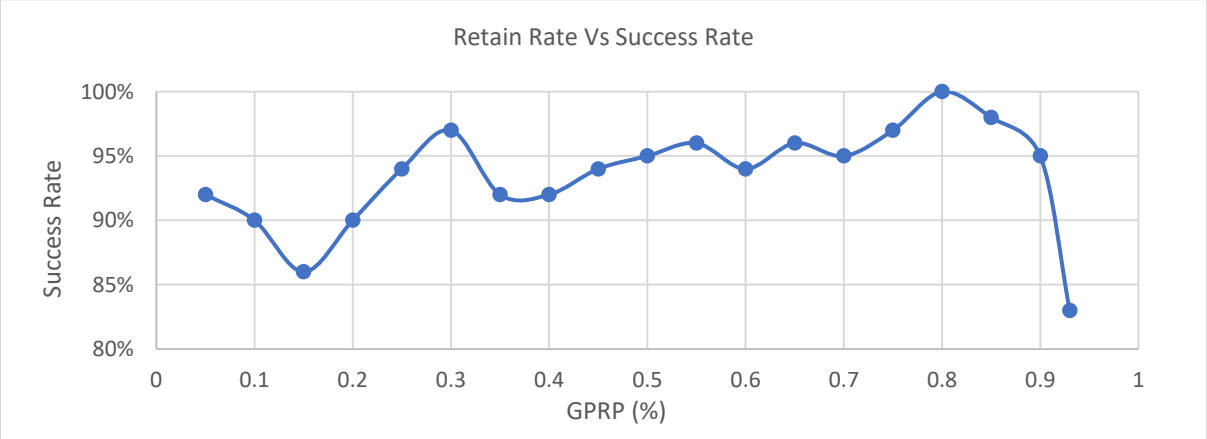

Figure 3. GPRP Vs Success Rate

The number of generations needed to reach the target increases exponentially (shown in Figure 2) as, without random mutations, when the GPRP = 1 the GA would never reach the desired target. This trend can be attributed to the fact that making the selection of the graded population bigger reduces the impact of the ranking in the parent selection. This means more generations are required to focus the population to trend to the desired output.

Success fluctuated randomly but remained predominantly above 90%, as shown in Figure 3. This is as the retain percentage doesn't affect the trending of the GA to reach the desired target, it just slows down the ranking approach as more unfit individuals are used as parents.

Varying Population Size

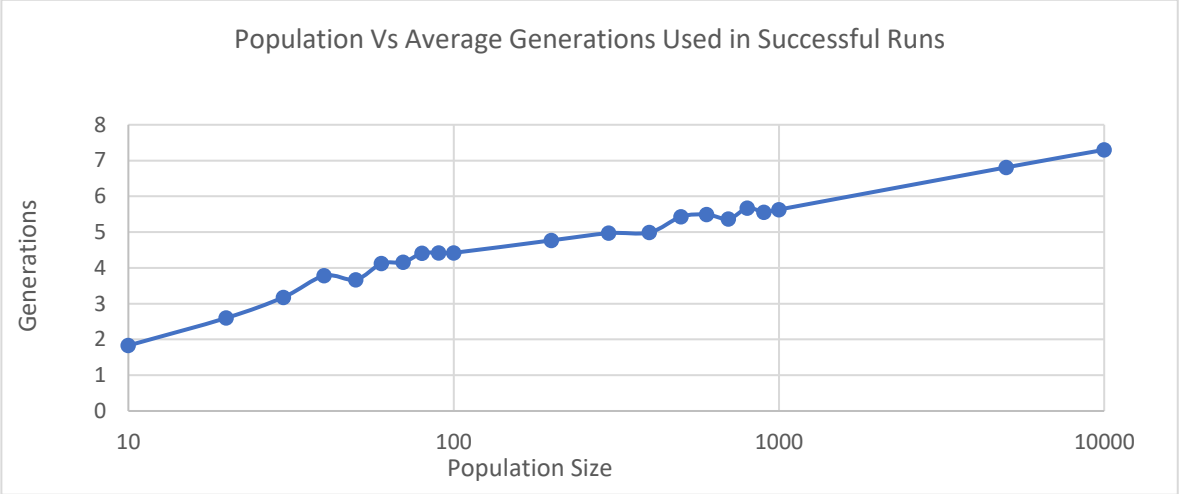

Figure 4. Population Vs Generations

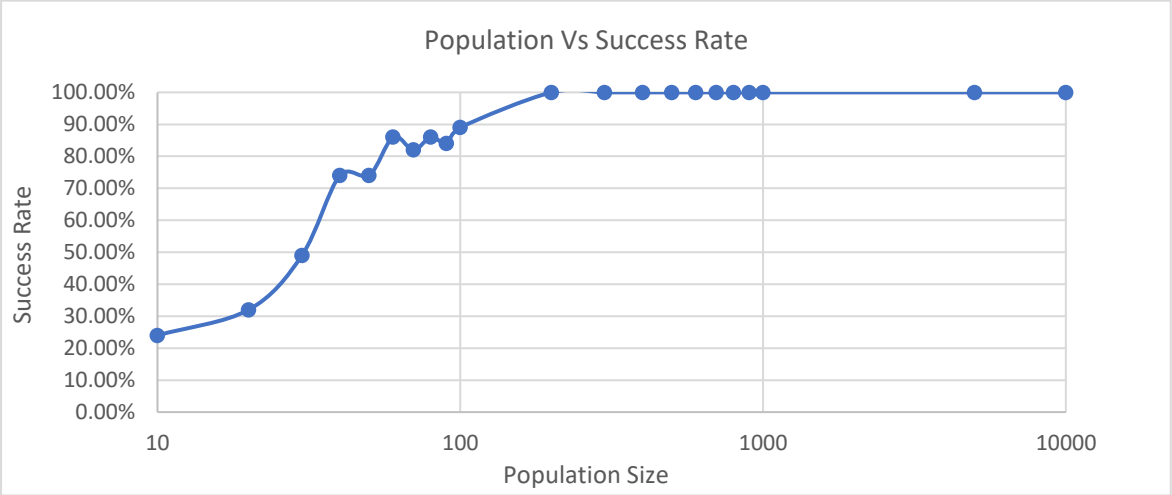

Figure 5. Population Vs Success Rate

A small positive linear trend is evident from Figure 4 between population size and generations used. The explanation of this is due to the way the GA is stopped when the target is reached. Here the GA was stopped when the overall population had a perfect fitness rather than just one individual. This means that as the size of the population increases the greater the number of individuals there are to reach the target. This is very small of only a max change of 5 generations between a population size of 10 to 10000 as the retain of 20% quickly tends the population to the desired target.

Population size also has a positive relationship with success rate and plateaus at the maximum of 100% from a population size of 200 onwards. This can be attributed by the fact that with a higher population size for such a simple task of finding one integer, the chances of guessing the correct answer in the first generation is much higher. Therefore, the GA is less reliant on random mutations and the efficacy of the crossover function.

Varying Mutation Probability

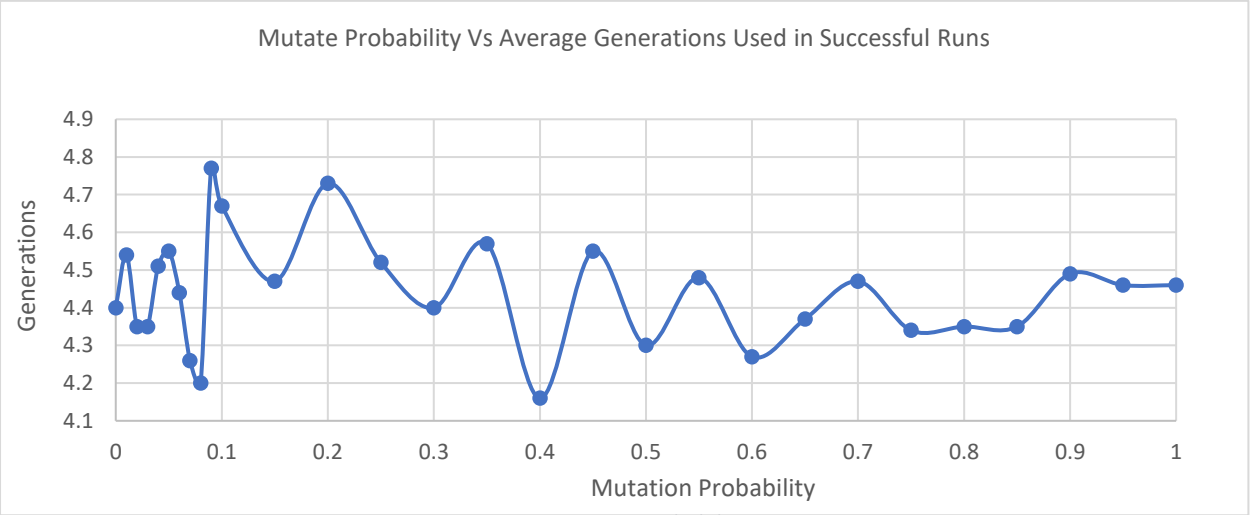

Figure 6. Mutation Probability Vs Generations

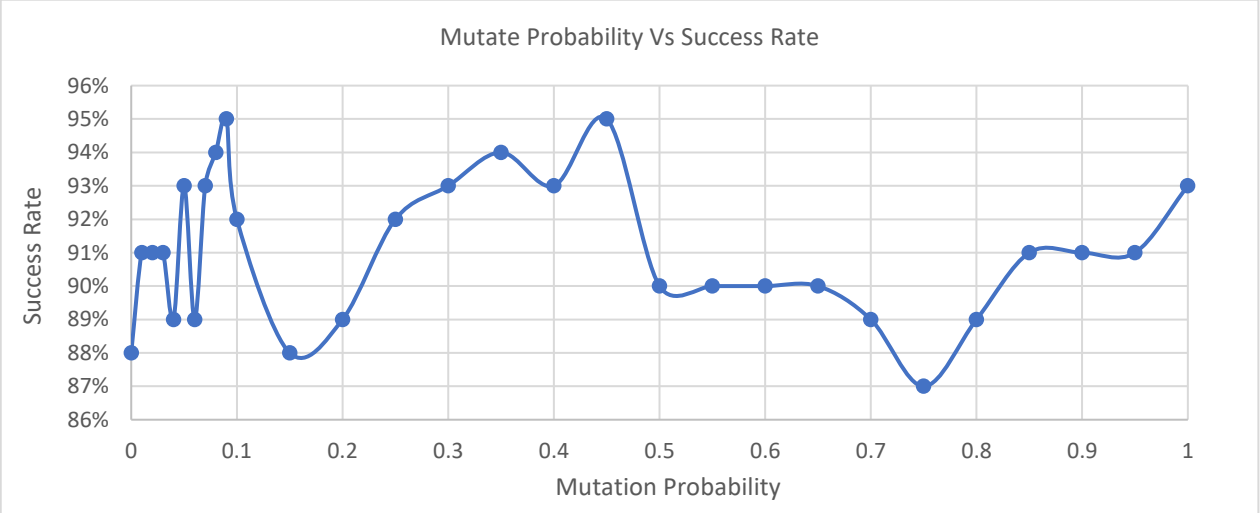

Figure 7. Mutation Probability Vs Success Rate

For both success rate and generations, mutation probability has no correlation. A reason for this could be the mutation function programmed in this GA. As any random number in the binary string can be changed to a random integer of either 0 or 1, a mutation could not result in a change of the individual as a 1 could be replaced with a 1. This results in the chances of a mutation to occur to be greatly reduced.

To make the mutation more effective, it could be changed that for what random integer is picked in the binary string has to be changed to its opposite counterpart so that a 0 must change into a 1 or a 1 into a 0.

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

Three methods were considered to stop the genetic algorithm rather than run each test with excessive generations.

Method 1: Stop when the population has an overall fitness of 0

```
Output = (nested_sum(p) / len(p))
# print('Average Output = ', Output, ' Average Fitness: ', "{:.2f}".format(abs(target - Output)))
if Output == target:
    break
```

Figure 8. Method 1 to reduce excessive computation

Where 'Output' is the average fitness of the population. When Output is exactly 0 (perfect) the GA stops running. This works well for when the answer is reached, however when the GA gets stuck and tends to an incorrect value it will carry on until it has completed the number of n generations have been completed.

Method 2: Stop when a specific individual has a fitness of 0

```
for x in pop:
    if fitness(x, target) == 0:
        flag = 1
        summed = np.NaN
        break
```

Figure 9. Method 2 to reduce excessive computation

When one of the individuals in the population has a perfect fitness (0) the target has been found and the flag is raised, and the GA is stopped. This makes more sense than 'Method 1' as it removes the wasted time of running the GA until the entire population has reached the target. There is also no risk of losing what is the correct answer in the population to a random mutation.

This method also endures the same problem for when a GA gets stuck and tends to the incorrect answer, as it has to still run through the number of n generations predefined until ending.

Method 3: Stop when the fitness is unchanged for X number of generations

```
for x in pop:
    if len(fitness_history) > 1:
        if fitness_history[-1] == (sum(fitness_history[-100:])/100):
            flag1 = 1
            summed = np.NaN
            break
```

Figure 10. Method 3 to reduce excessive computation

Stopping the GA when there has been no improvement for X number of generations covers all runs. This is as it stops the GA for when both it is correct and when it gets stuck on an incorrect guess. In Figure 10 the X value of generations chosen was 100. This is as the task is very simple so the number of generations on average needed to reach an output is around 2 to 7. 100 also allows for a possible mutation if the GA is stuck on an incorrect path as the mutation is 1% chance.

The final method used was to incorporate both 'Method 2' and 'Method 3'. This has all the benefits of not wasting time for the entire population to have to have a fitness of 0 when one individual already is the target value and stops the runs for then the GA gets stuck on an incorrect path before stopping at an excessive specified n number of generations. This is shown to be used in *Appendix A, Exercise 1*.

Exercise 4

"Create an optimization to optimize a set of parameters for a curve of a 5th -order polynomial as given below:"

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

Approach 1

An initial method of providing a target of the list [25, 18, 31, -14, 7, -19] was used, shown in Appendix B, Exercise 4 Approach 1. Being very similar to Exercise 1, it can find the exact list of coefficients of the curve but rather than an individual number target it is a list of 6. Rather than using binary for the mutation and crossover, each coefficient guess in the list is treated as a gene.

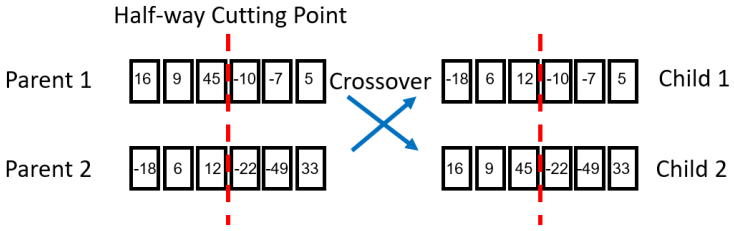

Figure 11. Polynomial Coefficient Crossover, Approach 1

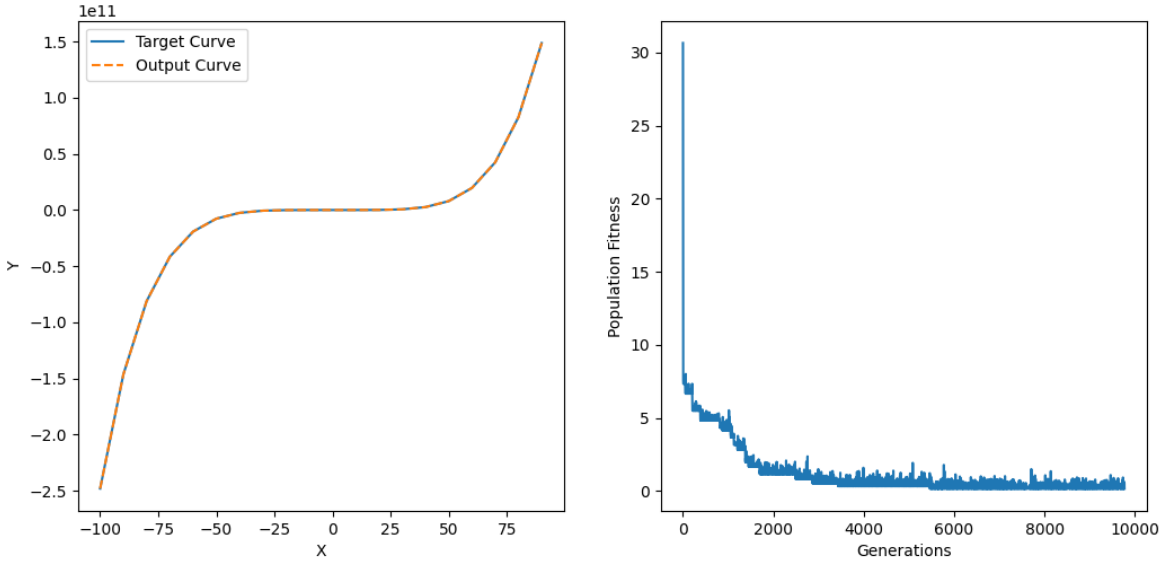

Figure 12. Output Curve compared to Target Curve, Population Fitness improvement with Generations

This approach took 9768 generations to reach the exact coefficients. The 'Target Curve' on the left of Figure 12 is exactly the 'Output Curve' as the exact coefficients were found, [25, 18, 31, -14, 7, -19]. This approach works however seems useless as it doesn't produce any new information. From this a better approach was produced to fit a curve with only providing a range of points on the 'Target Curve' as the targets.

Approach 2

Curve fitting using a genetic algorithm seems more useful as it produces a curve similar to or identical to the target polynomial with just a range of points from the 'Target Curve'. In order to produce a curve equation with the output curve, the genetic algorithm is also given a 5^{th} polynomial equation structure so that coefficients are found. The code can be found in Appendix C, .

A range of y-values are produced with a predetermined x- values initially from the given polynomial equation to act as the **Target** values.

The **sub-individuals** are an array of length 'n'. The sum of the array is the integer the sub-individual represents. Therefore, a single integer in the 'sub-individual's' array represents a **gene** These represent one individual coefficient.

The 'sub-individual' then produces a string of 6 by 'n', producing one **Individual**.

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

The **Fitness** is determined by the distance between the target y-values and an 'individual's' y-output using each 'sub-individual' as its specific coefficient in a 5^{th} order polynomial equation, in a pre-determined range of x-values.

The **Parent Selection** is the same method used as in Exercise 1 where the population is graded by ranking them in order of fitness and the top 20% of the graded population are kept and used as parents, when the rest are discarded. A random 5% of the population is also added to the parents to promote genetic diversity.

The **crossover** function used is a fixed single point crossover. Every 'sub-individual' array is halved in each male and female individual and is crossed over with its corresponding 'sub-individual' of the other parent. This is visually represented in Figure 13. This is beneficial compared to crossing over the entire 'Individual' array as it focuses on achieving each specific coefficient.

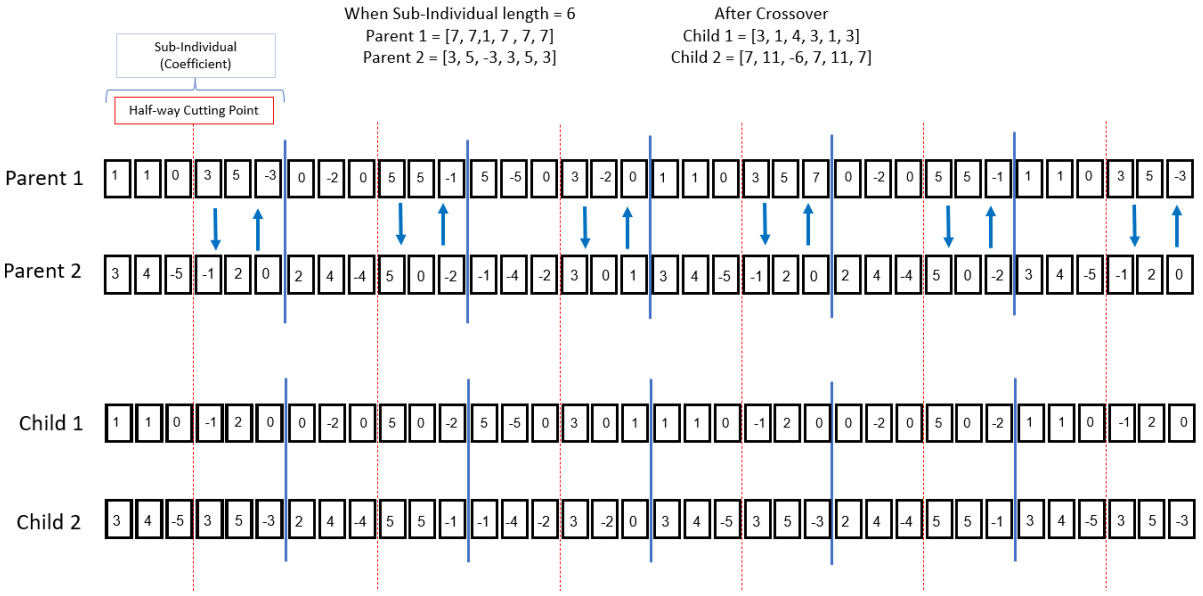

Figure 13. Polynomial Coefficient Crossover, Approach 2

The **mutation** changes a random single 'gene' in a random 'sub-individual' of a random 'individual'. The mutation is a random number between the predefined limit between 'i_max' and 'i_min'.

The initial parameters for a reasonable result, shown in Figure 14, were set as follows:

p_count = 100, Mutation = 0.01, Retain = 0.2, Random_select = 0.05, i_min = -10, i_max = 10, Generations = 100.

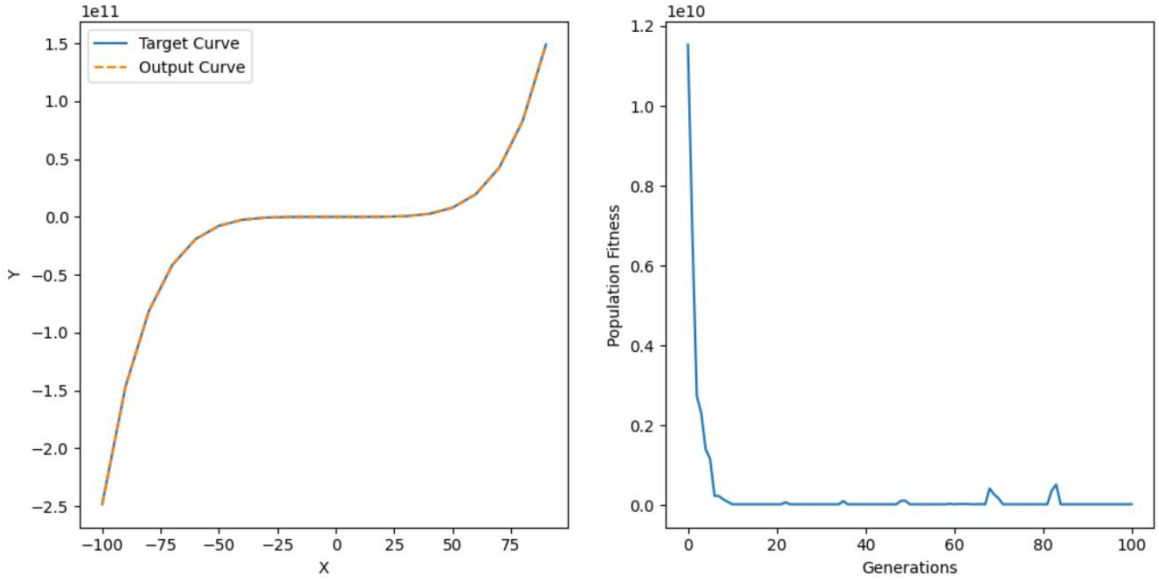

Figure 14. Output Curve compared to Target Curve, Population Fitness improvement with Generations, Approach 2

The output coefficients produced from this run were [**25, 18,** 13, -5, -19, -25]. Both the x^5 and x^4 coeffients have been found with a correlation of 0.9999999999999452 between the Target and Output curves.

The correlation was calculated using the 'Pandas' package which has an inbuilt correlation coefficient calculator for two curves. Using this coefficient was tested as the fitness output, however the computational time was much longer resulting in it being impractical to use.

Analysis of Approaches 1 & 2

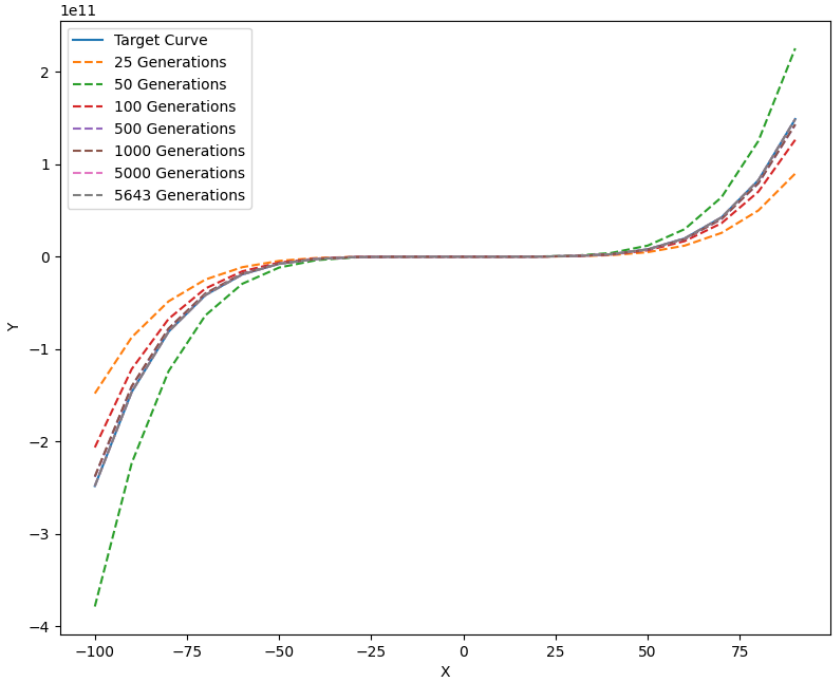

Figure 15. Results of Approach 1 with varying number of generations.

Running through using different numbers of generations for Approach 1, the results are what would be expected in Figure 15. The more generations used the closer the GA reaches the target array of coefficients, reaching the answer at 5643 generations in this example.

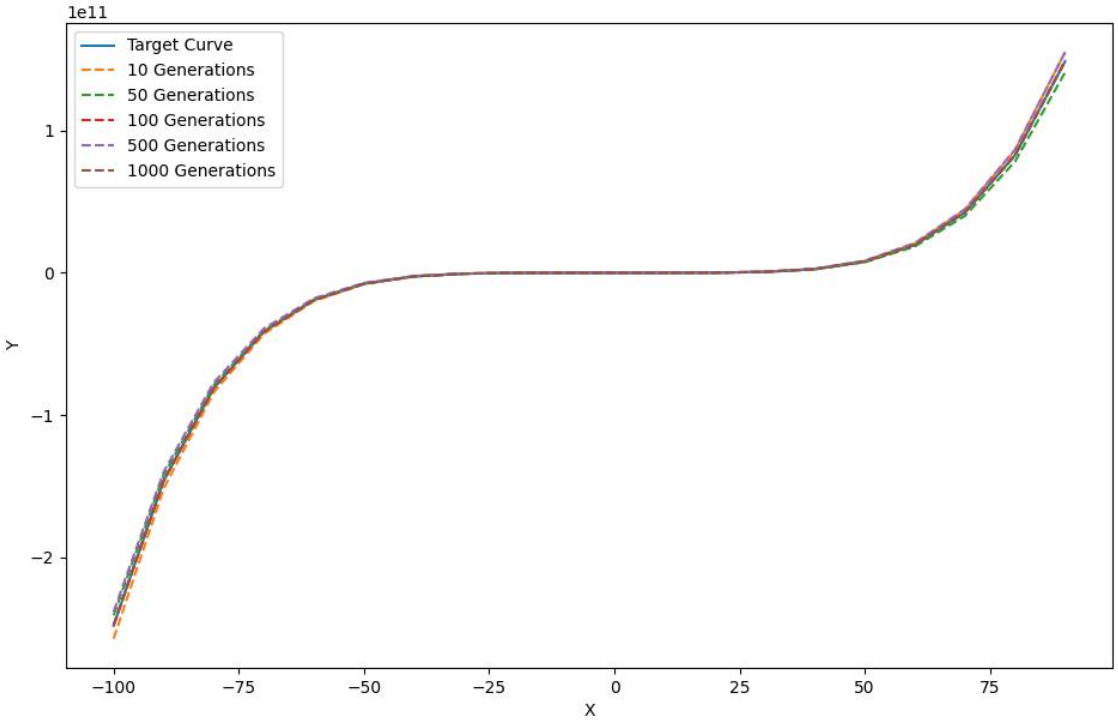

Figure 16. Results of Approach 2 with varying number of generations.

Running through using different numbers of generations for Approach 2, the same correlation is found as in Approach 1, however the variance of the results is less as Approach 2 find the x^5 and x^4 coefficients very quickly, which are the most impacting coefficients when following the curve of the given 5th order polynomial.

Exercise 5

"Understanding on Holland's Schema Theorem"

John Holland asked the question "How do Genetic Algorithms work?". To answer this, he developed what is now known as the "Schema Theorem".

The Schema Theorem can be explained considering Binary genetic algorithms, similar to Exercise 1 and can be summarised with,

"Short, low-order, above-average schemata receive exponentially increasing trials in subsequent generations of a genetic algorithm." [2]

A chromosome in the population will increase its number of copies of itself in the children in the successive generation. The fitter or better quality the chromosome is, the more likely it will be reproduced and survive throughout each generation. How this occurs can be explained using a schema [1].

A schema is a gene pattern that can exist within a chromosome [3]. For binary it is template that represents a subset of a binary string with '0's, '1's and '*'s, where *s are "don't care genes" (represents 0 or 1). If a chromosome has the specified bits of the schema, then it matches.

For example:

Schema 10101 matches 1 string: {10101}
Schema 1*011 matches the 2 strings: {10011, 11011}.
Schema *101* matches the 4 strings: {01010, 11010, 11011, 01011}
Schema ***** matches any 5-bit string.
String 10010 is matched by 2^5 schemas: {10010, *0010, 1*010, …, ****0, *****}

A string '11' has a length L=2 and belongs to 2^2different 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 order of the most significant bits.

In Exercise 1, the genetic algorithm used the crossing over and mutating of binary strings, representing a random integer between 1 and 100. The fitness was defined by the distance of the individual from the target. This means the most significant bit will be the bit that changes the size of the decimal number it represents by the most. In Exercise 1, with a range of a 100 it will use 7bit long strings. The dominating bit in this instance will be the 1st bit in the string as it represents 64 as an integer. Once the fittest schema of the 1st bit is dominant it will begin to look for the fittest schema with the 1st 2bits of the string as the 2nd bit is the second most significant bit. This continues until a sigma of L=7 is found through many generations, mutations, and crossovers.

References

[1] *Binary Genetic Algorithm - Part 7: Why do GAs work, Schema Theorem.* October, 2021. [Film] Directed by HK Lam. s.l.: s.n.

[2] Z. Michalewicz, D. D., 1997. *Evolutionary Algorithms in Engineering Applications.* Berlin: Springer.

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

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

Binary Genetic Algorithm - Part 7: Why do GAs work, Schema Theorem. October, 2021. [Film] Directed by HK Lam. s.l.: s.n.

Z. Michalewicz, D. D., 1997. *Evolutionary Algorithms in Engineering Applications.* Berlin: Springer.

Zhang, D. D., 2021. *EE40098_10-GAs Part 2,* s.l.: Bath University.

Appendix

Appendix A: Exercise 1

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

# Initial Variables
target = 550
p_count = 100
i_min = 0
i_max = 1000
generations = 1000


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


def population(count, min, max):
    return [individual(min, max) for x in range(count)]


def fitness(individual, target):
    """Determine the fitness of an individual. Lower is better."""
    Sum = 0
    Sum = sum(individual)
    return abs(target - Sum)


def grade(pop, target, fitness_history):
    """Find average fitness for a population."""
    summed = 0
    flag1 = 0
    flag2 = 0
    for x in pop:
        if len(fitness_history) > 1:
            if fitness_history[-1] == (sum(fitness_history[-100:])/100):
                flag1 = 1 # Flag raised when 100 iterations of fitness are
unchanged
                summed = np.NaN
                break
            if fitness(x, target) == 0:
                flag2 = 1 # Flag raised when an individual is found with a
fitness of 0
                summed = np.NaN
        else:
            summed = summed + fitness(x, target)
    return summed / (len(pop) * 1.0), flag1, flag2
```

```
def evolve(pop, target, retain=0.2, random_select=0.05, mutate=0.01):
    graded = [(fitness(x, target), x) for x in pop]
    # produces list of individuals, paired with fitness of whole population
    graded = [x[1] for x in sorted(graded)]
    # orders list of individuals in order of fittest to least fit
    retain_length = int(len(graded) * retain)
    # determines number of parents kept from 'retain' & population size
    parents = graded[:retain_length]
    # keeps fittest 'retain_length' of graded population
    # randomly add other individuals to promote genetic diversity
    for individual in graded[retain_length:]:
        if random_select > random():
            parents.append(individual)
    for x in parents:
        if mutate > random():
            pos_to_mutate1 = randint(0, len(bin(x[0])[2:]) - 1)
            # Random position is selected to mutate along the binary string
            x = list(bin(x[0])[2:])
            r1 = randint(0, 1)
            x[pos_to_mutate1] = r1
            # A random bit is changed to either a 1 or 0
            strings = [str(v) for v in x]
            x = "".join(strings)
            x = str('0b' + x)
            x = [int(x, 2)]
    # crossover parents 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]
            # Located male and female parents
            male = bin(male[0])[2:]
            female = bin(female[0])[2:]
            r = randint(0, len(male)-1)
            # Random cutting point for the crossover is selected
            child = str('0b' + male[:r] + female[r:])
            # The parents crossover
            child = [int(child, 2)]
            children.append(child)
    parents.extend(children)
    return parents


def nested_sum(L):
    total = 0
    for i in L:
        if isinstance(i, list):  # checks if `i` is a list
            total += nested_sum(i)
        else:
            total += i
    return total


# Example usage
BigF = []
p = population(p_count, i_min, i_max)
```

```
print('initial population: ', p)
fitness_history, flag1, flag2 = grade(p, target, BigF)
fitness_history = [fitness_history, ]
count = 0
for i in range(generations):
    p = evolve(p, target)
    average_error, flag1, flag2 = grade(p, target, fitness_history)
    fitness_history.append(average_error)
    Output = (nested_sum(p) / len(p))
    print('Average Output = ', Output, ' Average Fitness: ',
"{:.2f}".format(abs(target - Output)))
    count += 1
    if flag1 or flag2 == 1:
        break # When a flag is raised, the run is ended
if flag1 == 1:
    print('Number of generations used until trending: ', count - 100)
elif flag2 == 1:
    print('Number of generations used until trending: ', count)

Accuracy = (target / Output) * 100
if Accuracy > 100:
    Accuracy = 100 - (Accuracy - 100)

print(Accuracy, '% Accuracy')

# Create the plot

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

Appendix B: Exercise 4 Approach 1

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

# Example usage
target = [25, 18, 31, -14, 7, -19] # Target Coefficients
p_count = 100 # Population size
i_min = -50 # Lowest of range
i_max = 50 # Highest of range
generations = 100 # Number of generations


def objective(x):
    return 25 * x ** 5 + 18 * x ** 4 + 31 * x ** 2 - 14 * x ** 2 + 7 * x -
19

# Objective class is used to produce a range of y points on the given
polynomial between -100 and 99
yv = []
xv = []
xy = []
for x in range(-100, 100, 10):
    y = objective(x)
    yv.append(y)
    xv.append(x)


def individual(min, max):
    """Create a member of the population."""
    return [randint(min, max) for x in range(6)]


def population(count, min, max):
    """Create a number of individuals """
    return [individual(min, max) for x in range(count)]


def fitness(individual, target):
    """Determine the fitness of an individual. Lower is better."""
    target = np.array(target)
    individual = np.array(individual)
    fit = target - individual
    # print(np.average(abs(fit)))
    return np.average(abs(fit))


def grade(pop, target):
    """Find average fitness for a population."""
    summed = 0
    flag = 0
    for x in pop:
        if fitness(x, target) == 0:
            flag = 1 # Flag raised when an individual of fitness 0 is found
            summed = np.NaN
            break
        else:
            summed = summed + fitness(x, target)
    return summed / (len(pop) * 1.0), flag
```

```
def evolve(pop, target, retain=0.2, random_select=0.05, mutate=1):
    global male
    graded = [(fitness(x, target), x) for x in pop]
    graded = [x[1] for x in sorted(graded)]  # Ranking of population
    retain_length = int(len(graded) * retain)
    parents = graded[:retain_length]  # Retaining the fittest 20%
    # randomly add other individuals to promote genetic diversity
    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)
            # crossover parents 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]
            half = len(male) // 2
            child = male[:half] + female[half:]
            # Parents Crossover
            children.append(child)
    parents.extend(children)
    return parents


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

for i in tqdm(range(generations)):
    p = evolve(p, target)
    average_error, flag = grade(p, target)
    fitness_history.append(average_error)
    if flag == 1:
        break

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

a = []
b = []
# Producing the y values of the polynomial with the newly generated
coefficients
for x in range(-100, 100, 10):
    y = parents[0] * x ** 5 + parents[1] * x ** 4 + parents[2] * x ** 2 +
parents[3] * x ** 2 + \
        parents[4] * x - parents[5]
    a.append(x)
    b.append(y)

plt.subplot(121)
```

```
plt.plot(xv, yv, label="Target Curve")
plt.plot(a, b, '--', 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 C, Exercise 4 Approach 2

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


# Initial Variables
p_count = 100  # Population Size
i_length = 6  # Length of each coefficient array
i_min = -100  # Minima of Range
i_max = 100  # Maxima of Range
generations = 1000  # Number of Generations


def objective(x):
    return 25 * x ** 5 + 18 * x ** 4 + 31 * x ** 2 - 14 * x ** 2 + 7 * x -
19


# Objective class is used to produce a range of y points on the given
polynomial between -100 and 99 by steps of 10
yv = []
xv = []
xy = []
for x in range(-100, 100, 10):
    y = objective(x)
    yv.append(y)
    xv.append(x)


#  Coefficient array
def polyindividual(length, min, max):
    return [randint(min, max) for x in range(length)]


# Individual array
def polystring(length, min, max):
    return [polyindividual(length, min, max) for x in range(6)]


# Population Array
def population(count, length, min, max):
    return [polystring(length, min, max) for x in range(count)]


def fitness(ps):
    """Determine the fitness of an individual. Lower is better."""
    Sum = 0
    b = []
    for x in range(-100, 100, 10):
        a = sum(ps[0]) * x ** 5 + sum(ps[1]) * x ** 4 + sum(ps[2]) * x ** 2
- sum(ps[3]) * x ** 2\
            + sum(ps[4]) * x + sum(ps[5])
        b.append(a)
    np.array(b)
    f = np.subtract(yv,b)
    return abs(np.average(f))
```

```
def grade(pop, fitness_history):
    """Find average fitness for a population."""
    summed = 0
    flag = 0
    for x in pop:
        if len(fitness_history) > 10:
            if fitness_history[-1] == (sum(fitness_history[-
int(generations/10):])/int(generations)):
                flag = 1 # Flag raised when 100 iterations of fitness are
unchanged
                summed = np.NaN
                x = sum(x[0]), sum(x[1]), sum(x[2]), sum(x[3]), sum(x[4]),
sum(x[5])
                print(x) # Prints found result
                break
            elif fitness(x) == 0:
                flag = 1  # Flag raised when an individual is found with a
fitness of 0
                summed = np.NaN
                x = sum(x[0]), sum(x[1]), sum(x[2]), sum(x[3]), sum(x[4]),
sum(x[5])
                print(x)
                break
        else:
            summed = summed + fitness(x)
            error = summed / (len(pop) * 1.0)
    return error, flag


def evolve(pop, i_min, i_max, retain=0.4, random_select=0.05, mutate=0.01):
    graded = [fitness(x) for x in pop]  # produces list of individuals,
paired with fitness of whole population
    grades_sorted = np.argsort(graded)
    pop = np.array(pop)
    graded = pop[grades_sorted]  # orders list of individuals in order of
fittest to least fit
    retain_length = int(len(graded) * retain)  # determines number of
parents kept from 'retain' & population size
    graded = np.ndarray.tolist(graded)
    parents = graded[:retain_length]  # keeps fittest 'retain_length' of
graded population
    # randomly add other individuals to promote genetic diversity
    for individual in graded[retain_length:]:
        if random_select > random():
            parents.append(individual)
    for polystring in parents:
        if mutate > random():
            pos_to_mutate = randint(0, len(polystring) - 1)
            h = polystring[pos_to_mutate]
            h[pos_to_mutate] = randint(i_min, i_max)
            # crossover parents 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]
            half = len(polystring) // 2
```

```
            # Crossover crosses over every individual coefficient array at
the halfway cutting point
            child = [male[0][:half] + female[0][half:],male[1][:half] +
female[1][half:],
                     male[2][:half] + female[2][half:],male[3][:half] +
female[3][half:],
                     male[4][:half] + female[4][half:],male[5][:half] +
female[5][half:]]
            children.append(child)
    parents.extend(children)
    return parents


def nested_sum(L):
    total = 0
    for i in L:
        if isinstance(i, list):  # checks if `i` is a list
            total += nested_sum(i)
        else:
            total += i
    return total


# Example usage
BigF = []
p = population(p_count, i_length, i_min, i_max)
fitness_history, flag = grade(p, BigF)
fitness_history = [fitness_history, ]
count = 0
for i in tqdm(range(generations)):
    p = evolve(p, i_min, i_max)
    average_error, flag = grade(p, BigF)
    fitness_history.append(average_error)
    count += 1
    if flag == 1:
        break
    else:
        pass
graded = [fitness(x) for x in p]
grades_sorted = np.argsort(graded)
pop = np.array(p)
graded = pop[grades_sorted]
p = np.ndarray.tolist(graded[:1][0])
final_result = sum(p[0]), sum(p[1]), sum(p[2]), sum(p[3]), sum(p[4]),
sum(p[5])
print(final_result)
print('Number of Generations Used: ', count)
print('Final Fitness: ', fitness_history[len(fitness_history)-1])
yna = []
# Producing the y values of the polynomial with the newly generated
coefficients
for x in range(-100, 100, 10):
    yn = final_result[0] * x ** 5 + final_result[1] * x ** 4 +
final_result[2] * x ** 2 + final_result[3] * x ** 2 +\
         final_result[4] * x - final_result[5]
    yna.append(yn)

x = pd.Series(yv)
y = pd.Series(yna)
correlation = x.corr(y)
correlation2 = y.corr(x)
```

```
print('Correlation: ', correlation)

# Plotting of Results
plt.subplot(121)
yv = np.array(yv)
yna = np.array(yna)
plt.plot(xv, yv, label="Target Curve")
plt.plot(xv, yna, '--', 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()
```