Faculty of Engineering & Design

UNIVERSITY OF BATH

Coursework B: Genetic Algorithms

Candidate Number: 13359

25/11/2022

EXERCISE 1

The initial program provided used a population of 100 individuals, each with 6 elements between 0 and 100, and applied a genetic algorithm to change the population average (sum of elements in each individual divided by the number of individuals) towards the target value. This exercise aimed to optimise this program to reach the target in fewer generations with a lower failure rate. As can be seen in Figure 1, the number of generations taken to reach the final solution varies greatly, and steps toward the solution are unpredictable due to the random nature of the mutation and genetic diversity functions.

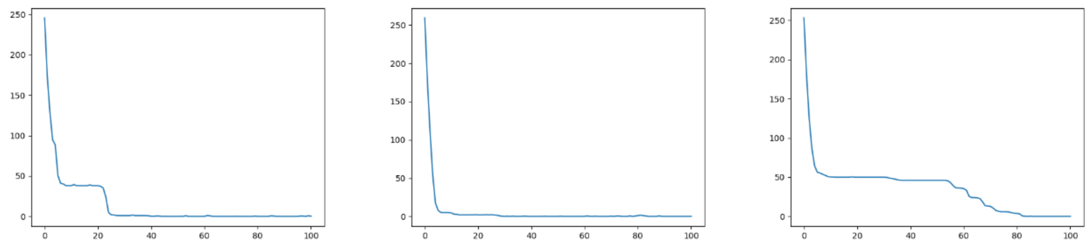


Figure 1 - Population average error against generations

Major Variables

Target – The objective of the function, in exercise 1 this is the number the average of the population is required to converge on.

Population – The number of individuals (initial: 100).

Elements – The number of separate integers within each individual (initial: 6). Each element has a minimum and maximum value (initial: min = 0, max = 100).

Generations – A hard limit on the number of iterations to reach convergence (initial: 100).

Major Functions

Fitness – Assess the error of each individual relative to the target.

Grade – Finds the average fitness of the population

Evolve – First collects the fittest X% of the population (X determined by retain length) and adds a selection of those individuals not chosen to promote genetic diversity. These individuals then become parents and have a chance to mutate. In initial, mutate randomly selects an element and randomises it between the maximum and minimum element values in that individual. The rest of the population is then filled in by crossover to create children, by selecting distinct males and females and taking half from each as shown in Figure 2. This type of selection is similar to elitism, as for low mutation rates, the majority of parents remain unchanged in the next generation.

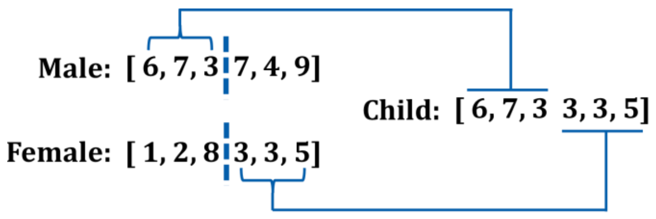


Figure 2 - Example crossover of parents to create a child

Optimisation

To optimise the function, each individual was reduced from 6 elements to 1, each with a starting value between 0 and 1000. The 'evolve' function was then modified to convert the selected 'parents' to 12-bit binary. A uniform crossover method was then applied, meaning that each bit of the child had a 50% chance of coming from the male and 50% from the female.

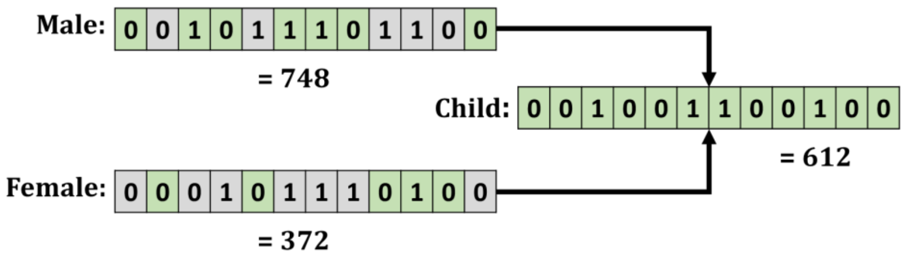

Male: 0 0 1 0 1 1 1 0 1 1 0 0
= 748
Child: 0 0 1 0 0 1 1 0 0 1 0 0
= 612
Female: 0 0 0 1 0 1 1 1 0 1 0 0
= 372

Figure 3 - 12-bit binary uniform crossover method

As the number of elements in each individual has been reduced to 1, the initial method of mutation is no longer valid. Instead, if an element is selected to be mutated, the single element is increased or decreased by up to 2% (element ± (element*0.02)).

To assess this algorithm against the initial, each was run 500 times, with the failure rate (number of times the hard limit of 100 generations was reached) and the average number of generations (successful runs only) taken to reach convergence being assessed. Convergence is defined as population average < 1% error from the target value.

Table 1 - Comparison of initial and modified genetic algorithms

	Dependent		Independent			
	Failure Rate	**Average Generations**	Target	Population of Individuals	Elements per Individual	Element min-max
Initial	8.8%	30.3	550	100	6	0-100
Modified	0.0%	4.56	550	100	1	0-1000

EXERCISE 2

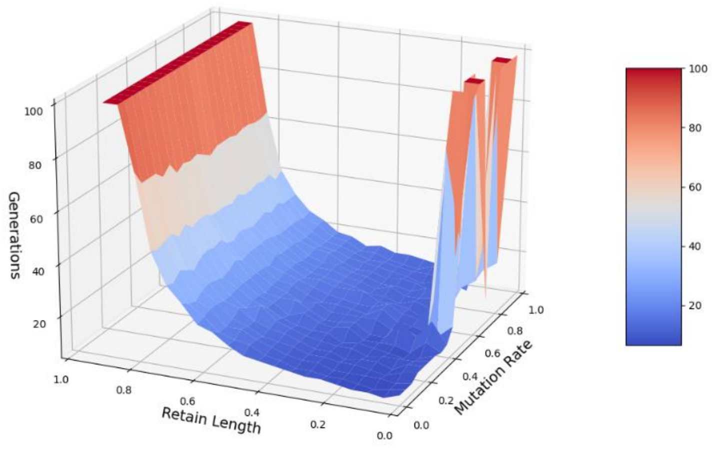
Initially, a parametric sweep of the number of parents retained from each generation and the mutation rate was performed, from 0 to 1 in steps of 0.05 for both parameters. This sweep was repeated 5 times, with the median value being recorded to eliminate anomalous results. The same sample population of 100 was used across the sweep to increase repeatability. The crossover probability [0-1] is defined as the proportion of the next generation created by the crossover of the parents (children).

Therefore, in this exercise, the crossover probability is given by:

Equation 1 - Crossover probability

$$Crossover\ probability\ =\ 1-(retain\ +\ randomselect)$$

Thus, by varying 'retain', crossover probability is varied. 'retain' [0-1] defines the proportion of parents (fittest of the current generation) retained in the next generation. 'randomselect' [0-1] is included to promote genetic diversity by randomly adding a percentage of the individuals not retained to the next generation. For this sweep, 'randomselect' was fixed at 0.05. 'Generations' shows the number of generations taken to reach convergence, where convergence is defined as population average < 1% error from the target value.



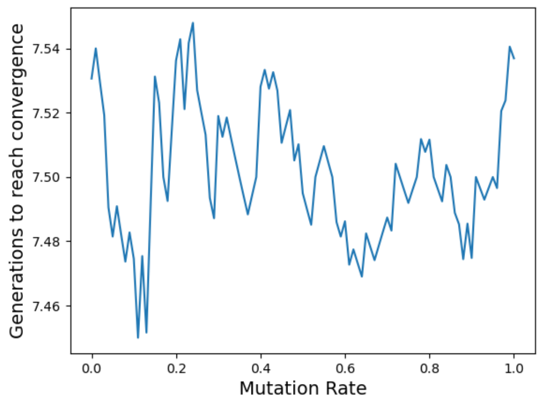
Mutation Rate	**(0, 1, 0.05)**
Retain Length	**(0, 1, 0.05)**
Random Select	0.05
Target	550
Population Count	100
Elements in Individual	6
Element Min-Max	0-100

Figure 4 - Parametric sweep of retain length and mutation rate

As can be seen in Figure 4, in general, lower retention and mutation rates allow the algorithm to converge faster. This is because for higher retention lengths, more 'unfit' parents are retained, and as such 'unfit' children also are produced at higher rates. However, lower retention rates are more computationally expensive, as more calculations are required to generate the 'children' required to fill the population. Therefore, in processes where time complexity is significant, a trade-off with error rate is considered.

At high retention lengths (>0.9), the user-defined hard limit of 100 generations is frequently reached. This is also true for higher mutation rates (>0.3) at low retention rates, demonstrating the need for the limits defined later in Exercise 3.

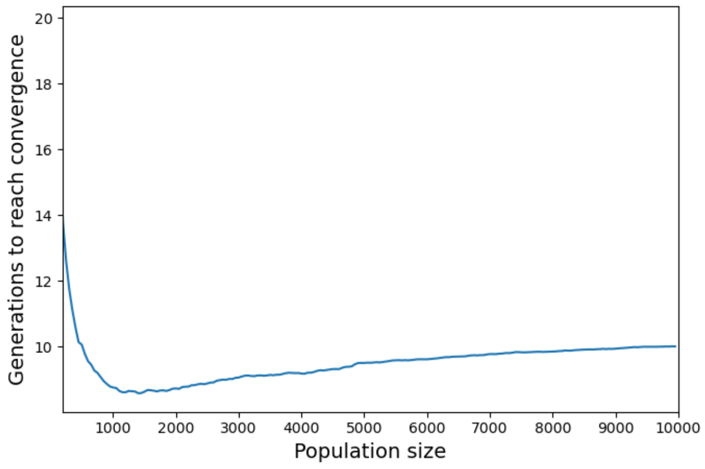
Due to the comparatively small variation caused by the mutation rate, a separate sweep was performed for mutation rate from 0 to 1 in steps of 0.01 to clarify the trend (Figure 5). In this sweep, all random values apart from mutation rate were seeded, the same sample population of 100 was used, and 50 repeats were used. Each As can be seen in Figure 5, for this application mutation rate has no discernible effect on the rate of convergence. There was also no discernible effect on the failure rate.



Mutation Rate	**(0, 1, 0.01)**
Retain Length	0.2
Random Select	0.05
Target	550
Population Count	100
Elements in Individual	6
Element Min-Max	0-100

Figure 5 - Generations vs Mutation Rate

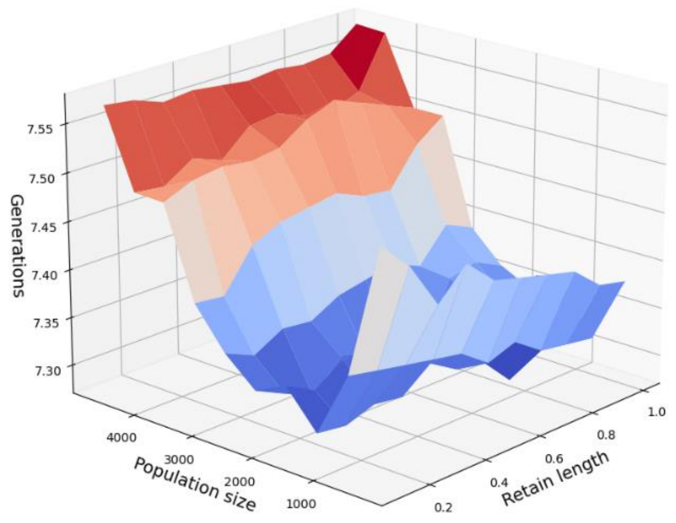
The effect of the population size on the rate of convergence was also plotted in Figure 6. This plot is an average of the generations taken over 50 attempts at each population count. It clearly shows the optimal population size for the minimum number of generations lies in the 1000 to 2000 range. However, the larger the population size the longer the computation time for each generation, which must be considered if your objective is to minimise runtime.



Mutation Rate	0.1
Retain Length	0.2
Random Select	0.05
Target	550
Population Count	**(50, 10000, 100)**
Elements in Individual	6
Element Min-Max	0-100

Figure 6 - Generations vs Population size

The interdependence of the population size and the retain length was also considered. This is because as the population size increases, the absolute amount retained by a given retain length also increases. As can be seen in Figure 7, at smaller population sizes (<1000), convergence is faster with higher retain lengths, as this is required to ensure suitable diversity within the parents. For larger population sizes (>2000), the trend generally appears to be the inverse.



Mutation Rate	0.1
Retain Length	**(0.1, 1, 0.1)**
Random Select	0.05
Target	550
Population Count	**(250, 4750, 250)**
Elements in Individual	6
Element Min-Max	0-100

Figure 7 - Interdependency of population size and retain length

Exercise 3

In order to prevent excessive computation time, three main stopping conditions are implemented:

Convergence – Once the program converges on the value the program can stop. This can be applied for either finding a single individual in the population that matches the target or for the average of the whole population. The definition of convergence can also be changed to reaching a value below a certain error threshold in the case of a small steady-state error.

Stagnation (Steady state error) – In the case that a number of consecutive generations X are produced without increasing significantly in fitness, the program can be stopped as it is likely o have reached a steady state error. A potential way to eliminate this steady-state error could be to apply integral principles from PID. By storing the changes caused by mutation at each generation, the magnitude of the last conditions large enough to overcome the stagnation condition could be re-applied to the current generation.

Hard limit of generations – By setting an upper limit of generations allowed to reach convergence, excess computation time can be prevented. In this case the attempt can be classified as 'failed'.

Exercise 4

Selection Methods

The code implementation for each of these methods can be found in appendix B.

Roulette wheel selection – Parents are selected at random with weights proportional to their relative fitness. Each parent (x) has a probability of being selected $p_s(x)$ equal to its fitness (f(x)) over the sum of all individual's fitness $\sum_{i=0}^{n} f(x_i)$ (Equation 2). n is the total number of individuals in the population.

Equation 2 - Roulette wheel selection

$$p_s(x) = \frac{f(x)}{\sum_{i=0}^{n} f(x_i)}$$

Ranked selection – Parents are selected at random with weights proportional to their ranking, ranked in order such that the worst fitness will be ranked the lowest. This can lead to slower convergence as the best individuals are less heavily weighted compared to roulette wheel selection but is less likely to fail by stepping over the optimal solution. When sorted best to worst, each parent (x) has a probability of being selected $p_s(x)$ equal to the sum of all indexes $\sum_{i=0}^{n} i$ minus its index i, all over the sum of all indexes.

Equation 3 - Ranked selection

$$p_s(x) = \frac{(\sum_{i=0}^{n} i) - i(x)}{\sum_{i=0}^{n} i}$$

Elitism selection – Parents are selected in order of their absolute fitness, with the top L parents being selected, where L is retain length multiplied by population size.

Method 1

The objective of exercise 4 is to find the coefficients for a given fifth-order polynomial:

Equation 4 - Target polynomial

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

Method 1 works by first identifying the target list of coefficients and applying a similar method to that used in exercise 1, using a crossover point to mate two parents to generate a child.

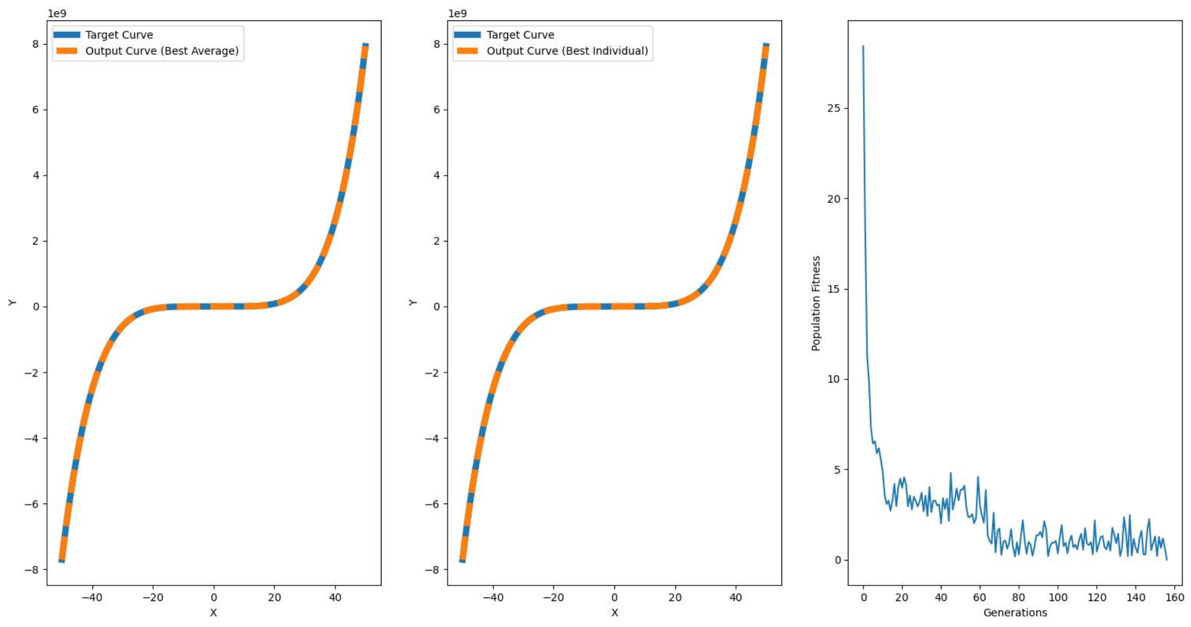


Figure 8 - Best population average curve (left), best individual curve (centre), population average vs generation (right)

From this method, there are two ways of producing a solution, the best individual and the best population average. For this method, both converged on the correct coefficients. For the example in Figure 8, the best individual: [25, 18, 31, -14, 7, -19] was produced in generation 73, with the best average: [25, 18, 31, -14, 7, -19] following later in generation 156. This trend was true over 50 runs, where the best individual was always produced before the best average.

By using a combination of parametric sweeps, the optimum parameters for method 1 (Table 2) were found.

Table 2 - Optimum Parameters for method 1

Mutation Rate	0.2
Retain Length	0.2
Random Select	0.05
Target	[25, 18, 31, -14, 7, -19]
Population Count	100
Elements in Individual	6
Element Min-Max	-50 to 50

Method 2

For method 2, instead of using the known coefficients as the target for the genetic algorithm, the curve is plotted and x,y coordinates are sampled from it. For each individual within the population, its fitness is assessed by putting the coefficients into the polynomial and evaluating the error in y for each value of x.

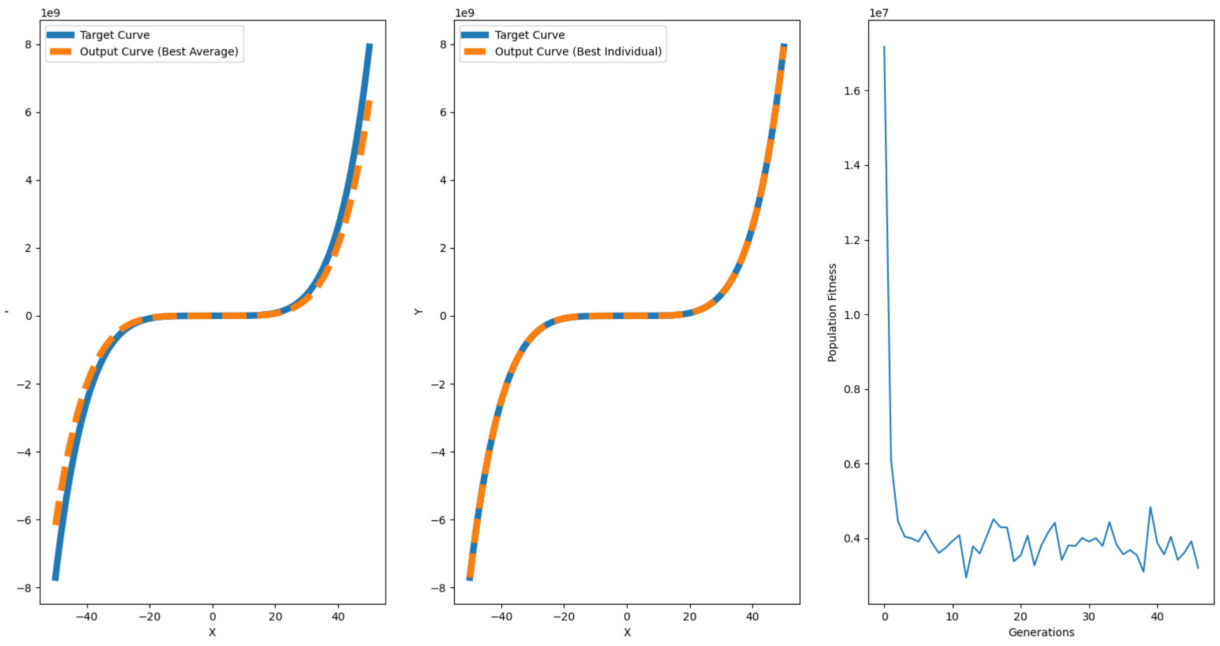


Figure 9 – Sample average curve (left), best individual curve (centre), population average vs generation (right)

As with the first method, there are two ways of generating the coefficients, taking the best individual, or the best average of the whole population. However, with this method, the average never converged on the correct solution, despite attempting a range of parameters and selection functions. The example coefficients for the curve shown in the left of Figure 9 are [20, 15, 24, -11, 5, -13].

The best individual: [25, 18, 31, -14, 7, -19] did perfectly converge on the solution after 46 generations.

By using a combination of parametric sweeps, the optimum parameters for method 2 (Table 3) were found.

Table 3 -Optimum Parameters for method 2

Mutation Rate	0.01
Retain Length	0.2
Random Select	0.05
Target	[25, 18, 31, -14, 7, -19]
Population Count	500
Elements in Individual	6
Element Min-Max	-50 to 50

Evaluation of methods

Table 4 shows the number of generations before the population average and single individual converged on the correct coefficients These values are averages of 50 runs for their respective with the optimised settings stated in Table 2 and Table 3. '-' represents when perfect convergence could not be achieved.

Table 4 - Generations for convergence for different methods

	Elitism		Roulette		Ranked	
	Average	**Single**	**Average**	**Single**	**Average**	**Single**
Method 1	461	331	787	699	1256	865
Method 2	-	39	-	35	-	-

As can be seen in Table 4, for single individual convergence using both elitism and roulette, method 2 requires significantly fewer generations for convergence. However, the runtime for method 2 was 7.9 seconds, compared to 3.8 seconds for method 1. This is likely partially due to the difference in population size.

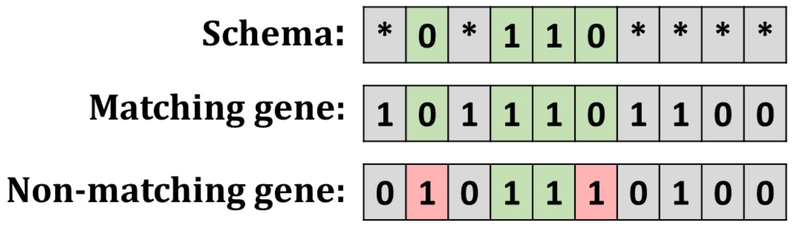
For method 1, a clear trend can be seen in which elitism is the fastest to converge, then roulette, then ranked. As the solution is fixed and well defined, once a perfect individual is found, it is optimal to retain it. Elitism guarantees it is retained, roulette gives it a high probability of being retained, and ranked has the lowest chance of retaining a perfect individual of the three. However, this is at the cost of genetic diversity, so for problems with less well defined or non-fixed solutions, this trend will likely not be present.

Method 2 is likely applicable to a wider range of problems, as it does not rely on knowing the equation of the curve in order to determine its fitness. Hence the genetic algorithm can be applied generally to find the equation of any fifth order polynomial and can easily be modified for an n-order polynomial.

Exercise 5

Schema Theorem

In order to apply the Schema theorem, the coefficients must be converted to their binary representations, and roulette wheel selection is required. A schema is defined as a gene pattern that exists within a chromosome [1]. In this case, after converting a coefficient to its binary representation, each can be compared to a schema (schema H = *0*110**** in Figure 10). A gene is said to 'match' if the specified bits are identical in the gene and schema.



Schema:	*	0	*	1	1	0	*	*	*	*
Matching gene:	1	0	1	1	1	0	1	1	0	0
Non-matching gene:	0	1	0	1	1	1	0	1	0	0

Figure 10 - Schema example

In Figure 10, the matching gene 1011101100 has a length l = 10. It therefore belongs to 2^l (2^{10} = 1024) schemas. For example, the schema in Figure 10, ********00 and 1**11**1100 etc [2].

Schema order o(H) is defined as the number of specified bits (non * bits) in a given schema. For the example in Figure 10, schema order o(H) = 4.

Schema defining length δ(H) is defined as the distance between the first and last specified bit. For the example in Figure 10, schema defining length δ(H) = 4 (Starts at bit, 9 ends at bit 5, 9-5=4).

Schema count m(H, k) is defined as the number of instances of H in the kth generation. For a given individual x, if it belongs to schema H, x is an instance of H(x ∈H).

Schema fitness f(H, k) is defined as the average fitness of H in the kth generation. f(x) is the fitness of the individual x.

Equation 5 - Schema fitness

$$f(H,k) = \frac{\sum_{x \in H} f(x)}{m(H,k)}$$

Genetic Algorithms & Schemas

The selection probability for an individual x is given above in Equation 2 - Roulette wheel selection. The expected number of instances (schema count m(H,k)) selected as parents (M(H,k))can be calculated using Equation 6 ($\bar{f}$ = average fitness of population):

Equation 6 - Expected instances of schema in parent population

$$M(H,k) = \frac{\sum_{x \in H} f(x)}{\bar{f}} = m(H,k)\frac{f(H,k)}{\bar{f}}$$

For single-point crossover, if the crossover point occurs within the schema defining length (p_{cd}), it is destroyed unless the other parent repairs the destroyed portion [2]. The probability that the crossover point occurs within the schema defining length for a string of length l and schema H is given by Equation 7:

Equation 7 - Probability crossover occurs within defining length

$$p_{cd} = \frac{\delta(H)}{l-1}$$

Therefore, the upper bound for the probability that schema H is destroyed is given by Equation 8, where p_c is the crossover probability. The lower bound for the probability $S_c(H)$ that schema H surviving is just the probability the schema is not destroyed (Equation 9).

Equation 8 - Probability of schema being destroyed in crossover

$$D_c(H) \leq p_c \frac{\delta(H)}{l-1}$$

Equation 9 - Probability of schema surviving crossover

$$S_c(H) = 1 - D_c(H) \geq 1 - p_c \frac{\delta(H)}{l-1}$$

From this, it can be determined that schemas with a lower order are more likely to survive.

Given a mutation operation that applies the mutation gene by gene, the probability a given gene not being changed is 1-p_m where p_m is the probability of mutation for a gene. In order for the schema to survive the mutation, the specified genes (bits) must not be mutated. The probability the schema survives mutation is given by:

Equation 10 - Probability of schema surviving mutation

$$S_m(H) = (1 - p)^{o(H)}$$

From this we can see the survival probability for mutation follows the trend identified for crossover; Schemas with a lower order are more likely to survive.

Finally, the expected number of schema instances in the next generation can be calculated:

Equation 11 - Expected number of schema instances in the next generation

$$E[m(H, k + 1)] \geq m(H, k) \frac{f(H, k)}{\bar{f}} \left(1 - P_c \frac{\delta(H)}{L - 1}\right) (1 - P_m)^{o(H)}$$

To verify Holland's Schema theorem, a binary encoding method was implemented to the genetic algorithm from exercise 4. This was achieved by converting each element of the individual to a signed 8-bit binary representation. This necessitated a change in the fitness function in order to weight the higher bits more heavily, meaning a change to bit 2 had an impact 64 times greater than bit 7 on the fitness. These binary representations were then concatenated into a 48 bit representation of the entire individual. This was then used to develop 5 different schema of the following defining lengths and orders:

1- δ(H) = 2, o(H) = 3 (Low defining length, Low order)
2- δ(H) = 4, o(H) = 3 (Medium defining length, Low order)
3- δ(H) = 8, o(H) = 3 (High defining length, Low order)
4- δ(H) = 4, o(H) = 5 (Medium defining length, Medium order)
5- δ(H) = 6, o(H) = 7 (High defining length, High order)

Single point crossover also had to be implemented so that the probability of cutting the schema during the mating process could be calculated.

Table 5 - Schema theorem testing (Population 500)

$\boldsymbol{m(H,k)}$	$\boldsymbol{E[m(H,k+1)]}$	$\boldsymbol{m(H,k+1)}$	**δ(H)**	**o(H)**
417	78	426	2	3
274	50	287	4	3
208	37	193	8	3
86	4.0	89	4	5
4	0.045	1	6	7

As can be seen in Table 5, the schema theorem holds true on all counts for the results generated between two random generations of the binary encoded genetic algorithm. The actual count of instances in the next generation is greater than or equal to the calculated estimate. It also shows the increased survivability of schemas with lower order and lower defining length. However, the estimated values are lower than expected, suggesting an error in the fitness calculation.

REFERENCES

[1] D. Zhang, “EE40098_10_GAs Part 2”.

[2] Perdue University, “Lecture 3: Schema Theory,” [Online]. Available: https://engineering.purdue.edu/~sudhoff/ee630/Lecture03.pdf. [Accessed 22 November 2022].

[3] A. Hassanat, K. Almohammadi, E. Alkafaween, E. Abunawas, A. Hammouri and V. S. Prasath, “Choosing Mutation and Crossover Ratios for Genetic Algorithms—A Review with a New Dynamic Approach,” 10 December 2019. [Online]. Available: https://pdfs.semanticscholar.org/5a25/a4d30528160eef96adbce1d7b03507ebd3d7.pdf. [Accessed 21 November 2022].

APPENDIX

A – Exercise 1 modified code

```
from random import randint, random
from functools import reduce
from operator import add
import matplotlib.pyplot as plt

def individual(length, min, max):

    'Create a member of the population.'
    return[randint(min, max) for x in range(length)]

def population(count, length, min,max):
    """
    Create a number of individuals (i.e. a population).

    count: the number of individuals in the population
    length: the number of values per individual
    min: the minimum possible value in an individual's list of values
    max: the maximum possible value in an individual's list of values
    """
    return[individual(length,min,max) for x in range(count)]

def fitness(individual, target):
    """
    Determine the fitness of an individual. Higher is better.

    individual: the individual to evaluate
    target: the target number individuals are aiming for
    """
    sum = reduce(add, individual, 0)
    return abs(target - sum)

def grade(pop,target):
    'Find average fitness for a population.'
    summed = reduce(add,(fitness(x,target) for x in pop))
    return summed/(len(pop) * 1.0)

def evolve(pop,target,retain=0.2,randomselect=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]

    #randomly add other individuals to promote genetic diversity
    for individual in graded[retain_length:]:
        if randomselect>random():
            parents.append(individual)

    #mutate some individuals
    for individual in parents:
        ########
        if mutate > random():
            position_to_mutate = randint(0, len(individual) - 1)
            #this mutation is not ideal, because it
            #restricts the range of possible values,
            #but the function is unaware of the min/max
            #values used to create the individuals,
            individual[position_to_mutate] = randint(int(individual[0]-
individual[0]/50), int(individual[0]+individual[0]/50))
        ########

    #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]
            #########
            #########
            male = format(male[0], '012b')
            male = [int(q) for q in str(male)]
            female = format(female[0], '012b')
            female = [int(q) for q in str(female)]
            child_bin = []
            child = 0

            for q in range(len(male)):
                if randint(0, 1) == 0:
                    child_bin.append(male[q])
                else:
                    child_bin.append(female[q])

            for q in child_bin:
                child = child*2 + int(q)

            children.append([child])

            #########
            #########
    parents.extend(children)

    return parents


#Example usage
target = 550
pcount = 100
i_length = 1
i_min = 0
i_max = 1000
generations = 100
gen_count = []
fail_count = 0
total_attempts = 0
for z in range(500):
    p = population(pcount, i_length, i_min, i_max)
    fitness_history = [grade(p,target), ]
    for i in range(generations):
        if grade(p,target) > target/100:
            p = evolve(p,target)
            fitness_history.append(grade(p,target))
        else:
            break
    if i < generations-1:
        gen_count.append(i)
    else:
        fail_count += 1
    total_attempts += 1
for datum in fitness_history:
    print(datum)

print(fail_count)
print(total_attempts)
print('Fail % = ', (fail_count/total_attempts)*100)
print('Average generations to find converge (Successful runs only):
',sum(gen_count)/len(gen_count))
```

B – Selection Functions

```
from numpy.random import choice

def roulette(retain_length, graded):
    weights = []
    candidates = []
    temp = []
    fitness_sum = 0
    graded = sorted(graded)

    for i in range(len(graded)):
        if graded[i][0] == 0:
            temp.append(0.0000000000000001)
        else:
            temp.append(graded[i][0])

        fitness_sum += 1/temp[i]

    for i in range(len(graded)):
        weights.append((1/temp[i])/fitness_sum)
        candidates.append(i)

    parent_indexes = choice(candidates, int(retain_length), p = weights)

    parents = []
    for i in range(len(parent_indexes)):
        parents.append(graded[parent_indexes[i]][1])

    return parents

def ranked(retain_length, graded):
    weights = []
    candidates = []
    fitness_sum = sum(range(1, len(graded)+1))
    graded = sorted(graded)
    for i in range(len(graded)):
        weights.append((len(graded)-i)/fitness_sum)
        candidates.append(i)

    parent_indexes = choice(candidates, int(retain_length), p = weights)

    parents = []
    for i in range(len(parent_indexes)):
        parents.append(graded[parent_indexes[i]][1])

    return parents


def elitism(retain_length, graded):
    parents = []
    for i in range(retain_length):
        parents.append(graded[i][1])

    return parents
```

C – Exercise 4

Ex4_1.py:

```
from random import randint, random
from functools import reduce
from operator import add
import numpy as np
import matplotlib.pyplot as plt
import GA
import time

start = time.time()
# Initial Variables
target = [25, 18, 31, -14, 7, -19]
pcount = 100            # Population size
i_length = 6
i_min = -50
i_max = 50
generations = 5000

retain = 0.2            # Percentage of parents to retain
random_select = 0.05    # Crossover probability
mutate = 0.2           # Mutation probability

max_error = 0      # Fitness level to stop at


p = GA.population(pcount, i_length, i_min, i_max)
```

```
fitness_history = [GA.grade(p, target)]
gen = 0
success = False
while 1:

    p = GA.evolve(p, target, retain, random_select, mutate, i_min, i_max)
    gen += 1
    average_fitness = GA.grade(p, target)
    print(average_fitness)
    fitness_history.append(average_fitness)
    graded = [(GA.fitness(x, target), x) for x in p]
    graded = [x[1] for x in sorted(graded)]
    best_ind = graded[0]
    best_fitness = GA.fitness(best_ind, target)
    if best_fitness <= 0 and success == False:
        first_ind_success = gen
        success = True

    if average_fitness <= max_error:
        break
    if gen >= generations:
        break



#print(p)
coeff = []
temp = []
for i in range(6):
    for q in range(len(p)):
        temp.append(0)
    coeff.append(temp)
    temp = []

for i in range(len(p)):
    for q in range(6):
        coeff[q][i] = p[i][q]

avg_individual = []
for i in range(6):
    avg_individual.append(sum(coeff[i])//len(p))
print('Best_Average: ', avg_individual, 'Generation', gen)


print('Best individual:', best_ind, 'Generation', first_ind_success )

# Create polynomial target curve
y_avg = []
y_best_ind = []
```

```
x_plot = []
print('Elapsed time: ', time.time()-start)
# Plot GA average curve
for x in np.arange(-50, 50, 0.01):
    ya = avg_individual[0]*x**5 + avg_individual[1]*x**4 +
avg_individual[2]*x**3 + avg_individual[3]*x**2 + avg_individual[4]*x +
avg_individual[5]
    x_plot.append(x)
    y_avg.append(ya)

# Plot best individual curve
for x in np.arange(-50, 50, 0.01):
    yb = best_ind[0]*x**5 + best_ind[1]*x**4 + best_ind[2]*x**3 +
best_ind[3]*x**2 + best_ind[4]*x + best_ind[5]
    y_best_ind.append(yb)

#Generate target curve
y_target = []
for x in np.arange(-50, 50, 0.01):
    y = 25*x**5 + 18*x**4 + 31*x**3 + 14*x**2 + 7*x + 19
    y_target.append(y)

plt.subplot(1, 3, 1)
plt.plot(x_plot, y_target, linewidth = 6, label = "Target Curve")
plt.plot(x_plot, y_avg, '--', linewidth = 6,  label = "Output Curve (Best
Average)")
plt.xlabel("X")
plt.ylabel("Y")
plt.legend()

plt.subplot(1, 3, 2)
plt.plot(x_plot, y_target, linewidth = 6,  label = "Target Curve")
plt.plot(x_plot, y_best_ind, '--', linewidth = 6,  label = "Output Curve (Best
Individual)")
plt.xlabel("X")
plt.ylabel("Y")
plt.legend()

plt.subplot(1, 3, 3)
plt.plot(fitness_history)
plt.xlabel("Generations")
plt.ylabel("Population Fitness")
plt.show()
```

GA.py:

```
from random import randint, random
from functools import reduce
```

```python
from operator import add
import numpy as np
import matplotlib.pyplot as plt
import Selection as select
def individual(length, min, max):

    'Create a member of the population.'
    return[randint(min, max) for x in range(length)]

def population(count, length, min, max):
    """
    Create a number of individuals (i.e. a population).

    count: the number of individuals in the population
    length: the number of values per individual
    min: the minimum possible value in an individual's list of values
    max: the maximum possible value in an individual's list of values
    """
    return[individual(length, min, max) for x in range(count)]

def fitness(individual, target):
    """
    Determine the fitness of an individual. Lower is better.

    individual: the individual to evaluate
    target: the target number individuals are aiming for
    """
    target = np.array(target)
    individual = np.array(individual)

    individual_fitness = target - individual
    return np.average(abs(individual_fitness))

def grade(pop, target):
    'Find average fitness for a population.'
    all_grades = [fitness(x, target) for x in pop]
    summed = reduce(add, all_grades)
    return summed / (len(pop) * 1.0)

def evolve(pop, target, retain, random_select, mutate, i_min, i_max):
    graded = [(fitness(x, target), x) for x in pop]
    #graded = [x[1] for x in sorted(graded)]
    retain_length = int(len(graded) * retain)

    graded = sorted(graded)
    #parents = select.roulette(retain_length, graded)
    #parents = select.ranked(retain_length, graded)
    parents = select.elitism(retain_length, graded)
    # parents = graded[:retain_length]

    # # Randomly add other individuals to promote genetic diversity
    # for individual in graded[retain_length:]:
    #     if random_select > random():
    #         parents.append(individual)

    # Mutate some individuals
    for individual in parents:
        if mutate > random():
            position_to_mutate = randint(0, len(individual) - 1)
            individual[position_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 = int(len(male) / 2)
            child = male[:half] + female[half:]
            children.append(child)
    parents.extend(children)
    return parents
```

Ex4_2:

```
from random import randint, random
from functools import reduce
from operator import add
import numpy as np
import matplotlib.pyplot as plt
import GA2 as GA
import time

start = time.time()
y_target = []
X = np.arange(-20, 20, 1)
for x in X:
    y = 25*x**5 + 18*x**4 + 31*x**3 - 14*x**2 + 7*x - 19
    y_target.append(y)

# Initial Variables
```

```
#target = [25, 18, 31, -14, 7, -19]
pcount = 500            # Population size
i_length = 6
i_min = -50
i_max = 50
generations = 5000

retain = 0.2            # Percentage of parents to retain
random_select = 0.05    # Crossover probability
mutate = 0.01          # Mutation probability

max_error = 0      # Fitness level to stop at


p = GA.population(pcount, i_length, i_min, i_max)
fitness_history = [GA.grade(p, y_target, X)]

for i in range(generations):
    p = GA.evolve(p, y_target, X, retain, random_select, mutate, i_min, i_max)
    average_fitness = GA.grade(p, y_target, X)
    print('          ', average_fitness)
    fitness_history.append(average_fitness)

    graded = [[GA.fitness(i, y_target, X), i] for i in p]
    graded = sorted(graded)
    graded = [x[1] for x in graded]

    top_fitness = GA.fitness(graded[0], y_target, X)
    print(top_fitness)


    if top_fitness <= max_error:
        print(i+1)
        break

coeff = []
temp = []
for i in range(6):
    for q in range(len(p)):
        temp.append(0)
    coeff.append(temp)
    temp = []

for i in range(len(p)):
    for q in range(6):
        coeff[q][i] = p[i][q]

avg_individual = []
for i in range(6):
```

```python
    avg_individual.append(sum(coeff[i])//len(p))
print('Best_Average: ', avg_individual)

graded = [x[1] for x in sorted([(GA.fitness(x, y_target, X), x) for x in p])]
best_ind = graded[0]
print('Best individual:', best_ind)

# Create polynomial target curve
y_avg = []
y_best_ind = []
x_plot = []
print('Elapsed time: ', time.time()-start)
# Plot GA average curve
for x in np.arange(-50, 50, 0.01):
    ya = avg_individual[0]*x**5 + avg_individual[1]*x**4 +
avg_individual[2]*x**3 + avg_individual[3]*x**2 + avg_individual[4]*x +
avg_individual[5]
    x_plot.append(x)
    y_avg.append(ya)

# Plot best individual curve
for x in np.arange(-50, 50, 0.01):
    yb = best_ind[0]*x**5 + best_ind[1]*x**4 + best_ind[2]*x**3 +
best_ind[3]*x**2 + best_ind[4]*x + best_ind[5]
    y_best_ind.append(yb)

# Plot target curve
y_target = []
for x in np.arange(-50, 50, 0.01):
    y = 25*x**5 + 18*x**4 + 31*x**3 + 14*x**2 + 7*x + 19
    y_target.append(y)

plt.subplot(1, 3, 1)
plt.plot(x_plot, y_target, linewidth = 6, label = "Target Curve")
plt.plot(x_plot, y_avg, '--', linewidth = 6,  label = "Output Curve (Best
Average)")
plt.xlabel("X")
plt.ylabel("Y")
plt.legend()

plt.subplot(1, 3, 2)
plt.plot(x_plot, y_target, linewidth = 6,  label = "Target Curve")
plt.plot(x_plot, y_best_ind, '--', linewidth = 6,  label = "Output Curve (Best
Individual)")
plt.xlabel("X")
plt.ylabel("Y")
plt.legend()
plt.subplot(1, 3, 3)
plt.plot(fitness_history)
plt.xlabel("Generations")
plt.ylabel("Population Fitness")
plt.show()
```

GA2:

```
from random import randint, random
from functools import reduce
from operator import add
import numpy as np
import matplotlib.pyplot as plt
import Selection as select
def individual(length, min, max):

    'Create a member of the population.'
    return[randint(min, max) for x in range(length)]

def population(count, length, min, max):
    """
    Create a number of individuals (i.e. a population).

    count: the number of individuals in the population
    length: the number of values per individual
    min: the minimum possible value in an individual's list of values
    max: the maximum possible value in an individual's list of values
    """
    return[individual(length, min, max) for x in range(count)]

def fitness(individual, y_target, X):
    """
    Determine the fitness of an individual. Lower is better.

    individual: the individual to evaluate
    y_target: the y_target number individuals are aiming for
    """
    y_guess = [0 for _ in range(len(X))]
    for i in range(len(X)):
        y_guess[i] = individual[0]*X[i]**5 + individual[1]*X[i]**4 +
individual[2]*X[i]**3 + individual[3]*X[i]**2 + individual[4]*X[i] +
individual[5]

    y_err = [0 for _ in range(len(X))]
    for i in range(len(X)):
        y_err[i] = abs(y_target[i] - y_guess[i])
    y_avg_err = np.average(y_err)


    return y_avg_err

def grade(pop, y_target, X):
    'Find average fitness for a population.'
    all_grades = [fitness(i, y_target, X) for i in pop]
    summed = reduce(add, all_grades)
    return summed / (len(pop) * 1.0)

def evolve(pop, y_target, X, retain, random_select, mutate, i_min, i_max):
    graded = [[fitness(i, y_target, X), i] for i in pop]
    # graded = sorted(graded)
    # graded = [x[1] for x in graded]
    retain_length = int(len(graded) * retain)

    graded = sorted(graded)
    #parents = select.roulette(retain_length, graded)
    #parents = select.ranked(retain_length, graded)
    parents = select.elitism(retain_length, graded)

    #parents = graded[:retain_length]

    # # Randomly add other individuals to promote genetic diversity
    # for individual in graded[retain_length:]:
    #     if random_select > random():
    #         parents.append(individual)



    # 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 = int(len(male) / 2)
            child = male[:half] + female[half:]
            children.append(child)

                # Mutate some individuals
        for individual in children:
            if mutate > random():
                position_to_mutate = randint(0, len(individual) - 1)
                individual[position_to_mutate] = randint(i_min, i_max)
    parents.extend(children)
    return parents
```

D – Exercise 5

Ex5:

```
from random import randint, random
from functools import reduce
from operator import add
import numpy as np
import matplotlib.pyplot as plt
import GA5 as GA
import time

start = time.time()
# Initial Variables
target = [25, 18, 31, -14, 7, -19]
binary_target = []
for i in target:
    binary_i = bin(abs(i))[2:].zfill(7)
    sign_i = i/abs(i)
    if sign_i == -1:
        binary_i = '1' + binary_i
    else:
        binary_i = '0' + binary_i
    binary_target += binary_i
for i in range(len(binary_target)):
    binary_target[i] = int(binary_target[i])

print(binary_target)


pcount = 500            # Population size
i_length = len(binary_target)
i_min = -50
i_max = 50
generations = 5000

retain = 0.2            # Percentage of parents to retain
random_select = 0.05    # Crossover probability
mutate = 0.5           # Mutation probability

max_error = 0      # Fitness level to stop at


p = GA.population(pcount, i_length, i_min, i_max)
fitness_history = [GA.grade(p, binary_target)]
gen = 0
```

```python
success = False
while 1:

    p = GA.evolve(p, binary_target, retain, random_select, mutate, i_min,
i_max)
    gen += 1
    average_fitness = GA.grade(p, binary_target)
    print('AVERAGE',average_fitness)
    fitness_history.append(average_fitness)
    graded = [(GA.fitness(x, binary_target), x) for x in p]
    graded = [x[1] for x in sorted(graded)]
    best_ind = graded[0]
    best_fitness = GA.fitness(best_ind, binary_target)
    print('BEST INDIVIDUAL', best_fitness)
    if best_fitness <= 0 and success == False:
        first_ind_success = gen
        success = True

    if average_fitness <= max_error:
        break
    if gen >= generations:
        break



#print(p)
coeff = []
temp = []
for i in range(6):
    for q in range(len(p)):
        temp.append(0)
    coeff.append(temp)
    temp = []

for i in range(len(p)):
    for q in range(6):
        coeff[q][i] = p[i][q]

avg_individual = []
for i in range(6):
    avg_individual.append(sum(coeff[i])//len(p))
print('Best_Average: ', avg_individual, 'Generation', gen)


print('Best individual:', best_ind, 'Generation', first_ind_success )

# Create polynomial target curve
y_avg = []
y_best_ind = []
```

```
x_plot = []
print('Elapsed time: ', time.time()-start)
# Plot GA average curve
for x in np.arange(-50, 50, 0.01):
    ya = avg_individual[0]*x**5 + avg_individual[1]*x**4 +
avg_individual[2]*x**3 + avg_individual[3]*x**2 + avg_individual[4]*x +
avg_individual[5]
    x_plot.append(x)
    y_avg.append(ya)

# Plot best individual curve
for x in np.arange(-50, 50, 0.01):
    yb = best_ind[0]*x**5 + best_ind[1]*x**4 + best_ind[2]*x**3 +
best_ind[3]*x**2 + best_ind[4]*x + best_ind[5]
    y_best_ind.append(yb)

#Generate target curve
y_target = []
for x in np.arange(-50, 50, 0.01):
    y = 25*x**5 + 18*x**4 + 31*x**3 + 14*x**2 + 7*x + 19
    y_target.append(y)

plt.subplot(1, 3, 1)
plt.plot(x_plot, y_target, linewidth = 6, label = "Target Curve")
plt.plot(x_plot, y_avg, '--', linewidth = 6,  label = "Output Curve (Best
Average)")
plt.xlabel("X")
plt.ylabel("Y")
plt.legend()

plt.subplot(1, 3, 2)
plt.plot(x_plot, y_target, linewidth = 6,  label = "Target Curve")
plt.plot(x_plot, y_best_ind, '--', linewidth = 6,  label = "Output Curve (Best
Individual)")
plt.xlabel("X")
plt.ylabel("Y")
plt.legend()

plt.subplot(1, 3, 3)
plt.plot(fitness_history)
plt.xlabel("Generations")
plt.ylabel("Population Fitness")
plt.show()
```

GA5:

```
from random import randint, random
from functools import reduce
```

```python
from operator import add
import numpy as np
import matplotlib.pyplot as plt
import Selection as select
def individual(length, min, max):

    'Create a member of the population.'
    return[randint(0, 1) for x in range(length)]

def population(count, length, min, max):
    """
    Create a number of individuals (i.e. a population).

    count: the number of individuals in the population
    length: the number of values per individual
    min: the minimum possible value in an individual's list of values
    max: the maximum possible value in an individual's list of values
    """
    return[individual(length, min, max) for x in range(count)]

def fitness(individual, target):
    fitness = 0
    weights = {0:128, 1:64, 2:32, 3:16, 4:8, 5:4, 6:2, 7:1}
    for i in range(len(target)):
        bit = i % 8
        if individual[i] != target[i]:
            fitness += weights[bit]
    return fitness

def grade(pop, target):
    'Find average fitness for a population.'
    all_grades = [fitness(x, target) for x in pop]
    summed = reduce(add, all_grades)
    return summed / (len(pop) * 1.0)

def evolve(pop, target, retain, random_select, mutate, i_min, i_max):
    graded = [(fitness(x, target), x) for x in pop]
    retain_length = int(len(graded) * retain)
    graded = sorted(graded)
    parents = select.roulette(retain_length, graded)

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

            if individual[position_to_mutate] == 1:
                individual[position_to_mutate] = 0
            else:
```

```
            individual[position_to_mutate] = 0

    # 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]
            crossover_point = randint(0, len(male)-1)
            child = male[:crossover_point] + female[crossover_point:]
            children.append(child)

    parents.extend(children)
        # Mutate some individuals

    LOLR = 0
    LOLR_FIT = []
    LOMR = 0
    LOMR_FIT = []
    LOHR = 0
    LOHR_FIT = []
    LRLO = 0
    LRLO_FIT = []
    LRMO = 0
    LRMO_FIT = []
    LRHO = 0
    LRHO_FIT = []
    for i in range(len(parents)):
        H = parents[i]

        #LOW ORDER LOW RANGE
        if H[0] == 0 and H[1] == 0 and H[2] == 0:
            LOLR += 1
            LOLR_FIT.append(fitness(individual, target))
        #LOW ORDER MEDIUM RANGE
        if H[0] == 0 and H[2] == 0 and H[4] == 1:
            LOMR += 1
            LOMR_FIT.append(fitness(individual, target))
        #LOW ORDER HIGH RANGE
        if H[0] == 0 and H[4] == 0 and H[8] == 0:
            LOHR += 1
            LOHR_FIT.append(fitness(individual, target))
        if H[0] == 0 and H[1] == 0 and H[2] == 0:
            LRLO += 1
            LRLO_FIT.append(fitness(individual, target))
        if H[0] == 0 and H[1] == 0 and H[2] == 0 and H[3] == 1 and H[4] == 1 :
            LRMO += 1
            LRMO_FIT.append(fitness(individual, target))
        if H[0] == 0 and H[1] == 0 and H[2] == 0 and H[3] == 1 and H[4] == 1
and H[5] == 0 and H[6] == 0:
            LRHO += 1
            LRHO_FIT.append(fitness(individual, target))
    print('LOLR :', LOLR)
    print('LOLR FITNESS: ',np.average(LOLR_FIT))
    print('LOMR :', LOMR)
    print('LOMR FITNESS: ',np.average(LOMR_FIT))
    print('LOHR :', LOHR)
    print('LOHR FITNESS: ',np.average(LOHR_FIT))
    print('LRLO :', LRLO)
    print('LRLO FITNESS: ',np.average(LRLO_FIT))
    print('LRMO :', LRMO)
    print('LRMO FITNESS: ',np.average(LRMO_FIT))
    print('LRHO :', LRHO)
    print('LRHO FITNESS: ',np.average(LRHO_FIT))

    return parents
# [0, 0, 0, 1, 1, 0, 0, 1, 0, 0, 0, 1, 0, 0, 1, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1,
0, 0, 0, 1, 1, 1, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 0, 0, 1, 0, 0, 1, 1]
#  0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15 16 17 18 19 20 21 22 23 24
25 26 27 28
```