EE40098 Computational Intelligence - Coursework B

Seb Hall samh25@bath.ac.uk, 21st November 2025
Department of Electronic & Electrical Engineering, University of Bath

1. Introduction

Genetic algorithms (GAs) are a type of iterative algorithm inspired by biological processes such as evolution [1]. They are used to find solutions to optimisation and search problems by implementing processes such as natural selection and genetic mutation to simulate a 'survival of the fittest' scenario over multiple generations.

2. Exercise 1

Implementation of a simple genetic algorithm to search for a target value.

A simple genetic algorithm was created in Python to search for a target number in the shortest number of iterations. This was achieved with an object-oriented approach that defined an 'Individual' class to represent a candidate solution, and a 'Population' class to manage individuals and evolution.

Three genetic processes were implemented [2]:

1. **Selection** - a proportion of the most fit individuals were selected to remain in the population.
2. **Mutation** - some individuals had their genes randomly modified to introduce genetic diversity.
3. **Crossover** - pairs of individuals were combined to produce offspring with a combination of genes.

An example plot showing the evolution of fitness over 10 generations with a population size of 10 is shown in Figure 1.

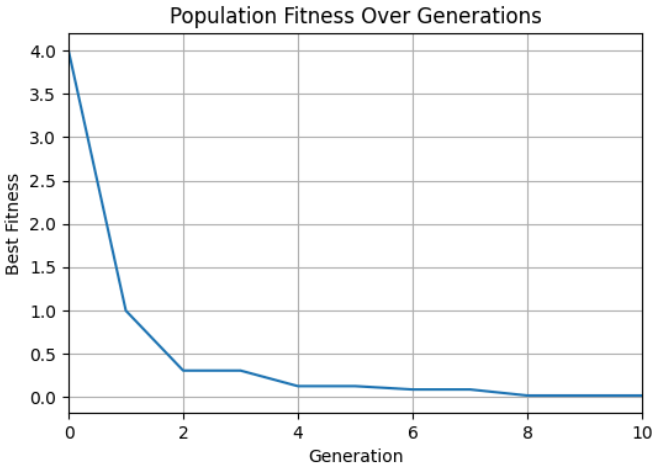

Figure 1: Example evolution over 10 generations with a population size of 10.

The source code for this exercise can be found in Section 9.1.

3. Exercise 2

Analysis of the genetic algorithm created in exercise 1.

The classes representing individuals and populations in exercise 1 were reused to perform a sensitivity analysis on several parameters of the genetic algorithm. These were:

1. **Population Size** - the number of individuals in the population.
2. **Mutation Proportion** - the proportion of surviving individuals that undergo mutation each generation.
3. **Mutation Limit** - the maximum amount by which an individual's gene can be mutated.
4. **Retain Proportion** - the proportion of the best individuals that are retained each generation.
5. **Crossover Variance** - the variance of blending genes from two parents when creating a child.

Each parameter was varied randomly over a range of values, with 10,000 samples taken for each. The modified Python scripts can be found in Section 9.2.

3.1. Population Size Analysis

Population size was the first parameter analysed. This varies the number of 'individuals' in the population, effectively changing the genetic diversity available to the algorithm. The results are shown in Figure 2.

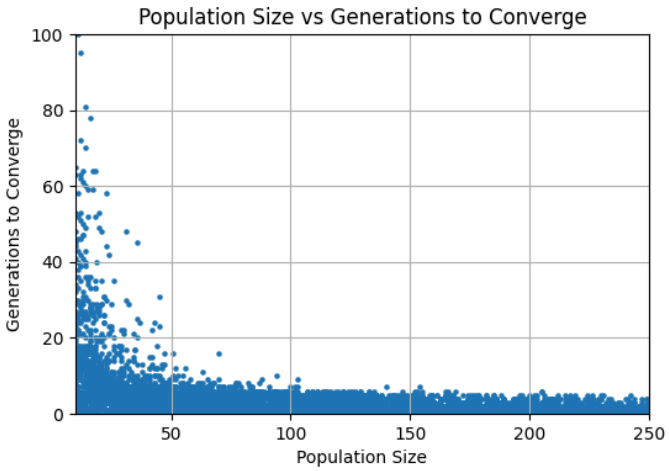

Figure 2: Performance comparison of different population sizes over 10,000 samples.

This shows a clear trend that larger populations lead to a faster convergence to the target value, as more genetic diversity allows the algorithm to explore a wider solution space.

3.2. Mutation Probability Analysis

The next parameter analysed was the mutation probability, corresponding to the proportion of individuals

that undergo mutation each generation. The results are shown in Figure 3.

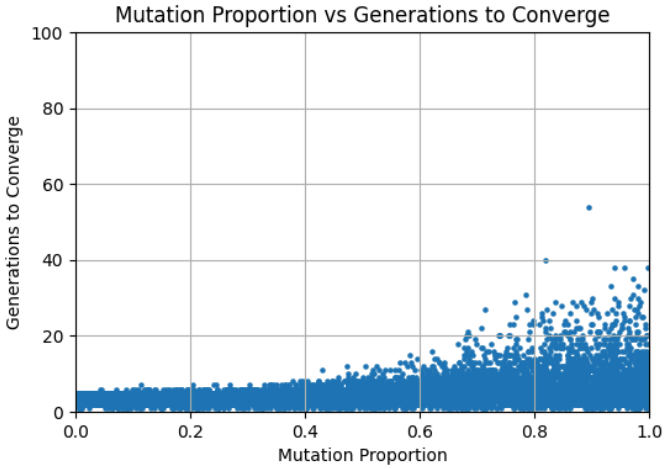

Figure 3: Performance comparison of different mutation proportions over 10,000 samples.

This shows a less clear trend, but suggests that high mutation rates hinder convergence, while low rates have little effect (in isolation).

3.3. Mutation Limit Analysis

The next parameter to be analysed was the mutation limit, referring to the range of values by which an individual's gene can be mutated. The results are shown in Figure 4.

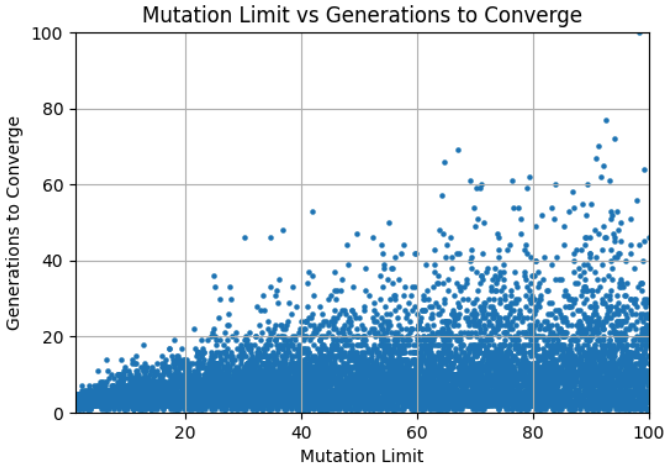

Figure 4: Performance comparison of different mutation limits over 10,000 samples.

The results suggest an inverse relationship, with lower mutation limits leading to faster convergence. This is likely because larger mutations move individuals further away from the optimal solution.

3.4. Retained Proportion Analysis

After analysing the effects of mutation, the next parameter analysed was the 'retain' proportion, which determines the proportion of the best individuals that are retained each generation. The results are shown in Figure 5.

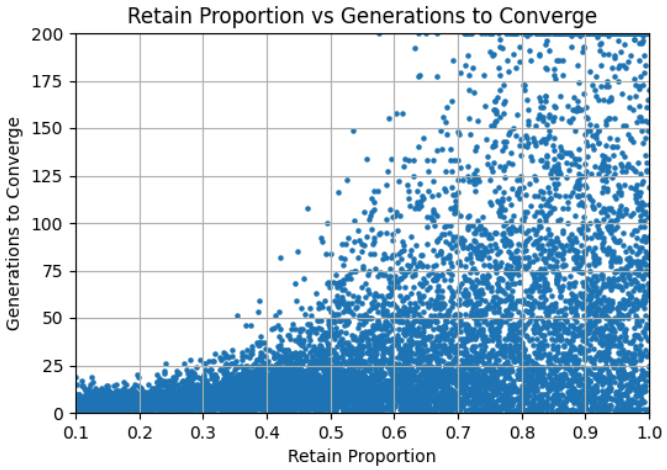

Figure 5: Performance comparison of different retained proportions over 10,000 samples.

These results show a clear trend that lower retained proportions lead to faster convergence, likely due to the rejection of suboptimal individuals bringing the population closer to the target.

3.5. Crossover Variance Analysis

The final parameter analysed was the crossover variance, referring to the variance of blending genes from two parents when creating a child. A lower variance ensures children have a close to 50:50 blend of their parents genes, while a higher variance could allow values closer to one parent. The results are shown in Figure 6.

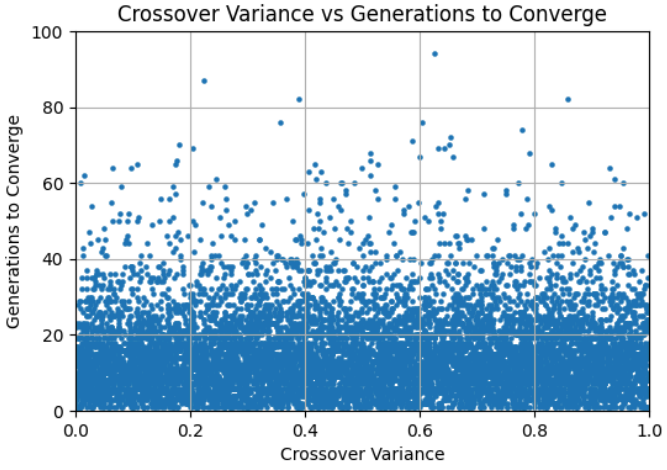

Figure 6: Performance comparison of different crossover variances over 10,000 samples.

This showed minimal effect on convergence, suggesting that gene blending plays a lesser role in the algorithm's performance.

4. Exercise 3

Implementation a stop condition for the algorithm created in exercise 1.

Stopping the algorithm early when a satisfactory solution is found can save computation time and resouces. This can be achieved quite simply by comparing the

smallest error in the population to a threshold at each generation. If the best error is below this threshold, the algorithm can terminate early as it has found an adequate solution. A study was performed to analyse the effect of different thresholds on convergence time, in the range of 0 to 0.5. The results are shown in Figure 7 and the source code can be found in Section 9.3.

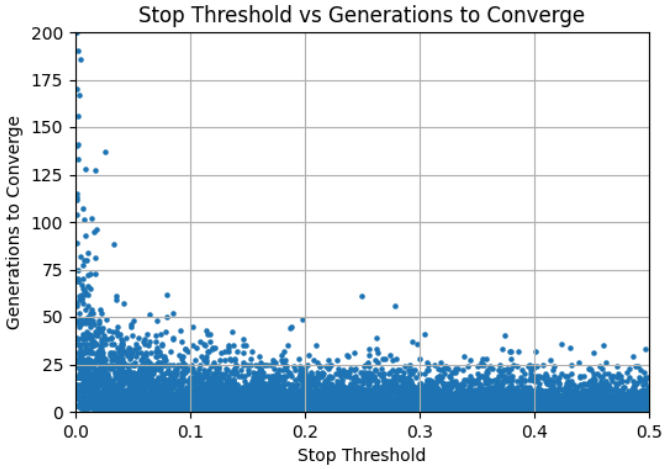

Figure 7: Performance comparison of different stop thresholds over 10,000 samples.

This plot shows a clear inverse relationship between stop threshold and convergence time. Lower thresholds require the algorithm to find a more accurate solution, taking longer to converge. Higher thresholds are more permissive, allowing the algorithm to terminate earlier.

5. Exercise 4

Using a genetic algorithm to optimise parameters for a 5th order polynomial.

A genetic algorithm was implemented to search for the co-efficients of a 5th order polynomial in the following form:

$$y = ax^5 + bx^4 + cx^3 + dx^2 + ex + f$$

In real-world applications, genetic algorithms can be used for curve-fitting tasks for empirical data, where the underlying relationship is unknown or complex.

For this exercise, the target polynomial was defined as:

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

The solution for this exercise was modelled around a more representative real-world application [3], and so the first step in solving the problem was to generate a dataset of sample points from the target polynomial.

5.1. Dataset Generation

Due to the polynomial being 5th order, any x values significantly larger than 1 result in extremely large y values, dominated by the $25x^5$ term. To avoid this, the dataset was sampled over the range −2 to 2, which would produce a range of y values that were more equally dependant on all co-efficients. A total of 1000 sample points were generated for the dataset.

5.2. Code Implementation

The classes for 'Individual' and 'Population' were modified to handle the shift from a single number search to a multi-variable polynomial co-efficient search.

The 'Individual' class was modified to have a set of genes corresponding to the co-efficients of the polynomial. The static property for target value was replaced with the target dataset, and the fitness evaluation method was modified to calculated the mean squared error (MSE) between the polynomial defined by the individual's genes and the target dataset. This is an industry standard metric for regression tasks [4]. The crossover logic and mutation logic were also updated to handle multiple genes.

The 'Population' class remained largely unchanged, with the exception of making mutation per-gene rather than per-individual. A helper method for managing the best individual was added.

The updated source code for this exercise can be found in Section 9.4.

5.3. Parameter Tuning

The genetic algorithm parameters were tuned to improve performance for this specific problem. The parameters studied were:

1. **Population Size**
2. **Retain Proportion**
3. **Mutation Proportion**
4. **Mutation Limit**

Each one of these was varied within a range, and the error after 10 generations was recorded as a measure of performance (10 being chosen for speed). An example plot of the search for population size is shown in Figure 8.

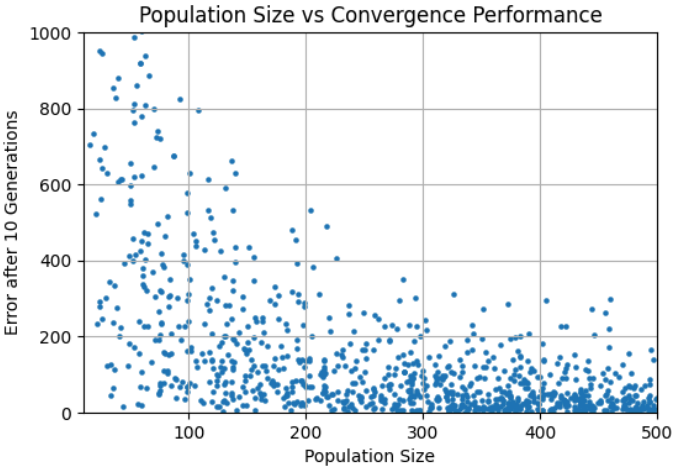

Figure 8: Performance comparison of different population sizes over 10,000 samples.

5.4. Final Results

After tuning the parameters, via direct analysis and strategic testing, the final configuration was found to be:

- Population Size: 200
- Retain Proportion: 0.2
- Mutation Proportion: 0.15
- Mutation Limit: 2.5

This resulted in a genetic algorithm that converged in approximately 250 generations to a mean squared error over the dataset of less than 1.0. The performance of the resultant configuration is shown in Figure 9, using a logarithmic scale for clarity and to show the rapid initial convergence.

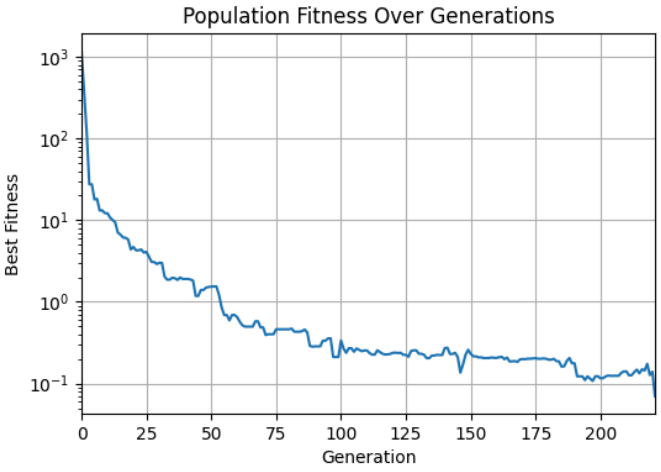

Figure 9: Plot of a single run of the resultant genetic algorithm.

The final co-efficients found by the genetic algorithm were as follows:

Co-efficient	Target	GA-Identified	Error
a	25	25.152	0.152
b	18	18.036	0.036
c	31	30.308	0.692
d	−14	−14.250	0.250
e	7	7.583	0.583
f	−19	−18.663	0.337

Table 1: Final genetic algorithm co-efficients

While most of these are close to their targets, some such as c and e are further off. The result is likely restricted by the size of test data. A larger set would likely result in a more accurate output, but would have taken significantly longer to compute.

6. Exercise 5

Explaining Holland's Schema Theorem based on exercise 4 using a genetic algorithm with binary encoding.

Holland's Schema Theorem [5] suggests that short, low-order schema with above-average fitness tend to increase exponentially in successive generations of a genetic algorithm. It can be expressed with the following equation:

$$m(H, t+1) \geq m(H, t)\frac{\overline{f}(H,t)}{\overline{f}(t)}\left(1 - p_c\frac{\delta(H)}{L-1} - o(H)p_m\right)$$

Where:

- $m(H,t)$ is the number of instances of schema H at generation t
- $\overline{f}(H,t)$ is the average fitness of schema H at generation t
- $\overline{f}(t)$ is the average fitness of the population at generation t
- $\delta(H)$ is the defining length of schema H
- $o(H)$ is the order of schema H
- L is the length of the individuals
- p_m is the mutation probability
- p_c is the crossover probability

In other words, schemas that are short (low defining length) and simple (low order) are less likely to be disrupted by crossover and mutation, allowing them to propagate through generations if they contribute positively to fitness.

We can use the genetic algorithm developed in exercise 4 to illustrate this in more detail. In order to apply the schema theorem with binary encoding, the 'Individual' class was further modified to represent genes with a known, fixed size type. 16 bits was chosen for the gene size, with a good trade-off between being large enough to represent a wide range of values, but short enough to allow schemas to be analysed.

To work better in the co-efficient seeking problem, the genes were treated as signed 16-bit integers with a fixed-point scaling factor of 0.001. This allows the genes to represent co-efficients in the range −32.768 to 32.767 with a fixed precision of three decimal places.

6.1. Demonstrating the Schema Theorem

To demonstrate Holland's Schema Theorem, we can chose 5 representative schemas to track over generations. Using the constant co-efficient as an example, we can define the following schemas, based on a decreasing defining length:

Schema	Pattern	Range
A	`1011010111001000`	−19.000 (exact)
B	`10110101********`	−19.200 to −18.945
C	`1011**********`	−20.000 to −16.385
D	`10**********`	−32.768 to −16.385
E	`1**********`	−32.768 to 0.000

Table 2: Schema Patterns for Constant Coefficient

Schema values were defined by the 2's compliment representation of signed 16-bit integers, scaled by 0.001, and with the '*' wildcard character representing bits outside of the schema.

The prevalence of these values during an optimisation run was then tracked, using the modified source code found in Section 9.5. A plot of the schema fitness using the 16-bit genes is shown in Figure 10 and the propagation of the schemas over time is shown in Figure 11.

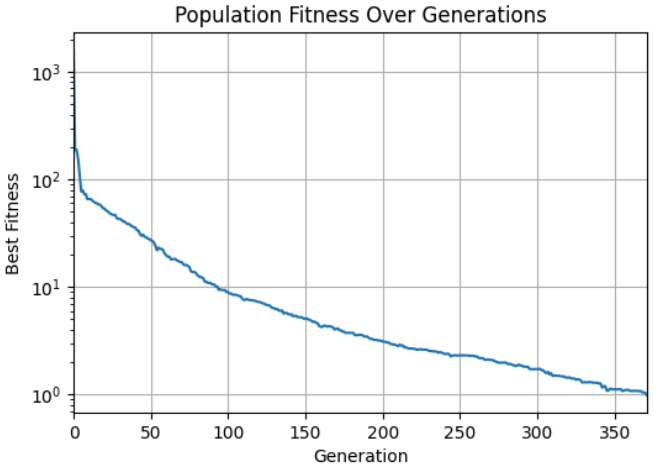

Figure 10: Best population fitness over generations with 16-bit genes.

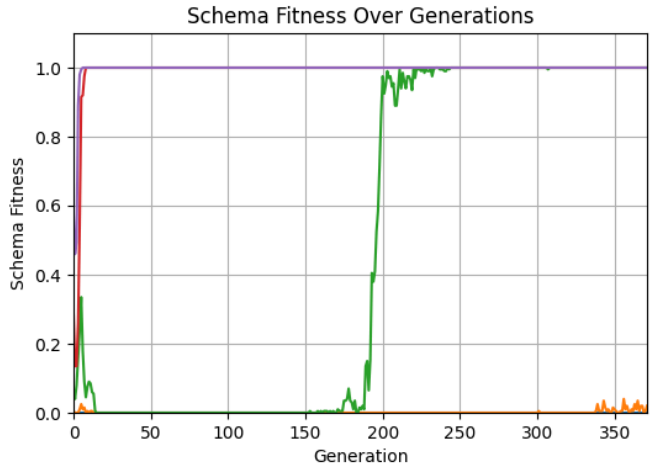

Figure 11: Schema fitness over generations - Schema A (purple), Schema B (red), Schema C (green), Schema D (orange) and Schema E (blue).

As shown in Figure 11, the trend between schema length and prevalence is extremely clear. The shortest schema (D and E) were almost immediately adopted by all individuals in the population, schema C was adopted by all around halfway through the run, schema B was only adopted by some of the population towards the end, and schema A was never adopted by any individuals (as the exact value was never found by the algorithm).

This is a clear demonstration of Holland's Schema Therem, showing that shorter, lower-order schemas with above-average fitness tend to propagate more successfully through generations of a genetic algorithm, and therefore demonstrating the capabilty of genetic algorithms to converge on optimal solutions over time.

7. Conclusion

This coursework has successfully demonstrated the implementation, analysis and application of genetic algorithms for optimisation tasks. While the problems and algorithms explored here are relatively simple, they illustrate the fundamental principles and capabilities of genetic algorithms, which could be extended to more complex and real-world problems in future work.

Futhermore, the demonstration of Holland's Schema Theorem provides insight into the underlying processes that drive convergence and optimisation in genetic algorithms, by focusing on a bit-level analysis of schema propagation.

8. References

[1] S. Forrest, "Genetic Algorithms," 1996. [Online]. Available: https://doi.org/10.1145/234313.234350

[2] J. H. Holland, "Genetic Algorithms," 1992. [Online]. Available: http://www.jstor.org/stable/24939139

[3] Y. M. M. Sevaux, "A curve-fitting genetic algorithm for a styling application," 2004. [Online]. Available: https://doi.org/10.1016/j.ejor.2005.03.065

[4] T. Y. O Koksoy, "Mean square error criteria to multiresponse process optimization by a new genetic algorithm," 2006. [Online]. Available: https://doi.org/10.1016/j.amc.2005.09.011

[5] J. H. Holland, *Adaptation in Natural and Artificial Systems*. 1975.

9. Appendices

9.1. Exercise 1: Source Code

9.1.1. individual.py

```python
######################################################################
##
## EE40098 Coursework B
##
## File         :  individual.py
## Exercise     :  1
## Author       :  samh25
## Created      :  2025-11-14 (YYYY-MM-DD)
## License      :  MIT
## Description  :  A class representing an individual in a
##                 genetic algorithm.
##
######################################################################

######################################################################
## MARK: INCLUDES
######################################################################

from random import uniform, random

######################################################################
## MARK: CLASS DEFINITIONS
######################################################################

class Individual:

    ##################################################################
    ## STATIC VARIABLES

    # example starting parameters
    min = 0
    max = 100
    mutation_limit = 5
    target = 42
    crossover_variance = 1

    ##################################################################
    ## STATIC METHODS

    # set parameters for all individuals
    def set_parameters(min, max, target, mutation_limit, crossover_variance):
        Individual.min = min
        Individual.max = max
        Individual.target = target
        Individual.mutation_limit = mutation_limit
        Individual.crossover_variance = crossover_variance

    # get the worst possible fitness value
    def get_worst_fitness():
        return Individual.target

    # create a child individual from two parents
    def crossover(male, female):

        child = Individual()

        # use blend crossover
        alpha = 0.5 - ((random() / 2) * Individual.crossover_variance)
        child.gene = (male.gene * alpha) + (female.gene * (1 - alpha))

        return child

```

```
    ############################################################
    ## CONSTRUCTOR

    # instantiate a new individual
    def __init__(self):
        self.gene = uniform(Individual.min, Individual.max)

    ############################################################
    ## INSTANCE METHODS

    # mutate this individual
    def mutate(self):

        # use a small, limited range mutation
        mutation = uniform(-Individual.mutation_limit, Individual.mutation_limit)
        self.gene = max(Individual.min, min(Individual.max, self.gene + mutation))

    # evaluate the fitness of this individual
    def evaluate_fitness(self):
        return abs(Individual.target - self.gene)

```

9.1.2. population.py

```
######################################################################
##
## EE40098 Coursework B
##
## File         :  population.py
## Exercise     :  1
## Author       :  samh25
## Created      :  2025-11-14 (YYYY-MM-DD)
## License      :  MIT
## Description  :  A class representing an population in a
##                 genetic algorithm.
##
######################################################################

######################################################################
## MARK: INCLUDES
######################################################################

from random import randint, random
import matplotlib.pyplot as plt

from .individual import Individual

######################################################################
## MARK: CLASS DEFINITIONS
######################################################################

class Population:

    ############################################################
    ## STATIC VARIABLES

    # example starting parameters
    retain = 0.2
    random_select = 0.05
    mutate = 0.01

    ############################################################
    ## STATIC METHODS

    # set parameters for all populations
    def set_parameters(retain, random_select, mutate):
        Population.retain = retain
        Population.random_select = random_select
        Population.mutate = mutate
```

```
46
47      ############################################################
48      ## CONSTRUCTOR
49
50      # instantiate a new population
51      def __init__(self, size):
52
53          # create a list of individuals
54          self.individuals = [Individual() for _ in range(size)]
55
56          # initialize fitness history
57          self.fitness_history = [self.evaluate_fitness()]
58
59      ############################################################
60      ## INSTANCE METHODS
61
62      # evaluate the fitness of this population
63      def evaluate_fitness(self):
64
65          # find the worst possible fitness value
66          min_error = Individual.get_worst_fitness()
67
68          # find the best fitness in the population
69          for i in range(len(self.individuals)):
70              min_error = min(min_error, self.individuals[i].evaluate_fitness())
71
72          return min_error
73
74      # evolve this population to the next generation
75      def evolve(self):
76
77          # evaluate fitness of all individuals and sort them
78          evaluated_individuals = [(individual.evaluate_fitness(), individual) for individual in
    self.individuals]
79          evaluated_individuals = [x[1] for x in sorted(evaluated_individuals, key=lambda x:
    x[0])]
80
81          # select the best individuals to be parents
82          retain_length = int(len(evaluated_individuals) * self.retain)
83          parents = evaluated_individuals[:retain_length]
84
85          # randomly individuals outside of the best to promote genetic diversity
86          for individual in evaluated_individuals[retain_length:]:
87              if self.random_select > random():
88                  parents.append(individual)
89
90          # mutate some individuals
91          for individual in parents:
92              if self.mutate > random():
93                  individual.mutate()
94
95          # identify number of children to create
96          parents_length = len(parents)
97          desired_length = len(self.individuals) - parents_length
98
99          # create children until we have a full population again
100         children = []
101
102         while len(children) < desired_length:
103             male = randint(0, parents_length - 1)
104             female = randint(0, parents_length - 1)
105             if male != female:
106                 male = parents[male]
107                 female = parents[female]
108
109                 child = Individual.crossover(male, female)
110                 children.append(child)
111
```

```
112         # create the new generation
113         parents.extend(children)
114         self.individuals = parents
115 
116         # evaluate fitness and record history
117         fitness = self.evaluate_fitness()
118         self.fitness_history.append(fitness)
119 
120     # get the current best fitness in the population
121     def get_fitness(self):
122         if self.fitness_history.__len__() > 0:
123             return self.fitness_history[-1]
124         else:
125             return Individual.get_worst_fitness()
126 
127     # plot the fitness history with matplotlib
128     def plot_fitness_history(self):
129         plt.figure(figsize=(6, 4))
130         plt.plot(self.fitness_history)
131         plt.title("Population Fitness Over Generations")
132         plt.xlabel("Generation")
133         plt.ylabel("Best Fitness")
134         plt.xlim(0, self.fitness_history.__len__() - 1)
135         plt.grid(True)
136         plt.show()
```

9.1.3. main.py

```
 1 ######################################################################
 2 ##
 3 ## EE40098 Coursework B
 4 ##
 5 ## File         :  main.py
 6 ## Exercise     :  1
 7 ## Author       :  samh25
 8 ## Created      :  2025-11-14 (YYYY-MM-DD)
 9 ## License      :  MIT
10 ## Description  :  Main program for exercise 1.
11 ##
12 ######################################################################
13 
14 ######################################################################
15 ## MARK: INCLUDES
16 ######################################################################
17 
18 from ga import Population, Individual
19 
20 ######################################################################
21 ## MARK: FUNCTIONS
22 ######################################################################
23 
24 # main program entry point
25 def main():
26 
27     # set parameters
28     target = 50
29     population_size = 10
30     individual_min = 0
31     individual_max = 100
32     generations = 10
33     retain = 0.2
34     random_select = 0.05
35     mutate = 0.3
36     mutation_limit = 10
37     crossover_variance = 1
38 
39     # configure individual and population parameters
40     Individual.set_parameters(min = individual_min, max = individual_max, target = target,
   mutation_limit = mutation_limit, crossover_variance=crossover_variance)
```

```
    Population.set_parameters(retain = retain, random_select = random_select, mutate = mutate)

    # create initial population
    population = Population(population_size)

    # evolve population over a number of generations
    for _ in range(generations):
        population.evolve()

    # plot fitness history
    population.plot_fitness_history()

# assign main function to entry point
if __name__ == '__main__':
    main()
```

9.2. Exercise 2: Source Code

9.2.1. main.py

```
######################################################################
##
## EE40098 Coursework B
##
## File         :  main.py
## Exercise     :  2
## Author       :  samh25
## Created      :  2025-11-14 (YYYY-MM-DD)
## License      :  MIT
## Description  :  Main program for exercise 2.
##
######################################################################

######################################################################
## MARK: INCLUDES
######################################################################

from ga import Population, Individual
import random
import matplotlib.pyplot as plt

######################################################################
## MARK: FUNCTIONS
######################################################################

# main program entry point
def main():

    # set parameters
    target = 50
    population_size = 10
    individual_min = 0
    individual_max = 100
    generations = 100
    retain = 0.2
    random_select = 0.05
    mutate = 0.3
    mutation_limit = 10
    crossover_variance = 1

    population_sizes_count = 10000
    mutation_proportions_count = 10000
    mutation_limits_count = 10000
    retain_proportions_count = 10000
    crossover_variances_count = 10000

    # configure individual and population parameters
    Individual.set_parameters(min = individual_min, max = individual_max, target = target,
mutation_limit = mutation_limit, crossover_variance=crossover_variance)
```

```
    Population.set_parameters(retain = retain, random_select = random_select, mutate = mutate)

    ############################################################
    ## FIRST PASS - POPULATION SIZE

    min_population_size = 10
    max_population_size = 250
    population_sizes = [random.randint(min_population_size, max_population_size) for _ in
range(population_sizes_count)]
    population_size_performance = []

    for i in range(len(population_sizes)):

        print("Testing population size:", i)

        population_size = population_sizes[i]

        # create initial population
        population = Population(population_size)
        seen_best = False

        # evolve population over a number of generations
        for j in range(generations):
            if population.get_fitness() < 0.01:
                population_size_performance.append(j + 1)
                seen_best = True
                break

            population.evolve()

        if not seen_best:
            population_size_performance.append(generations)

    # plot
    plt.figure(figsize=(6, 4))
    plt.scatter(population_sizes, population_size_performance, s=5)
    plt.title('Population Size vs Generations to Converge')
    plt.xlabel('Population Size')
    plt.xlim(min_population_size, max_population_size)
    plt.ylim(0, generations)
    plt.ylabel('Generations to Converge')
    plt.grid()
    plt.show()

    ############################################################
    ## SECOND PASS - MUTATION PROPORTION

    min_mutation_proportion = 0
    max_mutation_proportion = 1
    mutation_proportions = [random.uniform(min_mutation_proportion, max_mutation_proportion)
for _ in range(mutation_proportions_count)]
    mutation_proportion_performance = []

    for i in range(len(mutation_proportions)):

        print("Testing mutation proportion:", i)

        mutate = mutation_proportions[i]
        Population.set_parameters(retain = retain, random_select = random_select, mutate =
mutate)

        # create initial population
        population = Population(population_size)

        seen_best = False

        # evolve population over a number of generations
        for j in range(generations):
```

```
114             if population.get_fitness() < 0.01:
115                 mutation_proportion_performance.append(j + 1)
116                 seen_best = True
117                 break
118 
119             population.evolve()
120 
121         if not seen_best:
122             mutation_proportion_performance.append(generations)
123 
124     # plot
125     plt.figure(figsize=(6, 4))
126     plt.scatter(mutation_proportions, mutation_proportion_performance, s=5)
127     plt.title('Mutation Proportion vs Generations to Converge')
128     plt.xlabel('Mutation Proportion')
129     plt.xlim(min_mutation_proportion, max_mutation_proportion)
130     plt.ylim(0, generations)
131     plt.ylabel('Generations to Converge')
132     plt.grid()
133     plt.show()
134 
135     ############################################################
136     ## THIRD PASS - MUTATION LIMIT
137 
138     min_mutation_limit = 1
139     max_mutation_limit = 100
140     mutation_limits = [random.uniform(min_mutation_limit, max_mutation_limit) for _ in
    range(mutation_limits_count)]
141     mutation_limit_performance = []
142 
143     for i in range(len(mutation_limits)):
144 
145         print("Testing mutation limit:", i)
146 
147         mutation_limit = mutation_limits[i]
148         Individual.set_parameters(min = individual_min, max = individual_max, target = target,
    mutation_limit = mutation_limit, crossover_variance=crossover_variance)
149 
150         # create initial population
151         population = Population(population_size)
152 
153         seen_best = False
154 
155         # evolve population over a number of generations
156         for j in range(generations):
157             if population.get_fitness() < 0.01:
158                 mutation_limit_performance.append(j + 1)
159                 seen_best = True
160                 break
161 
162             population.evolve()
163 
164         if not seen_best:
165             mutation_limit_performance.append(generations)
166 
167     # plot
168     plt.figure(figsize=(6, 4))
169     plt.scatter(mutation_limits, mutation_limit_performance, s=5)
170     plt.title('Mutation Limit vs Generations to Converge')
171     plt.xlabel('Mutation Limit')
172     plt.xlim(min_mutation_limit, max_mutation_limit)
173     plt.ylim(0, generations)
174     plt.ylabel('Generations to Converge')
175     plt.grid()
176     plt.show()
177 
178     ############################################################
179     ## FORTH PASS - RETAIN PROPORTION
```

```
180
181     generations = 200
182
183     min_retain_proportion = 0.1
184     max_retain_proportion = 1
185     retain_proportions = [random.uniform(min_retain_proportion, max_retain_proportion) for _ in
    range(retain_proportions_count)]
186     retain_proportion_performance = []
187
188     for i in range(len(retain_proportions)):
189
190         print("Testing retain proportion:", i)
191
192         retain = retain_proportions[i]
193         Population.set_parameters(retain = retain, random_select = random_select, mutate =
    mutate)
194
195         # create initial population
196         population = Population(population_size)
197
198         seen_best = False
199
200         # evolve population over a number of generations
201         for j in range(generations):
202             if population.get_fitness() < 0.01:
203                 retain_proportion_performance.append(j + 1)
204                 seen_best = True
205                 break
206
207             population.evolve()
208
209         if not seen_best:
210             retain_proportion_performance.append(generations)
211
212     # plot
213     plt.figure(figsize=(6, 4))
214     plt.scatter(retain_proportions, retain_proportion_performance, s=5)
215     plt.title('Retain Proportion vs Generations to Converge')
216     plt.xlabel('Retain Proportion')
217     plt.xlim(min_retain_proportion, max_retain_proportion)
218     plt.ylim(0, generations)
219     plt.ylabel('Generations to Converge')
220     plt.grid()
221     plt.show()
222
223     ###############################################################
224     ## FIFTH PASS - CROSSOVER VARIANCE
225
226     generations = 100
227
228     min_crossover_variance = 0
229     max_crossover_variance = 1
230     crossover_variances = [random.uniform(min_crossover_variance, max_crossover_variance) for _
    in range(crossover_variances_count)]
231     crossover_variance_performance = []
232
233     for i in range(len(crossover_variances)):
234
235         print("Testing crossover variance:", i)
236
237         crossover_variance = crossover_variances[i]
238         Individual.set_parameters(min = individual_min, max = individual_max, target = target,
    mutation_limit = mutation_limit, crossover_variance=crossover_variance)
239
240         # create initial population
241         population = Population(population_size)
242
243         seen_best = False
```

```
244
245         # evolve population over a number of generations
246         for j in range(generations):
247             if population.get_fitness() < 0.01:
248                 crossover_variance_performance.append(j + 1)
249                 seen_best = True
250                 break
251
252             population.evolve()
253
254         if not seen_best:
255             crossover_variance_performance.append(generations)
256
257     # plot
258     plt.figure(figsize=(6, 4))
259     plt.scatter(crossover_variances, crossover_variance_performance, s=5)
260     plt.title('Crossover Variance vs Generations to Converge')
261     plt.xlabel('Crossover Variance')
262     plt.xlim(min_crossover_variance, max_crossover_variance)
263     plt.ylim(0, generations)
264     plt.ylabel('Generations to Converge')
265     plt.grid()
266     plt.show()
267
268
269 # assign main function to entry point
270 if __name__ == '__main__':
271     main()
```

9.3. Exercise 3: Source Code

9.3.1. main.py

```
 1 ######################################################################
 2 ##
 3 ## EE40098 Coursework B
 4 ##
 5 ## File         :  main.py
 6 ## Exercise     :  3
 7 ## Author       :  samh25
 8 ## Created      :  2025-11-17 (YYYY-MM-DD)
 9 ## License      :  MIT
10 ## Description  :  Main program for exercise 3.
11 ##
12 ######################################################################
13
14 ######################################################################
15 ## MARK: INCLUDES
16 ######################################################################
17
18 from ga import Population, Individual
19 import random
20 import matplotlib.pyplot as plt
21
22 ######################################################################
23 ## MARK: FUNCTIONS
24 ######################################################################
25
26 # main program entry point
27 def main():
28
29     # set parameters
30     target = 50
31     population_size = 10
32     individual_min = 0
33     individual_max = 100
34     generations = 200
35     retain = 0.2
36     random_select = 0.05
```

```
    mutate = 0.3
    mutation_limit = 10
    crossover_variance = 1

    stop_conditions_count = 10000

    # configure individual and population parameters
    Individual.set_parameters(min = individual_min, max = individual_max, target = target,
mutation_limit = mutation_limit, crossover_variance=crossover_variance)
    Population.set_parameters(retain = retain, random_select = random_select, mutate = mutate)

    min_stop_condition = 0
    max_stop_condition = 0.5
    stop_conditions = [random.uniform(min_stop_condition, max_stop_condition) for _ in
range(stop_conditions_count)]
    stop_conditions_performance = []

    for i in range(len(stop_conditions)):

        print("Testing stop limit:", i)
        stop_condition = stop_conditions[i]

        # create initial population
        population = Population(population_size)
        seen_best = False

        # evolve population over a number of generations
        for i in range(generations):
            population.evolve()

            best_fitness = population.evaluate_fitness()

            if best_fitness < stop_condition:
                stop_conditions_performance.append(i)
                seen_best = True
                break

        if not seen_best:
            stop_conditions_performance.append(generations)


    # plot
    plt.figure(figsize=(6, 4))
    plt.scatter(stop_conditions, stop_conditions_performance, s=5)
    plt.title('Stop Threshold vs Generations to Converge')
    plt.xlabel('Stop Threshold')
    plt.xlim(min_stop_condition, max_stop_condition)
    plt.ylim(0, generations)
    plt.ylabel('Generations to Converge')
    plt.grid()
    plt.show()


# assign main function to entry point
if __name__ == '__main__':
    main()
```

9.4. Exercise 4: Source Code

9.4.1. individual.py

```
################################################################
##
## EE40098 Coursework B
##
## File         :  individual.py
## Exercise     :  4
## Author       :  samh25
## Created      :  2025-11-17 (YYYY-MM-DD)
```

```
## License      :  MIT
## Description  :  A class representing an individual in a
##                 genetic algorithm.
##
#######################################################################

#######################################################################
## MARK: INCLUDES
#######################################################################

from random import uniform, random

#######################################################################
## MARK: CLASS DEFINITIONS
#######################################################################

class Individual:

    ###################################################################
    ## STATIC VARIABLES

    genes_count = 6

    # example starting parameters
    min = 0
    max = 100
    mutation_limit = 5
    target_data = []
    crossover_variance = 1

    ###################################################################
    ## STATIC METHODS

    # set parameters for all individuals
    def set_parameters(min, max, target_data, mutation_limit, crossover_variance, genes_count):
        Individual.min = min
        Individual.max = max
        Individual.target_data = target_data
        Individual.mutation_limit = mutation_limit
        Individual.crossover_variance = crossover_variance
        Individual.genes_count = genes_count

    # get the worst possible fitness value
    def get_worst_fitness():
        return float('inf')

    # create a child individual from two parents
    def crossover(male, female):

        child = Individual()

        for i in range(Individual.genes_count):

            # use blend crossover
            alpha = 0.5 - ((random() / 2) * Individual.crossover_variance)
            child.genes[i] = (male.genes[i] * alpha) + (female.genes[i] * (1 - alpha))

        return child

    ###################################################################
    ## CONSTRUCTOR

    # instantiate a new individual
    def __init__(self):
        self.genes = [uniform(Individual.min, Individual.max) for _ in
range(Individual.genes_count)]

    ###################################################################
```

```
    ## INSTANCE METHODS

    # mutate this individual per-gene
    def mutate(self, gene_index):

        # use a small, limited range mutation
        mutation = uniform(-Individual.mutation_limit, Individual.mutation_limit) * 1.0 # ensure float
        self.genes[gene_index] = max(Individual.min, min(Individual.max, self.genes[gene_index] + mutation))

    # evaluate the fitness of this individual, using absolute error over dataset
    def evaluate_fitness(self):

        total_error = 0.0
        a, b, c, d, e, f = self.genes

        for x, y_target in Individual.target_data:
            y_pred = (a*(x**5)) + (b*(x**4)) + (c*(x**3)) + (d*(x**2)) + (e*x) + f
            total_error += (y_pred - y_target) ** 2 # using squared error

        mean_error = total_error / len(Individual.target_data)

        return mean_error


```

9.4.2. population.py

```
##########################################################################
##
## EE40098 Coursework B
##
## File         :  population.py
## Exercise     :  4
## Author       :  samh25
## Created      :  2025-11-17 (YYYY-MM-DD)
## License      :  MIT
## Description  :  A class representing an population in a
##                 genetic algorithm.
##
##########################################################################

##########################################################################
## MARK: INCLUDES
##########################################################################

from random import randint, random, shuffle
import matplotlib.pyplot as plt

from .individual import Individual

##########################################################################
## MARK: CLASS DEFINITIONS
##########################################################################

class Population:

    ######################################################################
    ## STATIC VARIABLES

    # example starting parameters
    retain = 0.2
    random_select = 0.05
    mutate = 0.01

    ######################################################################
    ## STATIC METHODS

```

```
41      # set parameters for all populations
42      def set_parameters(retain, random_select, mutate):
43          Population.retain = retain
44          Population.random_select = random_select
45          Population.mutate = mutate
46
47      ##############################################################
48      ## CONSTRUCTOR
49
50      # instantiate a new population
51      def __init__(self, size):
52
53          # create a list of individuals
54          self.individuals = [Individual() for _ in range(size)]
55
56          # initialize fitness history
57          self.fitness_history = [self.evaluate_fitness()]
58          self.best_individual = None
59
60      ##############################################################
61      ## INSTANCE METHODS
62
63      # evaluate the fitness of this population
64      def evaluate_fitness(self):
65
66          # find the worst possible fitness value
67          min_error = Individual.get_worst_fitness()
68
69          # find the best fitness in the population
70          for i in range(len(self.individuals)):
71              min_error = min(min_error, self.individuals[i].evaluate_fitness())
72
73              # store the best individual
74              if min_error == self.individuals[i].evaluate_fitness():
75                  self.best_individual = self.individuals[i]
76
77          return min_error
78
79      # evolve this population to the next generation
80      def evolve(self):
81
82          # evaluate fitness of all individuals and sort them
83          evaluated_individuals = [(individual.evaluate_fitness(), individual) for individual in
    self.individuals]
84          evaluated_individuals = [x[1] for x in sorted(evaluated_individuals, key=lambda x:
    x[0])]
85
86          # select the best individuals to be parents
87          retain_length = int(len(evaluated_individuals) * self.retain)
88          parents = evaluated_individuals[:retain_length]
89
90          # randomly individuals outside of the best to promote genetic diversity
91          for individual in evaluated_individuals[retain_length:]:
92              if self.random_select > random():
93                  parents.append(individual)
94
95          # mutate some individuals
96          for individual in parents:
97              for gene_index in range(Individual.genes_count):
98                  if self.mutate > random():
99                      individual.mutate(gene_index)
100
101         # identify number of children to create
102         parents_length = len(parents)
103         desired_length = len(self.individuals) - parents_length
104
105         # Shuffle parents and breed sequentially (no infinite loop ever)
106         shuffle(parents)
```

```
107         children = []
108 
109         for i in range(desired_length):
110             # Cycle through parents if we run out
111             male = parents[i % parents_length]
112             female = parents[(i + 1) % parents_length]    # guaranteed different
113             child = Individual.crossover(male, female)
114             children.append(child)
115 
116         # create the new generation
117         parents.extend(children)
118         self.individuals = parents
119 
120         # evaluate fitness and record history
121         fitness = self.evaluate_fitness()
122         self.fitness_history.append(fitness)
123 
124     # get the current best fitness in the population
125     def get_fitness(self):
126         if self.fitness_history.__len__() > 0:
127             return self.fitness_history[-1]
128         else:
129             return Individual.get_worst_fitness()
130 
131     # get the current best individual in the population
132     def get_best_individual(self):
133         return self.best_individual
134 
135     # plot the fitness history with matplotlib
136     def plot_fitness_history(self):
137         plt.figure(figsize=(6, 4))
138         plt.plot(self.fitness_history)
139         plt.title("Population Fitness Over Generations")
140         plt.xlabel("Generation")
141         plt.ylabel("Best Fitness")
142         plt.yscale("log")
143         plt.xlim(0, self.fitness_history.__len__() - 1)
144         plt.grid(True)
145         plt.show()
146 
```

9.4.3. main.py

```
  1 ######################################################################
  2 ##
  3 ## EE40098 Coursework B
  4 ##
  5 ## File         :  main.py
  6 ## Exercise     :  4
  7 ## Author       :  samh25
  8 ## Created      :  2025-11-17 (YYYY-MM-DD)
  9 ## License      :  MIT
 10 ## Description  :  Main program for exercise 4.
 11 ##
 12 ######################################################################
 13 
 14 ######################################################################
 15 ## MARK: INCLUDES
 16 ######################################################################
 17 
 18 from ga import Population, Individual
 19 import random
 20 import matplotlib.pyplot as plt
 21 
 22 ######################################################################
 23 ## MARK: FUNCTIONS
 24 ######################################################################
 25 
 26 # sample polynomial: 25x^5 + 18x^4 + 31x^3 - 14x^2 + 7x - 19
```

```
27 def sample_poynomial(count, min_x, max_x):
28
29     data = []
30
31     for _ in range(count):
32
33         x = random.uniform(min_x, max_x)
34         y = 25*(x**5) + 18*(x**4) + 31*(x**3) - 14*(x**2) + 7*x - 19
35
36         data.append((x, y))
37
38     return data
39
40 # main program entry point
41 def main():
42
43     # set parameters
44     target = 50
45     population_size = 100
46     individual_min = -50
47     individual_max = 50
48     generations = 1000
49     retain = 0.2
50     random_select = 0.05
51     mutate = 0.15
52     mutation_limit = 1.0
53     crossover_variance = 0.5
54
55     genes_count = 6
56
57     dataset = sample_poynomial(100, -2, 2)
58
59     search_generations = 10
60
61     population_sizes_count = 1
62     retain_proportions_count = 1
63     mutation_proportions_count = 1
64     mutation_limits_count = 1
65
66     ################################################################
67     ## FIRST PASS - POPULATION SIZE
68
69     # configure individual and population parameters
70     Individual.set_parameters(min = individual_min, max = individual_max, target_data =
   dataset, mutation_limit = mutation_limit, crossover_variance=crossover_variance,
   genes_count=genes_count)
71     Population.set_parameters(retain = retain, random_select = random_select, mutate = mutate)
72
73     min_population_size = 10
74     max_population_size = 500
75     population_sizes = [random.randint(min_population_size, max_population_size) for _ in
   range(population_sizes_count)]
76
77     population_size_performance = []
78
79     for i in range(len(population_sizes)):
80
81         print("Testing population size:", i)
82
83         population_size = population_sizes[i]
84
85         # create initial population
86         population = Population(population_size)
87
88         fitness = 0
89
90         # evolve population over a number of generations
91         for i in range(search_generations):
```

```
92
93              population.evolve()
94              best_fitness = population.evaluate_fitness()
95
96          population_size_performance.append(best_fitness)
97
98      # plot
99      plt.figure(figsize=(6, 4))
100     plt.scatter(population_sizes, population_size_performance, s=5)
101     plt.title('Population Size vs Convergence Performance')
102     plt.xlabel('Population Size')
103     plt.xlim(min_population_size, max_population_size)
104     plt.ylim(0, generations)
105     plt.ylabel('Error after ' + str(search_generations) + ' Generations')
106     plt.grid()
107     plt.show()
108
109     population_size = 100 # reset for next tests
110
111     ###############################################################
112     ## SECOND PASS - RETAIN PROPORTION SIZE
113
114     min_retain_proportion = 0.1
115     max_retain_proportion = 0.5
116     retain_proportions = [random.uniform(min_retain_proportion, max_retain_proportion) for _ in
    range(retain_proportions_count)]
117     retain_proportion_performance = []
118
119     for i in range(len(retain_proportions)):
120
121         print("Testing retain proportion:", i)
122
123         retain = retain_proportions[i]
124         Population.set_parameters(retain = retain, random_select = random_select, mutate =
    mutate)
125
126         # create initial population
127         population = Population(population_size)
128
129         fitness = 0
130
131         # evolve population over a number of generations
132         for i in range(search_generations):
133
134             population.evolve()
135             fitness = population.evaluate_fitness()
136
137         retain_proportion_performance.append(fitness)
138
139     # plot
140     plt.figure(figsize=(6, 4))
141     plt.scatter(retain_proportions, retain_proportion_performance, s=5)
142     plt.title('Retain Proportion vs Convergence Performance')
143     plt.xlabel('Retain Proportion')
144     plt.xlim(min_retain_proportion, max_retain_proportion)
145     plt.ylim(0, generations)
146     plt.ylabel('Error after ' + str(search_generations) + ' Generations')
147     plt.grid()
148     plt.show()
149
150
151     ###############################################################
152     ## THIRD PASS - MUTATION PROPORTION
153
154     min_mutation_proportion = 0
155     max_mutation_proportion = 0.5
156     mutation_proportions = [random.uniform(min_mutation_proportion, max_mutation_proportion)
    for _ in range(mutation_proportions_count)]
```

```
157     mutation_proportion_performance = []
158 
159     for i in range(len(mutation_proportions)):
160 
161         print("Testing mutate proportion:", i)
162 
163         mutate = mutation_proportions[i]
164         Population.set_parameters(retain = retain, random_select = random_select, mutate =
    mutate)
165 
166         # create initial population
167         population = Population(population_size)
168 
169         fitness = 0
170 
171         # evolve population over a number of generations
172         for i in range(search_generations):
173 
174             population.evolve()
175             fitness = population.evaluate_fitness()
176 
177         mutation_proportion_performance.append(fitness)
178 
179     # plot
180     plt.figure(figsize=(6, 4))
181     plt.scatter(mutation_proportions, mutation_proportion_performance, s=5)
182     plt.title('Mutate Proportion vs Convergence Performance')
183     plt.xlabel('Mutate Proportion')
184     plt.xlim(min_mutation_proportion, max_mutation_proportion)
185     plt.ylim(0, generations)
186     plt.ylabel('Error after ' + str(search_generations) + ' Generations')
187     plt.grid()
188     plt.show()
189 
190 
191     ###############################################################
192     ## FOURTH PASS - MUTATION LIMIT
193 
194     min_mutation_limit = 0
195     max_mutation_limit = 50
196     mutation_limits = [random.uniform(min_mutation_limit, max_mutation_limit) for _ in
    range(mutation_limits_count)]
197     mutation_limit_performance = []
198 
199     for i in range(len(mutation_limits)):
200 
201         print("Testing mutate limit:", i)
202 
203         mutation_limit = mutation_limits[i]
204         Population.set_parameters(retain = retain, random_select = random_select, mutate =
    mutate)
205         Individual.set_parameters(min = individual_min, max = individual_max, target_data =
    dataset, mutation_limit = mutation_limit, crossover_variance=crossover_variance,
    genes_count=genes_count)
206 
207         # create initial population
208         population = Population(population_size)
209 
210         fitness = 0
211 
212         # evolve population over a number of generations
213         for i in range(search_generations):
214 
215             population.evolve()
216             fitness = population.evaluate_fitness()
217 
218         mutation_limit_performance.append(fitness)
219 
```

```
    # plot
    plt.figure(figsize=(6, 4))
    plt.scatter(mutation_limits, mutation_limit_performance, s=5)
    plt.title('Mutate Limit vs Convergence Performance')
    plt.xlabel('Mutate Limit')
    plt.xlim(min_mutation_limit, max_mutation_limit)
    plt.ylim(0, generations)
    plt.ylabel('Error after ' + str(search_generations) + ' Generations')
    plt.grid()
    plt.show()

    ################################################################
    ## FINAL PASS - BEST CONFIGURATION

    individual_min = -50
    individual_max = 50
    generations = 1000
    random_select = 0.05
    mutate = 0.15
    population_size = 200
    retain = 0.2
    mutation_limit = 2.5
    crossover_variance = 0.5

    dataset = sample_poynomial(100, -2, 2)

    Individual.set_parameters(min = individual_min, max =  individual_max, target_data =
dataset, mutation_limit = mutation_limit, crossover_variance=crossover_variance,
genes_count=genes_count)
    Population.set_parameters(retain = retain, random_select = random_select, mutate = mutate)

    # create initial population
    population = Population(population_size)
    fitness = 0

    # evolve population over a number of generations
    for i in range(generations):

        population.evolve()
        fitness = population.evaluate_fitness()
        print("Generation:", i, "Best Fitness:", fitness)

        if (fitness < 1):

            best_individual = population.get_best_individual()
            print(" Best Individual Genes:", best_individual.genes)
            break

    population.plot_fitness_history()



# assign main function to entry point
if __name__ == '__main__':
    main()
```

9.5. Exercise 5: Source Code

9.5.1. schema.py

```
################################################################
##
## EE40098 Coursework B
##
## File         :  schema.py
## Exercise     :  5
## Author       :  samh25
## Created      :  2025-11-20 (YYYY-MM-DD)
## License      :  MIT
```

```
## Description  :  A class representing a schema in a
##                 genetic algorithm.
##
#####################################################################

#####################################################################
## MARK: INCLUDES
#####################################################################

import numpy as np

#####################################################################
## MARK: CLASS DEFINITIONS
#####################################################################

class Schema:

    ################################################################
    ## CONSTRUCTOR

    # instantiate a new schema
    def __init__(self, gene_index, bit_mask, bit_pattern):

        self.gene_index = gene_index

        # ensure bit mask and pattern are uint16
        self.bit_mask = np.uint16(bit_mask)
        self.bit_pattern = np.uint16(bit_pattern)
```

9.5.2. individual.py

```
#####################################################################
##
## EE40098 Coursework B
##
## File         :  individual.py
## Exercise     :  5
## Author       :  samh25
## Created      :  2025-11-18 (YYYY-MM-DD)
## License      :  MIT
## Description  :  A class representing an individual in a
##                 genetic algorithm.
##
#####################################################################

#####################################################################
## MARK: INCLUDES
#####################################################################

from .schema import Schema
from random import uniform, random
import numpy as np

#####################################################################
## MARK: CLASS DEFINITIONS
#####################################################################

class Individual:

    ################################################################
    ## STATIC VARIABLES

    genes_count = 6

    # example starting parameters
    min = 0
    max = 100
    mutation_limit = 5
    target_data = []
```

```
    crossover_variance = 1

    ################################################################
    ## STATIC METHODS

    # set parameters for all individuals
    def set_parameters(min, max, target_data, mutation_limit, crossover_variance, genes_count):
        Individual.min = min
        Individual.max = max
        Individual.target_data = target_data
        Individual.mutation_limit = mutation_limit
        Individual.crossover_variance = crossover_variance
        Individual.genes_count = genes_count

    # get the worst possible fitness value
    def get_worst_fitness():
        return float('inf')

    # create a child individual from two parents
    def crossover(male, female):

        child = Individual()

        for i in range(Individual.genes_count):

            # use blend crossover
            alpha = 0.5 - ((random() / 2) * Individual.crossover_variance)
            child.genes[i] = np.int16((male.genes[i] * alpha) + (female.genes[i] * (1 - alpha)))

        return child

    ################################################################
    ## CONSTRUCTOR

    # instantiate a new individual
    def __init__(self):
        self.genes = [np.int16(uniform(Individual.min, Individual.max)) for _ in range(Individual.genes_count)]

    ################################################################
    ## INSTANCE METHODS

    # mutate this individual per-gene
    def mutate(self, gene_index):

        # use a small, limited range mutation
        mutation = uniform(-Individual.mutation_limit, Individual.mutation_limit) * 1.0 # ensure float
        self.genes[gene_index] = np.int16(max(Individual.min, min(Individual.max, self.genes[gene_index] + mutation)))

    # evaluate if this individual matches a given schema
    def evaluate_schema(self, schema):
        gene_u16 = np.uint16(self.genes[schema.gene_index]) # cast to uint16 for bitwise operations
        return (gene_u16 & schema.bit_mask) == (schema.bit_pattern & schema.bit_mask)

    # evaluate the fitness of this individual, using absolute error over dataset
    def evaluate_fitness(self):

        total_error = 0.0

        coeffs = [gene / 1000.0 for gene in self.genes]
        a, b, c, d, e, f = coeffs

        for x, y_target in Individual.target_data:
            y_pred = (a*(x**5)) + (b*(x**4)) + (c*(x**3)) + (d*(x**2)) + (e*x) + f
```

```
            total_error += (y_pred - y_target) ** 2 # using squared error

        mean_error = total_error / len(Individual.target_data)

        return mean_error
```

9.5.3. population.py

```
##########################################################################
##
## EE40098 Coursework B
##
## File         :  population.py
## Exercise     :  5
## Author       :  samh25
## Created      :  2025-11-17 (YYYY-MM-DD)
## License      :  MIT
## Description  :  A class representing an population in a
##                 genetic algorithm.
##
##########################################################################

##########################################################################
## MARK: INCLUDES
##########################################################################

from random import randint, random, shuffle
import matplotlib.pyplot as plt

from .individual import Individual
from .schema import Schema

##########################################################################
## MARK: CLASS DEFINITIONS
##########################################################################

class Population:

    ######################################################################
    ## STATIC VARIABLES

    # example starting parameters
    retain = 0.2
    random_select = 0.05
    mutate = 0.01

    ######################################################################
    ## STATIC METHODS

    # set parameters for all populations
    def set_parameters(retain, random_select, mutate):
        Population.retain = retain
        Population.random_select = random_select
        Population.mutate = mutate

    ######################################################################
    ## CONSTRUCTOR

    # instantiate a new population
    def __init__(self, size, schema_list):

        # create a list of individuals
        self.individuals = [Individual() for _ in range(size)]

        # initialize fitness history
        self.fitness_history = [self.evaluate_fitness()]
        self.best_individual = None

        # store schema
```

```
62          self.schema_list = schema_list
63          self.schema_history = [self.evaluate_schema_fitness()]
64
65      ##############################################################
66      ## INSTANCE METHODS
67
68      # evaluate the fitness of this population
69      def evaluate_fitness(self):
70
71          # find the worst possible fitness value
72          min_error = Individual.get_worst_fitness()
73
74          # find the best fitness in the population
75          for i in range(len(self.individuals)):
76              min_error = min(min_error, self.individuals[i].evaluate_fitness())
77
78              # store the best individual
79              if min_error == self.individuals[i].evaluate_fitness():
80                  self.best_individual = self.individuals[i]
81
82          return min_error
83
84      def evaluate_schema_fitness(self):
85
86          schema_fitness_results = []
87
88          for schema in self.schema_list:
89
90              match_count = 0
91
92              for individual in self.individuals:
93
94                  if individual.evaluate_schema(schema):
95                      match_count += 1
96
97              print("Schema Gene Index:", schema.gene_index, "Bit Mask:", schema.bit_mask, "Bit
    Pattern:", schema.bit_pattern, "Match Count:", match_count)
98
99              schema_fitness_results.append(match_count / len(self.individuals))
100
101         return schema_fitness_results
102
103     # evolve this population to the next generation
104     def evolve(self):
105
106         # evaluate fitness of all individuals and sort them
107         evaluated_individuals = [(individual.evaluate_fitness(), individual) for individual in
    self.individuals]
108         evaluated_individuals = [x[1] for x in sorted(evaluated_individuals, key=lambda x:
    x[0])]
109
110         # select the best individuals to be parents
111         retain_length = int(len(evaluated_individuals) * self.retain)
112         parents = evaluated_individuals[:retain_length]
113
114         # randomly individuals outside of the best to promote genetic diversity
115         for individual in evaluated_individuals[retain_length:]:
116             if self.random_select > random():
117                 parents.append(individual)
118
119         # mutate some individuals
120         for individual in parents:
121             for gene_index in range(Individual.genes_count):
122                 if self.mutate > random():
123                     individual.mutate(gene_index)
124
125         # identify number of children to create
126         parents_length = len(parents)
```

```
        desired_length = len(self.individuals) - parents_length

        # Shuffle parents and breed sequentially (no infinite loop ever)
        shuffle(parents)
        children = []

        for i in range(desired_length):
            # Cycle through parents if we run out
            male = parents[i % parents_length]
            female = parents[(i + 1) % parents_length]    # guaranteed different
            child = Individual.crossover(male, female)
            children.append(child)

        # create the new generation
        parents.extend(children)
        self.individuals = parents

        # evaluate fitness and record history
        fitness = self.evaluate_fitness()
        self.fitness_history.append(fitness)

        schema_fitness = self.evaluate_schema_fitness()
        self.schema_history.append(schema_fitness)

    # get the current best fitness in the population
    def get_fitness(self):
        if self.fitness_history.__len__() > 0:
            return self.fitness_history[-1]
        else:
            return Individual.get_worst_fitness()

    # get the current best individual in the population
    def get_best_individual(self):
        return self.best_individual

    # plot the fitness history with matplotlib
    def plot_fitness_history(self):
        plt.figure(figsize=(6, 4))
        plt.plot(self.fitness_history)
        plt.title("Population Fitness Over Generations")
        plt.xlabel("Generation")
        plt.ylabel("Best Fitness")
        plt.yscale("log")
        plt.xlim(0, self.fitness_history.__len__() - 1)
        plt.grid(True)
        plt.show()

    def plot_schema_history(self):
        plt.figure(figsize=(6, 4))

        for schema_index in range(len(self.schema_list)):
            schema_fitness_values = [generation[schema_index] for generation in
self.schema_history]
            plt.plot(schema_fitness_values, label=f"Schema {chr(ord('a') + schema_index)}")

        plt.title("Schema Fitness Over Generations")
        plt.xlabel("Generation")
        plt.ylabel("Schema Fitness")
        plt.ylim(0, 1.1)
        plt.xlim(0, self.schema_history.__len__() - 1)
        plt.grid(True)
        plt.show()
```

9.5.4. main.py

```
##############################################################
##
## EE40098 Coursework B
##
```

```
## File         :  main.py
## Exercise     :  5
## Author       :  samh25
## Created      :  2025-11-18 (YYYY-MM-DD)
## License      :  MIT
## Description  :  Main program for exercise 5.
##
#####################################################################

#####################################################################
## MARK: INCLUDES
#####################################################################

from ga import Population, Individual, Schema
import random
import matplotlib.pyplot as plt

#####################################################################
## MARK: FUNCTIONS
#####################################################################

# sample polynomial: 25x^5 + 18x^4 + 31x^3 - 14x^2 + 7x - 19
def sample_poynomial(count, min_x, max_x):

    data = []

    for _ in range(count):

        x = random.uniform(min_x, max_x)
        y = 25*(x**5) + 18*(x**4) + 31*(x**3) - 14*(x**2) + 7*x - 19

        data.append((x, y))

    return data

# main program entry point
def main():

    # set parameters

    individual_min = -32768
    individual_max = 32767
    generations = 1000
    random_select = 0.05
    mutate = 0.15
    population_size = 200
    retain = 0.2
    mutation_limit = 500
    crossover_variance = 0.5
    genes_count = 6


    dataset = sample_poynomial(100, -2, 2)

    Individual.set_parameters(min = individual_min, max = individual_max, target_data =
dataset, mutation_limit = mutation_limit, crossover_variance=crossover_variance,
genes_count=genes_count)
    Population.set_parameters(retain = retain, random_select = random_select, mutate = mutate)

    # define schemas to track

    # full value of -19
    schema_a = Schema(gene_index=5, bit_mask=0b1111111111111111,
bit_pattern=0b1011010111001000)

    # upper 8 bits, in range -19.200 to -18.945
    schema_b = Schema(gene_index=5, bit_mask=0b1111111100000000,
bit_pattern=0b1011010111001000)
```

```
69
70      # upper 4 bits, in range -20.480 to -16.385
71      schema_c = Schema(gene_index=5, bit_mask=0b1111000000000000,
    bit_pattern=0b1011010111001000)
72
73      # upper 6 bits, in range -32.768 to -16.385
74      schema_d = Schema(gene_index=5, bit_mask=0b1100000000000000,
    bit_pattern=0b1011010111001000)
75
76      # MSB only
77      schema_e = Schema(gene_index=5, bit_mask=0b1000000000000000,
    bit_pattern=0b1011010111001000)
78
79      schema_list = [schema_a, schema_b, schema_c, schema_d, schema_e]
80
81      # create initial population
82      population = Population(population_size, schema_list)
83      fitness = 0
84
85      # evolve population over a number of generations
86      for i in range(generations):
87
88          population.evolve()
89          fitness = population.evaluate_fitness()
90          print("Generation:", i, "Best Fitness:", fitness)
91
92          if (fitness < 1):
93
94              best_individual = population.get_best_individual()
95              print(" Best Individual Genes:", best_individual.genes)
96              break
97
98      population.plot_fitness_history()
99      population.plot_schema_history()
100
101
102 # assign main function to entry point
103 if __name__ == '__main__':
104     main()
```