COEN 244 – FINAL PROJECT

Problem Definition

The main purpose of the program is to make use of a genetic algorithm that evolves a population over time in order to obtain the desired fittest individual. The algorithm starts with a set of S individuals that form a population. There are two types of populations which differ by how their fitness is determined.

The first type of population is the XY-type where each individual has parameters (x,y). Its fitness is set by the Ackley's equation:

$$f(x) = -20\,exp\left(-0.2\sqrt{0.5\,(x^2+y^2)}\right) - exp(0.5\,(\cos 2\pi x + \cos 2\pi y)) - e - 20$$

The second type of individual is the ABCD-type who has parameters (A, B, C, D) which form a third degree curve. Its fitness is determined by how well this curve fits a give equation: $1x^3 + 4x^2 - 20x - 30$. The mean-squared error equation calculates this fitness: MSE = $\frac{1}{n}\sum_{i=1}^{n}(observed - desired)^2$.

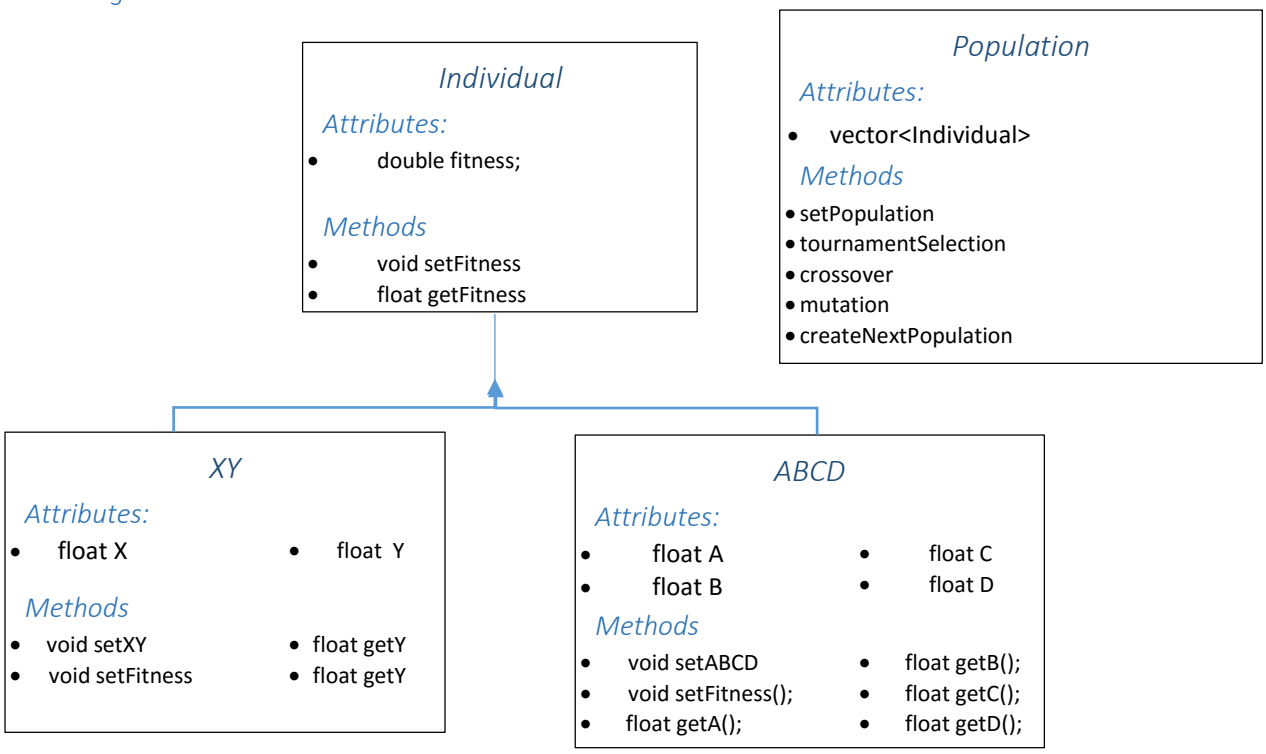
A parent pool is created by a tournament that select the fittest individual from three randomly picked individuals in the population. These parents then "crossover" to create 2 offspring whose parameters derive from the parents as such: α(parent 1) + (1- α) (parent 2) where alpha is a random percentage. Each offspring is "mutated" by adding noise. This process is repeated until there are 10 times more offspring than the initial population.

The best out of the parents and offspring go on to create the next population. The process is repeated until a desired fitness is obtained, solving the problem, or a maximum number of generations is attained.

Alternatives and Recommendations

Design 1 - Templates

Class Diagram:

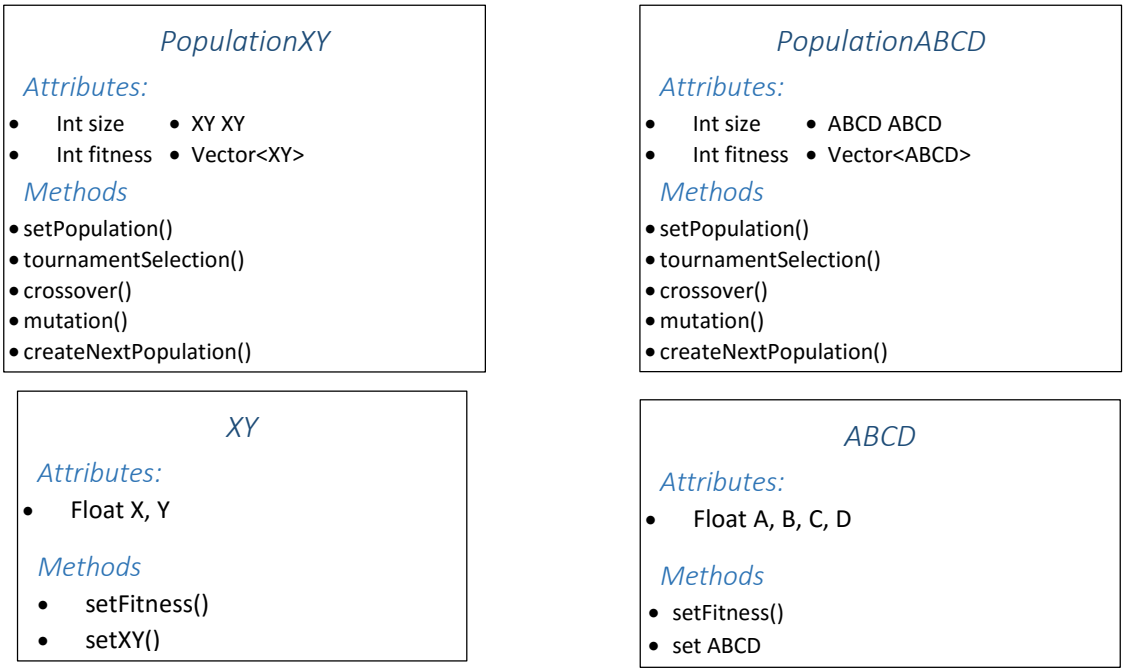


Pseudo-Code:

1. ınitialize one of the types of population with their respective parameters. Each individual is an object of a type.
2. Calculate the fitness of each individual of the population.
3. Perform tournament selection by selecting the fittest out of 3 random individuals in the population. The winners of the tournament remain in the population.
4. Two random parents are selected from the population to generate 2 offspring.
5. Each offspring is mutated.
6. The offspring fitness is set.
7. Steps 4 to 6 are repeated until there are 10 times more offspring than parents.
8. The population is sorted and the fittest form the next population
9. Steps 3 to 8 are repeated until the desired fitness is obtained or maximum of generations is reached.

Design 2 - Composition

Class Diagram:



Pseudo-Code:

1. ınitialize one of the types of population with their respective parameters. Each individual is an object of a type.
2. Calculate the fitness of each individual of the population.
3. Perform tournament selection by selecting the fittest out of 3 random individuals in the population. The winners of the tournament remain in the population.
4. Two random parents are selected from the population to generate 2 offspring.
5. Each offspring is mutated.
6. The offspring fitness is set.
7. Steps 4 to 6 are repeated until there are 10 times more offspring than parents.
8. The population is sorted and the fittest form the next population
9. Steps 3 to 8 are repeated until the desired fitness is obtained or maximum of generations is reached.

Comparison of Designs

	Templates	Composition	Inheritance
Advantages	Easy to implement different population	No use of pointer pointing to a type	Allows customization via virtual functions
Disadvantages	Less flexibility like does not allow different types of individuals within a population	Multiple redefinitions of the same methods	Requires a pointer pointing to a type, harder to manage memory

Comparing the different ways of implementing the algorithm, inheritance and templates provided a simpler style than composition. Both require redefinitions of methods specific to a type. However, inheritance proves to be the most coherent and simplest way of implementing the design. For main, the proposed algorithm was implemented in the final design as instructed, however, it would have been simpler to crossover, mutate and set the offspring fitness at the same time.

Design Description

Pseudo-Code of the main algorithm

1. Initialize one of the types of population with their respective parameters. Each individual is a pointer to a type.
2. Calculate the fitness of each individual of the population.
3. Perform tournament selection by selecting the fittest out of 3 random individuals in the population. The winners of the tournament remain in the population.
4. Two random parents are selected from the population to generate 2 offspring.
5. Each offspring is mutated.
6. The offspring fitness is set.
7. Steps 4 to 6 are repeated until there are 10 times more offspring than parents.
8. The population is sorted and the fittest form the next population.
9. Steps 3 to 8 are repeated until the desired fitness is obtained or the maximum number of generations is reached

Description of methods

- *static bool fittestCriteria(Individual*, Individual*)* : Takes in two Individual pointers and returns true if the second pointer has a lower fitness than the first.

- *void populationSort()* : Using the previous method, sorts the population in ascending fitness.

- *float getFittest()*: Returns the fitness of the fittest individual

- *void getMostFit()*: Prints the fitness and parameters of the fittest individual

- *void tournamentSelection(int):* Takes in one parameter to determine the parent population size. Three different random indexes are generated. The fittest individual in that selection is put into a temporary vector of Individual pointers. This process is repeated until the temp is contains the

desired number of parents. The parents are push backed into the population and the original population is erased from the population vector.

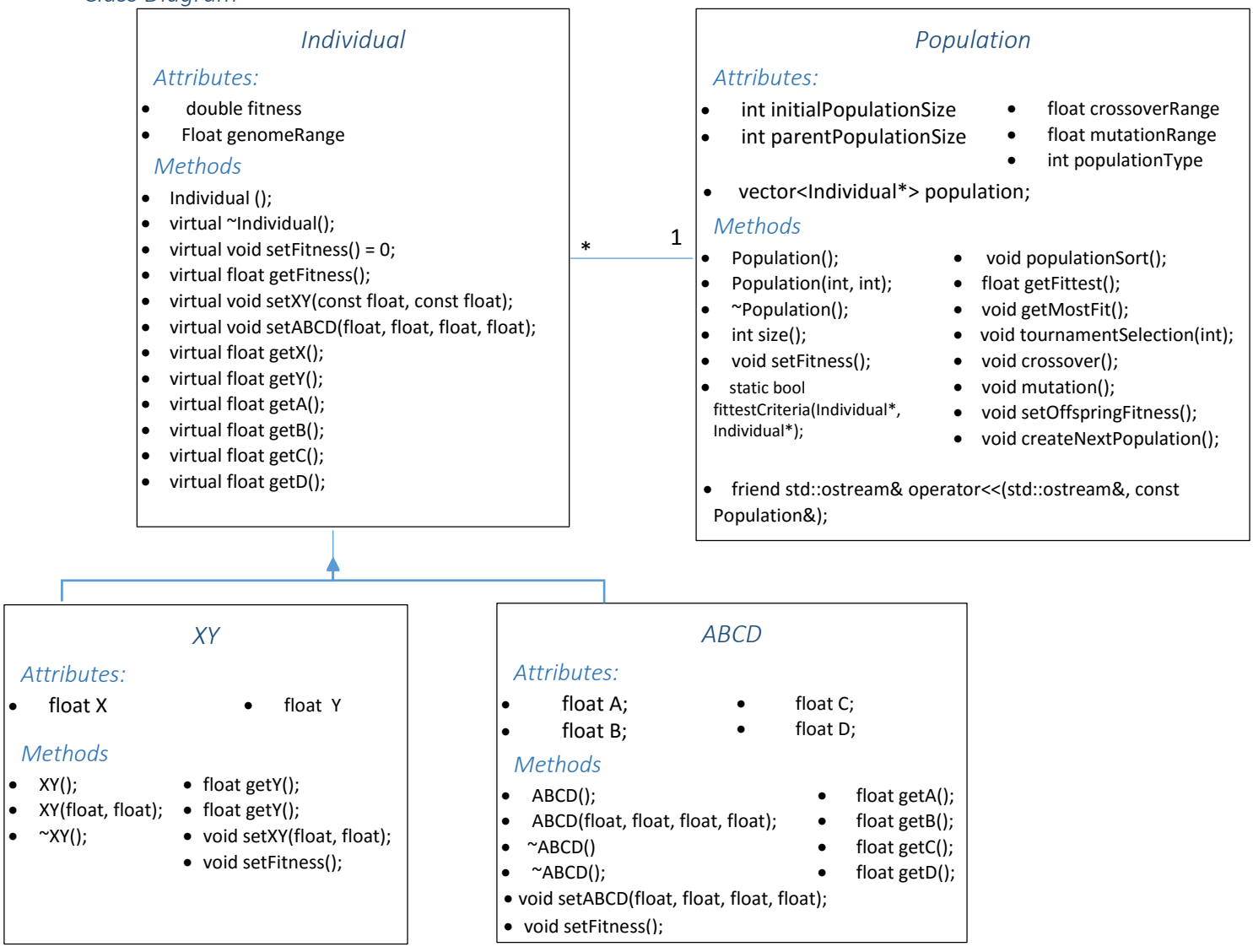
- *void crossover():* Two different and random indexes are generated to represent two parents. Parameters for the individual is determined by taking a random percentage of the parameters of one parent and the remaining from the other parent. The ratio is also determined randomly but within a crossover range. A new offspring is created and pushed back into the population and the process is repeated a second time.

- *void mutation():* A different random float number from a mutation range is added to each parameter of the individual.

- *void createNextPopulation():* Deletes the least fit individuals found in later indices of the vector.

- *friend std::ostream& operator<<(std::ostream&, const Population&):* Overloads the ouput operator in order to print each individual in the population in terms of fitness and parameters.

Class Diagram



Testing Results

Testing the XY individual

The ideal individual has values X=0 and Y=0. Setting the individual to these values returns the ideal fitness of 0 as desired.

Testing the ABCD individual

The ideal individual has values A=1, B=4, C=-20, D=-30. Setting the individual to these values returns the ideal fitness of 0 as desired.

Testing the Population methods

int populationSize(): After creating a population of a set size, the method returned the inputted size.

void populationSort(): The program successfully sorted the individuals of the population according to their fitness in ascending order.

float getFittest(): After sorting the individuals, the method returned the fitness of the fittest individual.

*void getMostFit():*After sorting the individuals, the method printed the fittest individual.

void tournamentSelection(int): The method only mainted the parents in the population. The same individual can be picked more than once.

void crossover(): The method created two new offspring s added at the back of the population.

*void mutation():*The method takes the last two individuals pushed into the population and adds noise.

void setOffspringFitness(): The last individuals pushed into the population had their default fitness of -1 set to their actual fitness..

void createNextPopulation(): After sorting, the fittest of the population were kept returning the population back to its original size.

Testing the algorithm

The only input the program receives one input and it is to decide the type of population to create '1' for XY and '2' for ABCD. The program is able to handle invalid inputs such as letters, repeated characters or numbers other than 1 or 2. Depending on the population, the results differ according to different settings such as the mutation range. The following results were found to be give the results: looking for a fitness less than 0.001 after 35 000 generations.

For the XY-type, a mutation range of 1 gave an initial best fitness starts at around 20 and quickly decreases to less than 1. Often the runs will achieve the desired fitness in less than 20 000 iterations:

```
Gen 19051 | Fittest | Fitness:0.000654088 | X:-4.87566e-005 | Y:0.000225544
Successful!
The fittest individual in the last population has a fitness of Fitness:0.000654088 | X:-4.87566e-005 | Y:0.000225544
```

As for the ABCD-type, a genome range of 0.25 gave a fitness that starts off in the millions and slowly decreases as desired. After 30 000 generations, the ABCD-type population is able to obtain the desired fitness after over 25 000 generations:

Overall, the program is able to generate the right output to valid inputs. The program also does not crash in response to valid inputs nor at the end of its run. Furthermore, if the user were to input a type of population that was not valid, the program is able to respond without failure. Finally, the ABCD-type population proved to be the most successful out of both types of populations, as observed by the slow rate at which the population tries to obtain the ideal.