Pierre Tran (40007601) & Niroigen Suntharam (27890540)

Problem definition:
-The problem is based upon genetic algorithm; it is asked to simulate/mimic the process of natural selection in a C++ code.

There is initially a **population** composed of ***S*** individuals, where each individual has a set of characteristics that determines their ***fitness***. According to this program, the concept of fitness can be described in two ways: by the langermann`s function or by mean square error(MSE).

Langermann's function: The langermann function is implemented in the C++ code. It is a function of two variables, where its global minima is of interest. In this project, each individual is attributed two characteristics, which correspond to the two independent variables of the function. The fitness,in this application, of an individual is considered to be the output of the function after inputting their characteristics into it. According to "infinity77.net", the global minima of the function in the z-axis is -5.1621259 using the variables x as 2.00299219 and y as 1.006096. The individual whose characteristics match the position of the global minima is considered to be the fittest. Therefore, the lower the output is, the fitter that individual will be.

MSE: The Mean Squared Error is a method to determine how precise a function is compared to what it should be. In Figure 1, the data which are shaded in blue are what the predicted function produces. The data in yellow is what the actual function should produce. The second application of this project is to determine a third degree polynomial where 20 points of the function are entered. Each individual will contain the parameters for polynomial (**a**, **b**, **c**, **d** where y = **a**x^3+**b**x^2+**c**x+**d**). After each generation, two possibilities can occur. The best individual can be a new offspring, or the best individual will be the same one as the one from the previous generation.

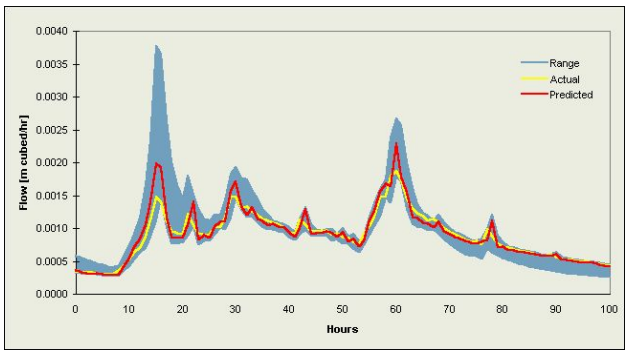
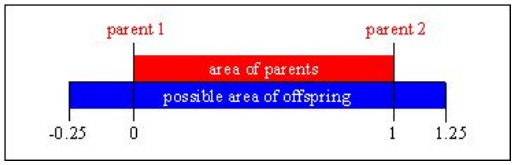


Figure 1 - Example of Mean Squared Error

Tournament Process: During the tournament process, 3 individuals from the population at random will be picked the **fittest** individual will be thrown into the parent pool. In other words, if the code runs 50 tournaments, 50 individuals are sent into the parent pool. The parents, in the parent pool, mate with each other to form **10 S** (where **S** is the initial population size) number of offsprings.

Intermediate recombination: For every offspring created each offspring's parameters will be altered using intermediate recombination. Each new characteristic of each new offspring is a combination of each corresponding characteristic of each parent. The ''crossover'' can be visualised as in the Figure 2 and described more accurately by the equation (1).



.

Figure 2 - Example of intermediate recombination

$$Var_i^O = Var_i^{P_1} \cdot a_i + Var_i^{P_2} \cdot (1 - a_i)\, i \in (1,2,\ldots,Nvar),$$
$$a_i \in [-d, 1+d] \text{ uniform at random, } d=0.25, a_i \text{ for each } i \text{ new}$$

Equation (1)

Real valued mutation: Once the offspring goes through the intermediate recombination, mutation will occur. Each characteristic of the offspring can either improve or worsen with equal probability, where ''small step-sizes'' of mutation are more likely to happen than ''big step-sizes ''. The real valued mutation can be visualized as the following in figure 2 and can be described by the equation (2).

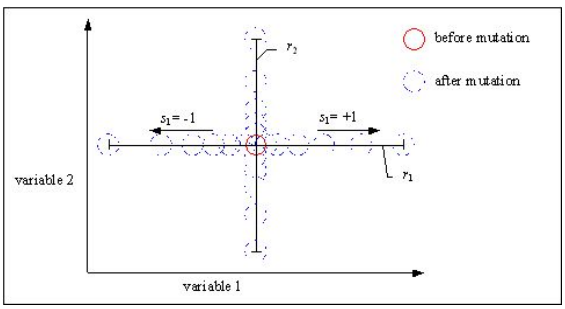


Figure 3 - Real Valued Mutation

$$Var_i^{Mut} = Var_i + s_i \cdot r_i \cdot a_i \, i \in (1,2,\ldots,n) \text{ uniform at random,}$$
$$s_i \in \{-1,+1\} \text{ uniform at random}$$
$$r_i = r \cdot domain_i,\ r : \text{mutation range (standard : 10\%)},$$
$$a_i = 2^{-u \cdot k},\ u \in [0,1] \text{ uniform at random, } k : \text{mutation precision},$$

Equation (2)

A joint pool is created, which contains **10 S offsprings** and **the number of tournaments which correspond to the number of parents**. It is then sorted in order of the fittest individual to the least fittest one. Only the fittest **S** number of individual remain; this will be the new population (i.e the new generation). The cases for both application is that individuals will be evaluated based on how close they are to the global minima. Since the program does not know what the value of the global minima for either of the application, a dummy will temporarily store the fittest individual of the current generation and will replace the latter by the fittest of the next generation and so on. This process will go on until the criteria has been reached .Since the program could potentially keep running to find the fittest individual we have written an algorithm where the program will stop after a certain amount of time. The algorithm is that if the best individual for next hundred generations is the same, it will stop the program and print out the parameters of the individual.

Alternatives and Recommendations:

1) Tool to stop the program if it is running too long :
   Our code takes a long time before its execution is ended and its final results output. Generally, the results are stuck at a certain point and the code outputs the same value after a certain number of generations. To fix this problem an if statement is

used. The parameter counter simply counts the number of times the same global minima/ lowest MSE is being shown the same time between generation to generation. If the number of times is equal to 200 generations then the answer will print out the answer according to the fittest individual. If for any chance the best individual had a better value, then the counter will be reset to zero.

2) Tool to combine both applications:

Our code initially were made to work independently, therefore putting them together would be difficult. The method to fix this problem is by creating the evaluate functions to two separate ones. For the first application the **evaluateapp1()** where it will be the output of the Langermann function and the second application will be the **evaluateapp2()**, where it will output the MSE value.

Pseudo-code:

If (past_fitness.setprecision(6) == fitness.setprecision(6))
// the values of the previous and current fitness have been stored. The two values are compared up to a precision of six numbers after the decimal.
{
Count++; //counter that counts the times when the two variables have the same value
If (count ==50) // after fifty times of having the same values, the statement ends the loop
{end the "while" loop;}
previous_fitness = fitness; //getting rid of the previous value of fitness
}
else
{previous_fitness = fitness;} //getting rid of the previous value of fitness

randomizer()
{
Randomize each of the elements in the array itself between a number between 0 and 10
}

evaluateapp1()
{
Find the value of the Langermann function and then store its value in the fitness of the individual
}

evaluateapp2()
{
Find the MSE between the function and the best individual and store that as its fitness
}

termination_criteria()
{

If the first individual is equal to the best individual, and the counter is equal to 100 then return true

If the first individual is not equal to the best individual then set the counter to zero and return false

Else add the counter by one and return false

}

Basic Class Diagram:

Figure 4

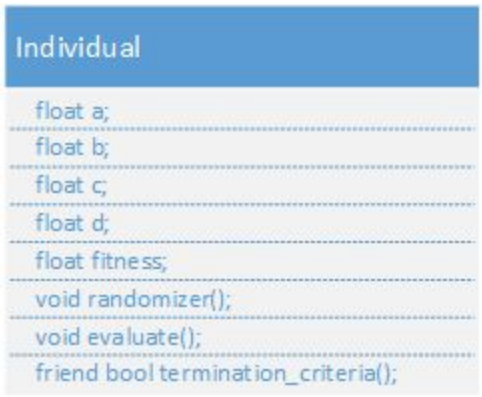


Figure 5

Pseudo-Code of the main algorithm:

A population will be initialized using the **randomizer()**, where each values in the array of each individual will be stored. Then the individual's fitness will be evaluated using the langermann's function and it will be stored in the data member **fitness**. Tournaments After there are 10S individuals in the offspring pool, each of the individuals including the parents will be sorted. Once sorted the **termination_criteria(vector <individual>:: iterator)** will be called where the parameter will be the first element of the new population since it will be the most fittest offspring.

A population will be initialized using the **randomizer()**, where each values of the randomizer will be stored in the data members a,b,c,d.Then the individual's fitness will be evaluated and it will be stored in the data member **fitness**. After there are 10S individuals in the offspring pool, each of the individuals including the parents will be sorted. Once sorted the **termination_criteria()** will be called and it will access the first element in the population pool, since the first element will have the fittest individual.

	Method 1	Method 2
Pros	• Data member array can be used for both application, since one will need only two parameters and the other would need four	• Data member a, b, c, d can be used whenever needed for each of the two applications

	• The termination_criteria uses an iterator so the parameter to be passed in can be the position of an iterator	
Cons	• The access of a particular element in the array of an individual can be confusing	• The termination_criteria() must access the population

The reason as to why the first method is chosen over the second is because the first method can be used universally. In other words, it can be used for the first application where the float array can use two parameters and for the second application the array can use four parameters.

Design Description:

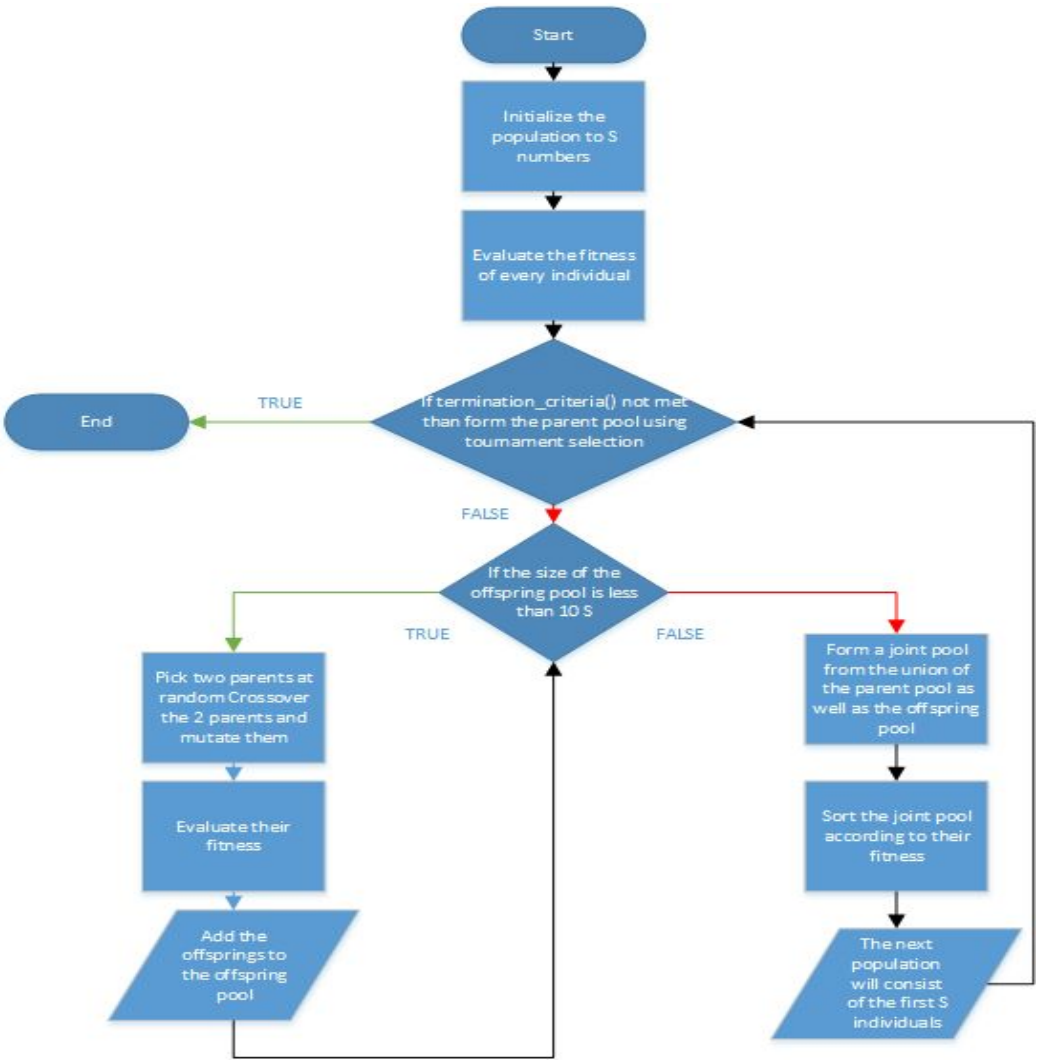


Figure 6

This figure shows the class diagram of the class Individual where the non-trivial methods being used are the randomizer, evaluate and the termination_criteria(). Since there is only on class in this program, it will mean that it will do all the interactions in the main program. The class individual is what holds all the information about the individual itself.

Design Description:

Black Box Testing: According to our program the only input that will be inputted by the user is that they need to enter a “1” or a “2” for the first application (FInding the global minima of the Langermann function) or for the second application (Finding the curve of a 3rd degree polynomial function). According to the input of the user, either of the two applications will run meaning that the output will be different for both cases.

1st application

```
Best individual is -5.16213
Generation 292
Best individual is -5.16213
Generation 293
Best individual is -5.16213
Generation 294
Best individual is -5.16213
Generation 295
Best individual is -5.16213
Generation 296
Best individual is -5.16213
Generation 297
Best individual is -5.16213
Generation 298
Best individual is -5.16213
Generation 299
Best individual is -5.16213
Generation 300
Best individual is -5.16213
The best individual has a fitness of 0
The x and y components are
2.00306
1.0062091
Program ended with exit code: 0
```

All Output

```
Generation 290
Best individual is -5.16213
Generation 291
Best individual is -5.16213
Generation 292
Best individual is -5.16213
Generation 293
Best individual is -5.16213
Generation 294
Best individual is -5.16213
Generation 295
Best individual is -5.16213
Generation 296
Best individual is -5.16213
Generation 297
Best individual is -5.16213
Generation 298
Best individual is -5.16213
Generation 299
Best individual is -5.16213
Generation 300
Best individual is -5.16213
The best individual has a fitness of
The x and y components are
2.00293
1.0061392
Program ended with exit code: 0
```

All Output

Here one can notice that after the 300th generation for both of the runs, the x and y values that outputs the lowest z value is when x = 2.00306 and y = 1.0062091 for the first run and for the second run, x = 2.0093 and y = 1.006192. According to “infinity77.net” the x and values that have the global minimum is at x = 2.00299219 and when y = 1.006096 which are off by a thousanth of a decimal and a millionth.

2nd application

```
Generation 206
Fittest individual 0.00116786
Generation 207
Fittest individual 0.00116786
Generation 208
Fittest individual 0.00116786
Generation 209
Fittest individual 0.00116786
Generation 210
Fittest individual 0.00116786
Generation 211
Fittest individual 0.00116786
Generation 212
Fittest individual 0.00116786
Generation 213
Fittest individual 0.00116786
Generation 214
Fittest individual 0.00116786
Generation 215
Fittest individual 0.00116786
Generation 216
Fittest individual 0.00116786
Generation 217
Fittest individual 0.00116786
The fittest individual is
0.00116786
a 3.00021
b 4.00032
c 1.98637
d 4.9809
Therefore the function is y = 3.00021x^3 + 4.00032x^2 + 1.98637x + 4.9809
Program ended with exit code: 0
```

Here one can notice that after the generation, the a, b, c, d values that outputs the best polynomial are when a = 3.00021, b=4.00032 , c=1.98637, d=4.9809. In the beginning of the code, the array “p” contained 20 sets of points for the function y = 3x^3+4x^2+2x+5. The parameters are roughly off by a thousandth.

```
Please enter a 1 or a 2 for the application number 1 or 2 respectively
3
You have not entered either a 1 or 2, please enter again
Please enter a 1 or a 2 for the application number 1 or 2 respectively
```

Here is a case when the user enters an invalid input and an error will be printed out where the user will have to enter again either a 1 or a 2 for the application 1 or 2 to happen.