# 

Optimization Algorithms

Benjamin Ell-Jones - 19016317

# Introduction

In this paper I intend to research and compare Artificial bee colony and genetic algorithms. Also compare several different genetic algorithm operators and to optimize the Rosenbrock and Ackley fitness functions.

# Background Research

## Introduction to ABCs

Another algorithm for solving optimization problems is the artificial bee colony. The algorithm works similar to a bee colony as you would expect, it has 3 types of bees, employed bees, onlookers and scouts (Karaboga, Basturk,2007). Within this algorithm possible solutions are not represented as individuals with genes as is the case with genetic algorithms and a fitness. Food source represents possible solutions in an optimization problem and the nectar represents the fitness (Karaboga, Basturk,2007). Using this information, it is clear that the representation of the problem is similar in concept both genetic algorithms and bee algorithms, have a problem solution they represent in their respective ways then they have a way of representing the fitness of said problem solution.

## ABC Structure

Breaking down the steps, the first step of the process is the initialization phase. This is where a randomly distributed initial population is created (Karaboga, Basturk, 2007).

The next phase is the employed bee phase, where in employed bee search for new solutions within the neighborhood of the food sources their memory, when they find one,

they evaluate the fitness of the food source (Karaboga, Gorkemli, Ozturk, 2014). The next step once a new food source is produced and greedy selection has been applied between it and its parent, employed bee share the information about the food source with onlooker bees in the hive that are currently dancing in the dancing area (Karaboga, Gorkemli, Ozturk, 2014).

The next phase is the onlooker phase. In this phase food sources are probabilistically chosen by onlooker bees, based on the information that they received from the employed bees (Karaboga, Gorkemli, Ozturk, 2014). In the case of this article, (Karaboga, Gorkemli, Ozturk, 2014) it is stated that a fitness-based selection algorithm like roulette wheel selection can be employed. Fitness based selection is also used in genetic algorithms they are just used in different ways, to select individuals for crossover as opposed to, using selected found food source to apply greedy selection between it and a neighborhood food source after its fitness has been determined (Karaboga, Gorkemli, Ozturk, 2014).

The next phase of the ABC algorithm is the scout phase. Any solutions that can’t be improved through a predetermined number of trials there employed bees become scouts and those solutions abandoned (Karaboga, Gorkemli, Ozturk, 2014). Scout will then start to find new solutions at random (Karaboga, Gorkemli, Ozturk, 2014).

## ABC parameters

Comparing parameter options for both Genetic algorithms, basic genetic algorithms can have multiple parameters, in the case of this research paper, including Generation, population, mutation rate and crossover rate compared with ABCs, in the case of this study “A comparative study of Artificial Bee Colony algorithm” the basic ABC used in this study employs only one control parameter, which is called limit (Karaboga, Akay, 2009). The limit described in the study is, when the limit is exceeded the food source will no longer be exploited and will be considered abandoned (Karaboga, Akay, 2009). However other basic parameters in the study are used such as, population number and maximum evaluation number (Karaboga, Akay, 2009).

## Comparison of Genetic algorithms and ABCs

Both genetic algorithms and artificial bee colonies can be used for global optimization. Although both genetic algorithms and ABC seem to be different on the surface, they do share some similarities in regard to the search process, for example the employed bee phase is comparable to the selection stage in a genetic algorithm, as in the employed bee phase solutions are selected and their information saved based on their fitness. Another example it could be argued that the onlooker phase is similar to genetic algorithms crossover. This is because, both phases are used to obtain new solutions to be operated on.

# Experimentation

My program’s structure is as follows, first I generate an initial population of individuals. Then test that populations initial fitness. Deep copying is used in all places where the population is being copied from one array to another. After initial fitness has beings calculated the selection process begins. After this step has been completed crossover then mutation is applied to the population. The final step is to calculate the minimum fitness of the generation and the mean fitness of the generation then the elitism function is called, and the best individual is selected and replaced with the worse individual for the next generation.

Several parameters are included at the top of the program they are as follows, N, Pop, MUTRATE, MUTSTEP, GEN, MIN and MAX. N is equal to the number of chromosomes in each individual. Pop is in reference to the max number of populations in each Generation. Gen or generation is how many times the above GA algorithm will run I.E how many times, selection, crossover, and mutation will happen. MIN and MAX are in reference to the minimum and maximum values and given chromosome can be in an individual. MUTRATE or mutation rate is how often an individual solution will be mutated. When dealing with real numbers as opposed to 1 and 0s you can’t simply flip a 1 to a 0 and vice versa to mutate them. Therefore, MUTSTEP or mutation step is required to determine what will be taken away or added to an individual chromosome to mutate it.

# Optimising Rosenbrock and Ackley fitness functions

The algorithm used for the mutation sweep for both Rosenbrock and Ackley fitness functions and other such tests, includes, one point crossover, tournament selection and uniform mutation. This is the case unless specified otherwise.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Rosenbrock fitness function | | | | | | |
|  | 10 | 20 | 30 | 40 | 50 | 60 |
| 0.01 | 6.6567779165177345 | 10.334487287471585 | 21.596542598085538 | 24.88742248527368 | 52.86235892915361 | 76.27828856181367 |
| 0.02 | 5.098354862184059 | 11.804474053218483 | 14.085477570082805 | 27.26965517794447 | 32.86292388650723 | 37.63483713803109 |
| 0.03 | 14.158465657304891 | 26.593004980532772 | 48.38988661082048 | 90.90760357089816 | 214.57228050102563 | 412.2038131474645 |
| 0.04 | 32.317574383798394 | 116.62564981102574 | 556.8254289757666 | 1271.4188608089955 | 4409.650837513189 | 15922.07132053572 |
| 0.05 | 112.95206411932922 | 1223.6938321477007 | 4874.225707327098 | 19315.478592853528 | 23738.321853579124 | 63726.964730178006 |

The table above shows a collection of data produced by a mutation rate and mutstep sweep to find the best possible combination of mutation rate and mutstep values. Each fitness value is produced by a test set contains a set of 10 individual test where the same test with the same parameters is run 10 times, this is to account for the inconsistent results obtained when these tests are run due to the fact that mutation rate and population generation is random the results can vary quite a lot.

The best fitness values for each run of the system, from all 10 tests for a set is added to one array and then averaged. This then gives one average fitness value for one pair of parameters for example 0.01 mutrate and 10 mutstep will give the average, 5.098354862184059

The other parameters are as follows, population: 100, generations: 500, and the min and max values for each individual chromosomes in an individual are min: -100 and max: 100.

As can be seen from the mutation sweep for the Rosenbrock fitness function the best fitness is 5.098354862184059 Meaning a mutation rate of 0.02 and a mutstep of 10 is a good starting point to tune the mutation rate and step further.

### 100 Pop, Mutstep 0.025 and 10 Mutstep

Graphical user interface, application

Description automatically generated

First testing 0.025 and 10 seemed like the most logical option to try first this is done with 100 Population and over 500 generations. As can be seen from both the graph above and the table below, with the increase of mutrate the fitness increased compared to the last run there by meaning it is a good idea to try a much lower mutrate value.

|  |  |
| --- | --- |
| Average | 6.675185664522118 |

A picture containing graphical user interface

Description automatically generated

As can be seen from this graph it appears to show premature convergence, however zooming in to the end of the graph shows a different story.

A picture containing graphical user interface

Description automatically generated

The graph shows however that this is not the case both the average and the best lines are running parallel to each other and separated not running on top of each other this indicates that the algorithm is still searching and has not converged on one solution.

### 100 Pop, Mutstep 0.015 and 10 Mutstep

Graphical user interface, application, Word

Description automatically generated

Decreasing the mutrate to 0.015 as the mutrate between 0.01 and 0.3 gave the best results we get a slight decrease in fitness.

|  |  |
| --- | --- |
| Average | 5.2300703115953135 |

A picture containing chart

Description automatically generated

As can be seen from this graph compared to the last, both lines have gotten closer together rather than further apart meaning we are on the right track in terms of mutrate and mutstep combination, as the opposite would be true if it was a poor combination of the two.

Text

Description automatically generated with medium confidence

The end of the graph lines being separated and running parallel still indicate the search process is continuing.

### 100 Pop, Mutstep 0.015 and 5 Mutstep

Graphical user interface, text, application

Description automatically generated

Decreasing the mutstep further produces less dramatic changes to genes resulting in even better gene fitness as can be seen bellow.

|  |  |
| --- | --- |
| Average | 2.395774327981293 |

A picture containing chart

Description automatically generated

Again, the lines show convergence however this is not the case as can be seen bellow. Further analysis of the graph

A picture containing application

Description automatically generated

As can be seen again from this graph although it appears to be converging, further analysis concludes that it is not, by zooming in to the end of the graph the lines are running parallel indicating it’s still searching.

### 100 Pop, Mutstep 0.015 and 2 Mutstep Application Description automatically generated with low confidence

After analysing both the graph and individual best fitness of each of the test runs the reduction of mutstep further produces to fewer changes to genes, resulting in less new genetic information being added to remove from genes resulting in a worse overall fitness as the GA is getting stuck in the local optimum rather than the global optimum.

|  |  |
| --- | --- |
| Average | 1330.460778658239 |

A picture containing graphical user interface

Description automatically generated

Both these graphs (Above and below) further clarify the fact decreasing the mutstep further has caused the fitness to increase and the search to be affected dramatically as the lines in the above graph are taking much longer to converge. The previous graph starts to converge around gen 25 whereas this run is starting to converge around 30 gens.

Chart

Description automatically generated with medium confidence

To achieve the optimum, it may require reverting to the previous best mutrate and mutesteps and increasing the population count as decreasing or increasing Mutrate and mutstep at this point does not yield any better results.

#### 200 pop, Mutstep 0.015 and 5 Mutstep

Graphical user interface, text, application

Description automatically generated

After reverting to the best mutrate and mutstep of previous tests and increasing the population it is clear now the fitness is decreasing to much lower values than previously seen in testing. Although increasing the population does affect the efficiency of the genetic algorithm meaning it will take longer to give an output.

|  |  |
| --- | --- |
| Average | 0.8083008653085122 |

A picture containing chart

Description automatically generated

Graphical user interface, text, application

Description automatically generated

Further increase to the population is the best course of action after analysing results as this should lead to a further reduction in fitness.

#### 300 Pop, Mutstep 0.015 and 5 Mutstep

Graphical user interface, application

Description automatically generated

After further increases to the population count the resulting fitness values have again decreased further. However, this has yet again made the GA more in efficient and has produced even slower run times for the system.

|  |  |
| --- | --- |
| Average | 0.36067163746478637 |

A picture containing diagram

Description automatically generated

# Graphical user interface Description automatically generated with low confidence

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Ackley fitness function | | | | | | |
|  | 10 | 20 | 30 | 40 | 50 | 60 |
| 0.01 | -21.900790251936318 | -21.18075517100894 | -20.450229537784217 | -19.797683405255146 | -19.54863429295534 | -19.316151031663168 |
| 0.02 | -22.24231355118912 | -21.63739055851879 | -20.83941549507453 | -20.484501109609862 | -19.98367085634009 | -19.575619054343935 |
| 0.03 | -22.20236553214861 | -21.63739055851879 | -20.364481890780386 | -19.965621849568883 | -19.208946615810508 | -19.403557403776418 |
| 0.04 | -21.7482207153077 | -20.398724832480294 | -19.469993370918665 | -19.246700000565465 | -18.45843400368334 | -18.63499842811235 |
| 0.05 | -20.87711000549546 | -19.655507843249897 | -18.719745294837402 | -18.290709081948624 | -17.626104175822988 | -16.918413741761803 |

Using the same testing method as the RosenBrock function we can see that -22.24231355118912 is the smallest fitness in this mutation sweep. Therefore, the best starting point for experimentation is 0.02, 10.

#### 100 Pop, Mutstep 0.025 and 10 Mutstep

Chart, line chart

Description automatically generated

First testing a higher mutrate than the given best combination in the mutation sweep, yield as expected worse results but this is done to get the upper bounds for testing to see how high we can increase the mutrate before drastic changes occur as the best results were found between a mutrate of 0.01 and 0.03, it is clear from this test that we should not increase the mutrate any higher than 0.025. The lines in the graph spreading apart would indicate that there is too much mutation going on this will be reduced in the next test.

|  |  |
| --- | --- |
| Average | -22.444862330907235 |

#### 100 Pop, Mutstep 0.015 and 10 Mutstep

Graphical user interface

Description automatically generated

After decreasing both mutrate and mutstep to 0.015 and 10 it is clear that the lines have come further together and starting from this point and working our way down in terms of mutstep is the best option, as in this case as a slightly lower mutrate and mutstep than the best combination in our mutation sweep has given better results.

|  |  |
| --- | --- |
| Average Fitness | -22.477837934574517 |

#### 100 Pop, Mutstep 0.015 and 5 Mutstep

Graphical user interface, application

Description automatically generated

Further testing with a smaller mutstep shows the lines coming closer together rather than further apart, however they still are not quite parallel indicating that too much mutation is still proving to be an issue a further reduction is still required.

|  |  |
| --- | --- |
| Average | -22.60814577345934 |

#### 100 Pop, Mutstep 0.015 and 2 Mutstep

A picture containing graphical user interface

Description automatically generated

After further reducing the mutstep the lines are running almost parallel indicating we are getting closer the right amount of mutation required to find the optimum solution. Further reduction of mutrate however would likely lead to less optimal results as genes would mutate rarely and with little change when they do mutate and further reduction of mutstep would cause similar issues and thereby increasing the average fitness. The most logical next step would be to increase the population count.

|  |  |
| --- | --- |
| Average | -22.67215876935784 |

#### 200 pop, Mutstep 0.015 and 1 Mutstep

Chart

Description automatically generated

After increasing the population count the lines continued to coverage and are now running parallel indicating it is very close to the optimal solution, a further increase to the population should yield even better results.

|  |  |
| --- | --- |
| Average | -22.702901846403094 |

#### 300 pop, Mutstep 0.015 and 1 Mutstep

Chart, line chart

Description automatically generated

Further increase to the population count has yielded better results but only slightly. We can there for conclude that this is indeed the optimum or extremely close.

|  |  |
| --- | --- |
| Average | -22.7079392982269 |

## Alternative Operators

All parameters chosen for testing these operators were the best parameters found in prior testing for the Rosenbrock and Ackley function respectively. Unless specified otherwise.

### Crossover – Multipoint

#### Rosen Brock fitness:

A picture containing application

Description automatically generated

|  |  |
| --- | --- |
| Average | 0.25844865889669955 |

A picture containing graphical user interface

Description automatically generated  
A picture containing application

Description automatically generated  
Ackley fitness:

Chart

Description automatically generated with medium confidence

|  |  |
| --- | --- |
| Average | -22.704808181649533 |

#### Analysis:

Comparing multipoint crossover to single point crossover we can see that it provides slightly better results in both tests with Rosenbrock and Ackley. Due to the low gene count however, crossover does not have as greater of an effect as mutation would have for example on the search process. If we were to increase this, a greater change would be observed.

### Selection – Roulette wheel

#### Rosen Brock fitness:

Graphical user interface, application

Description automatically generated

|  |  |
| --- | --- |
| Average | 3.6298063060367944 |

A picture containing chart

Description automatically generated  
Chart

Description automatically generated

Ackley Fitness:

Chart, line chart

Description automatically generated

|  |  |
| --- | --- |
| Average | -3.1221355335396783 |

#### Analysis:

From analysing the results, it is clear that roulette wheel selection has performed worse in this case, this is because this type of selection performs worse with higher amounts of chromosomes within genes, furthermore if all solutions to be selected from have almost identical fitness values, then the chances of each solution being selected are very similar resulting in slightly worse solutions being selected more often.

### Mutation – Gaussian Mutation Rosenbrock fitness

Graphical user interface, text, application

Description automatically generated

|  |  |
| --- | --- |
| Average | 4.532769917491803 |

Chart, line chart

Description automatically generated  
A picture containing chart

Description automatically generated

Ackley Fitness:

Graphical user interface

Description automatically generated with low confidence

|  |  |
| --- | --- |
| Average | -22.718281828458807 |

Chart, line chart

Description automatically generated  
Chart

Description automatically generated with medium confidence

#### Analysis:

The mean and standard deviation are parameters that must be adjusted like MUTSTEP, so the best mutrate values that were obtained through testing are not going to provide the best results necessarily. In the instance of the first test with gaussian mutation on the rosenbrock function however, it appears that it has done significantly worse, it is likely the deviation and will need to be increased to see better results with that function as currently the values that will be in the gaussian mutation value will be too small to see any effect when mutating, alternatively the mutation rate could be increased to induce mutation more often.

The Ackley function however is a different story and has been significantly improved with identical parameters to the parameters found in previous tests. This is likely because the Ackley function uses a smaller min and max values meaning less will have to be added or taken away from chromosomes to make a difference.

# Comparing ABC and GA performance

These are the results obtained by running Yarpiz (2015) artificial bee colony source code.

## Rosenbrock:

Text, letter

Description automatically generated

Rosenbrock function used in MATLAB code.

Chart

Description automatically generated  
Text

Description automatically generated  
  
Ackley:

#### Text, letter Description automatically generated

Ackley function used in MATLAB code.

# Chart, line chart Description automatically generated Text Description automatically generated with medium confidence

#### Analysis:

Both the ABC algorithm selected, and the genetic algorithm used in testing are using the same population count and epoch amount and number of values in each solution, 300, 500 and 20 respectively.

It is clear from the ABC Rosenbrock results that it performed significantly worse than the genetic algorithm and the opposite is true comparing the results of the Ackley functions.

This implementation of the bee algorithm utilises roulette wheel selection. In the tested genetic algorithm, this was proven to perform poorly compared with other selection operators, the reason why Rosen brock function performance was so poor could be related with this issue. I.E a greater number of chromosomes in genes results in poorer performance.

The better performance of the Ackley function in this case could be attributed to its much smaller search space, making it slightly easier to solve meaning the effects of the selection operator roulette wheels flaws less obvious.

# Conclusion:

In conclusion, from the tests run we can conclude a higher mutrate and mutstep does not provide better results because having too higher of either of these make to dramatic changes on genes resulting on a dramatically increased or decreased fitness. Furthermore, some genetic algorithm operators perform better in some circumstances and worse in others for example roulette wheel selection performing better with fewer chromosomes within genes. If given an opportunity to runs tests a second time, there are a few things I would do differently, including test a wider verity of GA operators and run similar comparisons between ABC and GA but test different ABC algorithms against the GA algorithm to see how they perform.

# References:

Karaboga, Akay, D.K, B.A. (2009) A comparative study of Artificial Bee Colony algorithm, Applied Mathematics and Computation [Online]. Volume 214, Pages 108-132 [Accessed 01 December 2021]

Karaboga, Basturk, D.K, B.B. (2007) A powerful and efficient algorithm for numerical function optimization: artificial bee colony (ABC) algorithm. Journal of Global Optimization[online]. Volume 39, Issue 3. [Accessed 29 November 2021]

Karaboga, Gorkemli, Ozturk, D.K, G.B, C.O. (2014) A comprehensive survey: artificial bee colony (ABC) algorithm and applications. Artificial Intelligence Review [online]. Volume 42, Issue 3. [Accessed 30 November 2021]

Yarpiz (2015) Artificial Bee Colony (ABC) in MATLAB (version 1.0.0.0) [Source Code]. Available from: https://uk.mathworks.com/matlabcentral/fileexchange/52966-artificial-bee-colony-abc-in-matlab [Accessed 4 December 2021]

# Appendix:

import math

import random

import array

import matplotlib.pyplot as plt

import numpy as np

import copy

class Individual:

    def \_\_init\_\_(self, gene,fitness):

        self.gene = gene

        self.fitness = fitness

class GenerationData:

    def \_\_init\_\_(self, best, mean, bestSolutions, MUTRATE):

        self.best = best

        self.mean = mean

        self.bestSolutions = bestSolutions

        self.MUTRATE = MUTRATE

class TestParameters:

    def \_\_init\_\_(self,testNum, N, Pop, MUTRATE, MUTSTEP, GEN, MIN, MAX,SELECROSS,SELESELECT, SELEFIT):

        self.testNum = testNum

        self.N = N

        self.Pop = Pop

        self.MUTRATE = MUTRATE

        self.MUTSTEP = MUTSTEP

        self.GEN = GEN

        self.MIN = MIN

        self.MAX = MAX

        self.SELECROSS = SELECROSS

        self.SELESELECT = SELESELECT

        self.SELEFIT = SELEFIT

def generatePopulationReals(Pop,N,MIN,MAX):

    population = []

    for i in range(0, Pop):

        gene = []

        for j in range(0, N):

            gene.append(random.uniform(MIN,MAX))

        population.append(Individual(gene, 0.0))

    return population

def fitnessFunc(population,Pop,N):

    for i in range(0, Pop):

        fitness = 0

        for x in range(0, N):

            fitness = fitness + population[i].gene[x]

        population[i].fitness = fitness

    return population

def fitnessFunc1(populationVar,Pop,N):

    for i in range(0, Pop):

        fitness = 10 \* N

        for x in range(0, N):

            fitness = fitness + (populationVar[i].gene[x] \*\* 2) - ( 10 \* math.cos((2 \* 3.14) \* populationVar[i].gene[x]) )

        populationVar[i].fitness = fitness

    return populationVar

def fitnessFunc2(population,Pop,N):

    for x in range(0, Pop):

        fitness = 0

        for i in range(0, N - 1):

            step1 = 0

            step2 = 0

            step1 = ((population[x].gene[i + 1] - (population[x].gene[i + 1] \*\* 2) ) \*\* 2)

            step2 = (1 - population[x].gene[i]) \*\* 2

            fitness = fitness + ((100 \* step1) + step2)

        population[x].fitness = fitness

    return population

def fitnessFunc3(population,Pop,N):

    for z in range(0, Pop):

        fitness = 0

        step1 = 0

        step2 = 0

        for i in range(N):

            step1 += population[z].gene[i] \*\* 2

            step2 += math.cos(2 \* math.pi \* population[z].gene[i])

        fitness = -20 \* math.exp(-0.2 \* math.sqrt(step1 / N)) - math.exp(step2 / N)

        population[z].fitness = fitness

    return population

def selectFitnessFunc(num, population,Pop,N):

    if (num == 0):

        return fitnessFunc(population,Pop,N)

    elif (num == 1):

        return fitnessFunc1(population,Pop,N)

    elif (num == 2):

        return fitnessFunc2(population,Pop,N)

    else:

        return fitnessFunc3(population,Pop,N)

def selectionTour(population,Pop):

    parent1 = 0

    parent2 = 0

    offspring = []

    for i in range(0,Pop):

        parent1 = random.randint(0, Pop - 1)

        parent2 = random.randint(0, Pop - 1)

        if (population[parent1].fitness < population[parent2].fitness):

            offspring.append(population[parent1])

        else:

            offspring.append(population[parent2])

    return offspring

def popFitTotal(population,Pop):

    totalFit = 0

    for i in range(0, Pop):

        totalFit = totalFit + (1 / population[i].fitness)

    return totalFit

def rouletteWheel(population,Pop):

    offspring = []

    population\_fitness\_total = popFitTotal(population,Pop)

    for i in range(0, Pop):

        selection\_point = random.uniform(0, population\_fitness\_total)

        running\_total  = 0

        j = 0

        while ( running\_total <= selection\_point ):

            running\_total += 1 / population[j].fitness

            j = j + 1

        offspring.append(population[j-1])

    return offspring

def selection(population,Pop, SELESELECT):

    if(SELESELECT == 0):

        return selectionTour(population, Pop)

    elif(SELESELECT == 1):

        return rouletteWheel(population, Pop)

def crossoverOne(offspring,Pop,N):

    for i in range( 0, Pop, 2 ):

        toff1 = copy.deepcopy(offspring[i])

        toff2 = copy.deepcopy(offspring[i+1])

        temp = copy.deepcopy(offspring[i])

        crosspoint = random.randint(1,N)

        for j in range (crosspoint, N):

            toff1.gene[j] = toff2.gene[j]

            toff2.gene[j] = temp.gene[j]

        offspring[i] = copy.deepcopy(toff1)

        offspring[i+1] = copy.deepcopy(toff2)

    return offspring

def twoPointCrossover(offspring,Pop,N):

    for i in range(0,Pop,2):

        offs1 = copy.deepcopy(offspring[i])

        offs2 = copy.deepcopy(offspring[i + 1])

        temp = copy.deepcopy(offspring[i])

        crossover1 = random.randint(1,N / 2)

        crossover2 = random.randint(crossover1, N)

        for j in range(crossover1, crossover2):

            offs1.gene[j] = copy.deepcopy(offs2.gene[j])

            offs2.gene[j] = copy.deepcopy(temp.gene[j])

        offspring[i] = copy.deepcopy(offs1)

        offspring[i + 1] = copy.deepcopy(offs2)

    return offspring

def crossover(offspring,Pop,N, SELECROSS):

    if (SELECROSS == 0):

        return crossoverOne(offspring,Pop,N)

    elif (SELECROSS == 1):

        return twoPointCrossover(offspring,Pop,N)

def mutationReals(offspring,Pop,N,MUTRATE,MUTSTEP,MAX,MIN):

    for i in range(0, Pop):

        for j in range(0, N):

            MUTPROB = random.uniform(0.0, 1.0)

            if (MUTPROB < MUTRATE):

                alter = random.uniform(0.0, MUTSTEP)

                coinFlip = random.randint(0,1)

                if (coinFlip == 1):

                    offspring[i].gene[j] = offspring[i].gene[j] + alter

                    if (offspring[i].gene[j] > MAX):

                         offspring[i].gene[j] = MAX

                else:

                    offspring[i].gene[j] = offspring[i].gene[j] - alter

                    if (offspring[i].gene[j] < MIN):

                         offspring[i].gene[j] = MIN

    return offspring

def mutationGauss(offspring,Pop,N,MUTRATE,MUTSTEP,MAX,MIN):

    for i in range(0, Pop):

        for j in range(0, N):

            MUTPROB = random.uniform(0.0, 1.0)

            if (MUTPROB < MUTRATE):

                gaussian = random.gauss(0,MUTSTEP)

                offspring[i].gene[j] = offspring[i].gene[j] \* gaussian

                if (offspring[i].gene[j] > MAX):

                    offspring[i].gene[j] = MAX

    return offspring

def minFitness(population,Pop):

    minFitness = population[0].fitness

    for x in range(0, Pop):

        if (population[x].fitness < minFitness):

            minFitness = population[x].fitness

    return minFitness

def bestSolutionOfGen(population,Pop):

    minFitness = population[0].fitness

    bestSolutionNo = []

    for x in range(0, Pop):

        if (population[x].fitness < minFitness):

            minFitness = population[x].fitness

            bestSolutionNo = copy.deepcopy(population[x].gene)

    return bestSolutionNo

def meanFitness(population,Pop):

    meanFitnessAdd = 0

    for x in range(0, Pop):

        meanFitnessAdd = meanFitnessAdd + population[x].fitness

    meanFitnessDiv = meanFitnessAdd / Pop

    return meanFitnessDiv

def minFitnessSave(population, offspring,Pop):

    minFitness = population[0].fitness

    bestIndervidual = 0

    for x in range(1, Pop):

        if (population[x].fitness < minFitness):

            minFitness = population[x].fitness

            bestIndervidual = x

    worseFitness = population[0].fitness

    worseIndervidual = 0

    for x in range(1, Pop):

        if (offspring[x].fitness > worseFitness):

            worseFitness = offspring[x].fitness

            worseIndervidual = x

    offspring[worseIndervidual] = copy.deepcopy(population[bestIndervidual])

    return offspring

def printMainPop(population,Pop,N):

    for i in range(0, Pop):

        print("Gene ")

        for x in range(0, N):

            print(population[i].gene[x])

        print("Fitness")

        print(population[i].fitness)

def printMin(minEachGen):

    for i in range(0, len(minEachGen)):

        print(minEachGen[i])

def printMean(meanFitnessGen):

    for i in range(0, len(meanFitnessGen)):

        print(meanFitnessGen[i])

def GA(N,Pop,MUTRATE, MUTSTEP, GEN, MIN,MAX,SELECROSS,SELESelect,SELEFIT):

    population = []

    offspring = []

    minEachGen = []

    meanFitnessGen = []

    theBestSolutions = []

    population = generatePopulationReals(Pop,N,MIN,MAX)

    population = selectFitnessFunc(SELEFIT, population,Pop,N)

    for i in range(0,GEN):

        offspring = selection(population,Pop,SELESelect)

        offspring = crossover(offspring,Pop,N,SELECROSS)

        offspring = mutationGauss(offspring,Pop,N,MUTRATE,MUTSTEP,MAX,MIN)

        offspring = selectFitnessFunc(SELEFIT, offspring,Pop,N)

        population = minFitnessSave(population,offspring,Pop)

        minEachGen.append(minFitness(population,Pop))

        meanFitnessGen.append(meanFitness(population,Pop))

    return GenerationData(minEachGen, meanFitnessGen,theBestSolutions,MUTRATE)

def plotGraph2D(testPara, genData):

    plt.figure(figsize=(20,10))

    plt.xlabel("Generation", fontsize=15)

    plt.ylabel("Min fitness",  fontsize=15)

    plt.title("GA Graph")

    for x in range(0, len(testPara)):

        ypoints = np.array(genData[x].best)

        zpoints = np.array(genData[x].mean)

        plt.plot(ypoints, label="Best" + " " + str(genData[x].MUTRATE))

        plt.plot(zpoints ,label="Mean" + " " + str(genData[x].MUTRATE))

    plt.legend(loc='best')

    plt.show()

def plotGraph2DAverage(genData,mean,MUTRATE,MUTSTEP):

    plt.figure(figsize=(20,10))

    plt.xlabel("Generations", fontsize=15)

    plt.ylabel("Average Fitness for each generation",  fontsize=15)

    plt.title("Genetic Algorithm results over 10 runs of the system")

    ypoints = np.array(genData)

    y2points = np.array(mean)

    plt.plot(ypoints, label="Best Fitness over 10 runs" + " " + str(MUTRATE) + " " + str(MUTSTEP))

    plt.plot(y2points, label="Average Fitness over 10 runs")

    plt.legend(loc='best')

    plt.show()

def lowestInEachTest(genData):

    for i in range(0,len(genData)):

        minFitness = genData[i].best[0]

        for x in range(0,len(genData[i].best)):

            if (genData[i].best[x] < minFitness):

                minFitness = genData[i].best[x]

        print(minFitness)

def lowestInEachTestAppend(genData):

    main = []

    for i in range(0,len(genData)):

        minFitness = genData[i].best[0]

        #bestSolution = []

        for x in range(0,len(genData[i].best)):

            if (genData[i].best[x] < minFitness):

                minFitness = genData[i].best[x]

                #bestSolution = copy.deepcopy(genData[i].bestSolutions[x])

        main.append(minFitness)

        #main.append(bestSolution)

    return main

"""

"""

"""

================================

SELECROSS:(Select Crossover)

One Point Crossover: 0

two Point Crossover: 1

================================

SELESELECT: (Select Selection)

Tournement Selection: 0

Roulette Wheel Selection: 1

================================

SELEFIT: (Select Fitness)

Adding Up Fitness: 0

Rastrigin function: 1

Rosenbrock function: 2

Ackley function: 3

================================

"""

def averageOverTests(testNo,test):

    allgen = []

    meanFitnessAdd = 0

    for z in range(0, len(test)):# Takes final element in the best fitness array for a given run of the system

        allgen.append(test[z].best[len(test[z].best) - 1])# appends to array

    #print(allgen)

    for x in range(0, len(allgen)):#Finds average

        meanFitnessAdd = meanFitnessAdd + allgen[x]

    meanFitnessDiv = meanFitnessAdd / len(allgen)

    return meanFitnessDiv

def averageOverGen(test,GEN):

    aver = []

    for i in range(0,GEN):

        allgen = []

        meanFitnessAdd = 0

        for z in range(0, len(test)):

            allgen.append(test[z].best[i])

        for x in range(0, len(allgen)):#Finds average

            meanFitnessAdd = meanFitnessAdd + allgen[x]

        aver.append(meanFitnessAdd / len(allgen))

    return aver

def averageMeanOverGen(test,GEN):

    aver = []

    for i in range(0,GEN):

        allgen = []

        meanFitnessAdd = 0

        for z in range(0, len(test)):

            allgen.append(test[z].mean[i])

        for x in range(0, len(allgen)):#Finds average

            meanFitnessAdd = meanFitnessAdd + allgen[x]

        aver.append(meanFitnessAdd / len(allgen))

    return aver

def testAverage(testNum,N,Pop,MUTRATE,MUTSTEP,GEN,MIN,MAX,SELECROSS,SELESELECT,SELEFIT):

    genData = []

    testPara = []

    testPara.append(TestParameters(testNum,N,Pop,MUTRATE,MUTSTEP,GEN,MIN,MAX,SELECROSS,SELESELECT,SELEFIT))

    testPara.append(TestParameters(testNum,N,Pop,MUTRATE,MUTSTEP,GEN,MIN,MAX,SELECROSS,SELESELECT,SELEFIT))

    testPara.append(TestParameters(testNum,N,Pop,MUTRATE,MUTSTEP,GEN,MIN,MAX,SELECROSS,SELESELECT,SELEFIT))

    testPara.append(TestParameters(testNum,N,Pop,MUTRATE,MUTSTEP,GEN,MIN,MAX,SELECROSS,SELESELECT,SELEFIT))

    testPara.append(TestParameters(testNum,N,Pop,MUTRATE,MUTSTEP,GEN,MIN,MAX,SELECROSS,SELESELECT,SELEFIT))

    testPara.append(TestParameters(testNum,N,Pop,MUTRATE,MUTSTEP,GEN,MIN,MAX,SELECROSS,SELESELECT,SELEFIT))

    testPara.append(TestParameters(testNum,N,Pop,MUTRATE,MUTSTEP,GEN,MIN,MAX,SELECROSS,SELESELECT,SELEFIT))

    testPara.append(TestParameters(testNum,N,Pop,MUTRATE,MUTSTEP,GEN,MIN,MAX,SELECROSS,SELESELECT,SELEFIT))

    testPara.append(TestParameters(testNum,N,Pop,MUTRATE,MUTSTEP,GEN,MIN,MAX,SELECROSS,SELESELECT,SELEFIT))

    testPara.append(TestParameters(testNum,N,Pop,MUTRATE,MUTSTEP,GEN,MIN,MAX,SELECROSS,SELESELECT,SELEFIT))

    for i in range(0,len(testPara)):

        genData.append(GA(testPara[i].N,testPara[i].Pop,testPara[i].MUTRATE,testPara[i].MUTSTEP,testPara[i].GEN,testPara[i].MIN,testPara[i].MAX,testPara[i].SELECROSS, testPara[i].SELESELECT,testPara[i].SELEFIT))

    return averageOverTests(testNum,genData)

def testIndiv(testNum,N,Pop,MUTRATE,MUTSTEP,GEN,MIN,MAX,SELECROSS,SELESELECT,SELEFIT):

    genData = []

    testPara = []

    testPara.append(TestParameters(testNum,N,Pop,MUTRATE,MUTSTEP,GEN,MIN,MAX,SELECROSS,SELESELECT,SELEFIT))

    testPara.append(TestParameters(testNum,N,Pop,MUTRATE,MUTSTEP,GEN,MIN,MAX,SELECROSS,SELESELECT,SELEFIT))

    testPara.append(TestParameters(testNum,N,Pop,MUTRATE,MUTSTEP,GEN,MIN,MAX,SELECROSS,SELESELECT,SELEFIT))

    testPara.append(TestParameters(testNum,N,Pop,MUTRATE,MUTSTEP,GEN,MIN,MAX,SELECROSS,SELESELECT,SELEFIT))

    testPara.append(TestParameters(testNum,N,Pop,MUTRATE,MUTSTEP,GEN,MIN,MAX,SELECROSS,SELESELECT,SELEFIT))

    testPara.append(TestParameters(testNum,N,Pop,MUTRATE,MUTSTEP,GEN,MIN,MAX,SELECROSS,SELESELECT,SELEFIT))

    testPara.append(TestParameters(testNum,N,Pop,MUTRATE,MUTSTEP,GEN,MIN,MAX,SELECROSS,SELESELECT,SELEFIT))

    testPara.append(TestParameters(testNum,N,Pop,MUTRATE,MUTSTEP,GEN,MIN,MAX,SELECROSS,SELESELECT,SELEFIT))

    testPara.append(TestParameters(testNum,N,Pop,MUTRATE,MUTSTEP,GEN,MIN,MAX,SELECROSS,SELESELECT,SELEFIT))

    testPara.append(TestParameters(testNum,N,Pop,MUTRATE,MUTSTEP,GEN,MIN,MAX,SELECROSS,SELESELECT,SELEFIT))

    for i in range(0,len(testPara)):

        genData.append(GA(testPara[i].N,testPara[i].Pop,testPara[i].MUTRATE,testPara[i].MUTSTEP,testPara[i].GEN,testPara[i].MIN,testPara[i].MAX,testPara[i].SELECROSS, testPara[i].SELESELECT,testPara[i].SELEFIT))

    #plotGraph2D(testPara,genData)

    print(averageOverTests(testNum,genData))

    print(lowestInEachTestAppend(genData))

    plotGraph2DAverage(averageOverGen(genData,GEN),averageMeanOverGen(genData,GEN),MUTRATE,MUTSTEP)

    #print(genData[0].best)

def mainAverageTests():

    GEN = 500

    MUTSTEP = 60

    SELECROSS = 0

    SELESELECT = 0

    SELEFIT = 2

    MIN = -100

    MAX = 100

    Pop = 50

    print(testAverage(1,20,Pop,0.01,MUTSTEP,GEN,MIN,MAX,SELECROSS,SELESELECT,SELEFIT))

    print(testAverage(1,20,Pop,0.02,MUTSTEP,GEN,MIN,MAX,SELECROSS,SELESELECT,SELEFIT))

    print(testAverage(1,20,Pop,0.03,MUTSTEP,GEN,MIN,MAX,SELECROSS,SELESELECT,SELEFIT))

    print(testAverage(1,20,Pop,0.04,MUTSTEP,GEN,MIN,MAX,SELECROSS,SELESELECT,SELEFIT))

    print(testAverage(1,20,Pop,0.05,MUTSTEP,GEN,MIN,MAX,SELECROSS,SELESELECT,SELEFIT))

def mainIndivTests():

    GEN = 500

    SELECROSS = 0

    SELESELECT = 0

    SELEFIT = 2

    MIN = -100

    MAX = 100

    testIndiv(1,20,300,0.015,20,GEN,MIN,MAX,SELECROSS,SELESELECT,SELEFIT)

#mainAverageTests()

mainIndivTests()