Exploring the Effect Parameters and Operators have on the Performance of Genetic Algorithms

Brendan Baker 17019052

Introduction

Genetic Algorithms came to fruition in the 1960s and 1970s when John Holland and his team took inspiration from Charles Darwin's theory of natural selection to create an algorithm which imitates natural selection. John Holland (2005) states that genetic algorithms "make it possible to explore a far greater range of potential solutions to a problem than conventional programs" (Holland, 2005). This is achieved through several genetic operators such as selection, crossover, and mutation. Following the Biocomputation module assignment, this report will follow how a genetic algorithm is developed for the optimisation pathway and cover minimisation problems using Ackley and Rosenbrock test functions to convey performance. This shall be with a fitness objective and be able to real values. In addition, the report will explore and evaluate different types of selection, crossover, and mutation.

Background Research

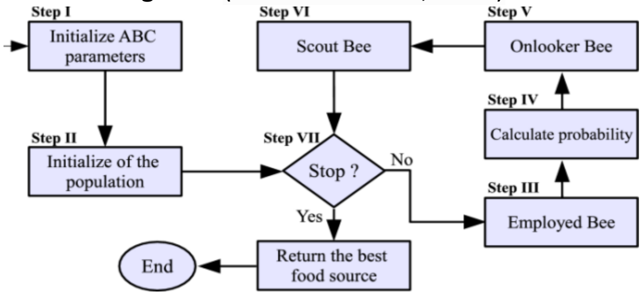
However, evolutionary algorithms are not the only way to solve optimisations problems. Another algorithm not covered in lectures that can be applied to optimisation problems is the Artificial Bee Colony (ABC) algorithm. Artificial Bee Colony is a Metaheuristics search technique which follows principles of swarm intelligence, it imitates the process of honeybee swarms foraging for food. The minimal model forage selection that transitions to the emergence of collective intelligence of honeybee swarms is identified by Dervis Karaboga (2005) who developed Artificial Bee Colony and describes the three essential components of the model as "food sources, employed foragers and unemployed foragers" (Karaboga, 2005). This is further accompanied by two principal behaviours "the recruitment of nectar source and abandonment of a source" (Karaboga, 2005).

In the Artificial Bee Colony algorithm, the position of the food sources are potential solutions to the given optimisation problem and the nectar (amount of food at the source) reflects the quality (fitness) of the associated fitness.

Employed Bees are bees which are actively associated with a food source, for example carrying nectar away from the source back to the hive. But they also carry with them information about the source such as location, profitability of the source and will convey this information with a certain probability. Information is conveyed through the waggle dance which is performed on the dance area. If the food source becomes diminished, the bee becomes a scout in the algorithm will pick a random point the search space to explore. There are 2 types of unemployed Bees: scouts and onlookers. These will be elaborated on in the implementation steps.

Step 1 to implementing Artificial Bee colony is as shown in figure 1 to initialise the parameters. There are three control parameters for any given optimisations problem which are essential to ABC these are "number of food sources (SN), the value of limit, the maximum cycle number (MCN)" (Kumar, Kumar and Jarial, 2016). The number of food sources corresponds to the population size in a GA, the MCN is the maximum number of generations in a GA, GA has more parameters to be adjusted than ABC to reach global minima.

Figure 1 (Awadallah et al., 2018)



Step 2 - The second step is to initialise/create the initial population following Karaboga (2005) each food source is created as depicted in figure 2.

Figure 2 (Awadallah et al., 2018)

$$x_i^j = x_{min}^j + rand(0,1)\left(x_{max}^j - x_{min}^j\right)$$

Here x^j_i denotes the ith employed bee and a food source position denoted in the jth position, x^j_{min} and x^j_{max} corresponds to the decision variable lower and upper limits whilst rand (0, 1) generates a random number between 0 and 1. After this, the food sources are designated at random to a "SN number of employed bees and their fitness is determined (Karaboga, 2005).

Step 3 This is the Employed Bee stage, we generate new solutions for the employee bee using the equation shown in figure 3. The equation depicts a search equation the employed bee adjusts the food source in its memory to produce a new food source in proximity.

Figure 4 | Figure 3

$$v_{ij} = x_{ij} + \phi_{ij}(x_{ij} - x_{kj}) \qquad P_i = f_i \Big/ \sum_{k=1}^{SN} f_k$$

Both Figures – Karaboga (2005)

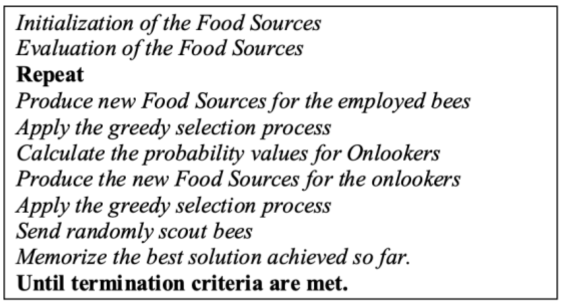
v_{ij} represents the new bee position, x_{ij} is the current employed bees' position, Phi denotes a random uniform number in the range of [-1, 1] this will control the generation of neighbouring food source around the current bee's position and is our step length. It will also "assist the bees visually in making comparison between two food sources" (Awadallah et al., 2018). Then the accumulative sum is multiplied by the current employed bees position subtracted by x_{kj} which is the decision variable located at j in a food source x_k, it's important that it's "chosen randomly by an employed bee other than the original food source" (Awadallah et al., 2018)

Now the employed bee needs to evaluate its current food source against the new food source, this is achieved via a tournament selection where it checks which food source has the better fitness the winner transitions into the population.

Onlooker's phase, the role of an onlooker is to establish the nectar information (fitness of solutions) collected from all the employed bees in the hive it then selects a food source based off the information as follows in figure 4.

P_i denotes the probability and where "f_i denotes the fitness value of the ith food source. The onlooker after selecting the food source x_i , modifies it by using Equation (2)." (Karaboga, 2005). Here when Karaboga (2005) refers to equation 2 this translates to the search space equation in figure 3 of the report. Similarly, a tournament selection is applied again, but this time between the current food source and the onlookers modified food source to see solution has the best fitness.

Figure 5 - (Kumar, Kumar and Jarial, 2016).



```
Initialization of the Food Sources
Evaluation of the Food Sources
Repeat
Produce new Food Sources for the employed bees
Apply the greedy selection process
Calculate the probability values for Onlookers
Produce the new Food Sources for the onlookers
Apply the greedy selection process
Send randomly scout bees
Memorize the best solution achieved so far.
Until termination criteria are met.
```

Scout Phase - When a food source becomes diminished after the limit of iterations it had to improve, it the becomes a scout and abandons its food source. The scout will search randomly for a food source in the search space using figure 3, this assists in inducing diversity into the Algorithm. Following the pseudo code in figure 5, the algorithm will run iterations of the employee stage, onlooker phase and scout phase until a termination criterion is met.

Implementation

The implementation stage began with implementing our own GA, the first iteration commenced with choosing a combination of operators that consisted of the following roulette wheel selection, single point crossover, uniformed mutation, and elitism. The performance will be tested by Rosenbrock (figure 5) and Ackley (figure 6). For Rosenbrock, where -100 < x < 100 with a gene size of 20 and a global minima of 0. And Ackley, where -32 < x < 32 and a gene size of 20 with a global minima of -22.7.

Figure 6

$$f_2 = 100(x_1^2 - x_2)^2 + (1 - x_1)^2$$

(Digalakis and Margaritis, 2001)

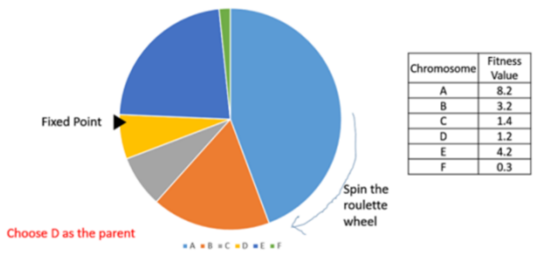
Figure 7

$$f(\mathbf{x}) = -a\exp\left(-b\sqrt{\frac{1}{d}\sum_{i=1}^{d} x_i^2}\right) - \exp\left(\frac{1}{d}\sum_{i=1}^{d}\cos(cx_i)\right)$$

(Ackley Function, 2021)

Roulette wheel selection is made of a wheel and has a fixed point. Fittest individuals take up a larger proportion of the wheel, a fixed point is allocated and when the wheel is span, the fitter individual has a higher chance of being in front of the fixed point.

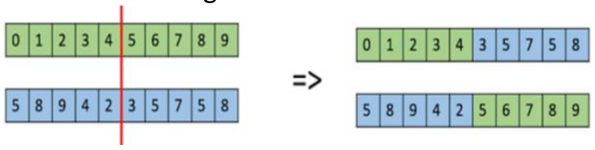
Figure 8 - (Genetic Algorithms Tutorial, 2021)



Chromosome	Fitness Value
A	8.2
B	3.2
C	1.4
D	1.2
E	4.2
F	0.3

Crossover is important in GA's as without crossover a chromosome may be represented several times in continuous generations leading to a population full of copies. To overcome this, crossover is aims to evolve populations towards the goal. Single point crossover involves two mating chromosomes to be cut at a random point with one another. At the cut-off point or crossover point, the information to the left (or right) of the crossover point are swapped between the two parent chromosomes to produce two offspring.

Figure 8



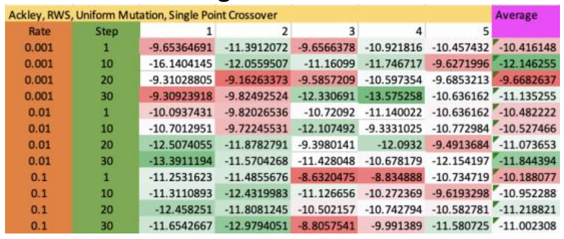
(Genetic Algorithms Tutorial, 2021)

Uniform mutation replaces genes when selected for mutation with a uniform distributed random value between 0 and the mutation step.

Elitism involves taking the best individual from the current population and overwriting the worst individual in the new population, this can improve convergence speed. However, can lead to a low diversity population in turn making offspring very similar to parent if not the same copy.

I started the with a parameter sweep for Ackley using roulette wheel selection, single point crossover, uniform mutation and elitism. However, as depicted in the below table the results where sporadic and random, their seemed to be no correlation and far from the global minima.

Figure 9



Ackley, RWS, Uniform Mutation, Single Point Crossover							Average
Rate	Step	1	2	3	4	5	
0.001	1	-9.65364691	-11.3912072	-9.6566378	-10.921816	-10.457432	-10.416148
0.001	10	-16.1404145	-12.0559507	-11.16099	-11.746717	-9.6271996	-12.146255
0.001	20	-9.31028805	-9.16263373	-9.5857209	-10.597354	-9.6853213	-9.6682637
0.001	30	-9.30923918	-9.82492524	-12.330691	-13.575258	-10.636162	-11.135255
0.01	1	-10.0937431	-9.82026536	-10.72092	-11.140022	-10.636162	-10.482222
0.01	10	-10.7012951	-9.72245531	-12.107492	-9.3331025	-10.772984	-10.527466
0.01	20	-12.5074055	-11.8782791	-9.3980141	-12.0932	-9.4913684	-11.073653
0.01	30	-13.3911194	-11.5704268	-11.428048	-10.678179	-12.154197	-11.844394
0.1	1	-11.2531623	-11.4855676	-8.6320475	-8.834888	-10.734719	-10.188077
0.1	10	-11.3110893	-12.4319983	-11.126656	-10.272369	-9.6193298	-10.952288
0.1	20	-12.458251	-11.8081245	-10.502157	-10.742794	-10.582781	-11.218821
0.1	30	-11.6542667	-12.9794051	-8.8057541	-9.991389	-11.580725	-11.002308

According to Rahman et al (2016) roulette wheel selection cannot handle negative values due to the "proportionality concept" (Rahman et al, 2016) as the pies would have negative portions and for a minimisation problem this is inadequate. An alternative, selection method is Tournament selection where individuals compete with eachother and the individual with the best fitness progresses to the crossover stage. An advantage of tournament selection is its ability to "handle either minimization or maximization problems without any structural changes" (Rahman et al, 2016) which I found was just changing a single operator. In addition, single point crossover is only used for binary encoded GA's and since we're handling real values, I decided to change to simple arithmetic crossover which handles real values by taking the weighted mean of two parents by using the following formula.

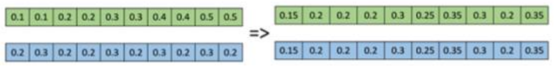
Figure 10

- Child1 = $\alpha.x + (1-\alpha).y$
- Child2 = $\alpha.x + (1-\alpha).y$

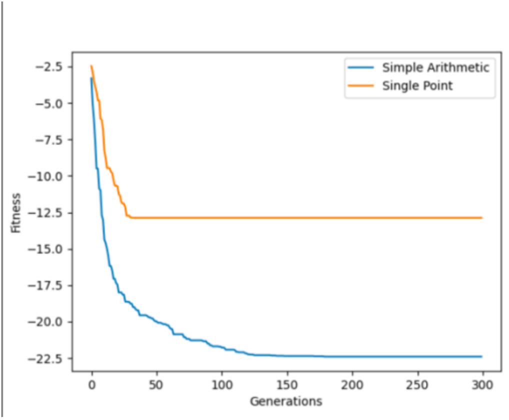
(Genetic Algorithms Tutorial, 2021)

In figure x 'a' represents the crossover weight or probability such as if a = '0.5' the following crossover will take place as depicted in figure x below. Simple Arithmetic crossover determines the point of change using a random number generator. Children are created using arithmetic operations of the parents' genes with a variable multiplier (crossover probability) that can ranges from 0 - 1.

Figure 11 (Genetic Algorithms Tutorial, 2021)

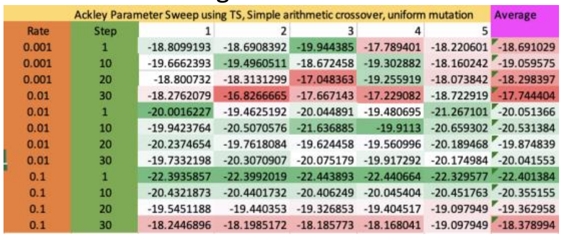


Side by side performance of crossovers using tournament selection, uniform mutation and elitism shown below. Supporting the theory that simple arithmetic handles real values better and as shown below escapes local optima implying it explored the search space more efficiently.



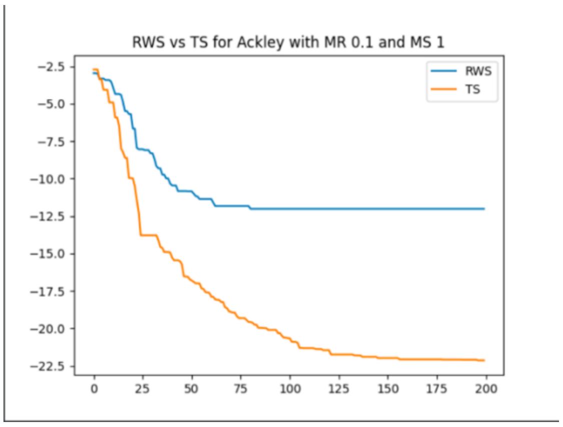
I decided to choose a weight/probability of 0.1 to attempt to maintain population diversity.

Figure 12



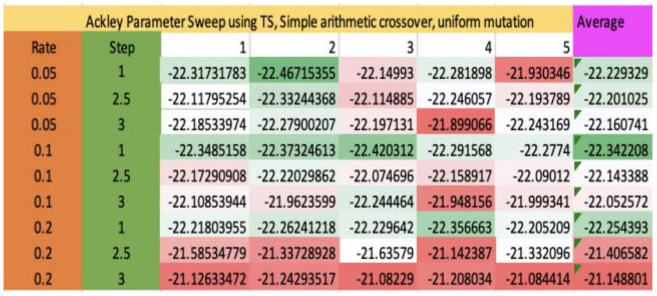
Ackley Parameter Sweep using TS, Simple arithmetic crossover, uniform mutation							Average
Rate	Step	1	2	3	4	5	
0.001	1	-18.8099193	-18.6908392	-19.944385	-17.789401	-18.220601	-18.691029
0.001	10	-19.6662393	-19.4960511	-18.672458	-19.302882	-18.160242	-19.059575
0.001	20	-18.800732	-18.3131299	-17.048363	-19.255919	-18.073842	-18.298397
0.01	30	-18.2762079	-16.8266665	-17.667143	-17.229082	-18.722919	-17.744404
0.01	1	-20.0016227	-19.4625192	-20.044891	-19.480695	-21.267101	-20.051366
0.01	10	-19.9423764	-20.5070576	-21.636885	-19.9113	-20.659302	-20.531384
0.01	20	-20.2374654	-19.7618084	-19.624458	-19.560996	-20.189468	-19.874839
0.01	30	-19.7332198	-20.3070907	-20.075179	-19.917292	-20.174984	-20.041553
0.1	1	-22.3935857	-22.3992019	-22.443893	-22.440664	-22.329577	-22.401384
0.1	10	-20.4321873	-20.4401732	-20.406249	-20.045404	-20.451763	-20.355155
0.1	20	-19.5451188	-19.440353	-19.326853	-19.404517	-19.097949	-19.362958
0.1	30	-18.2446896	-18.1985172	-18.185773	-18.168041	-19.097949	-18.378994

From the above parameter sweep, a mutation rate between 0.01 and 1 with a low step of 1 yielded good result. This implies that a relatively medium to high mutation rate and walking through the space little by little seems good.



The above graph shows that RWS has a slower gradient descent compared to TS whose descent is rapid, even though RWS started at a slightly better position. It appears TS method of choosing the best individuals each time is better than RWS as with RWS there's a possibility to pick an unfit or average individual which explains TS rapid gradient descent as only the best individuals make it through to crossover.

RWS then gets stuck in a local optimum far from the global minima, whereas TS continues towards the global minima. I believe RWS gets stuck due to lack of population diversity at less than 100 generations and premature convergence occurred figure 14

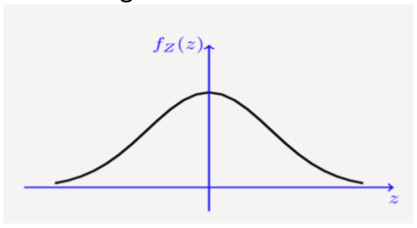


Ackley Parameter Sweep using TS, Simple arithmetic crossover, uniform mutation							Average
Rate	Step	1	2	3	4	5	
0.05	1	-22.31731783	-22.46715355	-22.14993	-22.281898	-21.930346	-22.229329
0.05	2.5	-22.11795254	-22.33244368	-22.114885	-22.246057	-22.193789	-22.201025
0.05	3	-22.18533974	-22.27900207	-22.197131	-21.899066	-22.243169	-22.160741
0.1	1	-22.3485158	-22.37324613	-22.420312	-22.291568	-22.2774	-22.342208
0.1	2.5	-22.17290908	-22.22029862	-22.074696	-22.158917	-22.09012	-22.143388
0.1	3	-22.10853944	-21.9623599	-22.244464	-21.948156	-21.999341	-22.052572
0.2	1	-22.21803955	-22.26241218	-22.229642	-22.356663	-22.205209	-22.254393
0.2	2.5	-21.58534779	-21.33728928	-21.63579	-21.142387	-21.332096	-21.406582
0.2	3	-21.12633472	-21.24293517	-21.08229	-21.208034	-21.084414	-21.148801

I decided to do a deeper analysis on the parameters to find the optimal combination (as shown in figure 14) and from my findings 0.1 mutation rate and a mutation step size of 1 remain the best with a population of 200 and 200 generations with the average of the best results being -22.34. Increasing the population size and generations has little effect on the results, perhaps using elitism every time eventually made the population lack diversity as values would get recycled (elitism was used on all variations and images shown).

Moving onto Rosenbrock, I initially carried over the same operators to the Rosenbrock test function. Using Tournament selection, simple arithmetic crossover uniform mutation and elitism. From what I learned from Ackley's a mutation rate of 0.05 performed quite well so I wanted to include this in the initial parameter search. This is supported by Piszcz and Soule (2016) who found that an optimal mutation rate "across a range of mutation types and level of difficulty is close to 1/C, where C is the maximum size of the individual" (Piszcz and Soule, 2016), C denoting the gene size of an individual in this case 1 / N, where N = 20. And since the upper and lower bounds are bigger, I increased my starting mutation step to 10% of the upper bound giving a step of 10.

Figure 15

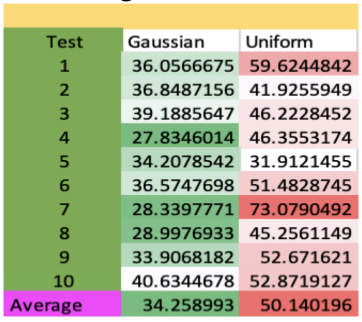


(Pishro-Nik, 2014)

I began to research variants of mutation and found that "The most commonly-applied mutation operator in this context is gaussian mutation" (Piszcz and Soule, 2006). After more research, I established that gaussian mutation works when a gene is selected for mutation, a random real value is generated based on gaussian distribution and appended to the gene value. Figure 15 is sometimes referred to as the bell curve, it shows the distribution and how values are chosen around the mean. Next the value is taken and appended onto the existing value of the gene. The advantages being that the mutated value will be near the original fitness but does allow for occasional larger changes as shown in figure 15. The GA will benefit hugely from this, as it will prevent the GA from being overly disruptive, whilst still retaining the ability to move out of local optima. The implementation was simple in Python by substituting the built in function random.uniform to random.gauss passing in a mu of 0 and sigma of step size.

I completed a test run to compare both uniform and gaussian mutations using a population size of 400 and 500 generations using 0.05 mutation rate and a mutation step of 10. As seen in figure 16 below, gaussian mutation performed better and shows it values correlating around the mean but allowed values such as test 10s result by creating a larger number to maintain even distribution. Uniform mutation also performed well but outliers such as test 7 and test 1 results made the average high compared to Gaussian.

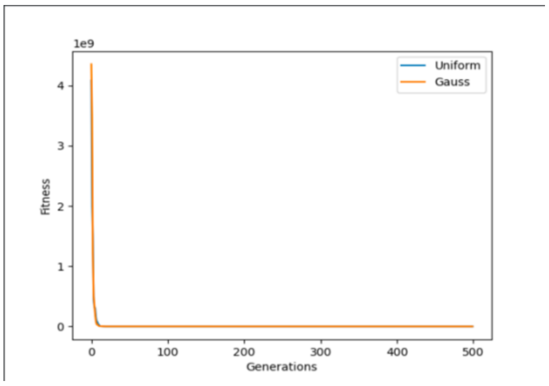
Figure 16



Test	Gaussian	Uniform
1	36.0566675	59.6244842
2	36.8487156	41.9255949
3	39.1885647	46.2228452
4	27.8346014	46.3553174
5	34.2078542	31.9121455
6	36.5747698	51.4828745
7	28.3397771	73.0790492
8	28.9976933	45.2561149
9	33.9068182	52.671621
10	40.6344678	52.8719127
Average	34.258993	50.140196

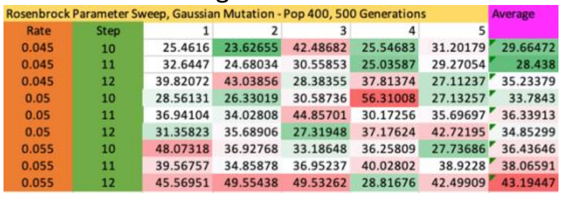
Figure 17 below shows a graph comparison of uniform and gaussian mutation using tournament selection, simple arithmetic crossover and elitism. As shown, they are almost identical in terms of gradient descent and as we are dealing with a small global minima the graph isn't sufficient to determine the results hence it's accompanied by the test table. My assumption is that uniform mutation got stuck in a local optimum whilst gaussian kept improving towards the global minima a while longer, escaping local optima. The reason for increasing population size and generations was to account for the bigger search space from -100 to 100 compared to Ackley -32 to 32.

Figure 17



Needing to know the best combination of mutation rate and mutation step I could achieve, a final sweep identified 0.045 with a step of 11. When running 5 runs, an average of averaged best was 28.438 as shown below.

Figure 18



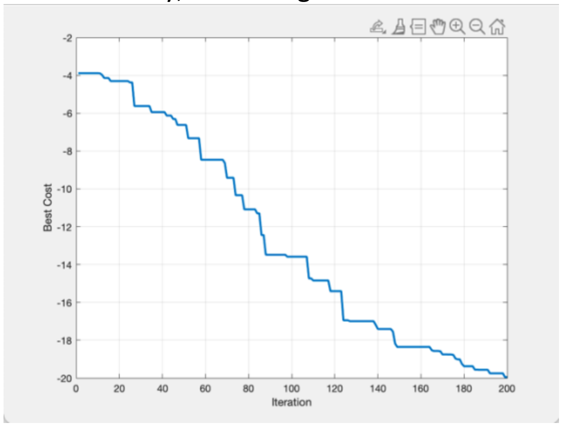
Rosenbrock Parameter Sweep, Gaussian Mutation - Pop 400, 500 Generations							Average
Rate	Step	1	2	3	4	5	
0.045	10	25.4616	23.62655	42.48682	25.54683	31.20179	29.66472
0.045	11	32.6447	24.68034	30.55853	25.03587	29.27054	28.438
0.045	12	39.82072	43.03856	28.38355	37.81374	27.11237	35.23379
0.05	10	28.56131	26.33019	30.58736	56.31008	27.13257	33.7843
0.05	11	36.94104	34.02808	44.85701	30.17256	35.69697	36.33913
0.05	12	31.35823	35.68906	27.31948	37.17624	42.72195	34.85299
0.055	10	48.07318	36.92768	33.18648	36.25809	27.73686	36.43646
0.055	11	39.56757	34.85878	36.95237	40.02802	38.9228	38.06591
0.055	12	45.56951	49.55438	49.53262	28.81676	42.49909	43.19447

Now to move onto increasing the generations to see if that would affect results, when increasing generations to 2000 over 5 runs the average best was 19.18. It implies the population was still diverse after 500 generations, and still escaped local optima's but never made it to the global minima. The trend in results show a bigger mutation step is needed for Rosenbrock compared to Ackley because the search space is bigger hence bigger mutation steps are needed to achieve good results.

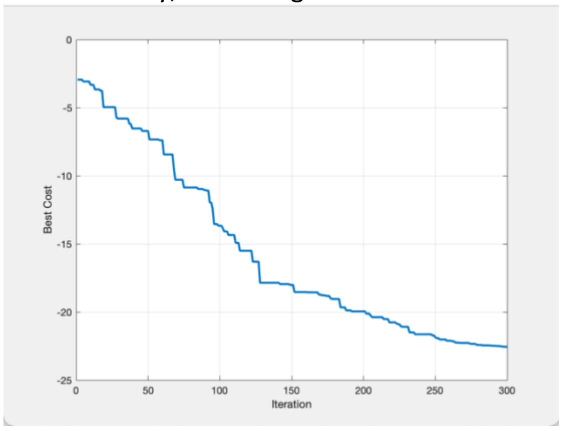
I acquired a MATLAB package to implement ABC through "Artificial Bee Colony in MATLAB" (Artificial Bee Colony in MATLAB - Yarpiz, 2021).

Comparing the performance for ABC Algorithm on Ackley test function with the same parameters 20 gene size, 200 population. We can see, in 200 generations it reached -19.95 which is a good result. But our GA performed better at -22.34, the gradual gradient in a step like manner shows the benefits of tournament selection to select only the fittest for the next generation. And as demonstrated in the graph below, when reaching past 160 generations the performance starts to decrease. However, the onlooker and scout phase keep the population diverse and avoid falling into a local optimum. But when increasing the generations to 300 as shown in the 2nd graph below, it does reach the global minima where my GA does not.

Ackley, ABC 200 generations

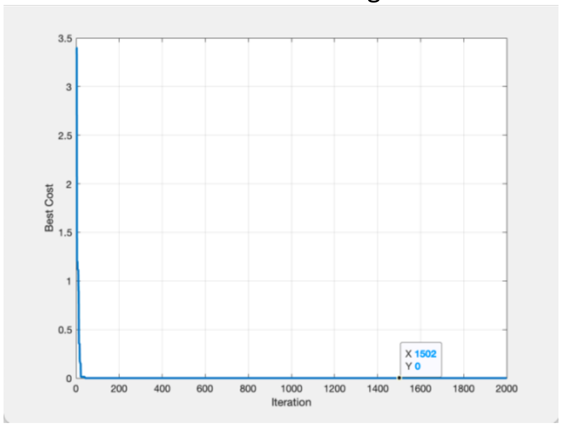


Ackey, ABC 300 generations



ABC was used to test Rosenbrock test functions performance, it performed better than my GA with same parameters where in 1500 generations it reached the global minima of 0 as shown below, whereas my GA reached 19.8. I believe yet again the onlooker phase is why ABC didn't get trapped in a local optimum like my GA.

ABC Rosenbrock 2000 generations



Conclusion

From the experiments on Ackley, Rosenbrock and ABC, the ABC algorithm outperformed my GA trials. However, my GA did improve significantly throughout the process through trying numerous operators and parameter sweeps. But unfortunately, it didn't manage to find the global minima for Rosenbrock and Ackley. ABC algorithm showed it's efficiently and simplicity as it reaches the global minima for both test function without parameter changes, compared to my GA which needed lots of research and tuning to be able to get close results. If I had a second chance at the experiment I would experiment with adaptive mutation. I'm aware my mutation could be mutating good fitness's and making them worse which is detrimental to GA. I would as well like to explore Gaussian Mutation more, as Temby, Vamplew, and Berry (2005) found that the sigma should be "1/15 of the range of each allele" (Temby, Vamplew, and Berry, 2015) whereas I was using my mutation step size as my sigma, they also convey that the mu could be adjusted from 0-1, I didn't explore the effects of changing the mu from 0, I could conduct a parameter sweep of both if I did the experiments again. I would also, not used Elitism for every population, as I think it made it lose diversity, a set probability would have been better.

Bibliography

Abd Rahman, R., Ramli, R., Jamari, Z. and Ku-Mahamud, K., 2016. Evolutionary Algorithm with Roulette-Tournament Selection for Solving Aquaculture Diet Formulation. *Mathematical Problems in Engineering*, 2016, pp.1-10.

Awadallah, M., Al-Betar, M., Bolaji, A., Alsukhni, E. and Al-Zoubi, H., 2018. Natural selection methods for artificial bee colony with new versions of onlooker bee. *Soft Computing*, 23(15), pp.6455-6494.

Digalakis, J.G. and Margaritis, K.G., 2001. On benchmarking functions for genetic algorithms. *International journal of computer mathematics*, *77*(4), pp.481-506.

Holland, J.H., 2005. Genetic algorithms: Computer programs that "evolve" in ways that resemble natural selection can solve complex problems even their creators do not fully understand. *Sci. Am*, *267*, p.1992.

Kumar, A., Kumar, D. and Jarial, S., 2016. A Comparative Analysis of Selection Schemes in the Artificial Bee Colony Algorithm. *Computación y Sistemas*, 20(1).

Pishro-Nik, H., 2014. *Introduction to probability, statistics, and random processes*.

Piszcz, A. and Soule, T., 2006. Genetic Programming: Analysis of Optimal Mutation Rates in a Problem with Varying Difficulty. In *FLAIRS Conference* (pp. 451-456).

Tutorialspoint.com. 2021. *Genetic Algorithms Tutorial*. [online] Available at: <https://www.tutorialspoint.com/genetic_algorithms/index.htm> [Accessed 19 December 2021].

Sfu.ca. 2021. *Ackley Function*. [online] Available at: <https://www.sfu.ca/~ssurjano/ackley.html> [Accessed 19 December 2021].

Temby, L., Vamplew, P. and Berry, A., 2005, December. Accelerating real-valued genetic algorithms using mutation-with-momentum. In *Australasian Joint Conference on Artificial Intelligence* (pp. 1108-1111). Springer, Berlin, Heidelberg.

Yarpiz. 2021. *Artificial Bee Colony in MATLAB - Yarpiz*. [online] Available at: <https://yarpiz.com/297/ypea114-artificial-bee-colony> [Accessed 19 December 2021].

Appendix

Program code for Rosenbrock

```
import random
import matplotlib.pyplot as plt
import copy

N = 20  # Gene Size
P = 400  # Population size
mutations = [0.07]  # Mutation Rate
steps = [1]  # List to hold mutation steps
MIN = -100  # Lower Bound
MAX = 100  # Upper bound
probability = 0.1  # Crossover Probability

"""
class that create a individuals attributes
"""
class Individual:
    def __init__(self):
        self.gene = [0] * N  # initialise gene size
        self.fitness = 0  # initialise fitness



def generate_genes():
    population = []
    # Create random population of genes
    for i in range(0, P):
        temp_gene = []  # List to hold a temp gene
        for j in range(0, N):
            temp_gene.append(random.uniform(MIN, MAX))  # appending a random value from 
between the bounds
        new_ind = Individual()  # New instance of an individual
        new_ind.gene = temp_gene.copy()
        new_ind.fitness = rosenbrock(new_ind)  # new individual is assigned a fitness value 
from rosenbrock
        population.append(new_ind)  # new individual gets appended to the population

    return population
```

```
def tournament_selection(population):
    offspring = []

    for i in range(0, P):
        parent_1 = random.randint(0, P - 1)  # Generate 1st random integer in population
        off_1 = population[parent_1]  # Get random individual in population
        parent_2 = random.randint(0, P - 1)  # Generate 1st random integer in population
        off_2 = population[parent_2]  # Get random individual in population
        if off_1.fitness > off_2.fitness:  # Compete to get best
            offspring.append(off_2)
        else:
            offspring.append(off_1)

    return offspring  # Winner is returned for crossover


def arithmetic_combination(parent_1, parent_2, cross_prob):
    child_1 = (cross_prob * parent_1) + (1 - cross_prob) * parent_2  # create child 1 from
parents
    child_2 = (cross_prob * parent_2) + (1 - cross_prob) * parent_1  # create child 2 from
parents

    return child_1, child_2


def simple_arithmetic_combination(offspring, cross_prob):
    for i in range(0, len(offspring), 2):
        parent_1 = offspring[i].gene  # Get offspring gene for parent 1
        parent_2 = offspring[i + 1].gene  # Get offspring gene for parent 1

        cross_point = random.randint(0, N - 1)  # Generate crossover point

        for j in range(cross_point, N):  # Get range of gene to change
            produce_child = arithmetic_combination(parent_1[j], parent_2[j], cross_prob)  # get
children
            parent_1[j] = produce_child[0]
            parent_2[j] = produce_child[1]
    return offspring


def mutation(offspring):
    for i in range(0, P):
        new_individual = Individual()  # New instance of individual
        new_individual.gene = []  # Clear genes
        for j in range(0, N):
            gene = offspring[i].gene[j]
            mutation_probability = random.random()  # Get mute prob
            if mutation_probability < MUTATION_RATE:  # condition to check if mutation
should occur
                alter = random.uniform(0.0, step)  # 0 1.0
                if random.choice([0, 1]) == 1:
                    offspring[i].gene[j] = offspring[i].gene[j] + alter  # alter gene
                    if offspring[i].gene[j] > MAX:
                        offspring[i].gene[j] = MAX  # alter gene value
                else:
                    offspring[i].gene[j] = offspring[i].gene[j] - alter  # alter gene value
                    if offspring[i].gene[j] < MIN:
                        offspring[i].gene[j] = MIN  # alter gene value

            new_individual.gene.append(gene)
        new_individual.fitness = rosenbrock(new_individual)  # new
        offspring[i] = new_individual
    return offspring


def mutation_gauss(offspring):
    for i in range(0, P):
        new_individual = Individual()  # New instance of individual
        new_individual.gene = []  # Clear genes
        for j in range(0, N):
            gene = offspring[i].gene[j]
            mutation_probability = random.random()  # Get mute prob

            if mutation_probability < MUTATION_RATE:  # condition to check if mutation
should occur
                alter = random.gauss(0.0, MUTATION_STEP)  # mu 0.0, sigma mutation step for
gaussian distribution
                if random.choice([0, 1]) == 1:
                    offspring[i].gene[j] = offspring[i].gene[j] + alter  # alter gene
                    if offspring[i].gene[j] > MAX:
                        offspring[i].gene[j] = MAX  # alter gene value
                else:
                    offspring[i].gene[j] = offspring[i].gene[j] - alter  # alter gene value
                    if offspring[i].gene[j] < MIN:
                        offspring[i].gene[j] = MIN  # alter gene value

            new_individual.gene.append(gene)
        new_individual.fitness = rosenbrock(new_individual)  # new
        offspring[i] = new_individual
    return offspring


def rosenbrock(individual) -> float:
    # Rosenbrock minimisation
    fitness = 0
    for i in range(1, N - 1):
        fitness += 100 * pow(individual.gene[i + 1] - individual.gene[i] ** 2, 2) + pow(1 -
individual.gene[i], 2)
    return fitness



def elitism(population, offspring):
    # Sorts the population in order of fitness
    population.sort(key=lambda individual: individual.fitness, reverse=True)
    # The best individual for minimisation is at the end of list
    bestIndividual = population[-1]
    # Take the offspring and overwrite population with offspring
    new_population = copy.deepcopy(offspring)

    # Sort the population in order of fitness again
    new_population.sort(key=lambda individual: individual.fitness, reverse=True)

    # Take the worst individual in the new population and overwrite it with the best from the
old pop
    new_population[0] = bestIndividual

    return new_population



def myGA(population, mutation):
    # Set the amount of generations
    generations = 500
    best_fitness = []
    mean_fitness_plot = []
    # iterate through generations
    for i in range(generations):
        # Get offspring from tournament selection
        offspring = tournament_selection(population)
        # Get crossed over offspring
        offspring_crossover = simple_arithmetic_combination(offspring, probability)
        # Get mutated offspring
        offspring_mutated = mutation(offspring_crossover)
        # Apply elitism
        population = elitism(population, offspring_mutated)

        fitness = []
        for individual in population:
            fitness.append(individual.fitness)
        min_fitness = min(fitness)  # get the best fitness in population for generation

        # Allows to see via command line what the specific fitness are for each step
        if i == generations - 1:
            print("Mutation Rate:", str(MUTATION_RATE), f" | Step
Size:{MUTATION_STEP}", " | Fitness:", min_fitness)

        #mean_fitness = (sum(fitness) / P)  # optional get mean, not used
        best_fitness.append(min_fitness)
        # mean_fitness_plot.append(mean_fitness) # not in use

    return best_fitness


test_gauss = [] # list to hold gauss mutation results
for i in range(5): # Amount of runs
    for rate in mutations: # loop through mutation rates
        for step in steps: # loop through mutation steps
            MUTATION_RATE = rate
            MUTATION_STEP = step
            best_fitness_data = myGA(generate_genes(), mutation) # Call GA for uniform
mutation operator
            test_gauss = myGA(generate_genes(), mutation_gauss) # Call GA for Gauss mutation
operator

plt.plot(best_fitness_data, label=str("Uniform")) # plot uniform mutation results
plt.plot(test_gauss, label='Gauss') # plot gauss mutation results
plt.title("Uniform vs Gauss") # plot title
plt.ylabel('Fitness') # y label
plt.xlabel('Generations')# x label
plt.legend()
plt.show()
```

Program code for Ackley

```
import random
import matplotlib.pyplot as plt
import copy
from numpy import exp
from numpy import sqrt
from numpy import cos
from numpy import pi

N = 20 # gene size
P = 200 # population size
mutations = [0.1] # List of mutation rates
MIN = -32.0 # Lower Bound
MAX = 32.0  # Upper bound
steps = [1] # List of mutation steps

"""
class that create a individuals attributes
"""
```

```
class Individual:
    def __init__(self):
        self.gene = [0] * N  # initialise gene size
        self.fitness = 0  # initialise fitness


def single_point_crossover(offspring):
    for i in range(0, P, 2):
        off1 = copy.deepcopy(offspring[i]) # Copy two individuals
        off2 = copy.deepcopy(offspring[i + 1])
        temp = copy.deepcopy(offspring[i])
        crossover_point = random.randint(1, N) # generate crossover point in gene
        for j in range(crossover_point, N):
            off1.gene[j] = off2.gene[j]
            off2.gene[j] = temp.gene[j]
        off1.fitness = ackley(off1) # designate fitness
        off2.fitness = ackley(off2)
        offspring[i] = copy.deepcopy(off1)
        offspring[i + 1] = copy.deepcopy(off2)
    return offspring


def generate_genes():
    population = []
    # Create random population of genes
    for i in range(0, P):
        temp_gene = []  # List to hold a temp gene
        for j in range(0, N):
            temp_gene.append(random.uniform(MIN, MAX))  # appending a random value from
between the bounds
        new_ind = Individual()  # New instance of an individual
        new_ind.gene = temp_gene.copy()
        new_ind.fitness = ackley(new_ind)  # new individual is assigned a fitness value from
rosenbrock
        population.append(new_ind)  # new individual gets appended to the population

    return population


def tournament_selection(population):
    offspring = []

    for i in range(0, P):
        parent_1 = random.randint(0, P - 1)  # Generate 1st random integer in population
        off_1 = population[parent_1]  # Get random individual in population
        parent_2 = random.randint(0, P - 1)  # Generate 1st random integer in population
        off_2 = population[parent_2]  # Get random individual in population
        if off_1.fitness > off_2.fitness:  # Compete to get best
            offspring.append(off_2)
        else:
            offspring.append(off_1)

    return offspring  # Winner is returned for crossover


def RWS(population):
    # total fitness of initial pop
    total = 0
    for individual in population:
        total += abs(individual.fitness) # abs adapts RWS for negative values

    offspring = []

    for i in range(0, P):
        selection_point = random.uniform(0.0, total) # Generating crossover point
        count_total = 0
        j = 0
        while count_total <= selection_point:
            count_total += abs(population[j].fitness) # keep running total
            j += 1
            if (j == P):
                break
        offspring.append(copy.deepcopy(population[j - 1])) # Add the individual who got
selected by the wheel

    return offspring


def arithmetic_combination(parent_1, parent_2, cross_prob):
    child_1 = (cross_prob * parent_1) + (1 - cross_prob) * parent_2  # create child 1 from
parents
    child_2 = (cross_prob * parent_2) + (1 - cross_prob) * parent_1  # create child 2 from
parents

    return child_1, child_2


def simple_arithmetic_combination(offspring, cross_prob):
    for i in range(0, len(offspring), 2):
        parent_1 = offspring[i].gene  # Get offspring gene for parent 1
        parent_2 = offspring[i + 1].gene  # Get offspring gene for parent 1

        cross_point = random.randint(0, N - 1)  # Generate crossover point

        for j in range(cross_point, N):  # Get range of gene to change
            produce_child = arithmetic_combination(parent_1[j], parent_2[j], cross_prob)  # get
children
            parent_1[j] = produce_child[0]
            parent_2[j] = produce_child[1]
    return offspring
```

```
def mutation(offspring):
    for i in range(0, P):
        new_individual = Individual()  # New instance of individual
        new_individual.gene = []  # Clear genes
        for j in range(0, N):
            gene = offspring[i].gene[j]
            mutation_probability = random.random()  # Get mute prob

            if mutation_probability < MUTATION_RATE:  # condition to check if mutation
should occur
                alter = random.uniform(0.0, step)  # 0 1.0
                if random.choice([0, 1]) == 1:
                    offspring[i].gene[j] = offspring[i].gene[j] + alter  # alter gene
                    if offspring[i].gene[j] > MAX:
                        offspring[i].gene[j] = MAX  # alter gene value
                else:
                    offspring[i].gene[j] = offspring[i].gene[j] - alter  # alter gene value
                    if offspring[i].gene[j] < MIN:
                        offspring[i].gene[j] = MIN  # alter gene value

            new_individual.gene.append(gene)
        new_individual.fitness = ackley(new_individual)  # new
        offspring[i] = new_individual
    return offspring


# Ackely
def ackley(individual) -> float:
    # Ackley minimisation function
    fitness = 0
    a = 0
    b = 0
    # Execute loop part of equation
    for i in range(1, N):
        a += (individual.gene[i] ** 2)

        b += (cos(2 * pi * individual.gene[i]))
    # Calculate the first half for easier understanding
    part1 = -20 * exp(-0.2 * sqrt((1 / N) * a))
    # Calculate the 2nd half for easier understanding
    part2 = exp((1 / N) * b)
    # sum of the 2
    fitness = part1 - part2
    return fitness
```

```
def elitism(population, offspring):
    # Sorts the population in order of fitness
    population.sort(key=lambda individual: individual.fitness, reverse=True)
    # The best individual for minimisation is at the end of list
    bestIndividual = population[-1]
    # Take the offspring and overwrite population with offspring
    new_population = copy.deepcopy(offspring)

    # Sort the population in order of fitness again
    new_population.sort(key=lambda individual: individual.fitness, reverse=True)

    # Take the worst individual in the new population and overwrite it with the best from the
old pop
    new_population[0] = bestIndividual

    return new_population



def myGA(population, crossover):
    # Set the amount of generations
    generations = 300
    best_fitness = []
    mean_fitness_plot = []
    for i in range(generations):
        # Get offspring from tournament selection
        offspring = tournament_selection(population)
        # Get crossed over offspring
        offspring_crossover = crossover(offspring)
        # offspring_crossover = single_point_crossover(offspring, cross_prob)
        # Get mutated offspring
        offspring_mutated = mutation(offspring_crossover)
        # Apply elitism
        population = elitism(population, offspring_mutated)

        fitness = []
        for individual in population:
            fitness.append(individual.fitness)
        min_fitness = min(fitness)  # get the best fitness in population for generation

        # Allows to see via command line what the specific fitness are for each step
        if i == generations - 1:
            print("Mutation Rate:", str(MUTATION_RATE), f" | Step
Size:{MUTATION_STEP}", " | Fitness:", min_fitness)

        # mean_fitness = (sum(fitness) / P)  # optional get mean, not used

        best_fitness.append(min_fitness)
        # mean_fitness_plot.append(mean_fitness) # not in use

    return best_fitness
```

```
test_fit = []
for i in range(1):
    for rate in mutations:  # iterate through mutations
        for step in steps:  # iterate through mutation steps
            MUTATION_RATE = rate
            MUTATION_STEP = step
            best_fitness_data = myGA(generate_genes(), simple_arithmetic_combination) # GA
for crossover variation
            test_fit = myGA(generate_genes(), single_point_crossover) # GA for crossover
variation

    plt.plot(best_fitness_data, label='Simple Arithmetic')
    plt.plot(test_fit, label='Single Point')
    plt.title("Single vs Simple Arithmetic")
    plt.ylabel('Fitness')
    plt.xlabel('Generations')
    plt.legend()
    plt.show()
```