**Generational Genetic Algorithms**

**Abdulrahman Ali 21065685**

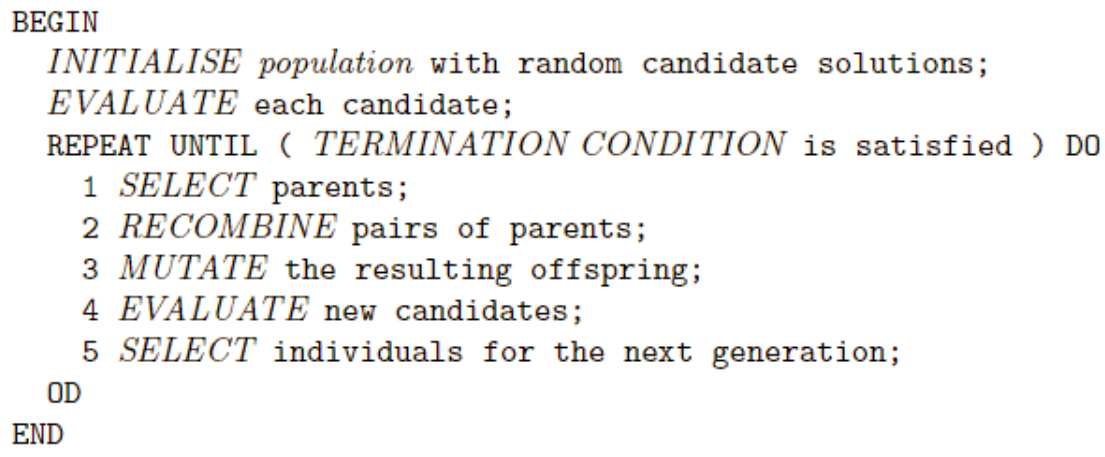
**1 INTRODUCTION**

This report explores the effectiveness of using evolutionary algorithms, specifically genetic algorithms (GAs), to solve a pair of minimisation problems. The aim is to solve the minimisation problems with the best efficiency possible using a set of different parameters, which include natural selection, crossover and mutation as the fundamental components of the used GA.

The following sections will dive deeper into GA’s and the abovementioned parameters regarding how they work, their effect on the performance of the GA, and how they contribute to the variance in optimality for the solutions obtained by the minimisation problems.

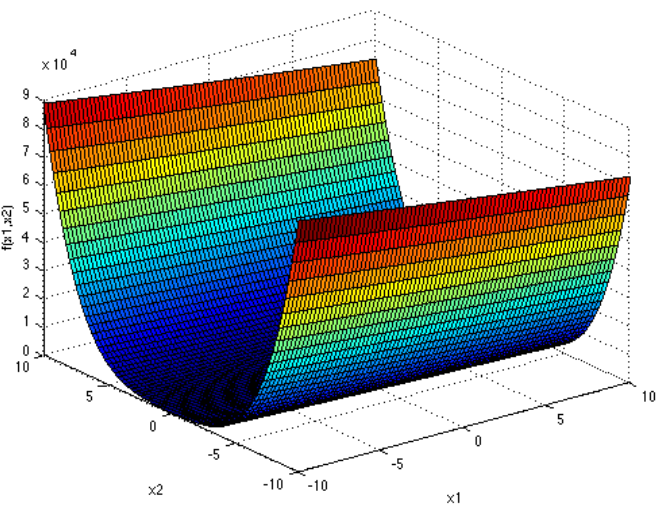
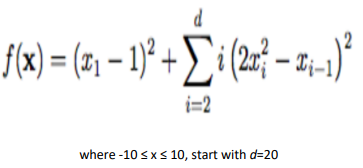
**2 EXPERIMENTATION**

2.1 Explaining the GA

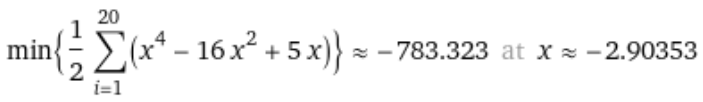
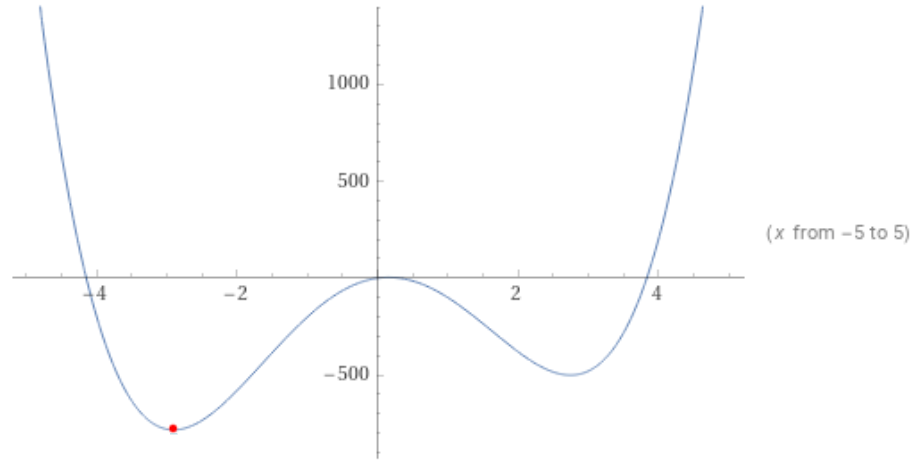
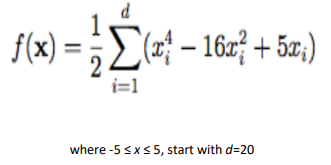


(Figure 1)

Figure 1 shows the pseudocode for this specific algorithm. To begin with we need to initialise the population using the parameters provided by the objective function (minimisation problem). The objective function provides you with the number of genes/bits in every individual within a population and the range of each of the genes. For this algorithm, two objective functions were provided as shown in Figure 2 & 3.



(Figure 2. First Objective Function and plot, where global minimum is 0)

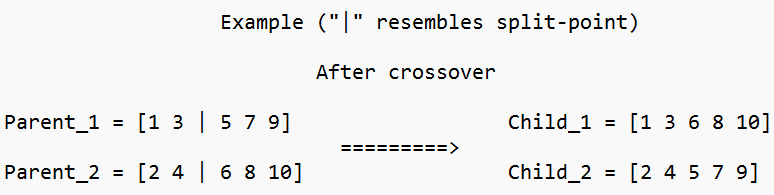


(Figure 3. Second Objective Function and Plot, where -738 is the global minimum)

Figures 2 and 3 both state that the the number of genes/bits (represented by *d*) per individual should be 20, where f(x) is the calculated fitness. However, they differ in range, where Fig. 2 has a value range between -10 to 10 in each gene, and Fig. 3 has a range of -5 to 5 in each gene. Each gene will have a random value that will be appended to each individual in the population. Afterwards, the function provides the fitness value to the individual passed through the parameters. The desired fitness would be the global minimum.

The next step is evaluation, a process that checks the fitness of each individual by passing the genes of each individual through the optimisation task (objective function). The preferred outcome would be the individual with the least fitness. Once a low fitness is found, the algorithm redefines the “best” individual and reduces the fitness range to look for an even better individual in a continuous loop until the entire generation has undergone this process.

After each candidate has undergone the evaluation process, the parents are then selected from the initial population based on their fitness. Once both parents are selected, the crossover function takes place (Figure 4).



(Figure 4)

Thereafter, bits/genes are altered to a random value or flipped, this is also known as mutation.. It should be noted that the crossover and mutation operators are both probabilistic, meaning they may or may not take place. This helps prevent premature convergence to undesired solutions and more importantly, maintain diversity, to prevent the algorithm from being trapped in a local minimum.

Next step involves evaluation of the new children produced by crossover and selecting them to prepare them for the next generation. The selection method picks the individuals with the highest fitness. Finally, the algorithm repeats for remaining number of generations until the an approximate of the global minimum is reached as the desired outcome.

RESULTS

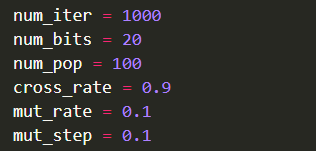
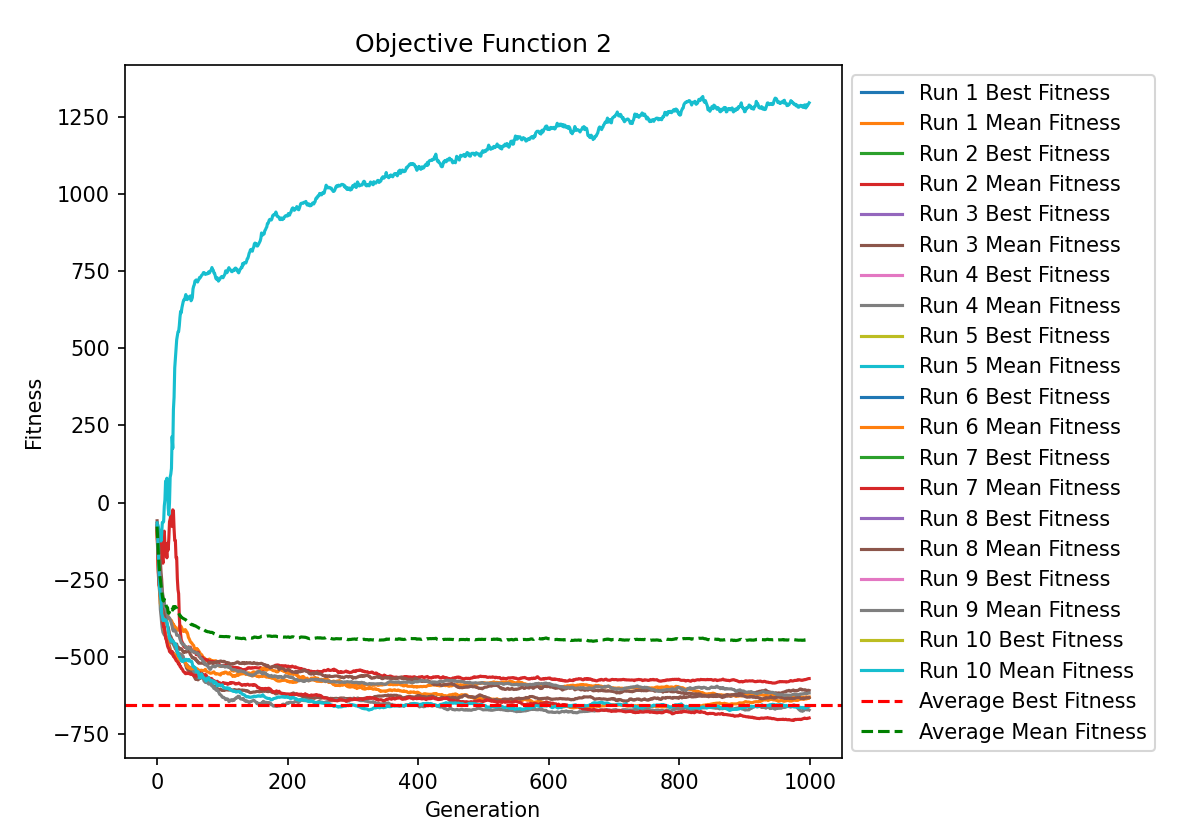
This section of the report will show the results when the algorithm is run and shows how the manipulation of parameters affects fitness average and efficiency of the algorithm. It consists of four runs on each minimisation fitness function (starting with the second one (Figure 3)). The first three runs will involve manually modifying the hyperparameters to see the effects of hyperparameter manipulation on the fitness result, and applying three different types of crossovers and two different types of selection. These include single-point, partially-mapped, and cycle crossovers, and tournament and roulette wheel selection. However, the last run will use a random search algorithm to find the best hyperparameters and methods to use. The random search algorithm will be explained later on in the comparison section.

HYPERPARAMETERS

The hyperparameters that were used were:

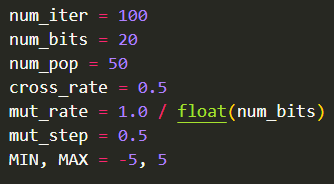
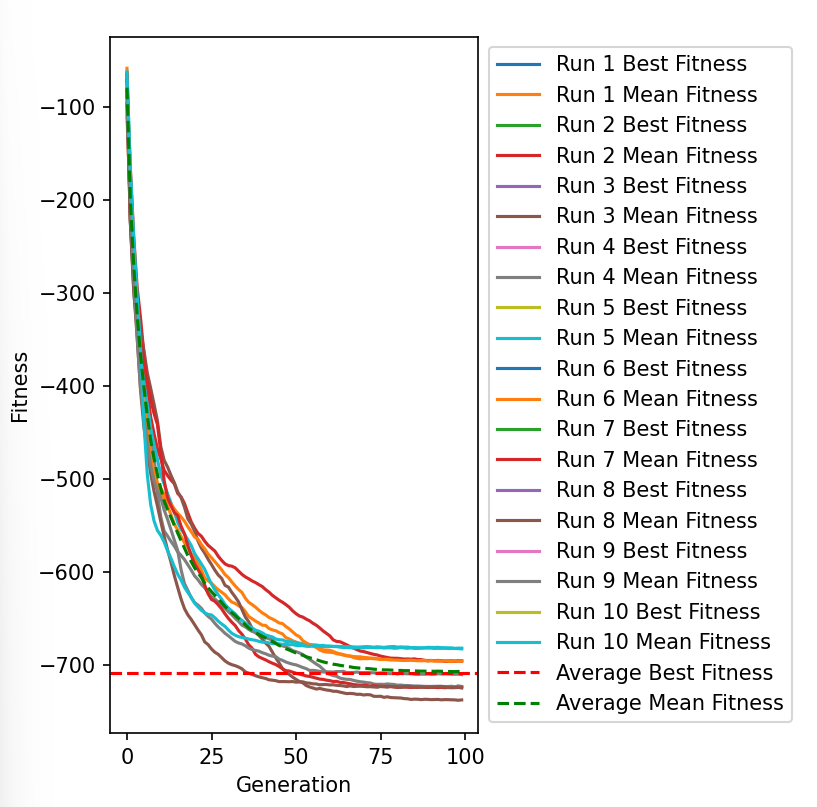
* num\_iter: the number of generations/iterations to run the algorithm. Generally speaking, a larger value will provide more accurate results.
* num\_bits: the number of genes/bits per individual. This value is fixed as the provided minimisation problems can only be solved with 20 bits per individual.
* num\_pop: the size of the population of individuals. A larger population can help find better solutions as it involves working with a large space. A smaller population can reach convergence faster, but can also hinder the ability to find the global minimum.
* cross\_rate: The probability of crossover occurring between bits of individuals represented as a value between 0 and 1. A higher cross-rate can better explore the problem during searching, while a lower value would focus refining current existing solutions.
* mut\_rate: similar to cross\_rate, it is the probability of mutation occurring on the bits/genes of an individual.
* mut\_step: the maximum amount by which a bit/gene can be mutated by.

SECOND MINIMISATION FUNCTION



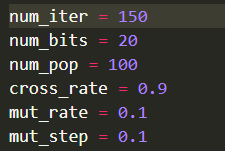
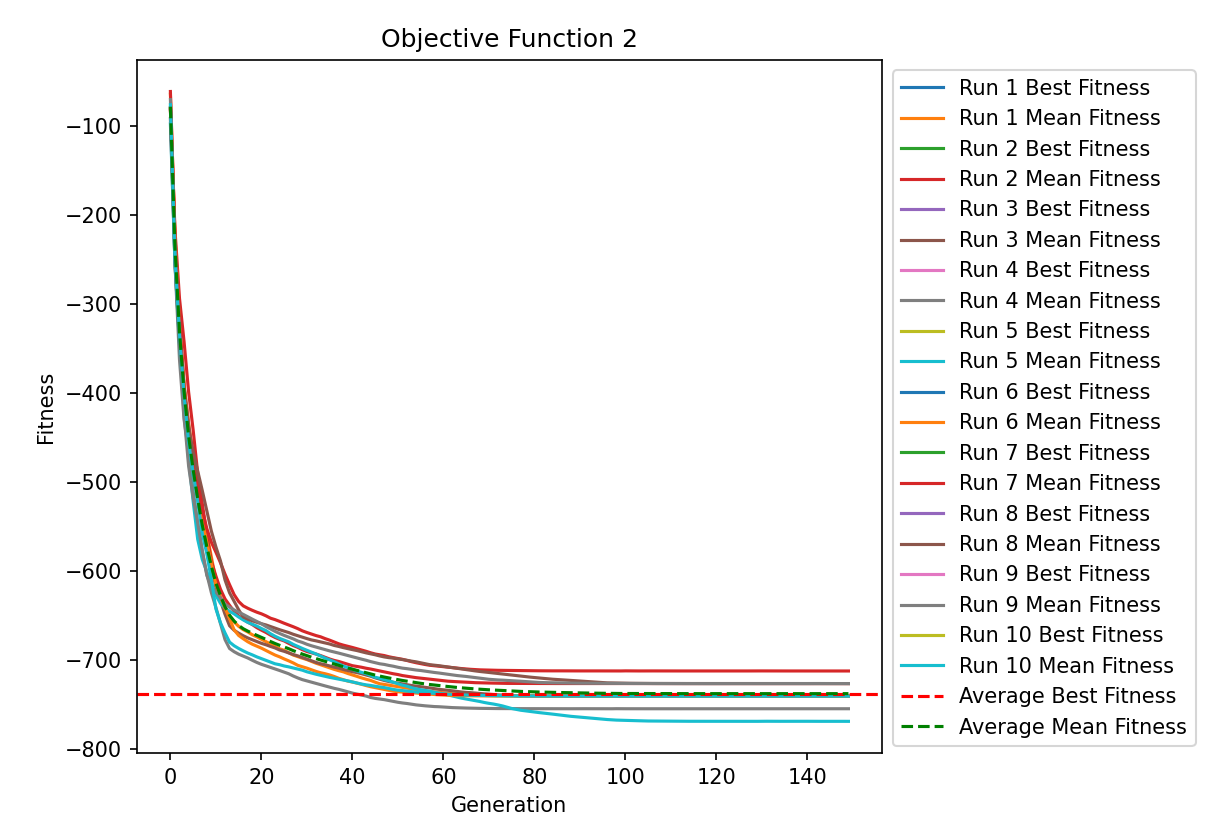
(Figures 5.1 & 5.2)

The operators used were cycle crossover and roulette selection. It can be seen in Fig 5.1 that there is one distinct outlier after changing the parameters. There could be multiple causes for this outlier including the stochastic nature of GA’s, variability of crossover and mutation operators and more importantly the probabilistic nature of the roulette-wheel selection method. The results had an average fitness of -657.6, which is sub-optimal.



(Figures 5.3 & 5.4)

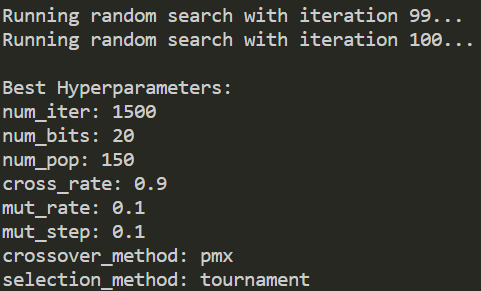
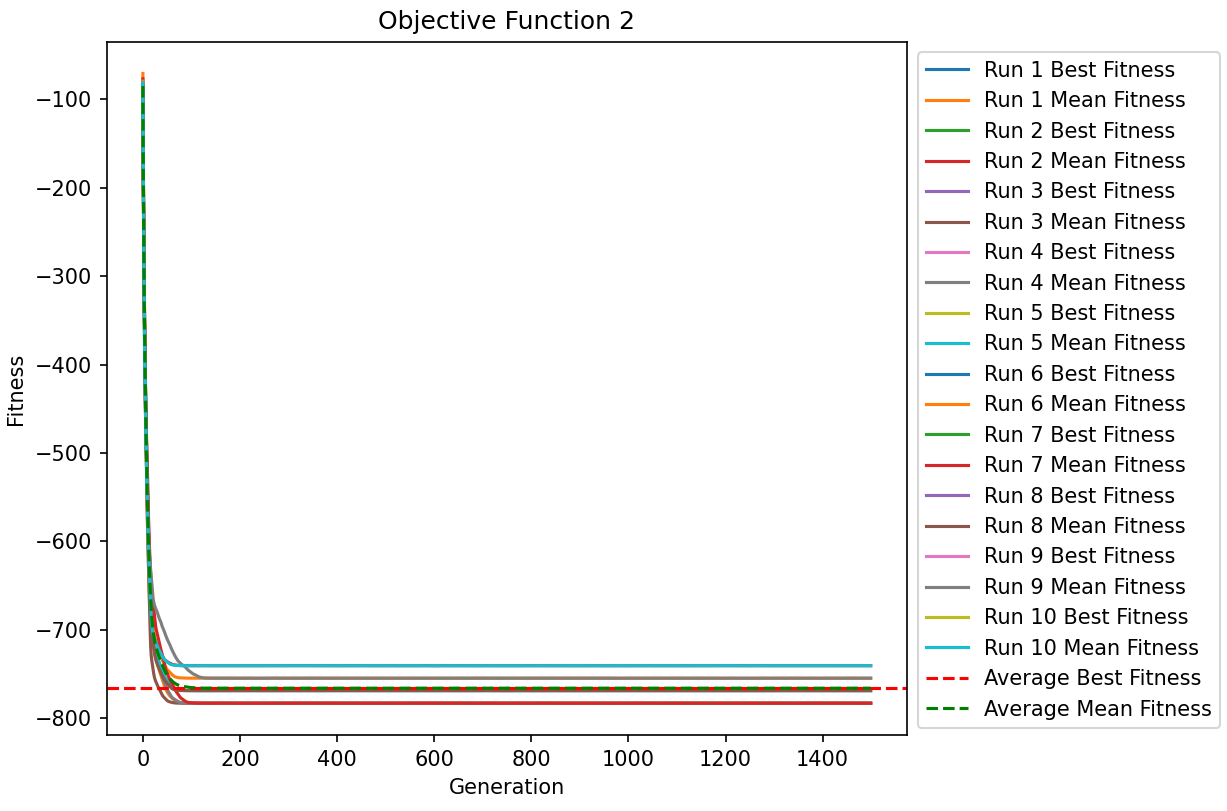
The values of the parameters used in figure 5.4 are quite basic since they are normally the starting point for parameter testing. Figure 5.3 used single-point crossover alongside tournament selection of size 3 over 100 generations. The average fitness was -708.8 which is very close to the global minimum. It can be seen by the graph that exceeding 100 generation for the current hyperparameters would be unbeneficial because the best fitness of the current parameters was always achieved at around 80 generations.



(Figures 5.5 & 5.6)

The parameters shown in Fig 5.6 have helped achieve an average fitness of -738.0, which is exactly the global minimum. The operators used were partially-mapped crossover and tournament selcetion of size 5.

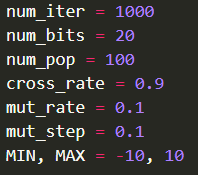
RANDOM SEARCH RESULTS



(Figures 5.7 & 5.8)

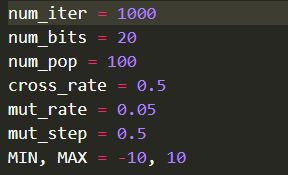
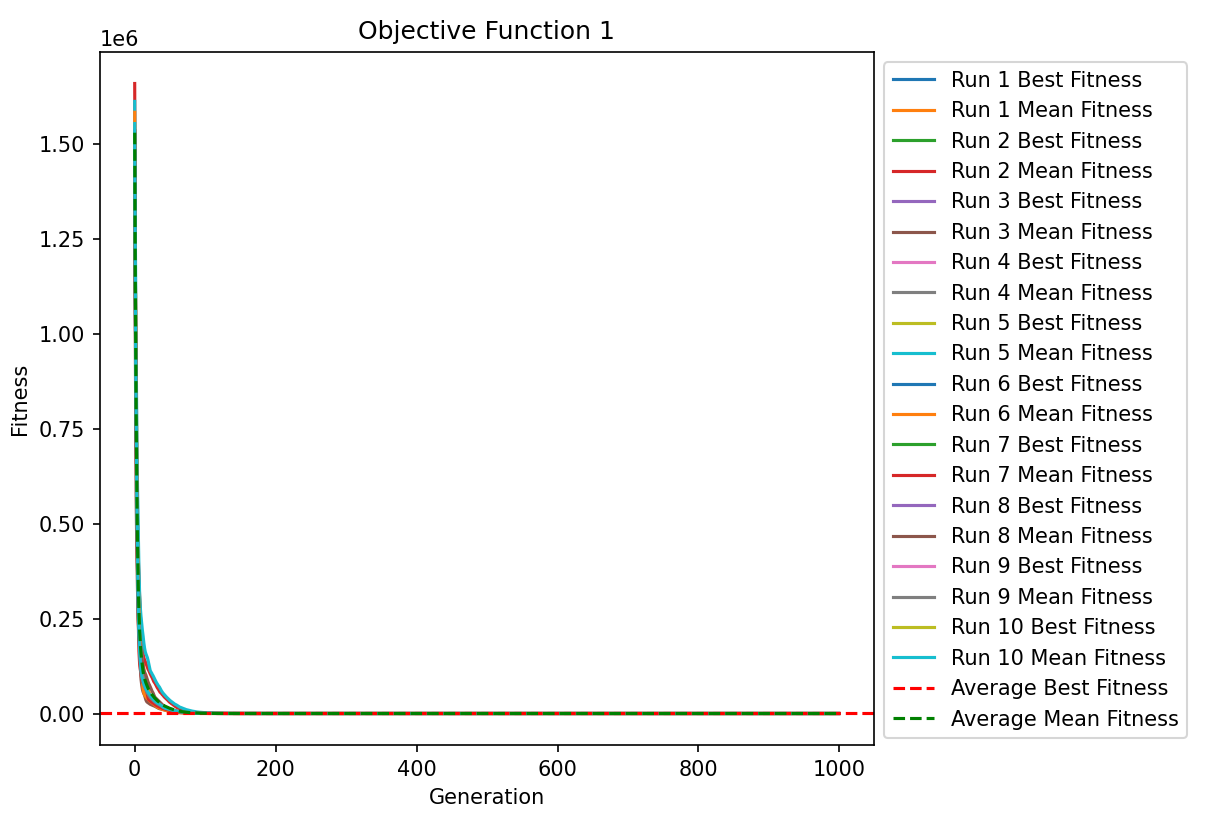
The hyperparameters provided by the random search algorithm were inefficient, the algorithm took ~5 minutes to run. The graph shows stability in fitness for the runs around the 150th generation, thus the other iterations were inefficient. The average fitness achieved was -766.3. The possible reasons for this were mentioned earlier, causing the random search to fail to solve the problem.

FIRST MINIMISATION FUNCTION



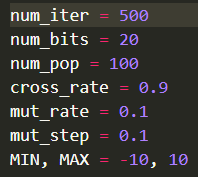
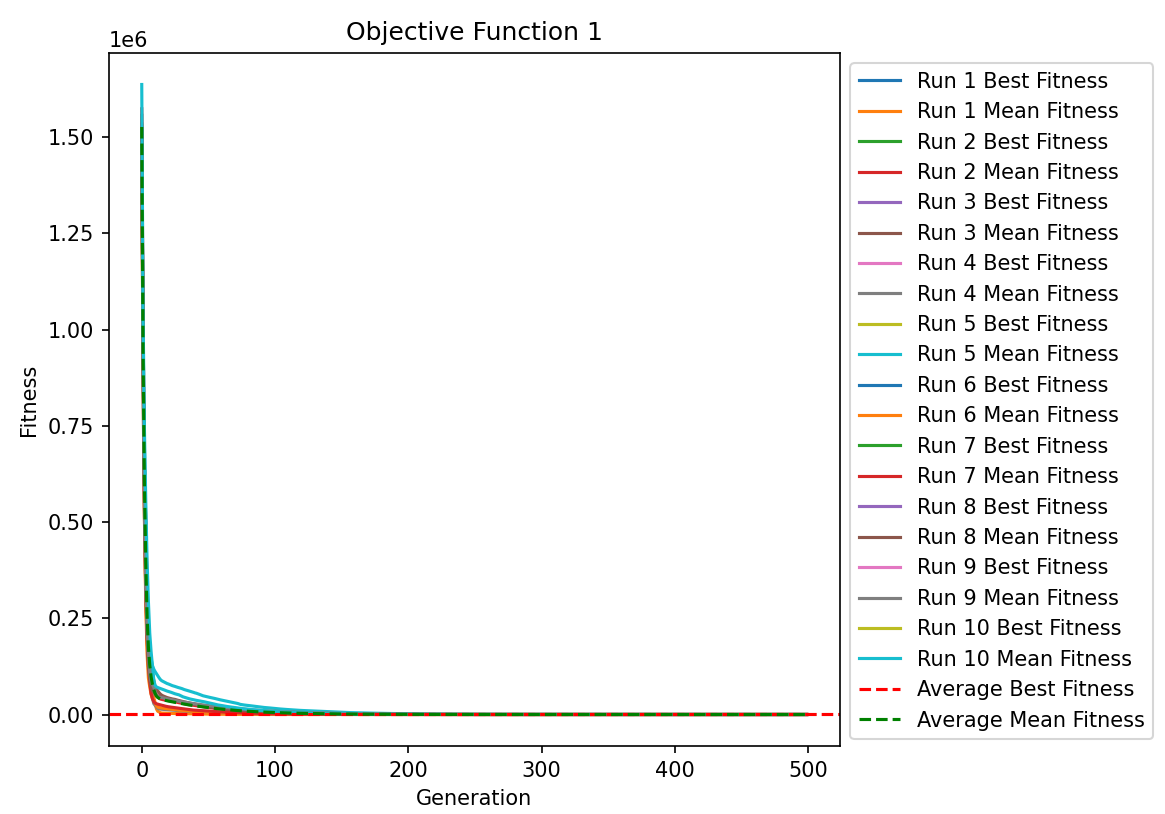
(Figures 6.1 & 6.2)

An undesirale average fitness of 442555.2 was achieved using cycle crossover and roulette-wheel selection with the parameters in Figure 6.2.



(Figures 6.3 & 6.4)

The results of Fig 6.3 are very optimal with an average fitness of 0.5. The reason this run has 1000 generations, unlike other runs, with a flat line of fitness after ~120 generations is because fitness value is very close to the global minimum, thus fine tuning the number of iterations will cause a significant yet small change in the fitness value. Tournament selection of size 3 and single-point crossover were the selected operators.



(Figures 6.5 & 6.6)

The last run used partially-mapped crossover with tournament selection of size 5. The average fitness was 1.0, achieving slighly worse than the single-point crossover run, however the result is still desirable compared to the initial run.

THE EFFECT OF PARAMETER MANIPULATION

The genetic algorithm that was used included several parameters/operators that could have a significant impact on the performance and optimisation of the algorithm.

The number of generations and individuals will always produce a better outcome if they are large in size. However, this comes at the cost of computational resources when in trade-off with the other parameters, increasing the memory usage and complexity.

CROSSOVER

The crossover functions, as mentioned earlier are probabilistic. The way this is implemented in code is by making sure that the crossover rate is larger than a randomly generated percentage when checking for fitness. This recombination causes there to be much more diversity when looking for the fittest individual since the purpose of the function is to try and find another bitstring that may prove to be the fittest, allowing the algorithm to explore more options and arrive at a much fitter individual than it would have, had the crossover rate been neglected. The three implemented crossover methods are single-point, partially-mapped and cycle crossover.

Partially-mapped crossover addresses single-point crossover’s limitations (Figure 4) by preserving the relative order of elements in parents while swapping segments between two randomly chosen crossover points, thus maintaining diversity and prevents duplicates, improving exploration. (Kora, P. and Yadlapalli, P. 2017).

Moreover, cycle crossover also preserves relative order and operates by selecting a random element from one parent and tracing its cycle of associations with corresponding elements in the other parent. This process creates diverse offspring by combining cycles from both parents, promoting exploration. (Kora, P. and Yadlapalli, P. 2017).

SELECTION

Tournament selection and roulette-wheel selection are the two selected selection methods in this GA. In simple terms, tournament selection randomly groups individuals into tournaments, with the best or worst individual chosen as a parent. It is simple and fair but may lack diversity. One the other hand, roulette-wheel selection assigns probabilities to individuals based on their fitness, providing a balance between exploration and exploitation. However, it can be biased towards stronger individuals.

3 COMPARISON

Random Search is a simple optimization technique that relies on random sampling to explore the search space. Unlike Genetic Algorithms (GA), which use a population-based approach and genetic operators, Random Search makes no use of problem-specific information or gradients. Instead, it randomly generates candidate solutions within the specified search space and evaluates their fitness based on the objective function. The process is repeated for a fixed number of iterations or until a termination condition is met. (C. I. Marrison, R. F. Stengel, 1997)

The GA demonstrated a faster convergence compared to the Random Search Algorithm. It achieved convergence in approximately 800 iterations, whereas the Random Search Algorithm required around 2000 iterations ie. generations, and 100 iterations. to reach a similar level of convergence. This highlights the efficiency of the GA in exploring the search space and converging towards promising solutions more rapidly. (Thierens, D., Goldberg, D. 1994)

Moreover, the GA outperformed the Random Search Algorithm in terms of solution quality. The GA obtained a final fitness value of -750, while the Random Search Algorithm resulted in a fitness value of -770. This signifies that the GA was successful in finding a better-quality solution closer to the global minimum, indicating its superior performance.

4 CONCLUSIONS

Generational genetic algorithms were concisely discussed and explained, and matters regarding the methods used were described while showing their significance in providing a desirable result for these kinds of algorithms.

And based on the comparative analysis, we conclude that the Genetic Algorithm is a more effective and reliable optimization technique for the given problem when compared to the Random Search Algorithm. Its ability to strike a balance between exploration and exploitation, along with its adaptability to problem-specific characteristics, makes it a valuable choice for solving complex optimization problems.

REFERENCES

* Kora, P. and Yadlapalli, P. (2017) Crossover Operators in Genetic Algorithms: A Review. International Journal of Computer Applications. 162 (10)
* C. I. Marrison and R. F. Stengel, "Robust control system design using random search and genetic algorithms," in *IEEE* *Transactions on Automatic Control*, vol. 42, no. 6, pp. 835-839, June 1997, doi: 10.1109/9.587338.
* Thierens, D., Goldberg, D. (1994). Convergence models of genetic algorithm selection schemes. In: Davidor, Y., Schwefel, HP., Männer, R. (eds) Parallel Problem Solving from Nature — PPSN III. PPSN 1994. Lecture Notes in Computer Science, vol 866. Springer, Berlin, Heidelberg. https://doi.org/10.1007/3-540-58484-6\_256

# APPENDIX

import matplotlib.pyplot as plt

from numpy.random import randint, rand, uniform

import numpy as np

from pprint import pprint

num\_iter = 500

num\_bits = 20

num\_pop = 100

cross\_rate = 0.9

mut\_rate = 0.1

mut\_step = 0.1

MIN, MAX = -10, 10

best\_ind = []

mean = []

# first minimization fitness function

def objective\_1(x):

    sum = (x[0] - 1) \*\* 2

    for i in range(1, num\_bits):

        sum += i \* (2 \* (x[i]) \*\* 2 - x[i - 1]) \*\* 2

    return sum

# second minimization fitness function

def objective\_2(x):

    sum = 0

    for i in range(num\_bits):

        sum += x[i] \*\* 4 - 16 \* (x[i]) \*\* 2 + 5 \* x[i]

    return sum \* 0.5

# Tournament Selection Function

def tournament\_selection(population, fitness, tournament\_size):

    selected = []

    for \_ in range(len(population)):

        tournament\_indices = np.random.choice(len(population), size=tournament\_size, replace=False)

        tournament\_fitness = [fitness[i] for i in tournament\_indices]

        winner\_index = tournament\_indices[np.argmin(tournament\_fitness)]

        selected.append(population[winner\_index])

    return selected

def roulette\_wheel\_selection(population, fitness):

    abs\_fitness = np.abs(fitness)

    total\_fitness = sum(abs\_fitness)

    probabilities = [fit / total\_fitness for fit in abs\_fitness]

    selected\_indices = np.random.choice(len(population), size=len(population), p=probabilities)

    selected\_pop = [population[i] for i in selected\_indices]

    return selected\_pop

def selection(population, fitness, selection\_method="tournament"):

    if selection\_method == "tournament":

        return tournament\_selection(population, fitness, tournament\_size=5)

    elif selection\_method == "roulette":

        return roulette\_wheel\_selection(population, fitness)

    else:

        raise ValueError("Invalid selection method")

def crossover(parent\_1, parent\_2, cross\_rate, crossover\_method="spx"):

    if crossover\_method == "spx":

        return sp\_crossover(parent\_1, parent\_2, cross\_rate)

    elif crossover\_method == "pmx":

        return pmx\_crossover(parent\_1, parent\_2, cross\_rate)

    elif crossover\_method == "cx":

        return cx\_crossover(parent\_1, parent\_2, cross\_rate)

    else:

        raise ValueError("Invalid crossover method")

def sp\_crossover(parent\_1, parent\_2, cross\_rate):

    # making children copies of parents in case crossover doesn't occur

    # check for recombination probability

    if rand() < cross\_rate:

        # select crossover point that is not on the end of the string

        cross\_point = randint(1, len(parent\_1) - 2)

        # placing split point from number generated from crosspoint

        child\_1 = parent\_1[:cross\_point] + parent\_2[cross\_point:]

        child\_2 = parent\_2[:cross\_point] + parent\_1[cross\_point:]

        for i in range(len(child\_1)):

            if child\_1[i] > MAX:

                child\_1[i] = MAX

            if child\_1[i] < MIN:

                child\_1[i] = MIN

            if child\_2[i] > MAX:

                child\_2[i] = MAX

            if child\_2[i] < MIN:

                child\_2[i] = MIN

    else:

        child\_1, child\_2 = parent\_1.copy(), parent\_2.copy()

    return [child\_1, child\_2]

def pmx\_crossover(parent\_1, parent\_2, cross\_rate, max\_iterations=100):

    if rand() < cross\_rate:

        cross\_point1 = randint(1, len(parent\_1) - 2)

        cross\_point2 = randint(cross\_point1 + 1, len(parent\_1) - 1)

        child\_1 = [-1] \* len(parent\_1)

        child\_2 = [-1] \* len(parent\_2)

        # Copy the middle segment from parents to children

        child\_1[cross\_point1:cross\_point2] = parent\_1[cross\_point1:cross\_point2]

        child\_2[cross\_point1:cross\_point2] = parent\_2[cross\_point1:cross\_point2]

        # Fill the remaining genes using PMX

        iteration = 0

        while iteration < max\_iterations:

            for i in range(cross\_point1, cross\_point2):

                if child\_1[i] == -1:

                    if parent\_1[i] not in child\_2:

                        child\_1[i] = parent\_1[i]

                    else:

                        idx = parent\_2.index(parent\_1[i])

                        while child\_1[idx] != -1:

                            idx = parent\_2.index(parent\_1[idx])

                        child\_1[idx] = parent\_1[i]

                if child\_2[i] == -1:

                    if parent\_2[i] not in child\_1:

                        child\_2[i] = parent\_2[i]

                    else:

                        idx = parent\_1.index(parent\_2[i])

                        while child\_2[idx] != -1:

                            idx = parent\_1.index(parent\_2[idx])

                        child\_2[idx] = parent\_2[i]

            iteration += 1

            # Check if all genes are filled

            if -1 not in child\_1 and -1 not in child\_2:

                break

        # Fill the remaining genes with the mapped values

        for i in range(len(parent\_1)):

            if child\_1[i] == -1:

                child\_1[i] = parent\_2[i]

            if child\_2[i] == -1:

                child\_2[i] = parent\_1[i]

        return [child\_1, child\_2]

    return [parent\_1, parent\_2]

def cx\_crossover(parent\_1, parent\_2, cross\_rate):

    if rand() < cross\_rate:

        child\_1 = [-1] \* len(parent\_1)

        child\_2 = [-1] \* len(parent\_2)

        cycle\_start = randint(len(parent\_1))

        idx = cycle\_start

        # Create a cycle

        while True:

            child\_1[idx] = parent\_1[idx]

            child\_2[idx] = parent\_2[idx]

            if parent\_1[idx] == parent\_2[cycle\_start]:

                break

            idx = parent\_1.index(parent\_2[idx]) if parent\_2[idx] in parent\_1 else None

            if idx is None:

                break

        # Fill the remaining genes using CX

        for i in range(len(parent\_1)):

            if child\_1[i] == -1:

                child\_1[i] = parent\_2[i]

            if child\_2[i] == -1:

                child\_2[i] = parent\_1[i]

        return [child\_1, child\_2]

    return [parent\_1, parent\_2]

def mutation(individual, mut\_rate, mut\_step, MIN, MAX):

    if isinstance(individual, list):

        # Mutation for a list of genes

        for i in range(len(individual)):

            # Check for mutation probability

            if rand() < mut\_rate:

                mutate = uniform(-mut\_step, mut\_step)  # Select how much the gene should mutate by

                mutated\_value = individual[i] + mutate

                individual[i] = min(max(mutated\_value, MIN), MAX)

                # Check that gene is not out of bounds

                if individual[i] > MAX:

                    individual[i] = MAX

                if individual[i] < MIN:

                    individual[i] = MIN

    else:

        if rand() < mut\_rate:

            mutate = uniform(-mut\_step, mut\_step)  # Select how much the gene should mutate by

            mutated\_value = individual + mutate

            # Check that mutated value is not out of bounds

            individual = min(max(mutated\_value, MIN), MAX)

# main GA function

def GA(objective\_1, num\_bits, num\_iter, num\_pop, cross\_rate, mut\_rate, mut\_step, crossover\_method, selection\_method):

    population = []

    for i in range(num\_pop):

        individual = [uniform(MIN, MAX) for \_ in range(num\_bits)]

        population.append(individual)  # Initialize pop with individuals filled with genes

    # Keep track of the best solution and run the objective function on the first index to use as a comparison

    best, best\_num = population[0], objective\_1(population[0])

    best\_ind.append((best, best\_num))  # Append the initial best individual to the list

    # Enumerate over multiple generations

    for \_ in range(num\_iter):

        # Evaluate candidates in the pop by using the objective func on each individual

        fitness = [objective\_1(individual) for individual in population]

        mean.append(sum(fitness) / num\_pop)  # Append the average fitness of every individual to plot on the graph

        # Check for a new best solution continuously

        for i in range(num\_pop):

            if fitness[i] < best\_num:

                best, best\_num = population[i], fitness[i]

                best\_ind.append((best, best\_num))  # Getting the best individuals' genes and fitness and appending to a list to plot on the graph

        # Select parents using Tournament Selection

        selected = selection(population, fitness, *selection\_method*=selection\_method)

        # Create the next generation of children

        children = *list*()

        for i in range(0, num\_pop, 2):

            # Get selected parents in pairs

            parent\_1, parent\_2 = selected[i], selected[i + 1]

            # Crossover & mutation (explained above)

            for c in crossover(parent\_1, parent\_2, cross\_rate, *crossover\_method*=crossover\_method):

                # Mutation

                if isinstance(c, *list*):

                    mutation(c, mut\_rate, mut\_step, MIN, MAX)

                else:

                    mutation(c, mut\_rate, mut\_step, MIN, MAX)

                # Store for the next gen to run the func on again to get an even better result

                children.append(c)

        # Replace population

        population = children

    return (best, best\_num)  # Returns the best individual with the best fitness

num\_runs = 10

# Lists to store best fitness and mean fitness for each generation across all runs

best\_fitness\_per\_run = []

mean\_fitness\_per\_run = []

for run in range(num\_runs):

    # Perform GA for objective\_1 with given hyperparameters

    \_, best\_fitness = GA(objective\_1, num\_bits, num\_iter, num\_pop, cross\_rate, mut\_rate, mut\_step,

*crossover\_method*="pmx", *selection\_method*="tournament")

    # Append results for this run

    best\_fitness\_per\_run.append(best\_fitness)

    mean\_fitness\_per\_run.append(mean[:])  # Append a copy of the 'mean' list

    # Clear the 'mean' list for the next run

    mean.clear()

# Calculate the average best fitness over all runs

average\_best\_fitness = sum(best\_fitness\_per\_run) / num\_runs

# Calculate the average mean fitness for each generation across all runs

average\_mean\_fitness = [sum(mean\_fitness\_per\_run[j][i] for j in range(num\_runs)) / num\_runs for i in range(num\_iter)]

# Print the average fitness of the 5 runs to the console

# pprint(mean\_fitness\_per\_run)

print("Average Best Fitness:", average\_best\_fitness)

# Plotting the averaged best fitness and mean fitness in one graph

plt.figure(figsize=(10, 6))

for i in range(num\_runs):

    plt.plot(best\_fitness\_per\_run[i], label=f"Run {i+1} Best Fitness")

    plt.plot(mean\_fitness\_per\_run[i], label=f"Run {i+1} Mean Fitness")

# Plotting the average fitness

plt.axhline(y=average\_best\_fitness, color='r', linestyle='--', label="Average Best Fitness")

plt.plot(average\_mean\_fitness, color='g', linestyle='--', label="Average Mean Fitness")

plt.title("Objective Function 1")

plt.xlabel("Generation")

plt.ylabel("Fitness")

plt.legend(loc="upper left", bbox\_to\_anchor=(1, 1))

plt.show()

RANDOM SEARCH CODE

from GA import \*

import random

grid\_num\_iter = [500, 1000, 1500]

grid\_num\_bits = [20]

grid\_num\_pop = [50, 100, 150]

grid\_cross\_rate = [0.5, 0.7, 0.9]

grid\_mut\_rate = [0.01, 0.1, 0.5]

grid\_mut\_step = [0.001, 0.01, 0.1]

grid\_crossover\_method = ["spx", "pmx", "cx"]

grid\_selection\_method = ["tournament", "roulette"]

# Perform random search

best\_fitness = float('inf')

best\_hyperparameters = None

total\_iterations = 100

for iteration in range(total\_iterations):

    # Randomly choose hyperparameters from the grid

    num\_iter = random.choice(grid\_num\_iter)

    num\_pop = random.choice(grid\_num\_pop)

    cross\_rate = random.choice(grid\_cross\_rate)

    mut\_rate = random.choice(grid\_mut\_rate)

    mut\_step = random.choice(grid\_mut\_step)

    crossover\_method = random.choice(grid\_crossover\_method)

    selection\_method = random.choice(grid\_selection\_method)

    print(*f*"Running random search with iteration {iteration + 1}...")

    \_, best\_num = GA(objective\_2, num\_bits, num\_iter, num\_pop, cross\_rate, mut\_rate, mut\_step, crossover\_method,

                     selection\_method)

    if best\_num < best\_fitness:

        best\_fitness = best\_num

        best\_hyperparameters = (num\_iter, num\_bits, num\_pop, cross\_rate, mut\_rate, mut\_step, crossover\_method,

                                selection\_method)

print("\nBest Hyperparameters:")

print("num\_iter:", best\_hyperparameters[0])

print("num\_bits:", best\_hyperparameters[1])

print("num\_pop:", best\_hyperparameters[2])

print("cross\_rate:", best\_hyperparameters[3])

print("mut\_rate:", best\_hyperparameters[4])

print("mut\_step:", best\_hyperparameters[5])

print("crossover\_method:", best\_hyperparameters[6])

print("selection\_method:", best\_hyperparameters[7])

# Run the GA again with the best hyperparameters and selection method and get the best individual and fitness

best, best\_num = GA(objective\_2, \*best\_hyperparameters[1:7], *crossover\_method*=best\_hyperparameters[6],

*selection\_method*=best\_hyperparameters[7])

print("\nBest Genes:", best)

print("Best Fitness:", best\_num)