CSE/IEE 598: BIO-INSPIRED AI AND OPTIMIZATION
MINI PROJECT 1

Tanmay Khandait
Student ID : 1219385830
CIDSE, Arizona State University
tkhandai@asu.edu

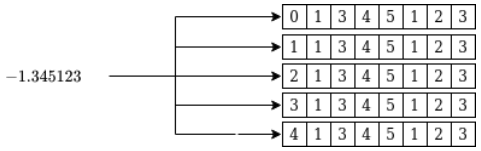
February 6, 2022

1 Answers to Question 1

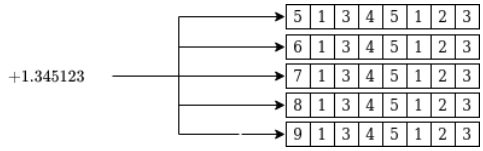
1.1 Answers to Question 1A

The genetic algorithm for this mini-project was implemented in the following fashion:

1. Encoding:
   The first step is to ask the user for accuracy/resolution. Ths accuracy is basically the number of digits after the decimal point. For ex, both 0.123 and -0.394 has an accuracy of 3, both 0.21939 and -4.59202 has an accuracy of 5, and both -1 and -2 has an accuracy of 0. Let us refer to accuracy with the variable N.
   Every phenotype is of length $N + 2$, 1 gene for representing the sign of the genotype, 1 gene for representing the integer part of the genotype and N genes for representing the genotype after the decimal point. We use the base 10 encoding, thus every gene in the chromosome can have values/alleles in the range $[0 - 9]$. For the sign gene, any value/allele in the range $[0 - 4]$ denotes a positive sign and $[5 - 9]$ represent a negative sign. Some examples of this encoding are shown in Figure 1.



(a) All possible Base-10 Encodings of a negative number



(b) All possible Base-10 Encodings of a positive number

Figure 1: The acccuracy of both the examples is 5. We therefore have a phenotype of length 7 with 5 representing the sign with 1 sting, integer part 1 gene, and decimal part with 5 genes.

2. Selection Algorithm
   The tournament selection algorithm was implemented. We choose 2 individuals at random and choose the individual with the best fitness value to survive in the next generation.
3. Reproduction Algorithm:
   For the reproduction Phase, out of a population of M individuals, we choose R elite individuals, which have the best fitness. These individuals are passed on to both the next generation and also for reproduction (crossover and mutation). This number is determined based on a ratio provided by the user, called the *elite_population_ratio*. This *elite_population_ratio* denoted by P_e can take values between 0 and 1, where 0 denotes no elite individuals and 1 denotes choosing all the individuals. All other values denote the proportion, for ex, if the population size is POP_SIZE and elite individual ratio is P_e, we choose $floor(P_e * POP\_SIZE)$ number of individuals are passed on.
   We then perform the crossover and mutation as follows.
   (a) Crossover
   We sample a number from $x = U(0, 1)$ and if this number is less than *crossover probability* denoted by P_c,

we perform a crossover, else we do not perform a crossover. A single point crossover was implemented with probability.

(b) Mutation
Every individual that had been through the crossover algorithm went through mutation. The mutation was performed based on the mutation probability denoted by P_m. For every gene in the phenotype, a random number was sampled from $x = U(0, 1)$ and if $P_m \leq x$, the gene was mutated with a random integer in the range $[0 - 9]$

4. Reducing Population
The population was reduced by keeping the fittest individuals from the new generation.

A basic framework has been shown in Figure 2. We terminate the algorithm based on the number of generation elapsed. In figure 2, R represent the population size at that phase. Assume that we start here with 20 individuals and $P_e = 0.4$. using tournament selection, we choose 20 individuals. Then $POP\_SIZE * P_e = 8$ individuals are chosen for reproduction and also passed on to next generation. These 8 individuals go through crossover to get 16 individuals and then mutation is performed on all of them. We now have a population of 24 individuals. The best 20 individuals are kept to repeat the algorithm. It should also be noted that solution outside the domain of the objective function are assigned a fitness of 0.

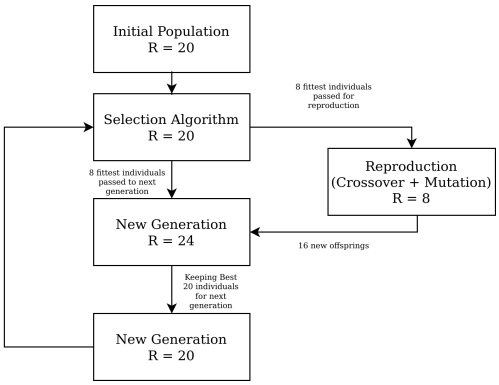
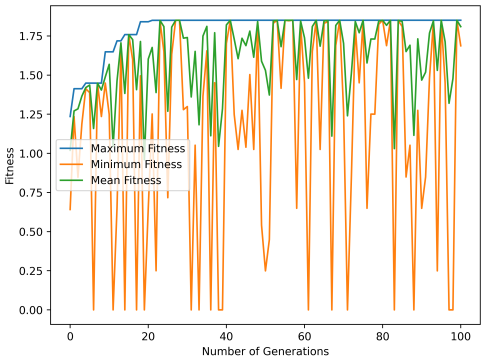


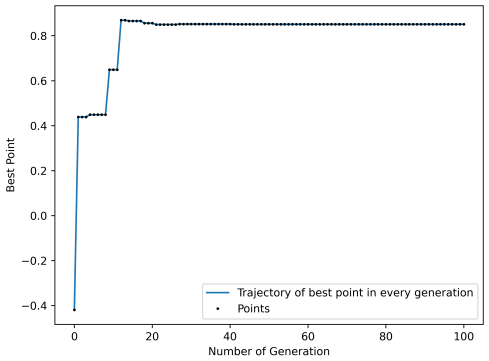
Figure 2: Basic implementation of GA.

1.2 Answer to Question 1B

For our experiment, we set accuracy to 3, to initial population size to 10, elite individual ratio $P_e = 0.4$, crossover probability $P_c = 0.3$ and mutation probability $P_m = 0.3$ and the number of generation to 100. The results from this experiment are shown in Figure 3



(a) Best, worst, and average fitness for each successive generation of the GA



(b) Best individual for each successive generation generation of the GA

Figure 3: Plots for Question 1B

1.3 Answer to Question 1C

From visual inspection in Figure 4, we can say the implementation was able to find the optima. I used scipy libraries to find the true optima and the true optima is at $x_{true} = 0.85118978$ and $f(x_{true}) = 1.85059524$. Using the base 10 encoding, the implementation gives us the solution $x_{base-10} = 0.851$ and $f(x_{base-10}) = 1.85058008$.

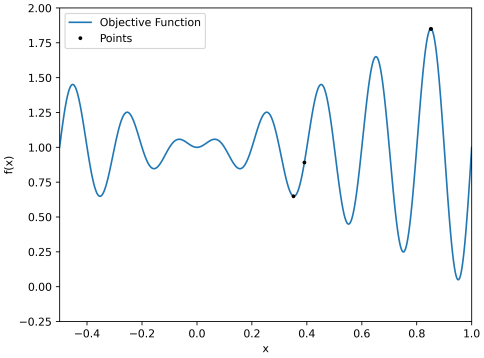


Figure 4: Plots for question 1C. The black dots represents the population after 100 generations. We can clearly see that 8/10 points have converged to the true maxima of the function. The maxima reported from the algorithm is found at $x = 0.851$ and $f(x) = 1.85058008$.

2 Answer to Question 2

In terms of convergence to the solution, both the algorithms converged to almost the same solution. However, there are subtle differences.

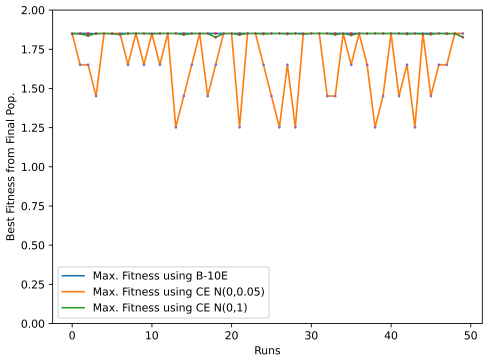
In comparison w.r.t. time, the continuous encoding (CE) performs better than base-10 encoding (B-10E) since there is only one gene as compared to five genes in B-10E. Since we perform only mutation in CE, this is an intuitive result.

In comparison to the quality of solution, I think the genetic algorithm implementation with B-10E performs better than CE. If we look at Table 1in the columns for B-10 encoding with noise from $\mathcal{N}(0, 0.05)$ the best solution is found by CE which is slightly better than the solution from B-10E. However, since mean of both mean and minimum fitness from population is higher and variance is lower for B-10E in comparison to CE, this encoding seems to be more robust in finding the better solution over the 50 runs. The scenario is completely reversed if we chose the noise for mutation from $\mathcal{N}(0, 1)$. The poor results for genetic algorithm with CE is because of high dependency on initial conditions and the parameters.

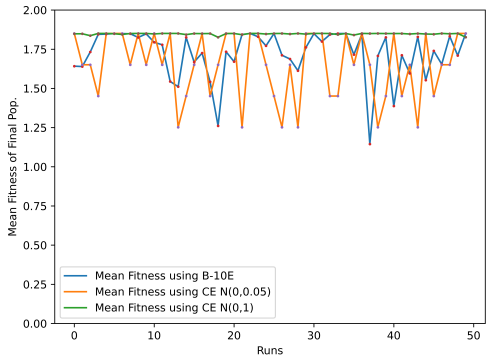
Therefore, based on the evidence from multiple runs, the genetic algorithm with base-10 encoding gives us a better solution mainly because of it's robustness to initial conditions, in comparison to continuous encoding. The above argument can also be visualized in Figure 5.

	Statistic over 50 generations	Base 10 encoding	Continuous Encoding $\mathcal{N}(0, 0.05)$	Continuous Encoding $\mathcal{N}(0, 1)$
Time	Mean	2.20266E-01	9.41929E-02	1.03454E-01
Max. of Final Population	Mean	1.85050E+00	1.65090E+00	1.84761E+00
	Variance	1.53600E-07	4.30234E-02	2.79425E-05
	Min.	1.84858E+00	1.25199E+00	1.82609E+00
	Max	**1.85058E+00**	**1.85060E+00**	**1.85059E+00**
Mean of Final Population	Mean	1.72378E+00	1.65090E+00	1.84761E+00
	Variance	2.34631E-02	4.30234E-02	2.79425E-05
	Min.	1.14452E+00	1.25199E+00	1.82609E+00
	Max	1.85058E+00	1.85060E+00	1.85059E+00
Min. of Final Population	Mean	1.20837E+00	1.65090E+00	1.84761E+00
	Variance	3.81625E-01	4.30234E-02	2.79425E-05
	Min.	0.00000E+00	0.00000E+00	1.82609E+00
	Max	1.85058E+00	1.85060E+00	1.85059E+00

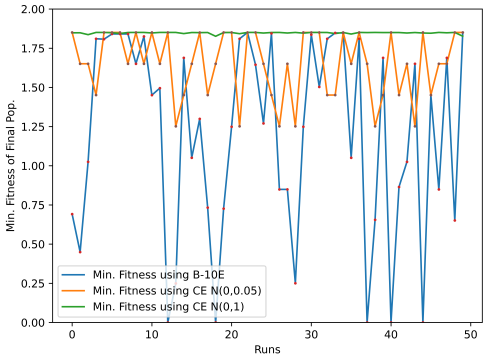
Table 1: The maximum fitness, mean fitness and minimum fitness of the final population is collected over 50 runs of the algorithm for 200 generations and the mean, variance, minimum and maximum of the final population over those 50 runs is shown here. The initial points were differently initialized for all the 50 runs of both the algorithms, but it was made sure that the initial values during a run were same for both the experiments. The numbers in bold indicate the best fitness found over all solution in the 50 runs.



(a) Best Fitness of final population from 50 runs



(b) Mean Fitness of final population from 50 runs



(c) Min. Fitness of final population from 50 runs

Figure 5: Plots for question 2

3 Answer to Question 3

3.1 Answer to Question 3a

The encoding is in the form x_1, x_2, x_3, x_4 and can be decoded as a floating point number $0.x_1x_2x_3x_4$. If the points are in vicinity of 0.001 of each other, any two individuals will have the same values for x_1 and x_2. Crossing two parents with such a structure will always create off-springs with same values of x_1 x_2 and the only difference will be in x_3 and x_4 and is basically equivalent to local search in that region.

Another way to look at it are that since the individuals are clustered around each other, two parents will exchange traits and since most of the parents are nearby, the exchange of traits will also lead to offsprings with similar traits, and no new offsprings will be generated.

3.2 Answer to Question 3b

The behavior of recombination is such that:

- If two parents are at different peaks, recombination of such parents will lead to a interchange of traits from both the parents and will lead to offsprings such that if they are healthy will survive during selection, and if are unhealthy will filter out during the selection process. This will be similar to the exploration phase of the algorithm.
- If two parents are nearby, they will crossover and create offsprings which have similar traits, and thus new solutions will be closer to each other. This is similar to exploitation phase of the algorithm.

This behavior is good behavior and is in a way, automatic way of performing exploration and exploitation. This will ensure that both exploration takes place in order to explore new solution from combination of possibly healthy traits from parents and exploit when these parents nearby to find better solutions in that region.

4 Appendix

4.1 Parameters for Base 10 Encoding and Continuous Encoding for question 2

		Question 1	Question 2	
		Base-10 Encoding	Base-10 Encoding	Continuous Encoding
Initial Population	-	10	10	10
Selection Algorithms	Type	Tournament Selection	Tournament Selection	Tournament Selection
Elitism Ratio	-	0.4	0.4	0.4
Crossover	Type	Single Point Crossover	Single Point Crossover	Single Point Crossover
	Crossover Probability	0.3	0.3	0
Mutation	Mutation Probability	0.3	0.3	0.3
	Noise mean	-	-	0
	Noise variance	-	-	0.05, 1
Number of Generations	-	100	200	200

Table 2: Parameters for Genetic Algorithm

4.2 Codes

4.2.1 Codes for Question 1

```
from dataclasses import replace
from platform import release
import random
import numpy as np
import warnings

class ga_options:
    def __init__(self, accuracy, bounds, fitness_function, initial_population_size, seed) -> None:
        self.accuracy = accuracy
        self.lower_bound, self.upper_bound = bounds
        self.encoding_vector = 10.0 ** (-1*np.array(range(0,accuracy+1)))
        self.fitness_function = fitness_function
        self.initial_population_size = initial_population_size
        self.rng = np.random.default_rng(seed)
        warnings.warn("Ensure fitness function is non-negative")

class chromosome():
    def __init__(self, phenotypes, ga_options) -> None:
        self.ga_options = ga_options
        assert isinstance(phenotypes, np.ndarray), "Phenotype should be a 1 dimensional Numpy.ndarray of
        assert all(isinstance(phenotype, str) for phenotype in phenotypes), "All Phenotype should be str
        for phenotype in phenotypes:
            # print(phenotype)
            assert len(phenotype) == (ga_options.accuracy+2), "Phenotype is of length {}. Expected lengt

        self.phenotypes = phenotypes


    def convert_phenotype_to_matrix(self, phenotypes):
        all_pheno_types = np.array([np.array(list(map(int,list(x)))) for x in phenotypes])
        return all_pheno_types

    def get_genotypes(self):
```

```
        phenotype_matrix = self.convert_phenotype_to_matrix(self.phenotypes)
        sign_allele = phenotype_matrix[:,0]
        sign = np.multiply(sign_allele >= 5, 1)
        sign[sign == 0] = -1
        other_alleles = phenotype_matrix[:,1:]
        real_val = np.round(sign * (other_alleles @ self.ga_options.encoding_vector.T),self.ga_options.a
        return real_val

    def get_fitness(self):
        genotypes = self.get_genotypes()
        fitness = np.array([self.ga_options.fitness_function(x) for x in genotypes])
        cond_1 = genotypes < self.ga_options.lower_bound
        cond_2 = genotypes > self.ga_options.upper_bound
        fitness[cond_1 | cond_2] = 0
        return fitness

    def add_phenotypes(self, new_phenotypes):
        self.phenotypes = np.hstack((self.phenotypes, new_phenotypes.phenotypes))

    def rank_by_fitness(self):
        fitness = self.get_fitness()
        return self.phenotypes[np.argsort(-1 * fitness)], fitness[np.argsort(-1 * fitness)]

    def get_chromosome_size(self):
        return self.phenotypes.shape[0]

    def reduce_population(self, number_needed):
        new_pop, _ = self.rank_by_fitness()
        return chromosome(new_pop[0:number_needed], self.ga_options)


import re
class geneticAlgorithmOperators:
    def __init__(self, ga_options):
        self.ga_options = ga_options

    def generate_legal_phenotypes(self):
        legal_phenotypes = self.ga_options.rng.uniform(self.ga_options.lower_bound, self.ga_options.uppe
        return chromosome(self.phenotype_to_chromosome(legal_phenotypes), self.ga_options)

    def phenotype_to_chromosome(self, phenotype):
        aes = str(abs(np.round(phenotype,self.ga_options.accuracy))).replace(".","").replace("[","").rep
        for iterate in range(len(phenotype)):
            if phenotype[iterate] >= 0:
                val = str(self.ga_options.rng.integers(5,10,1)[0])
            elif phenotype[iterate] < 0:
                val = str(self.ga_options.rng.integers(0,5,1)[0])
            aes[iterate] = val + aes[iterate]

            if len(aes[iterate]) < self.ga_options.accuracy+2:

                aes[iterate] += "0"*abs(len(aes[iterate])-self.ga_options.accuracy-2)

        return np.array(aes)

    def generate_random_population(self):
        legal_phenotypes = self.generate_legal_phenotypes()
        return legal_phenotypes
```

```
def tournament_selection(self, current_population, selection_algorithm_options):
    num_players = selection_algorithm_options["num_players"]
    number_of_offsprings = selection_algorithm_options["number_of_offsprings"]

    fitness = current_population.get_fitness()
    offsprings = []
    for _ in range(number_of_offsprings):
        select_index = self.ga_options.rng.choice(len(fitness), size = num_players, replace = False)
        offsprings.append(current_population.phenotypes[select_index[np.argmax(fitness[select_index]

    return chromosome(np.array(offsprings), self.ga_options)

def selection(self, current_population, selection_algorithm_options):
    if selection_algorithm_options["type"] == "tournament":
        new_offsprings = self.tournament_selection(current_population, selection_algorithm_options)
    return new_offsprings, current_population

def single_point_crossover(self, elite_children_object, number_of_offsprings):
    elite_children_phenotypes = elite_children_object.phenotypes
    elite_children_fitness = elite_children_object.get_fitness()
    crossover_offsprings = []
    for _ in range(number_of_offsprings):
        select_index = self.ga_options.rng.choice(len(elite_children_fitness), size = 2, replace = F
        parent_1 = elite_children_phenotypes[select_index[0]]
        parent_2 = elite_children_phenotypes[select_index[1]]
        coin_toss = self.ga_options.rng.uniform(0,1,1)[0]

        if coin_toss < reproduce_options["crossover_probablity"]:
            point = self.ga_options.rng.choice((self.ga_options.accuracy+2), size = 1, replace=False
            offspring_1 = parent_1[0:point] + parent_2[point:]
            offspring_2 = parent_2[0:point] + parent_1[point:]

        else:
            offspring_1 = parent_1
            offspring_2 = parent_2


        crossover_offsprings.append(offspring_1)
        crossover_offsprings.append(offspring_2)

    return chromosome(np.array(crossover_offsprings), self.ga_options)


def mutation(self, new_population, reproduce_options):
    population_pheno = new_population.phenotypes

    mutated_offsprings = []
    for iterate in range(population_pheno.shape[0]):
        enc_string = population_pheno[iterate]
        temp = list(enc_string)
        coin_toss = self.ga_options.rng.uniform(0,1,len(enc_string))
        mutation_bits = self.ga_options.rng.integers(0,10, size = len(enc_string))
        for iterate_2 in range(len(enc_string)):
            if coin_toss[iterate_2] < reproduce_options["mutation_probablity"]:

                temp[iterate_2] = str(mutation_bits[iterate_2])


        mutated_os = "".join(temp)
```

```
            mutated_offsprings.append(mutated_os)


        return chromosome(np.array(mutated_offsprings), self.ga_options)

    def reproduce(self, current_population, reproduce_options):
        num_pop = current_population.get_chromosome_size()
        ranked_fit_individuals, _ = current_population.rank_by_fitness()
        elite_children_count = int(np.floor(num_pop * reproduce_options["prob_elite_children"]))
        if elite_children_count < 2:
            raise Exception("Elite children count is {}. Try increasing the prob_elite_children paramete
        elite_children = chromosome(ranked_fit_individuals[0:elite_children_count], self.ga_options)
        number_of_offsprings = num_pop - elite_children_count
        crossover_offsprings = self.single_point_crossover(elite_children, number_of_offsprings)
        mutated_offsprings = self.mutation(crossover_offsprings, reproduce_options)
        elite_children.add_phenotypes(mutated_offsprings)


        return elite_children


def objective_function(x):
    return (x * np.sin(10*np.pi*x) + 1)

import time
def run_ga(num_generations, ga_options, selection_algorithm_options, reproduce_options):


    x_genotype = []
    corres_y = []

    ga = geneticAlgorithmOperators(ga_options)

    pop_1 = ga.generate_legal_phenotypes()
    fitness = pop_1.get_fitness()

    x_genotype.append(pop_1.get_genotypes())
    corres_y.append(fitness)
    t0 = time.clock()
    for _ in range(num_generations):
        selected_pop,_ = ga.selection(pop_1, selection_algorithm_options)
        new_pop_1 = ga.reproduce(selected_pop, reproduce_options)
        pop_1 = new_pop_1.reduce_population(options.initial_population_size)
        fitness = pop_1.get_fitness()
        x_genotype.append(pop_1.get_genotypes())
        corres_y.append(fitness)
    time_elapsed = time.clock() - t0
    population = pop_1
    return population, x_genotype, corres_y, time_elapsed



x_history = []
y_history = []
time_history = []
for j in range(50):

    bounds = (-0.5,1)
    accuracy = 3
```

```
    seed = 123 + j
    print(seed)
    initial_pop_size = 10
    num_generations = 100

    options = ga_options(accuracy, bounds, objective_function, initial_pop_size, seed)

    reproduce_options = {"crossover_probablity":0.3, "prob_elite_children":0.4, "mutation_probablity":0.

    selection_algorithm_options = {"type":"tournament", "num_players":2, "number_of_offsprings":initial_

    pop_1, x_genotype, corres_y, time_elapsed_indi = run_ga(num_generations, options, selection_algorith
    x_history.append(x_genotype)
    y_history.append(corres_y)
    time_history.append(time_elapsed_indi)

x_history = np.array(x_history)
y_history = np.array(y_history)
time_history = np.array(time_history)

print(x_history.shape)
print(y_history.shape)

import matplotlib.pyplot as plt

for iterate in range(1):
    plt.plot(np.max(y_history[iterate,:,:],1), label = "Maximum Fitness")
    plt.plot(np.min(y_history[iterate,:,:],1), label = "Minimum Fitness")
    plt.plot(np.mean(y_history[iterate,:,:],1), label = "Mean Fitness")
    plt.xlabel("Number of Generations")
    plt.ylabel("Fitness")

plt.legend()
plt.tight_layout()
plt.savefig("min_max_mean_q1.png", dpi = 500, format = 'png')
# plt.show()

from matplotlib import animation
import matplotlib.pyplot as plt
fig = plt.figure()
ax = plt.axes(xlim=(-0.5,1.), ylim=(-0.25, 2))
x = np.linspace(-0.5,1,10000)
y = objective_function(x)
line_1, = ax.plot(x,y, label = "Objective Function")
ax.plot(x_history[0,-1,:], y_history[0,-1,:],".k", markersize = 5, label = "Points")
ax.set_xlabel("x")
ax.set_ylabel("f(x)")
ax.legend()
fig.tight_layout()
plt.savefig("Evalute_best_point_q1.png", dpi = 500, format = 'png')
# plt.show()


fig = plt.figure()
ax = plt.axes()

x_best = []
for iter, y in enumerate(y_history[0,:,:]):
    ind = np.argmax(y)
```

```
    x_best.append(x_history[0,iter,ind])

ax.plot(x_best, label = "Trajectory of best point in every generation")
ax.plot(x_best, "ok", markersize = 1.5, label = "Points")
ax.set_xlabel("Number of Generation")
ax.set_ylabel("Best Point")
ax.legend()
fig.tight_layout()
plt.savefig("point_trajectory_q1.png", dpi = 500, format = 'png')


import pickle

with open("question_1.pickle", "wb") as output_file:
    pickle.dump((x_history, y_history, time_history), output_file)
```

4.2.2 Codes for Question 2

```
from dataclasses import replace
import numpy as np
import warnings
from rsa import sign

class ga_options:
    def __init__(self, accuracy, bounds, fitness_function, initial_population_size, seed) -> None:
        self.accuracy = accuracy
        self.lower_bound, self.upper_bound = bounds
        self.encoding_vector = 10.0 ** (-1*np.array(range(0,accuracy+1)))
        self.fitness_function = fitness_function
        self.initial_population_size = initial_population_size
        self.rng = np.random.default_rng(seed)
        warnings.warn("Ensure fitness function is non-negative")

class chromosome():
    def __init__(self, phenotypes, ga_options) -> None:
        self.ga_options = ga_options
        assert isinstance(phenotypes, np.ndarray), "Phenotype should be a 1 dimensional Numpy.ndarray of
        assert all(isinstance(phenotype, float) for phenotype in phenotypes), "All Phenotype should be s
        self.phenotypes = phenotypes
        # self.phenotype_matrix =

    def convert_phenotype_to_matrix(self, phenotypes):
        all_pheno_types = np.array([np.array(list(map(int,list(x)))) for x in phenotypes])
        return all_pheno_types

    def get_genotypes(self):
        real_val = self.phenotypes
        return real_val

    def get_fitness(self):
        genotypes = self.get_genotypes()
        fitness = np.array([self.ga_options.fitness_function(x) for x in genotypes])
        cond_1 = genotypes < self.ga_options.lower_bound
        cond_2 = genotypes > self.ga_options.upper_bound
        fitness[cond_1 | cond_2] = 0
        return fitness

    def add_phenotypes(self, new_phenotypes):
        self.phenotypes = np.hstack((self.phenotypes, new_phenotypes.phenotypes))
```

```
    def rank_by_fitness(self):
        fitness = self.get_fitness()
        return self.phenotypes[np.argsort(-1 * fitness)], fitness[np.argsort(-1 * fitness)]

    def get_chromosome_size(self):
        return self.phenotypes.shape[0]

    def reduce_population(self, number_needed):
        new_pop, _ = self.rank_by_fitness()
        return chromosome(new_pop[0:number_needed], self.ga_options)


import re
class geneticAlgorithmOperators:
    def __init__(self, ga_options):
        self.ga_options = ga_options

    def generate_legal_phenotypes(self):
        legal_phenotypes = self.ga_options.rng.uniform(self.ga_options.lower_bound, self.ga_options.uppe
        return chromosome(legal_phenotypes, self.ga_options)

    def generate_random_population(self):
        legal_phenotypes = self.generate_legal_phenotypes()
        return legal_phenotypes

    def tournament_selection(self, current_population, selection_algorithm_options):
        num_players = selection_algorithm_options["num_players"]
        number_of_offsprings = selection_algorithm_options["number_of_offsprings"]

        fitness = current_population.get_fitness()
        offsprings = []
        for _ in range(number_of_offsprings):
            select_index = self.ga_options.rng.choice(len(fitness), size = num_players, replace = False)
            offsprings.append(current_population.phenotypes[select_index[np.argmax(fitness[select_index]

        return chromosome(np.array(offsprings), self.ga_options)

    def selection(self, current_population, selection_algorithm_options):
        if selection_algorithm_options["type"] == "tournament":
            # print("Playing Tournament")
            new_offsprings = self.tournament_selection(current_population, selection_algorithm_options)
        return new_offsprings, current_population

    def single_point_crossover(self, elite_children_object, number_of_offsprings, reproduce_options):
        elite_children_phenotypes = elite_children_object.phenotypes
        elite_children_fitness = elite_children_object.get_fitness()
        crossover_offsprings = []
        for _ in range(number_of_offsprings):
            select_index = self.ga_options.rng.choice(len(elite_children_fitness), size = 2, replace = F
            parent_1 = elite_children_phenotypes[select_index[0]]
            parent_2 = elite_children_phenotypes[select_index[1]]

            coin_toss = self.ga_options.rng.uniform(0,1,1)[0]

            if coin_toss < reproduce_options["crossover_probablity"]:
                point = self.ga_options.rng.choice((self.ga_options.accuracy+2), size = 1, replace=False
                # print(point)
                offspring_1 = parent_1[0:point] + parent_2[point:]
```

```
                offspring_2 = parent_2[0:point] + parent_1[point:]

            else:
                # print("Not Crossing")
                offspring_1 = parent_1
                offspring_2 = parent_2


            crossover_offsprings.append(offspring_1)
            crossover_offsprings.append(offspring_2)


        return chromosome(np.array(crossover_offsprings), self.ga_options)


    def mutation(self, new_population, reproduce_options):
        population_pheno = new_population.phenotypes
        noise_mean = reproduce_options["mutation_noise_mean"]
        noise_std = reproduce_options["mutation_noise_std"]
        mut = self.ga_options.rng.normal(noise_mean, noise_std, size = population_pheno.shape[0])

        coin_toss = self.ga_options.rng.uniform(0,1,population_pheno.shape[0])
        mutated_offsprings = []
        for iterate, prob in enumerate(coin_toss):
            if prob < reproduce_options["mutation_probablity"]:

                mutated_offsprings.append(population_pheno[iterate] + mut[iterate])
            else:
                mutated_offsprings.append(population_pheno[iterate])


        return chromosome(np.array(mutated_offsprings), self.ga_options)

    def reproduce(self, current_population, reproduce_options):
        num_pop = current_population.get_chromosome_size()
        ranked_fit_individuals, _ = current_population.rank_by_fitness()
        elite_children_count = int(np.floor(num_pop * reproduce_options["prob_elite_children"]))
        # print(elite_children_count)
        if elite_children_count < 2:
            raise Exception("Elite children count is {}. Try increasing the prob_elite_children paramete

        elite_children = chromosome(ranked_fit_individuals[0:elite_children_count], self.ga_options)

        number_of_offsprings = num_pop - elite_children_count
        crossover_offsprings = self.single_point_crossover(elite_children, number_of_offsprings, reprodu
        mutated_offsprings = self.mutation(crossover_offsprings, reproduce_options)
        # print(mutated_offsprings.phenotypes.shape)
        elite_children.add_phenotypes(mutated_offsprings)


        return elite_children


def objective_function(x):
    return (x * np.sin(10*np.pi*x) + 1)

import time
def run_ga(num_generations, ga_options, selection_algorithm_options, reproduce_options):
```

```
    x_genotype = []
    corres_y = []

    ga = geneticAlgorithmOperators(ga_options)

    pop_1 = ga.generate_legal_phenotypes()
    fitness = pop_1.get_fitness()

    x_genotype.append(pop_1.get_genotypes())
    corres_y.append(fitness)
    t0 = time.clock()
    for _ in range(num_generations):
        selected_pop,_ = ga.selection(pop_1, selection_algorithm_options)
        new_pop_1 = ga.reproduce(selected_pop, reproduce_options)

        pop_1 = new_pop_1.reduce_population(options.initial_population_size)
        fitness = pop_1.get_fitness()
        x_genotype.append(pop_1.get_genotypes())
        corres_y.append(fitness)
    time_elapsed = time.clock()-t0
    population = pop_1
    return population, np.array(x_genotype), np.array(corres_y), time_elapsed



x_history = []
y_history = []
time_history = []
num_runs = 1
for j in range(50):
    seed = 123 + j
    print(seed)
    bounds = (-0.5,1)
    accuracy = 5

    initial_pop_size = 10
    num_generations = 100

    options = ga_options(accuracy, bounds, objective_function, initial_pop_size, seed)

    reproduce_options = {"crossover_probablity":0, "prob_elite_children":0.4, "mutation_probablity":0.3,

    selection_algorithm_options = {"type":"tournament", "num_players":2, "number_of_offsprings":initial_

    pop_1, x_genotype, corres_y, time_elapsed_indi = run_ga(num_generations, options, selection_algorith
    # print(corres_y)
    x_history.append(x_genotype)
    y_history.append(corres_y)
    time_history.append(time_elapsed_indi)

x_history = np.array(x_history)
y_history = np.array(y_history)
time_history = np.array(time_history)



import matplotlib.pyplot as plt
```

```
for iterate in range(1):
    plt.plot(np.max(y_history[iterate,:,:],1), label = "Maximum Fitness")
    plt.plot(np.min(y_history[iterate,:,:],1), label = "Minimum Fitness")
    plt.plot(np.mean(y_history[iterate,:,:],1), label = "Mean Fitness")
    plt.xlabel("Number of Generations")
    plt.ylabel("Fitness")

plt.legend()
plt.tight_layout()
plt.savefig("min_max_mean_q2.png", dpi = 500, format = 'png')

from matplotlib import animation
import matplotlib.pyplot as plt
fig = plt.figure()
ax = plt.axes(xlim=(-0.5,1.), ylim=(-0.25, 2))
x = np.linspace(-0.5,1,10000)
y = objective_function(x)
line_1, = ax.plot(x,y, label = "Objective Function")
ax.plot(x_history[0,-1,:], y_history[0,-1,:],".k", markersize = 5, label = "Points")
ax.set_xlabel("x")
ax.set_ylabel("f(x)")
ax.legend()
fig.tight_layout()
plt.savefig("Evalute_best_point_q2.png", dpi = 500, format = 'png')


fig = plt.figure()
ax = plt.axes()

x_best = []
for iter, y in enumerate(y_history[0,:,:]):
    ind = np.argmax(y)
    x_best.append(x_history[0,iter,ind])

ax.plot(x_best, label = "Trajectory of best point in every generation")
ax.plot(x_best, "ok", markersize = 5, label = "Points")
ax.set_xlabel("Number of Generation")
ax.set_ylabel("Best Point")
ax.legend()
fig.tight_layout()
plt.savefig("point_trajectory_q2.png", dpi = 500, format = 'png')

import pickle
with open("question_2.pickle", "wb") as output_file:
    pickle.dump((x_history, y_history, time_history), output_file)
```

Code for collecting data and generate plots and result.

```
import pickle
import numpy as np

with open("/home/daittan/bio-inspired-optimization/super-funicular/question_1.pickle", "rb") as output_f
    x_history_q1, y_history_q1, time_q1 = pickle.load(output_file)

with open("/home/daittan/bio-inspired-optimization/super-funicular/question_2.pickle", "rb") as output_f
    x_history_q2, y_history_q2, time_q2 = pickle.load(output_file)

def objective_function(x):
    return (x * np.sin(10*np.pi*x) + 1)
```

```
mean_pop_over_gen_q1 = [np.round(np.mean(y_history_q1[i,-1,:]),5) for i in range(y_history_q1.shape[0])]
min_pop_over_gen_q1 = [np.round(np.min(y_history_q1[i,-1,:]),5) for i in range(y_history_q1.shape[0])]
max_pop_over_gen_q1 = [np.round(np.max(y_history_q1[i,-1,:]),5) for i in range(y_history_q1.shape[0])]

mean_pop_over_gen_q2 = [np.round(np.mean(y_history_q2[i,-1,:]),5) for i in range(y_history_q2.shape[0])]
min_pop_over_gen_q2 = [np.round(np.min(y_history_q2[i,-1,:]),5) for i in range(y_history_q2.shape[0])]
max_pop_over_gen_q2 = [np.round(np.max(y_history_q2[i,-1,:]),5) for i in range(y_history_q2.shape[0])]


result = np.zeros((13,2))
result[0,0], result[0,1] = (np.mean(time_q1), np.mean(time_q2))
result[1,0], result[1,1] = (np.mean(max_pop_over_gen_q1), np.mean(max_pop_over_gen_q2))
result[2,0], result[2,1] = (np.var(max_pop_over_gen_q1), np.var(max_pop_over_gen_q2))
result[3,0], result[3,1] = (np.min(max_pop_over_gen_q1), np.min(max_pop_over_gen_q2))
result[4,0], result[4,1] = (np.max(max_pop_over_gen_q1), np.max(max_pop_over_gen_q2))

result[5,0], result[5,1] = (np.mean(mean_pop_over_gen_q1), np.mean(mean_pop_over_gen_q2))
result[6,0], result[6,1] = (np.var(mean_pop_over_gen_q1), np.var(mean_pop_over_gen_q2))
result[7,0], result[7,1] = (np.min(mean_pop_over_gen_q1), np.min(mean_pop_over_gen_q2))
result[8,0], result[8,1] = (np.max(mean_pop_over_gen_q1), np.max(mean_pop_over_gen_q2))


result[9,0], result[9,1] = (np.mean(min_pop_over_gen_q1), np.mean(min_pop_over_gen_q2))
result[10,0], result[10,1] = (np.var(min_pop_over_gen_q1), np.var(min_pop_over_gen_q2))
result[11,0], result[12,1] = (np.min(min_pop_over_gen_q1), np.min(min_pop_over_gen_q2))
result[12,0], result[12,1] = (np.max(min_pop_over_gen_q1), np.max(min_pop_over_gen_q2))

import pandas as pd
pd.set_option('display.float_format', '{:.5E}'.format)
df = pd.DataFrame(result, columns = ['q1','q2'])
print(df.to_latex(index=False))



# print(mean_pop_over_gen_q1)
# print(mean_pop_over_gen_q2)
from matplotlib import animation

import matplotlib.pyplot as plt
fig = plt.figure()
ax = plt.axes()
ax.plot(max_pop_over_gen_q1, "-", label = "Max. Fitness using B-10E")
ax.plot(max_pop_over_gen_q2, "-", label = "Max. Fitness using CE")
ax.plot(max_pop_over_gen_q1, ".", markersize = 3)
ax.plot(max_pop_over_gen_q2, ".", markersize = 3)
ax.set_xlabel("Runs")
ax.set_ylabel("Best Fitness from Final Pop.")

ax.legend()
fig.tight_layout()
plt.savefig("q2_maxplot.png", dpi = 1000, format = 'png')


fig = plt.figure()
ax = plt.axes()
ax.plot(mean_pop_over_gen_q1, "-", label = "Mean Fitness using B-10E")
ax.plot(mean_pop_over_gen_q2, "-", label = "Mean Fitness using CE")
```

```
ax.plot(mean_pop_over_gen_q1, ".", markersize = 3)
ax.plot(mean_pop_over_gen_q2, ".", markersize = 3)
ax.set_xlabel("Runs")
ax.set_ylabel("Mean Fitness of Final Pop.")
ax.legend()
fig.tight_layout()
plt.savefig("q2_meanplot.png", dpi = 1000, format = 'png')

fig = plt.figure()
ax = plt.axes()
ax.plot(min_pop_over_gen_q1, "-", label = "Min. Fitness using B-10E")
ax.plot(min_pop_over_gen_q2, "-", label = "Min. Fitness using CE")
ax.plot(min_pop_over_gen_q1, ".", markersize = 3)
ax.plot(min_pop_over_gen_q2, ".", markersize = 3)
ax.set_xlabel("Runs")
ax.set_ylabel("Min. Fitness of Final Pop.")
ax.legend()
fig.tight_layout()
plt.savefig("q2_minplot.png", dpi = 1000, format = 'png')

# plt.show()

import matplotlib.pyplot as plt
fig = plt.figure()
ax = plt.axes()
ax.plot(np.mean(y_history_q1[10,:,:],1), linewidth = 0.5, label = "Maximum Fitness using base-10 encodi
ax.plot(np.mean(y_history_q2[10,:,:],1), linewidth = 0.5, label = "Maximum Fitness using continuous enc
ax.set_xlabel("Number of Generations")
ax.set_ylabel("Fitness")

ax.legend()
fig.tight_layout()
plt.savefig("min_max_mean_q1q2.png", dpi = 1000, format = 'png')
# plt.show()


fig = plt.figure()
ax = plt.axes()

x_best_q1 = []
x_best_q2 = []
for iter, y in enumerate(y_history_q1[0,:,:]):
    ind = np.argmax(y)
    x_best_q1.append(x_history_q1[10,iter,ind])
    x_best_q2.append(x_history_q2[10,iter,ind])

ax.plot(x_best_q1, label = "Trajectory of best point - Base 10 Encoding")
# ax.plot(x_best_q1, "ok", markersize = 1, label = "Points base 10 Encoding")
ax.plot(x_best_q2, label = "Trajectory of best point - Cont. Encoding")
# ax.plot(x_best_q2, "or", markersize = 1, label = "Points Cont. Encoding")
ax.set_xlabel("Number of Generation")
ax.set_ylabel("Best Point")
ax.legend()
fig.tight_layout()
plt.savefig("point_trajectory_q1q2.png", dpi = 500, format = 'png')
```