CSE 598 - Bio Inspired AI & Optimisation

Ajay Kannan - ASU ID 1219387832

Problem - Suppose that you are given the function

$$f(x) = x \sin(10 \pi x) + 1$$

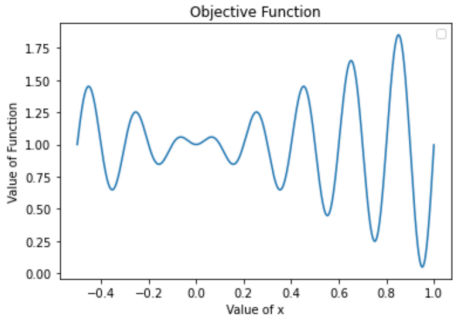
To maximise f (x) subject to $x \in [-0.5, 1]$

---

1) Implement a simple genetic algorithm that encodes the decision variable x as a string of a sign variable and 4 base-10 digits (e.g., -0.1234 is (-1,1,2,3,4)). You are free to choose whatever GA parameters would like (e.g., crossover operator, mutation operator, population size, number of parents, crossover probability, mutation probability), but your algorithm must have both crossover and mutation.

- Describe your GA-implementation choices
- Generate two plots: Best, worst, and average fitness for each successive generation of the GA. Best individual for each successive generation generation of the GA
- Plot the function f and evaluate the quality of the solution found by the GA

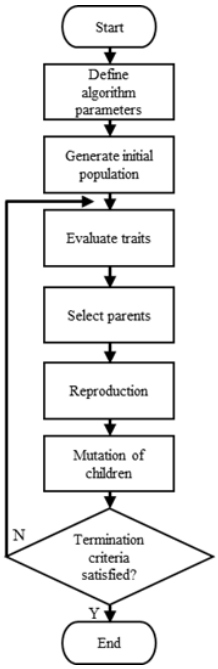
Ans) Plot of the function -



Range from $x \in [-0.5, 1]$

i) Description of the Solution:

- **Population size** - Even integer that represents the number of candidate solutions considered in each generation - in this case, we go with population size 100. This is the collection of candidate solutions, each represented by an array where each element represents a digit of the solution.
- **Evaluating the fitness** of each candidate solution in the population by restricting to a range of -0.5 to 1. After the first iteration, if any number of population goes out of bounds, it is reset to the centre (probability 0.25).
- For the next iterations, **Selection** operation is performed comparing fitness of 2 genes and selecting the winner. According to the algorithm, after evaluating the population, 150 will be there. 100 are taken from this set and children from the previous iteration is given preference and it is competed with the parents. If they have more fitness, there are preferred.
- **Double point Crossover** - Samples, with replacement, from parents and performs double-point crossover to produce offsprings for next generation. The children are a set of the candidate solutions that will be used in the next generation. The parents and children are taken for the iteration. The point of crossover is kept constant.
- **Mutation** - This is a Randomly adjusted genotypes of some of the offspring produced for each generation. Randomness is gauged upon a probability that is set to 0.01 (This is a variable. Based on the output performance, it can be changed).
- **Number of generation** is specified before the operation.

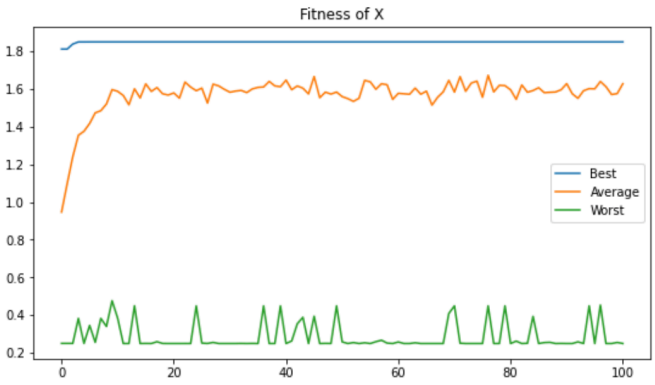


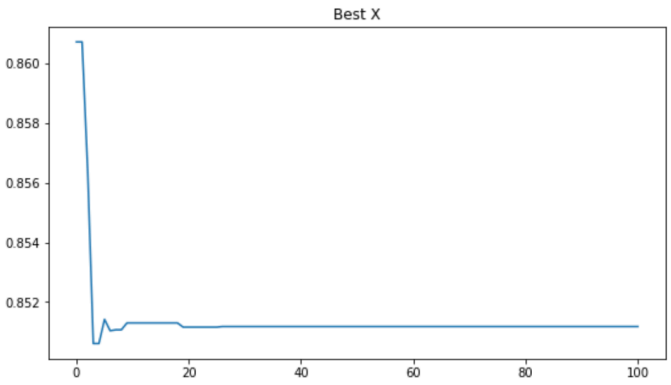
ii) The below graphs shows,

a) The best, average and worst value of the objective function found by the algorithm is generated after each iteration. (Objective Function)

b) The best solution -- the candidate solutions giving the best value of the objective function found by the algorithm (Best X)

The largest value of the objective function is: 1.850595202592346
The corresponding solution is x = 0.85118

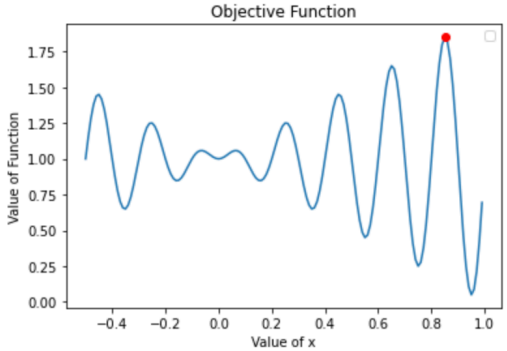




Since the bast case graph does not show any change for 4-base-10 digits, I have taken, 5-base-10 digits, increasing the efficiency and increasing the randomness of the graph. For the code (given separately in a python notebook), I have given 4-base.

iii) There are 3 forces acting on the genetic algorithm, Drift, Mutation and Selection.

1. The mutation is judged based on average objective function which lies at 1.5~ (probability 0.01 as above), where it is plateaued. Mutation happens only if the probability is low. Therefore, drift is lower.
2. Selection pressure is random in this instance, although the selection is done where 2 are selected and highest fitness of the two is taken. But the base criteria of the selection is random.



This GA is maxed out on y = 1.85 and it is the same as maximum objective function.

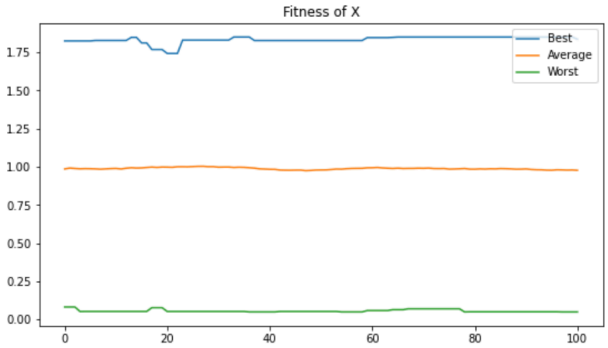
This means the GA has accurately predicted the real objective function.

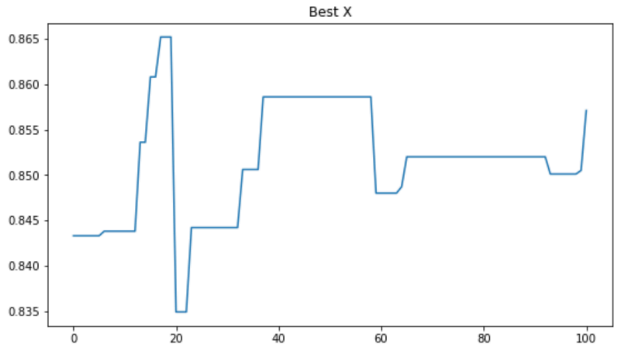
---

2) Instead of encoding the decision variable as a string of base-10 digits, use a genotype with a single gene representing the continuous value of the decision variable. Because there is only one gene, there is no crossover, and parents selected for crossover are simply copied without any recombination. In this case, the GA is effectively only doing mutation.

Implement this mutation-only GA, and compare the efficiency (i.e., number of generations needed to find a solution) of your mutation-only GA to the base-10 discrete GA from Question 1. For the same number of generations, is one algorithm favourable over the other for some reason? Justify your response.

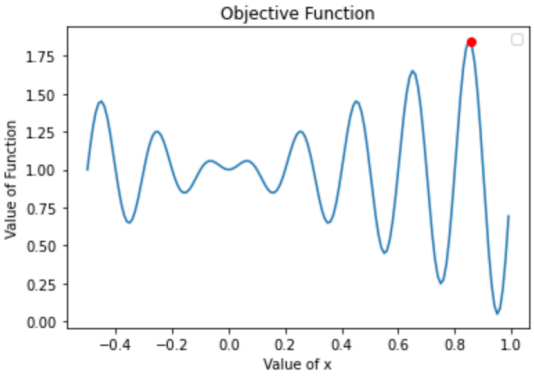
Ans) The graphs turn out to be comparable with the first GA.
The largest value of the objective function is: 1.8358667444951955
The corresponding solution is x = 0.8571





Though it is slightly off make with the first GA, it produces decent values. **The objective function is also shown below. Since the mutation didn't have any effect for the iterations specified, the mutation probability is raised to 0.1 from 0.01 in the question 1 GA.**

This is because is no crossover and selection, and this is evident from the fact that best, average and worst case scenarios are turning out to be pretty constant from the first question.



It starts off with the best solution and inches close to the average solution. Since it is very slow, the average solution is hindered and it is a flat line for 100 iterations. This proves that you can't expect average solution within less time. Though mutation is effective to a certain extent, it is slow in finding out the optimal solution and does not explore like crossover does. You can't expect a new solution very often.

If you can see the fitness graph off the first question, it is hovering at around 1.85 (best) and 1.6(Avg). But in the 2nd question, 1.75 to 1.8 (Best) and 1(Average). Which proves my conclusion that it is not exploring or it takes lot of time to explore. In Q1, initially it takes more time to explore them to exploit. Then after reaching a certain level, it begins to exploit. Drift is also lower in this case. That makes it a solution and a faster one.

---

3) Assume that a base-10 genotype (x1, x2, x3, x4) represents the number 0.x1x2x3x4. While analysing successive population generations from the operation of a GA, you observe that once all members of the population are within 0.001 of each other, the recombination operator very rarely produces offspring that differ from the parent generation, and all new variation in the population is driven by mutation only.

1. Explain why this occurs.

2. Why might it be useful for the recombination operator to act this way? HINT: Think about about "global"
and "local" search as well as the exploration–exploitation tradeoff when forming your answer.

Ans) 1) This is because of the exploration takes time as the recombination step occurs within 0.001 and probability of it happening is next to zero. Also, crossover doesn't happen in this GA, because if a child is produced already, there would be another parent with the same genotype or similar to it. As the iterations go, the population of local maxima, thereby while evaluating the fitness, it won't be selected. So, mutation only occurs. This is known as recombination loss.

2) Take the real world example of City roads. To explore the entire city, you have to explore the neighbourhood first. Then go into the rest of the areas one by one. That takes time. Just like that the mutation operator explores. In the cases of thorough exploration, it is called exploitation. When exploitation becomes more, the drift becomes more where you can't get anywhere. In this question

since the recombination operator very rarely produces new offsprings we can take that drift occurs and very very low exploration happens. Rather than that, algorithm can be adaptive. First explore, then exploit.

Appendix -

1. Initialisation Q1 (initialize_population)- Q2 (initialixe_population1)

```
def initialize_population(population_size):
    population = []
    for j in range(population_size):
        x1 = np.random.uniform(low =-0.5, high =1)   #Value
        x2 = np.random.randint(low=0, high=9, size = 4)
        x = np.append(x1,x2)
        population = np.append(population,x)
    population = np.split(population, population_size)
    return population
```

```
def initialize_population1(population_size):
    population = []
    for j in range(population_size):
        x = np.random.uniform(low =-0.5, high =1)
        x = np.around(x,4)
        population.append(x)
    return population
```

2. Evaluation and Selection Q1 and Q2-

```
def evaluate_fitness(population):
    population_size = len(population)
    fitness = []
    bestx, fitx = -1, 0
    for j in range(population_size):
        ind = population[j]
        if ind[0]>=0:
            sign = 1
        else:
            sign = -1
        x = sign*float("0"+"."+str((int(ind[1])))+str((int(ind[2])))+str((int(ind[3])))+str((int(ind[4])))
        if float(x)<-0.5 or float(x)>1:
            fit = f(0.25)
        else:
            fit = f(x)
        fitness = np.append(fitness,fit)
        if bestx:
            if fit > fitx:
                fitx = fit
                bestx = x
        else:
            fitx = fit
            bestx = x
    return fitness, bestx
```

```
def evaluate_fitness1(population):
    population_size = len(population)
    fitness = []
    bestx, fitx = -1, 0
    for j in range(population_size):
        if float(population[j])<-0.5 or float(population[j])>1:
            fit = f(0.25)
        else:
            fit = f(population[j])
        fitness.append(fit)
        if bestx:
            if fit > fitx:
                fitx = fit
                bestx = population[j]
        else:
            fitx = fit
            bestx = population[j]
    return fitness, bestx
```

```
def selection(population, fitness, num_parents):
    parents =[]
    for i in range(int(num_parents)):
        if fitness[i] > fitness[-i]:
            parents = np.append(parents, population[i])
        else:
            parents = np.append(parents, population[-i])
    parents = np.split(parents, num_parents)
    return parents
```

```
def selection1(population, fitness, num_parents):
    parents =[]
    for i in range(num_parents):
        if fitness[i] > fitness[num_parents*2-i-1]:
            parents.append(population[i])
        else:
            parents.append(population[num_parents*2-i-1])
    return parents
```

3. Crossover and Mutation Question 1 and 2 -

```
def double_point_crossover(parents, num_offspring):
    children = []
    for i in range(int(num_offspring/2)):
        index1 = np.random.randint(low=0, high=9)
        index2 = np.random.randint(low=0, high=9)
        parent1 = parents[index1]
        parent2 = parents[index2]
        offspring1 = np.append(parent1[0:2],parent2[2:4])
        offspring1 = np.append(offspring1 ,parent1[4:])
        offspring2 = np.append(parent2[0:2],parent1[2:4])
        offspring2 = np.append(offspring2, parent2[4:])
        children = np.append(children, offspring1)
        children = np.append(children, offspring2)
    children = np.split(children, num_offspring)
    return children
```

```
def mutation1(children):
    num_offspring = len(children)
    for i in range(num_offspring):
        probability = np.random.uniform(low=0, high=1)
        if probability < 0.1:
            random_value = np.random.random_sample()
            children[i] = np.around((children[i] + 0.01 * random_value),4)
    return children
```

```
def mutation(children, prop):
    num_offspring = len(children)
    for i in range(num_offspring):
        for j in range(1,5):
            probability = np.random.uniform(low=0, high=1)
            if probability < prop:
                random_value = np.random.randint(low=0, high=9)
                children[i][j] = random_value
    return children
```