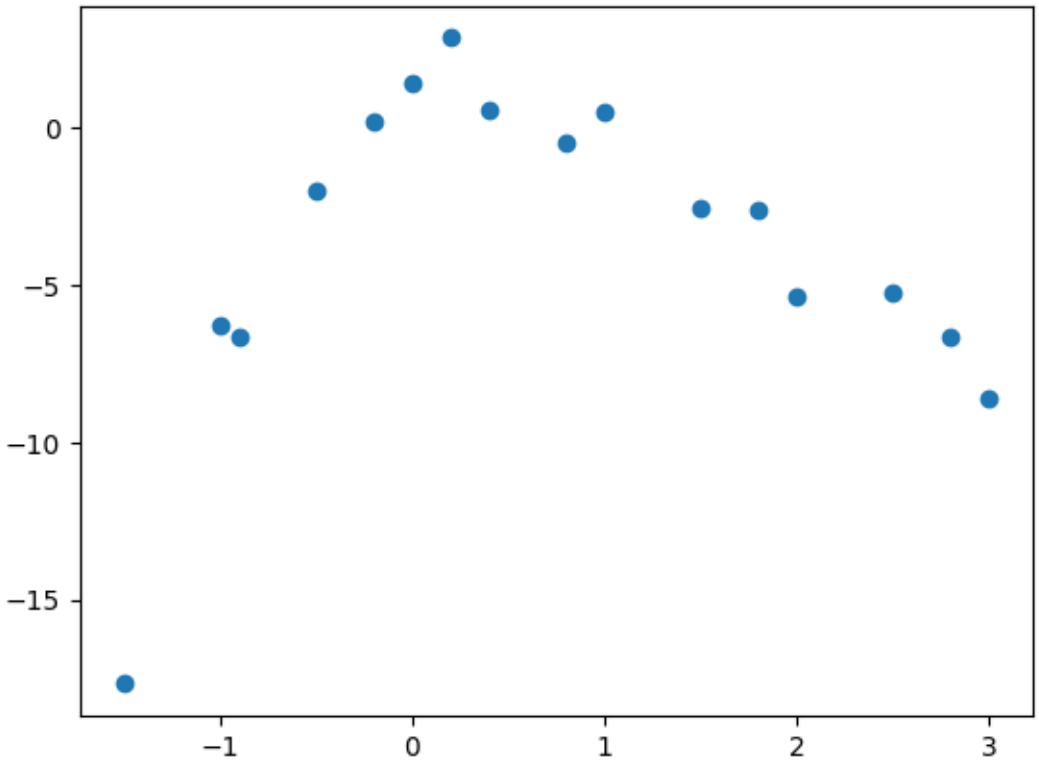
Genetic Algorithm

Task 1:

Curve fitting is the process of finding a mathematical function in an analytic form that best fits a set of data.

Data Visualization:



The data is made of 16 spots. What We need to do is introduce a mathematical function and optimize the scalars in that function. At first I used the sinus or cosine function because they can make most other functions by optimizing the coefficients in the Fourier as I studied in Signals and Systems.

In my algorithm genes are the coefficients and the chromosomes are lists of genes. Each generation is a list of all the possible solutions we could randomly accomplish.

However, the conclusion was not satisfactory! I tried using polynomials. It was discovered that using the polynomial and polyıd is for some reason more efficient than writing a polynomial by hand! This made the algorithm performance better:

```
def select_parents(self, population):
    # Tournament selection
    selected_parents = random.sample(population, self.tournament_size)

    # Sort the selected parents based on fitness score in descending order
    sorted_parents = sorted(selected_parents, key=self.fitness, reverse=True)

    # Return the two best parents based on fitness score
    return sorted_parents[0], sorted_parents[1]
```

For selecting the best parents the tournament selection method was used. At first, a specific amount of chromosomes was chosen using the sample function. Then I decided to

select the best chromosome twice! Combining the qualities of the best individuals with themselves enhanced the result by preserving the best individuals and killing the worse ones.

However, after changing some values and changing the selection function the performance was once again enhanced using 2 different top parents chosen from each tournament.

For the crossover method, Single-point crossover was performed. A random point of chromosomes was chosen by the randint function and a child was made by the 2 parents. Later in the ga function you'll see that this method is called twice, interchanging inputs of the function as the parent1 and parent 2 will get swapped.

```
def crossover(self, parent1, parent2):
    # Single-point crossover
    crossover_point = random.randint(0, len(parent1))
    child = np.concatenate((parent1[:crossover_point], parent2[crossover_point:]))
    return child
```

For mutation, a random number is called from the random function and the mutation rate states that weather that gene will change or not. The lower the mutation rate the less it's possible for the chromosome to have mutated genes. After that a small amount is added to the gene.

In the ga method, the genetic algorithm is performed. First, an initial population was generated. The genes were generated in the range(-5,5) since I found out that the best answers were in this range!

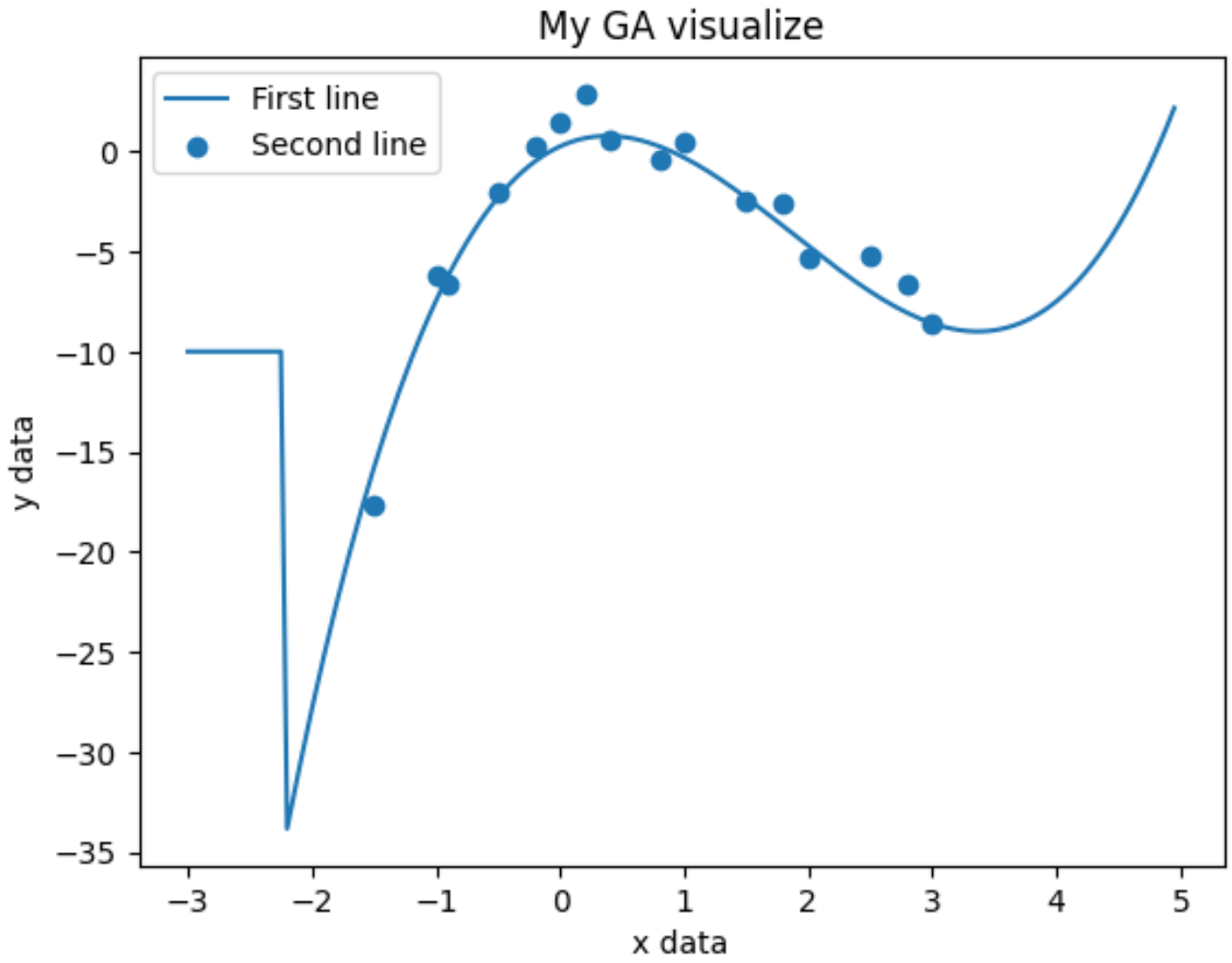
In the second for loop for every chromosome in the population, all the methods were called and children were made. The new population was generated and then became the population list itself. This was done equal to the number of generations.

The best chromosomes have a fitness close to zero. So I printed them and the average fitness in each generation.

```
for z in final_y:
  if z<-35:
    final_y[final_y.index(z)] = -20
  if z>15:
    final_y[final_y.index(z)] = 10
```

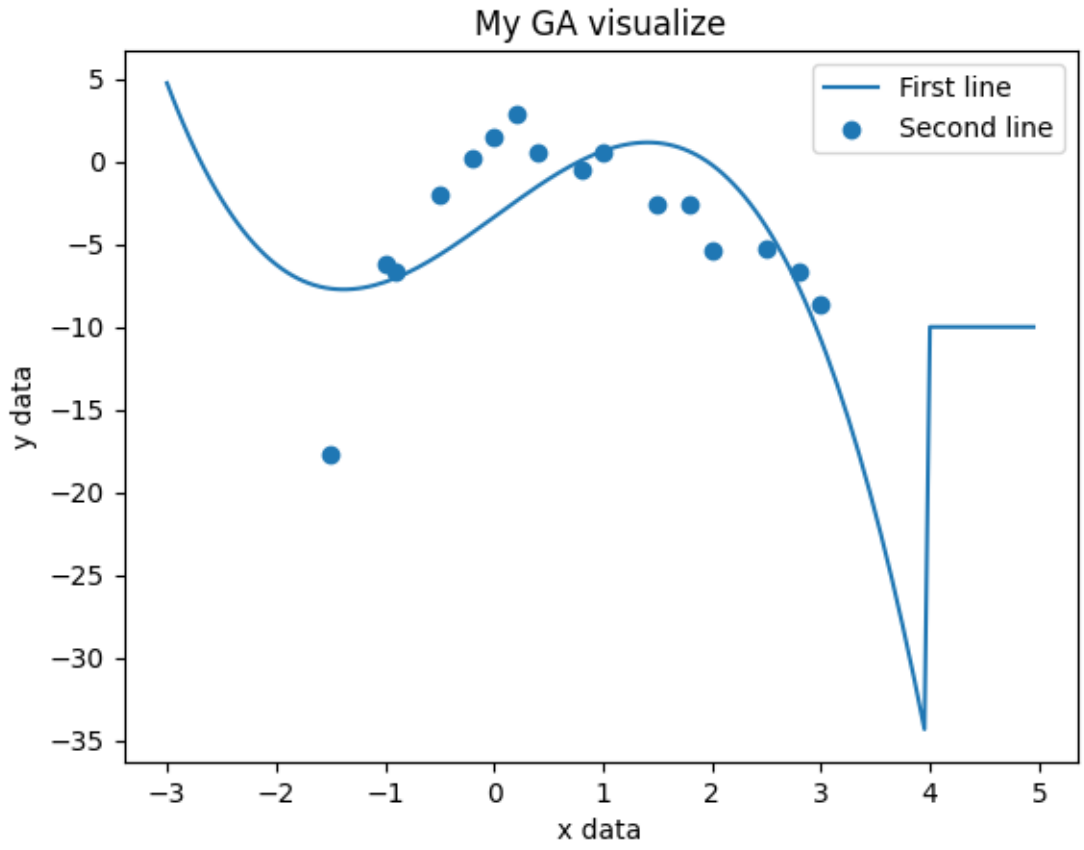
This line of code is intended to limit the y axis for visualization which can sometimes exceed -20 or 10 which will make the visualization invalid!

This was one of the best results: 300 generations and 100 chromosome each generation.



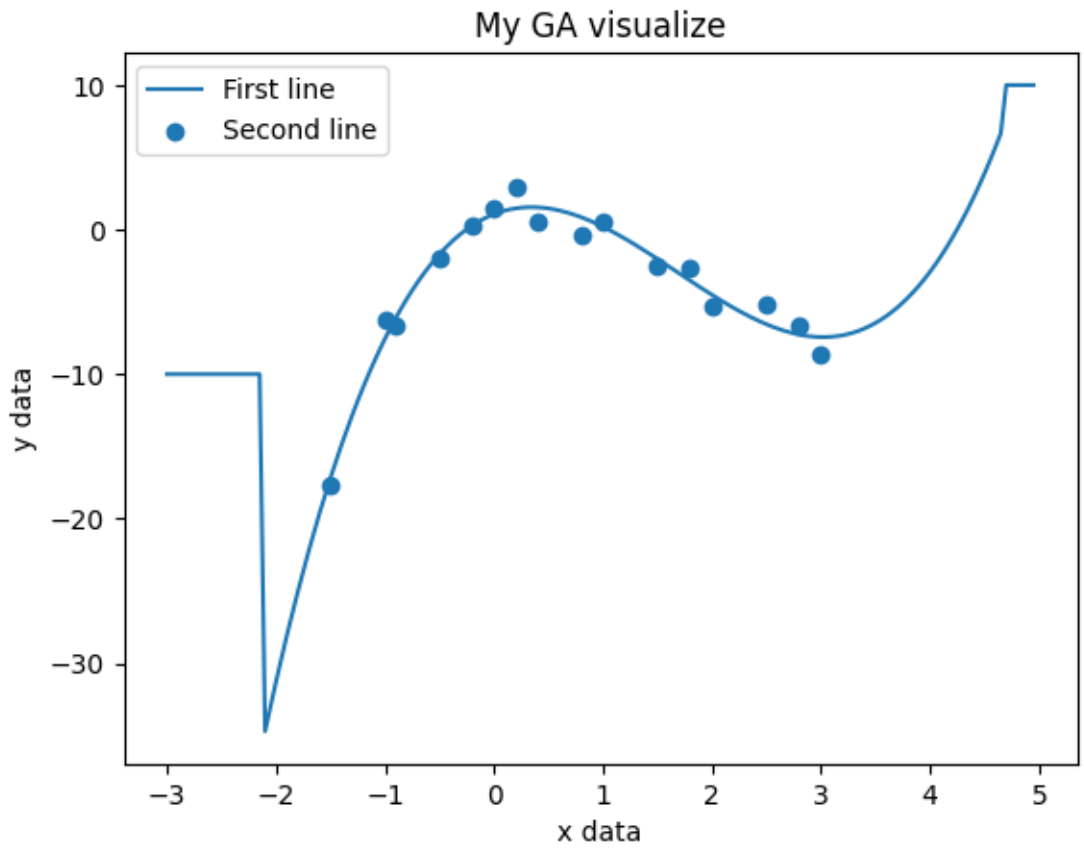
Q1:

In the code I put very low population and very high.



For low population as you can see the algorithm simply doesn't have enough sample to work on so it'll fail. 10 chromosomes are not enough for optimizing 4 values!

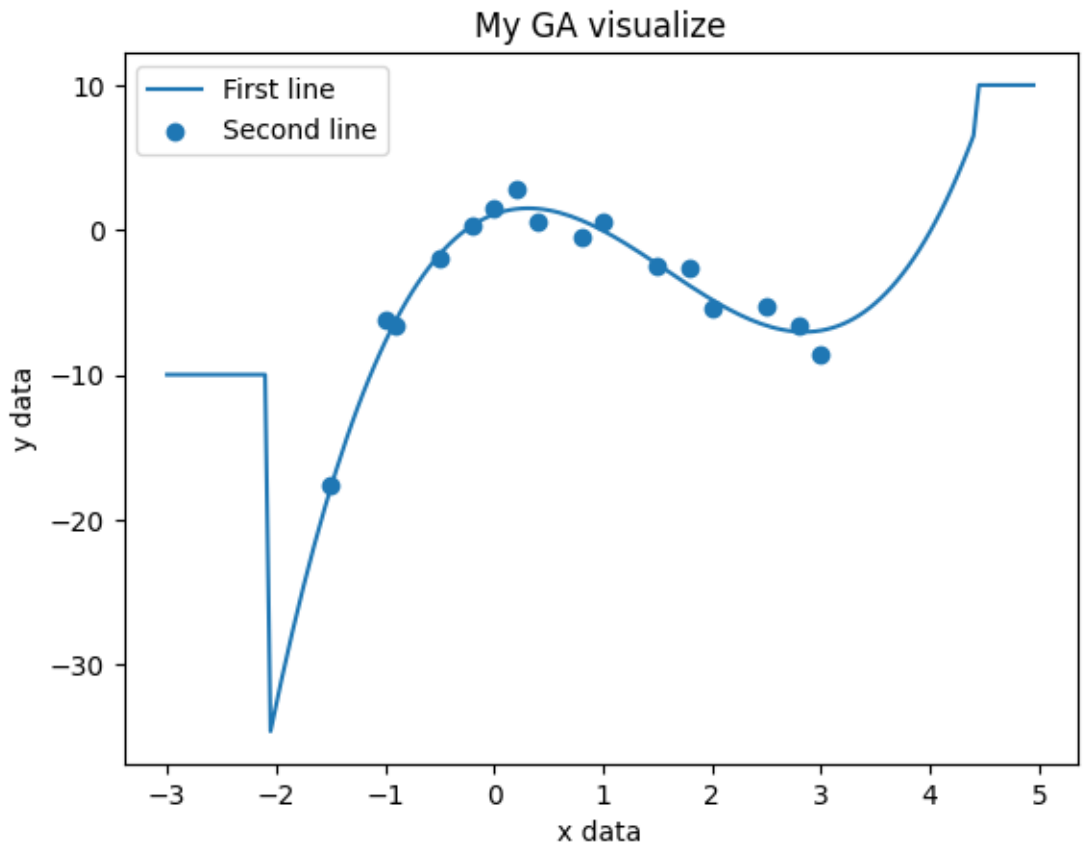
For high population will be very very slow. The system had difficulty calculating everything! I took a lot from cpu but eventually the result was satisfactory!



As you can see the algorithm performed better but it didn't make so much progress because due to low data this is the best it can do the best fitness I found with 20000 generations was -0.7816620393129854 which is as close as possible to 0!

Q2:

I added 50 individuals per generation in the ga method. It won't be so different. However, in the last generation due to a higher population, the algorithm will run slower than it used to in the beginning. The fitness values will rise a bit:



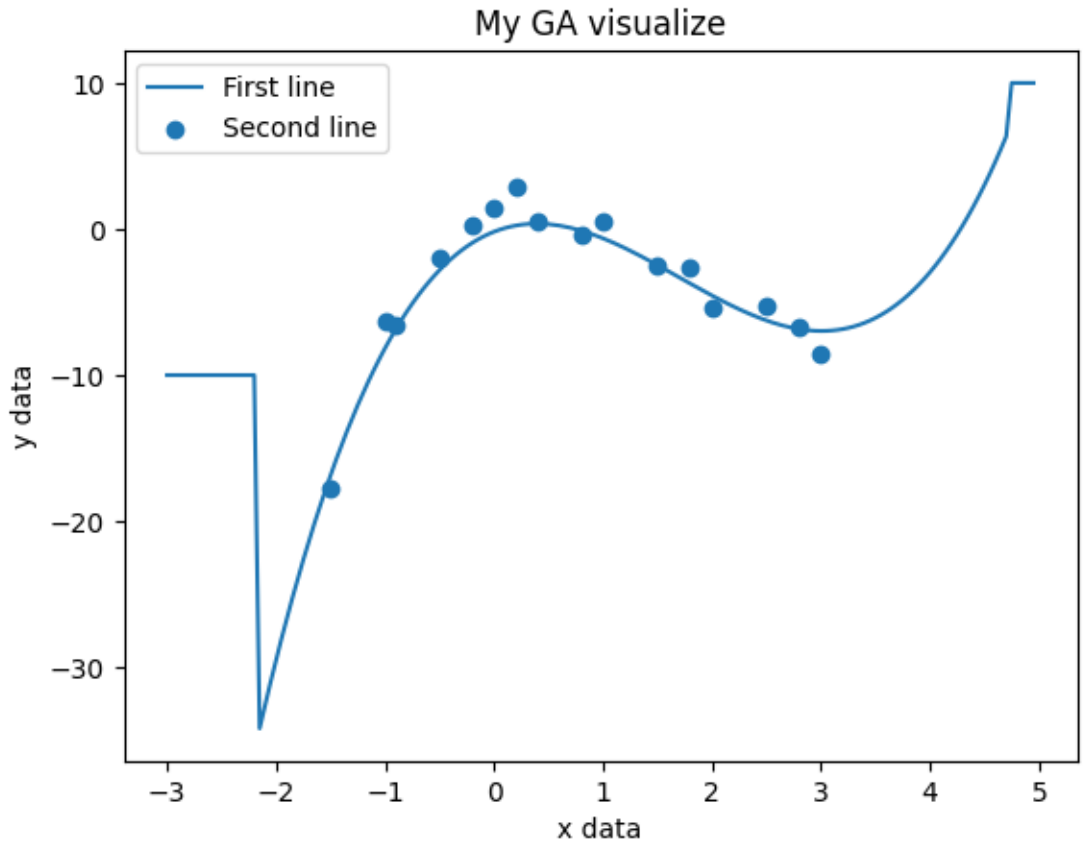
Q3:

The crossover operation combines the best chromosomes together to make better chromosomes. In terms of our problem the coefficients of the best individuals are shared among them and better children will be generated. The crossover point is random to increase algorithm's ability to explore search space.

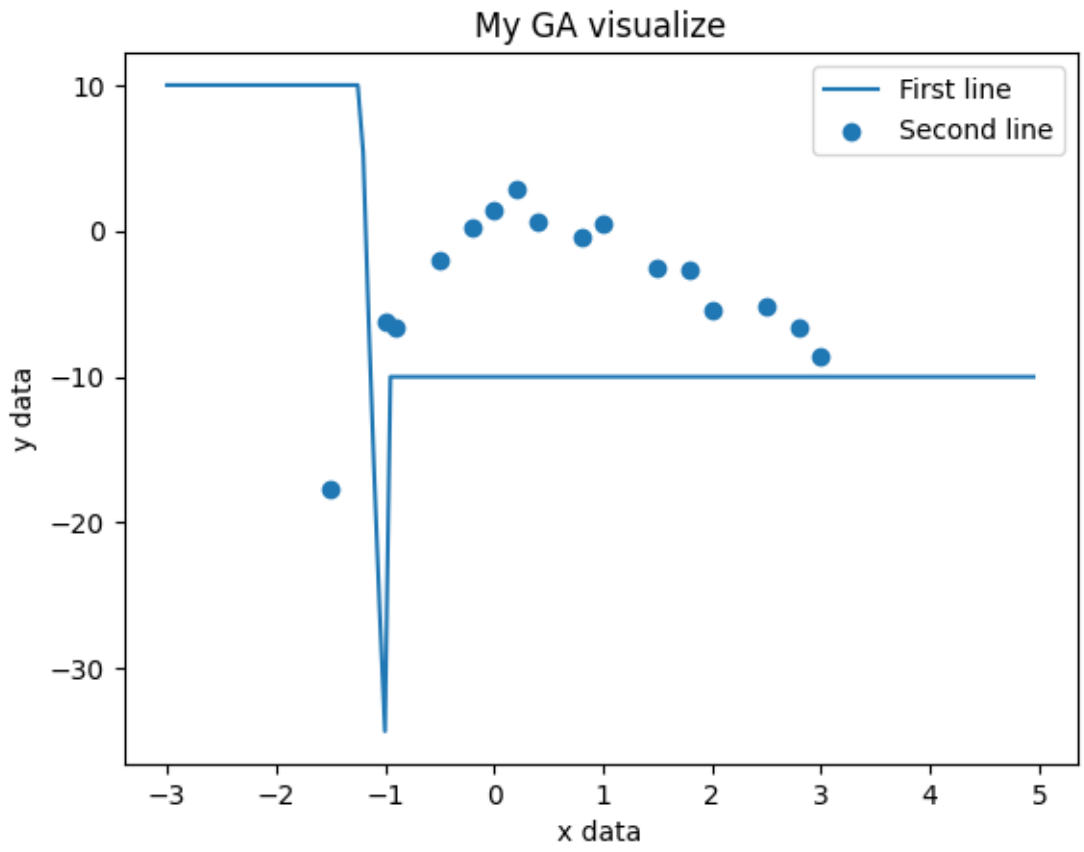
The mutation operation is intended to increase algorithm's ability to explore search space by changing the genes by 0.5.

This algorithm in this problem can perform without the mutation method. In fact I tried this while programming my ga and it didn't make too much difference. However, it'll have better performance with mutation since it doesn't get stuck on one minimum! The crossover method is the main part of ga. Without it there'll be no progress.

Without mutation:



Without crossover: It's all just random values.



Q4:

Q5:

For this problem less population and less generations can make the code run faster. We can also use better built in functions in the code or less loops. I can make it better and faster! There's just no need. 😊

Q6:

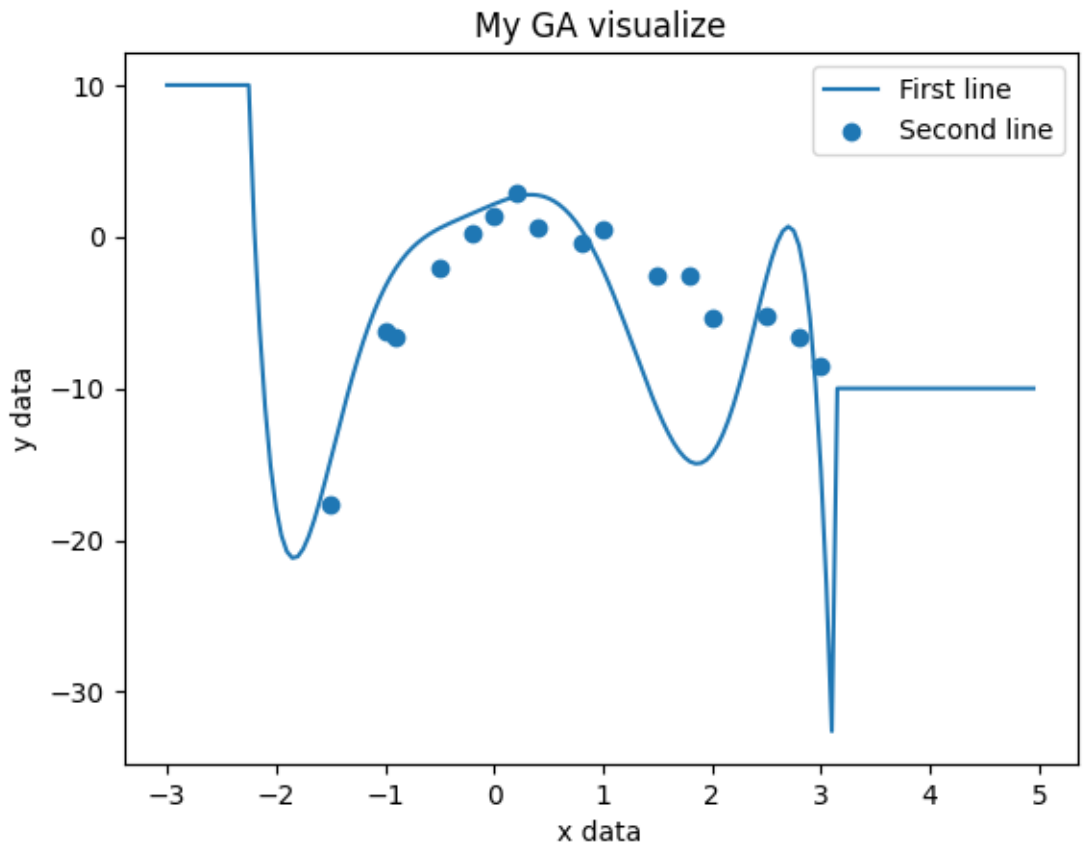
Callbacks are efficient. When an algorithm gets stuck on a local minimum it should jump out of it. There's a waiting value. Like when we wait for 3 generations and if the value doesn't make any changes in the fitness value we change something big like making new chromosomes. Other solutions involve raising the mutation rate or mutation value.

Q7:

We need a callback function that stops the algorithm if the best fitness does not rise and stays the same in 0.01 for example 1.8432 and 1.18485 after four decimals the the algorithm will stop running simply by putting an if statement and break in the loop in the ga method. I tried something like this for the TSP problem.

Q8:

The num_genes variable is the power of the polynomial. By rasing it from 4 to 8 the algorithm did worse and fitness decreased, so more genes don't make ga work better not always at least.



Q9:

The initial data didn't have enough spots. Low data means bad performance no matter what the programmer does!!! More points means much better performance.

Q10:

I used real coded encoding because binary is more complicated for this problem and using it has more computation than real coded which makes the algorithm slower.

TSP exercise 1 part 2

First I calculated the distance of every city from the others using the cdist function from scipy. This functions takes 2 vectors full of positions and calculates the distance and puts them in a matrix.

For convenience I converted this np array in to dataframe format which you can see in the code.

There are 20 cities and in total 200 distances. There's in total 20! (20factorial) possibilities . The algorithm should find the least total distance possible.

My genes are numbers from 1 to 20 plus 21 which is the first city that we should come back to at the end.

My chromosomes are possible permutations of the numbers 1 to 21 which are the genes.

For my fitness function, I used the np array of distances. The function takes chromosome as an input then calculates the distances from one city to the next and returns the total distance.

The lower the fitness the better the chromosome.

For selection once again I used the tournament selection and selecting the fittest chromosome from the chosen ones.

For Crossver once again I used one_point selection. A point was randomly selected and the 2 parents were combined.

In mutation, all I did was swap the positions of 2 genes a specific number of times which is the mutation rate.

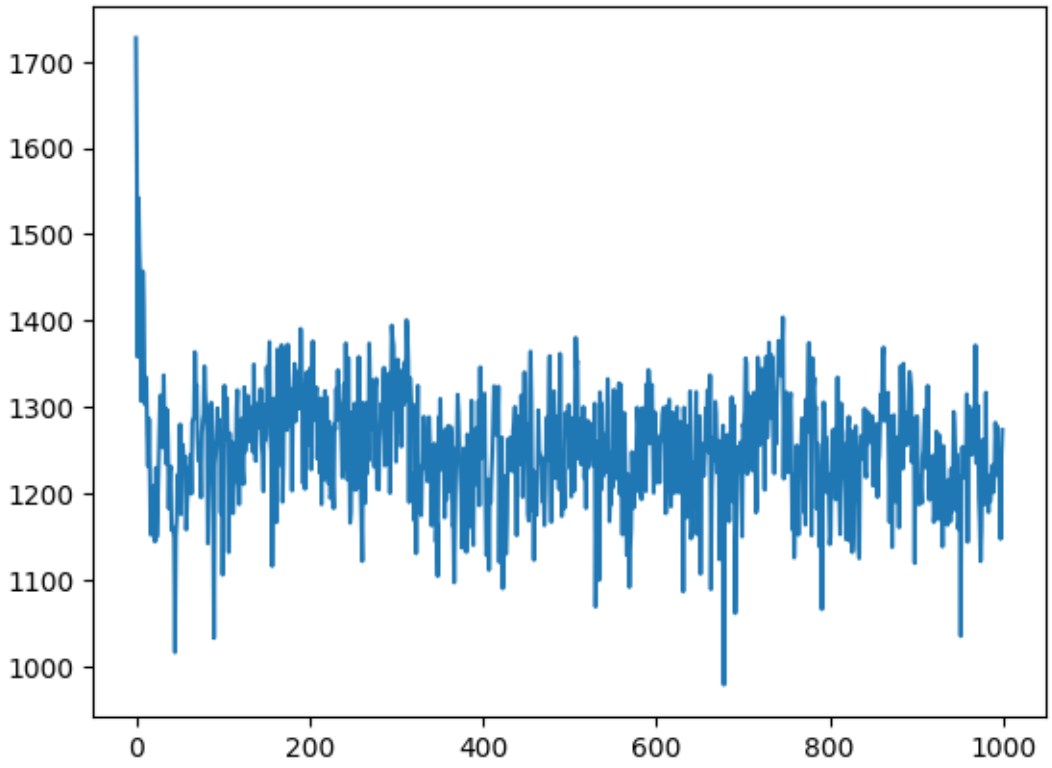
The callback method is intended to prevent the algorithm from getting stuck on a local minimum. The performance was enhanced when this method was introduced. The best_firness a list of the best fitness values from the previous generations as you'll see in the ga method. The waiting value for this callback is 3 as I tried other values and this was the best. Then if the average of differences between the 4 fitnesses is less than one it'll generate a new population! While preserving the best individuals from the previous population of course.

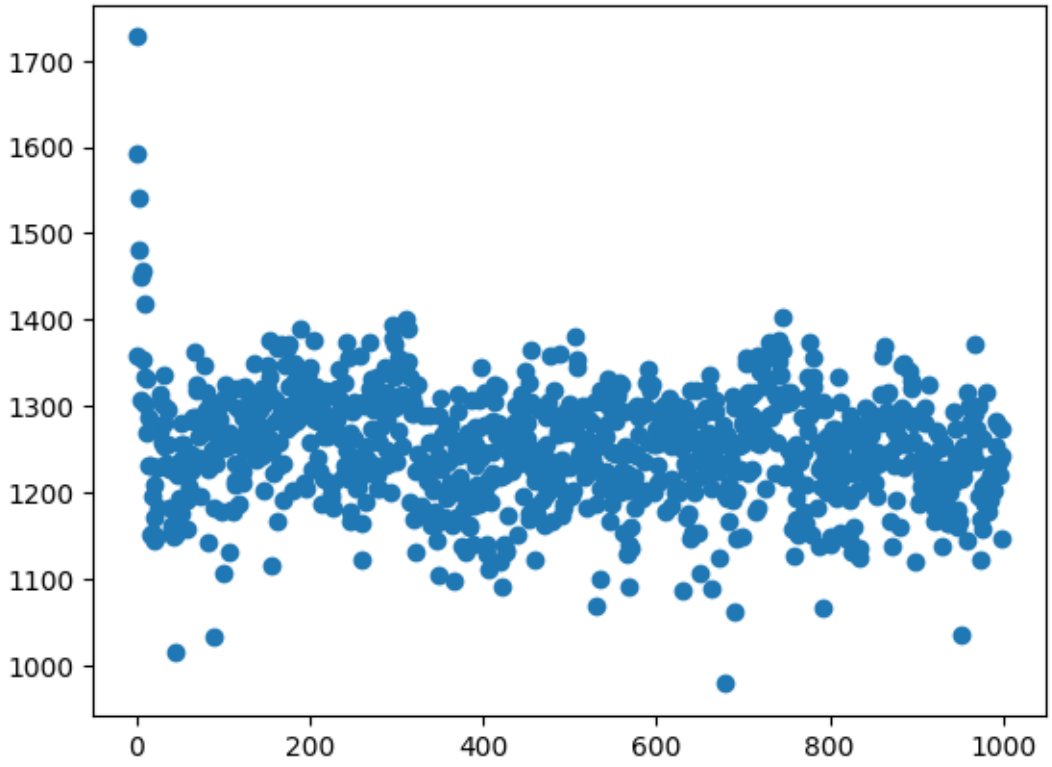
This was the best result I came upon.

([19, 16, 13, 10, 15, 12, 1, 6, 11, 14, 17, 18, 2, 9, 5, 7, 3, 0, 8, 4], 964.7847790484965)

964 made sense to me! 😊

To visualize I plotted and scattered the best fitnesses in the algorithm throughout the generations.


1700
1600
1500
1400
1300
1200
1100
1000
0
200
400
600
800
1000


1700
1600
1500
1400
1300
1200
1100
1000
0
200
400
600
800
1000

Q3:

Again without the crossover method this algorithm does not evolve and it will work based on random values. Mutaition is meant to prevent the algorithm from getting stuck in a local minimum which happens a lot. Mutation rate 2 was the best.

Q5:

For this problem to be solved faster there can be another representation for chromosomes or again we can use more built in function from python or methods from different libraries. All to use less loops which will make the algorithm faster.

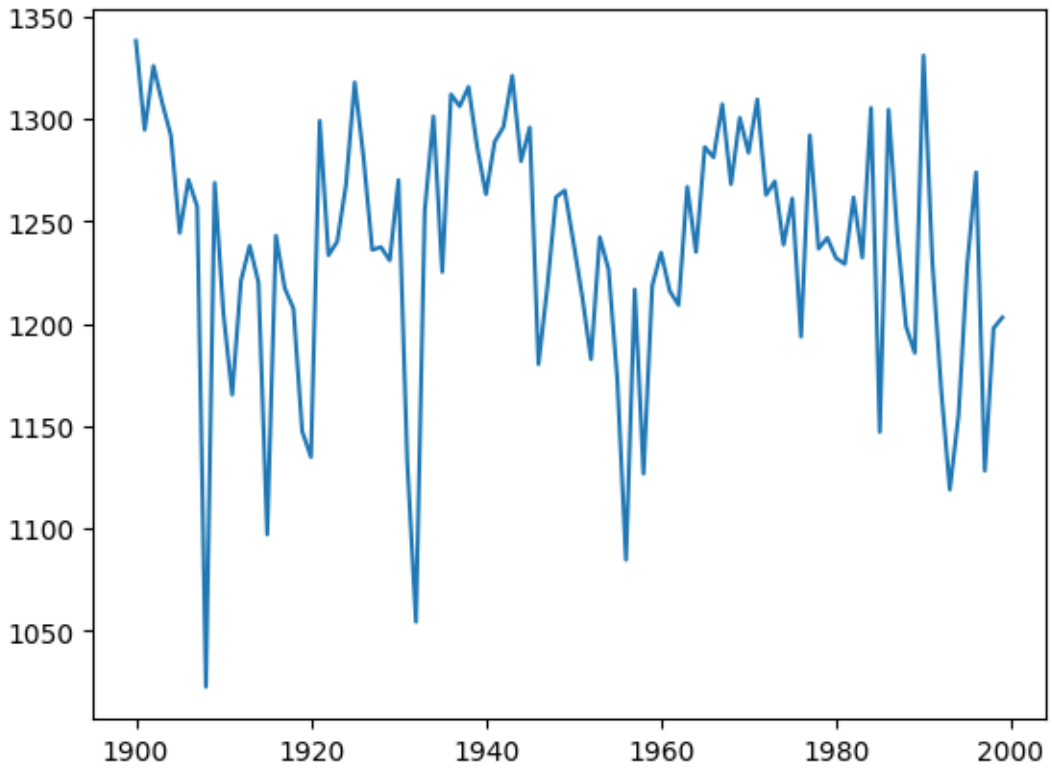
Q6:

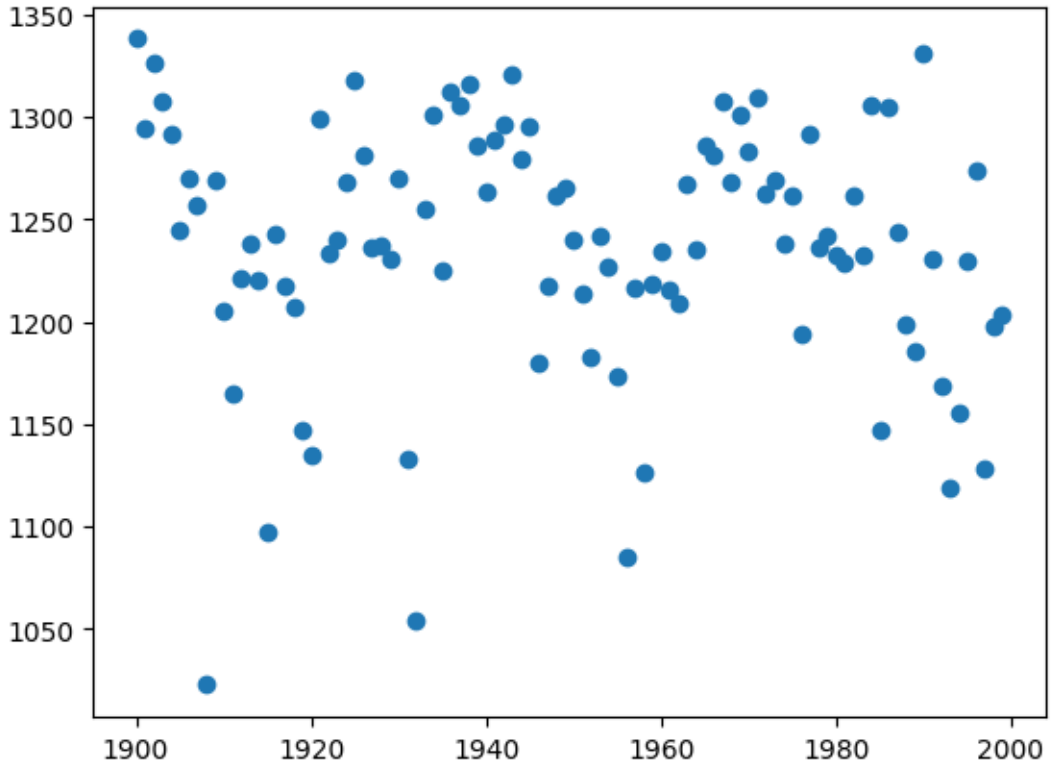
I already have! Using the callback method and mutation I prevented the algorithm from being stuck in one specific minimum or fitness. I even got the distance 882 once!!

Q10:

I used real_coded encoding because this problem is based on decimal numbers and to calculate the distance easily and using cdist function numbers have to be represented as decimals rather than binary. Real-coded encoding is faster and easier in python!!

Last 100 generations plotted and scattered.





Exercise 2

This exercise was mostly done in the notebook and it's all simple machine learning concepts! I'll explain it in detail in a video.

Thank you for reading! 😊