Instituto Politécnico Nacional

Escuela Superior de Cómputo

Evolutionary Computing

Practice 3. Introduction to Genetic Algorithms

Martínez Ramírez Sergi Alberto

Rosas Trigueros Jorge Luis

Date when the practice was made: 24, September 2021

Date when the report was committed: 8, October 2021

1 Theoretical Setting

1.1 Genetic Algorithms

Genetic Algorithms are a family of computational models inspired by evolution. These algorithms encode a potential solution to a specific problem on a simple chromosome-like data structure and apply recombination operators to these structures as as to preserve critical information. Genetic algorithms are often viewed as function optimizer, although the range of problems to which genetic algorithms have been applied are quite broad.

An implementation of genetic algorithm begins with a population of (typically random) chromosomes. One then evaluates these structures and allocated reproductive opportunities in such a way that these chromosomes which represent a better solution to the target problem are given more chances to 'reproduce' than those chromosomes which are poorer solutions. The 'goodness' of a solution is typically defined with respect to the current population.

1.2 Test Functions

In applied mathematics, test functions, known as artificial landscapes, are useful to evaluate characteristics of optimization algorithms, such as:

- Convergence rate.
- Precision.
- Robustness.
- General performance.

On the website [2], some test functions are presented with the aim of giving an idea about the different situations that optimization algorithms have to face when coping with these kinds of problems. But in this paper, the objective of work with these functions is to prove the performance of the Genetic Algorithms made by the author.

1.2.1 Rastrigin Function

In mathematical optimization, the Rastrigin function is a non-convex function used as a performance test problem for optimization algorithms. It is a typical example of non-linear multimodal function.

On an n-dimensional domain it is defined by:

$$f(x) = An + \sum_{i=1}^{n}[x_i^2 - Acos(2\pi x_i)] \tag{1}$$

where $A = 10$ and $x_i \in [-5.12, 5.12]$. It has a global minimum at $x = 0$ where $f(x) = 0$ In this paper it will work on a 3 dimensional field, then $n = 3$

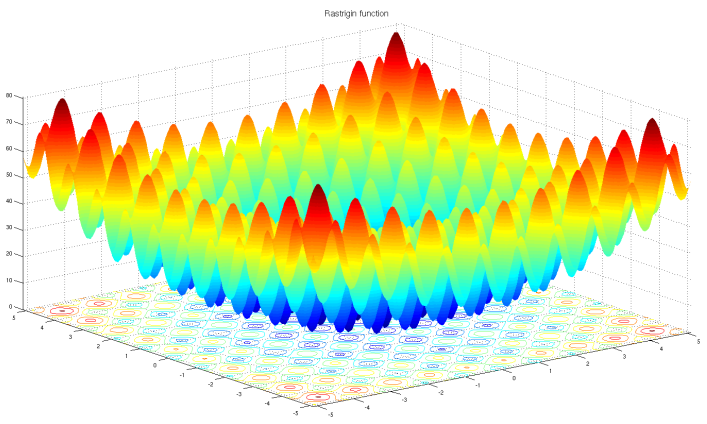


Figure 1: Rastrigin Function.

1.2.2 Ackley Function

In mathematical optimization, the Ackley function is a non-convex function used as a performance test problem for optimization algorithms. It was proposed by David Ackley in his 1987 PhD Dissertation.

On a $2-dimensional$ domain it is defined by:

$$f(x,y) = -20exp[-0.2\sqrt{0.5(x^2+y^2)}] - exp[0.5(\cos(2\pi x)+\cos(2\pi y))] + e + 20 \tag{2}$$

Its global minimum apoint $f(0,0) = 0$

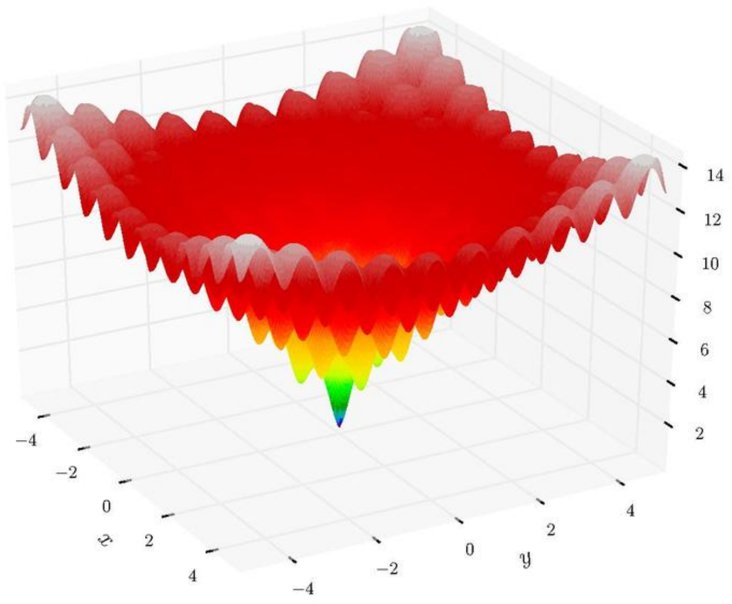


Figure 2: Ackley Function.

2 Material and Equipment

- **Material**
  - Google Collaboratory or Jupyter Notebook.
  - Linux, Windows or other SO that supports google or Jupyter Lab.
- **Equipment**
  - Computer with:
    - 1 Ghz CPU or more.
    - 4 GB of RAM or more.

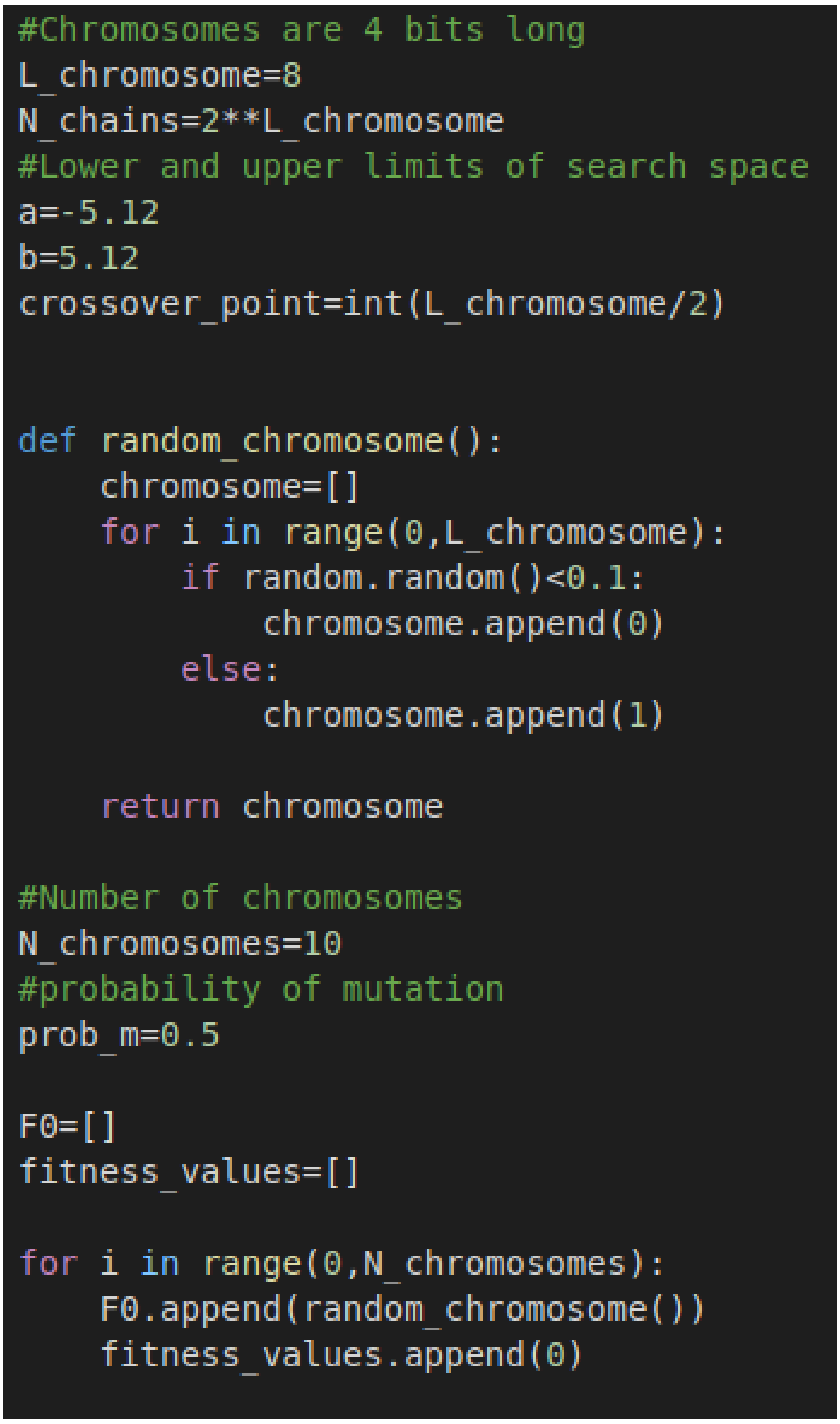
3 Procedure

3.1 Rastrigin Function

I started from the file given by the teacher, which has the definition of a Genetic Algorithm, and first I needed to define the population, after running some tests with the algorithm finished I determined that the best settings for the population would be the following:

- Length of each chromosome: 8.
- Number of chains: 2^8.
- Crossover point: 8/2.
- Number of chromosomes: 10.
- Mutation probability: 50%

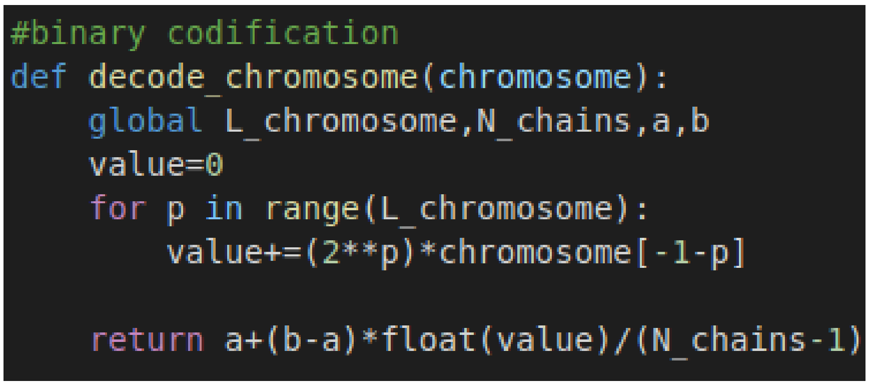
Then I defined the "birth" of every chromosome which it's shown in the following image.



```
#Chromosomes are 4 bits long
L_chromosome=8
N_chains=2**L_chromosome
#Lower and upper limits of search space
a=-5.12
b=5.12
crossover_point=int(L_chromosome/2)


def random_chromosome():
    chromosome=[]
    for i in range(0,L_chromosome):
        if random.random()<0.1:
            chromosome.append(0)
        else:
            chromosome.append(1)

    return chromosome

#Number of chromosomes
N_chromosomes=10
#probability of mutation
prob_m=0.5

F0=[]
fitness_values=[]

for i in range(0,N_chromosomes):
    F0.append(random_chromosome())
    fitness_values.append(0)
```

Figure 3: Generation of random chromosomes code.

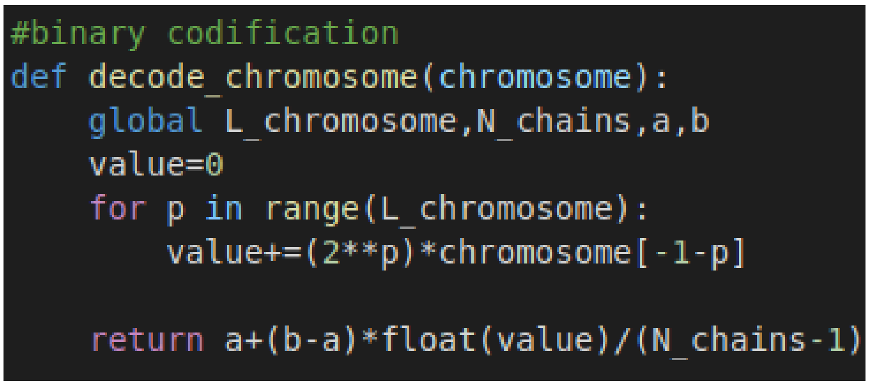
Then, as you may be noticed, the chromosomes are binary coded so I had to define a function to decode them.



```
#binary codification
def decode_chromosome(chromosome):
    global L_chromosome,N_chains,a,b
    value=0
    for p in range(L_chromosome):
        value+=(2**p)*chromosome[-1-p]

    return a+(b-a)*float(value)/(N_chains-1)
```

Figure 4: Decode chromosomes function.

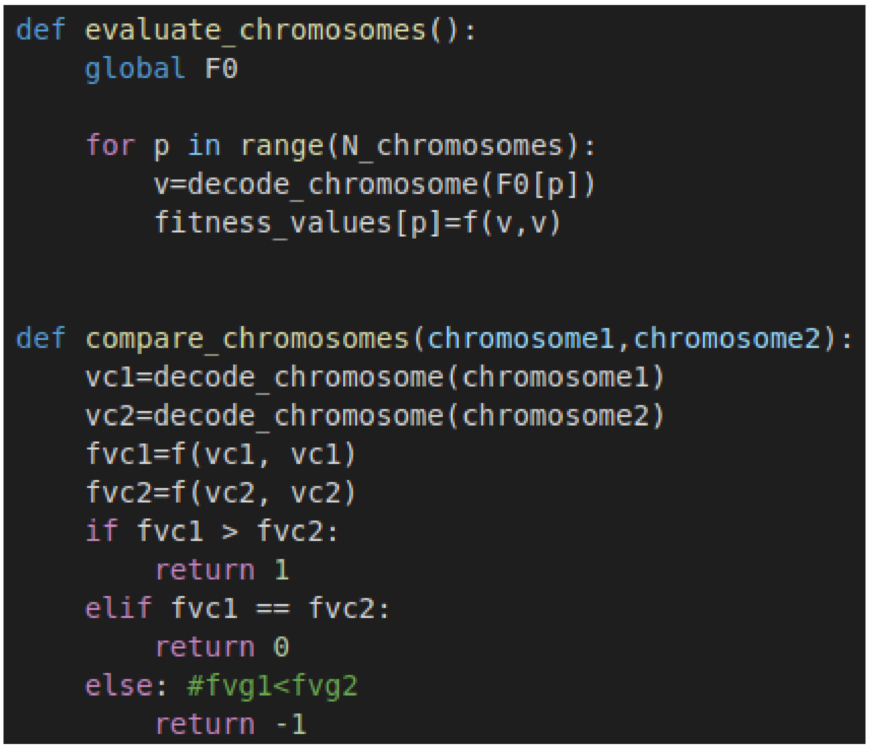
For this example, I needed to set a function that returns the corresponding value for the given values of the chromosome, and for that, the equation (1) was needed.



```
#binary codification
def decode_chromosome(chromosome):
    global L_chromosome,N_chains,a,b
    value=0
    for p in range(L_chromosome):
        value+=(2**p)*chromosome[-1-p]

    return a+(b-a)*float(value)/(N_chains-1)
```

Figure 5: Rastrigin Function coded.

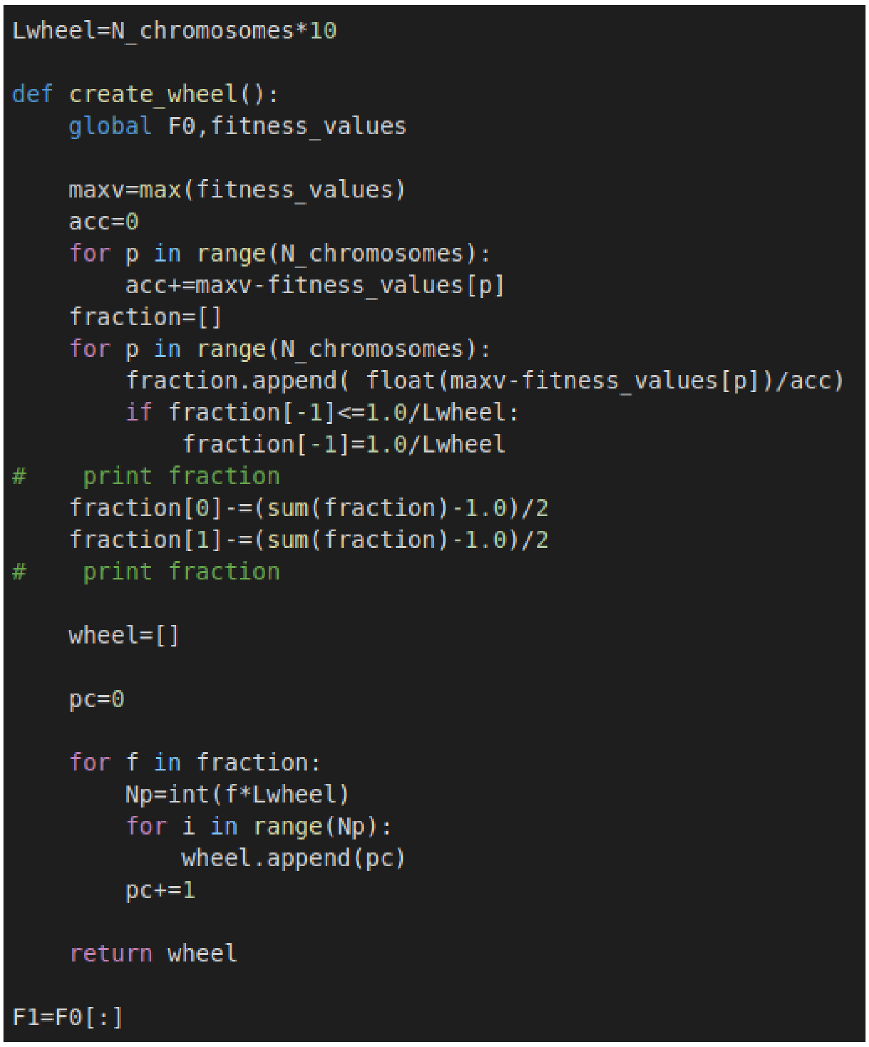
For the next generation, I needed to evaluate the chromosomes to see if they are in a near solution or if they are lost (so they could die the next generation), and then it has to compare the chromosomes to see which one is better. And that's what the next code does.



```python
def evaluate_chromosomes():
    global F0

    for p in range(N_chromosomes):
        v=decode_chromosome(F0[p])
        fitness_values[p]=f(v,v)


def compare_chromosomes(chromosome1,chromosome2):
    vc1=decode_chromosome(chromosome1)
    vc2=decode_chromosome(chromosome2)
    fvc1=f(vc1, vc1)
    fvc2=f(vc2, vc2)
    if fvc1 > fvc2:
        return 1
    elif fvc1 == fvc2:
        return 0
    else: #fvg1<fvg2
        return -1
```

Figure 6: Evaluation and Compare function.

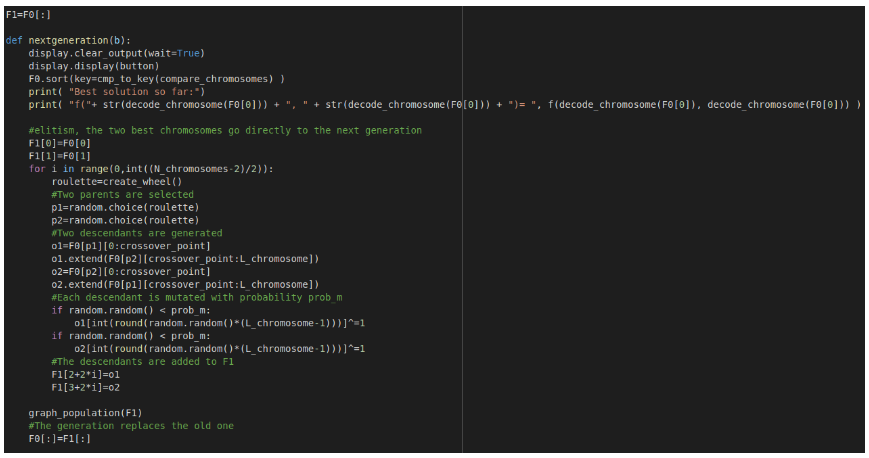
Then for the next generation, it's important to define which chromosomes are best adapted (the closest to the solution) so they'll have more chances to survive and pass to the next generation, and which are the worst adapted (the furthest to the solution) so they'll have more chances to die. For that, a roulette wheel was made to choose "randomly" which chromosomes will pass to the next gen and which won't. And I said "randomly" because as I said previously, the best-adapted chromosomes have more chances to survive. The following code creates the wheel and selects the fortunate chromosomes that will survive.



```
Lwheel=N_chromosomes*10

def create_wheel():
    global F0,fitness_values

    maxv=max(fitness_values)
    acc=0
    for p in range(N_chromosomes):
        acc+=maxv-fitness_values[p]
    fraction=[]
    for p in range(N_chromosomes):
        fraction.append( float(maxv-fitness_values[p])/acc)
        if fraction[-1]<=1.0/Lwheel:
            fraction[-1]=1.0/Lwheel
#    print fraction
    fraction[0]-=(sum(fraction)-1.0)/2
    fraction[1]-=(sum(fraction)-1.0)/2
#    print fraction

    wheel=[]

    pc=0

    for f in fraction:
        Np=int(f*Lwheel)
        for i in range(Np):
            wheel.append(pc)
        pc+=1

    return wheel

F1=F0[:]
```

Figure 7: Roulette Wheel function.

Finally, the next generation has to be created, and it has to pass the same process that the previous one in order to find a solution. The following code does that.



```
F1=F0[:]

def nextgeneration(b):
    display.clear_output(wait=True)
    display.display(button)
    F0.sort(key=cmp_to_key(compare_chromosomes) )
    print( "Best solution so far:")
    print( "f("+ str(decode_chromosome(F0[0])) + ", " + str(decode_chromosome(F0[0])) + ")= ", f(decode_chromosome(F0[0]), decode_chromosome(F0[0])) )

    #elitism, the two best chromosomes go directly to the next generation
    F1[0]=F0[0]
    F1[1]=F0[1]
    for i in range(0,int((N_chromosomes-2)/2)):
        roulette=create_wheel()
        #Two parents are selected
        p1=random.choice(roulette)
        p2=random.choice(roulette)
        #Two descendants are generated
        o1=F0[p1][0:crossover_point]
        o1.extend(F0[p2][crossover_point:L_chromosome])
        o2=F0[p2][0:crossover_point]
        o2.extend(F0[p1][crossover_point:L_chromosome])
        #Each descendant is mutated with probability prob_m
        if random.random() < prob_m:
            o1[int(round(random.random()*(L_chromosome-1)))]^=1
        if random.random() < prob_m:
            o2[int(round(random.random()*(L_chromosome-1)))]^=1
        #The descendants are added to F1
        F1[2+2*i]=o1
        F1[3+2*i]=o2

    graph_population(F1)
    #The generation replaces the old one
    F0[:]=F1[:]
```

Figure 8: Next gen function.

The rest of the code it's the plotting of the function and the chromosomes, so it's easier to see the solutions given by the actual generation.

Next, we have some examples of the execution of the algorithm.

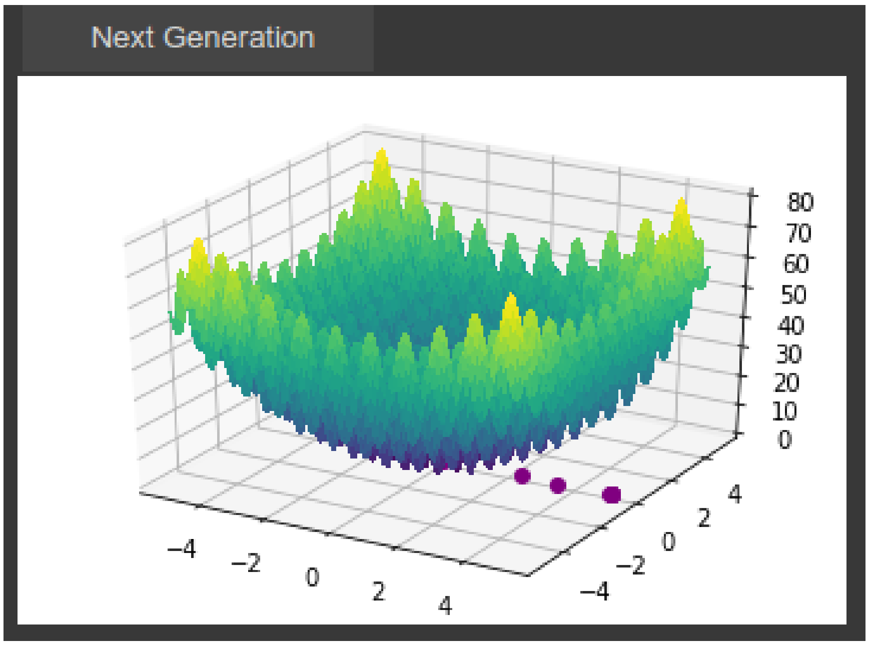
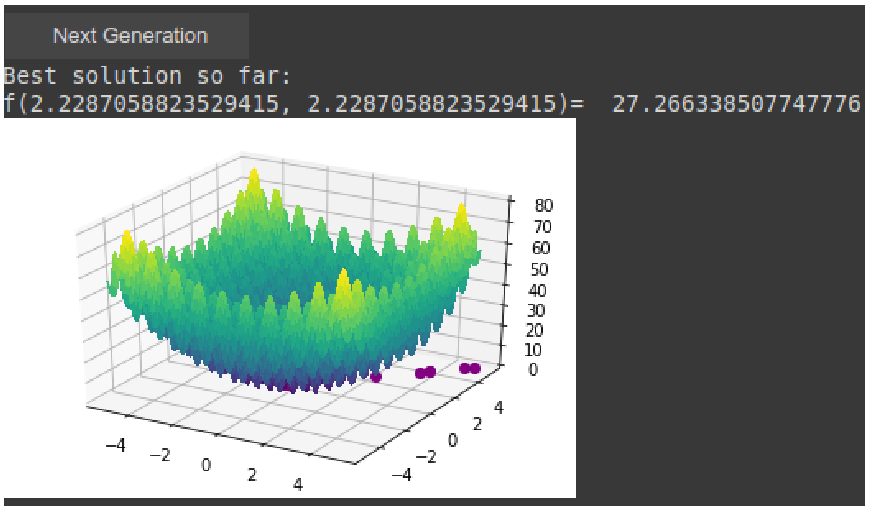
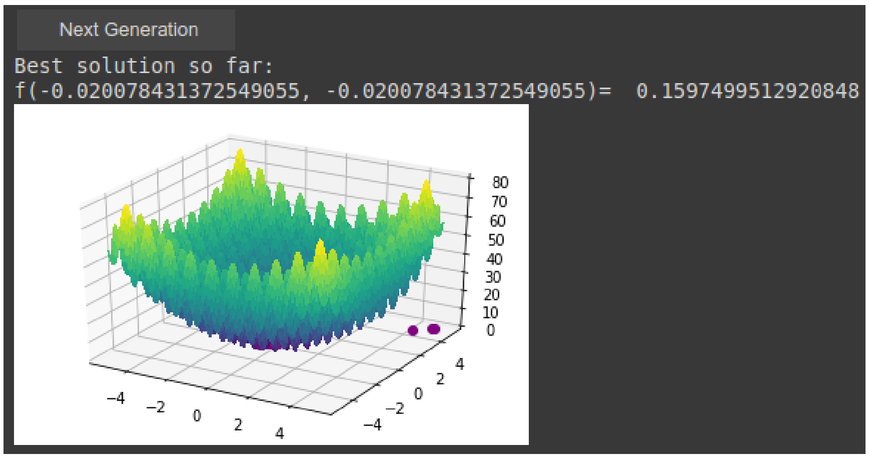


Figure 9: First run of the program.



```
Best solution so far:
f(2.2287058823529415, 2.2287058823529415)=  27.266338507747776
```

Figure 10: Second run of the program.

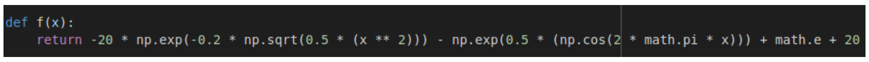


```
Best solution so far:
f(-0.020078431372549055, -0.020078431372549055)=  0.1597499512920848
```

Figure 11: Fifth run of the program a solution has been found.

3.2 Ackley Function

For this function, the procedure was the same as the previous function, with the exception that the function "f" was changed so it returns the same value as equation (2), and the plot was in 2 dimensions.



```
def f(x):
    return -20 * np.exp(-0.2 * np.sqrt(0.5 * (x ** 2))) - np.exp(0.5 * (np.cos(2 * math.pi * x))) + math.e + 20
```

Figure 12: Ackley Function Code.

Next we have some examples of the execution of the algorithm.

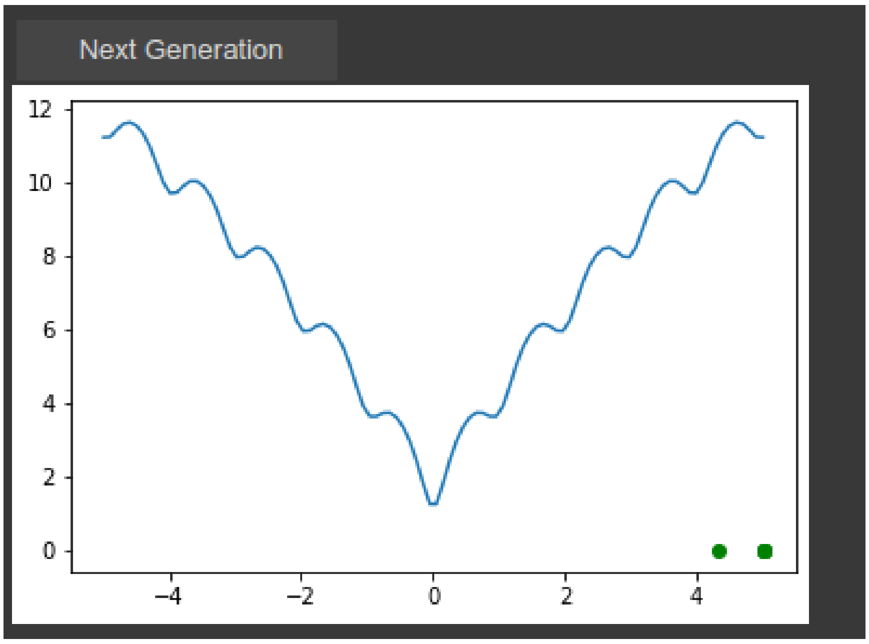
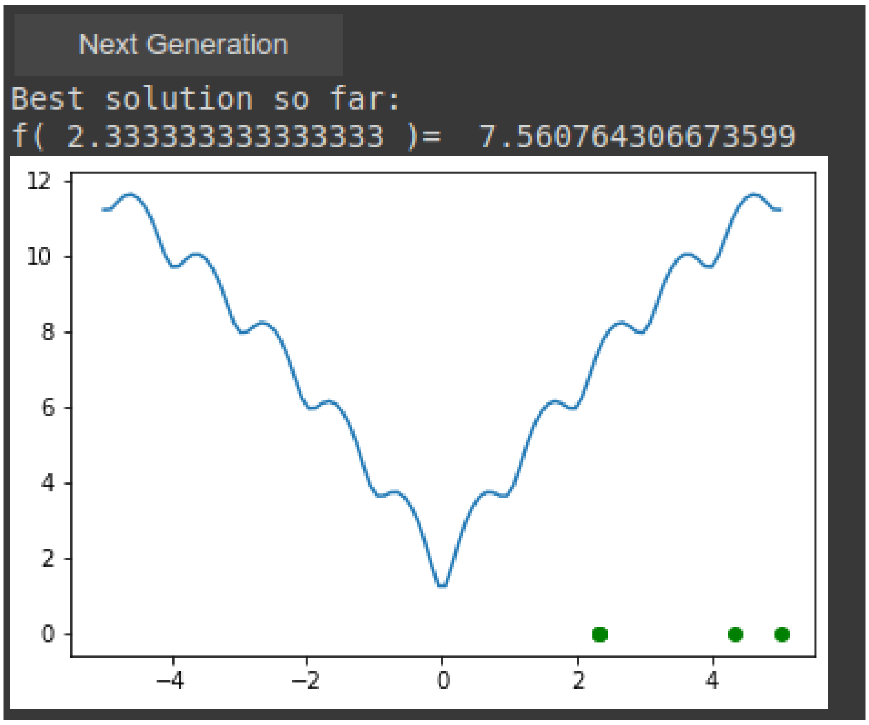
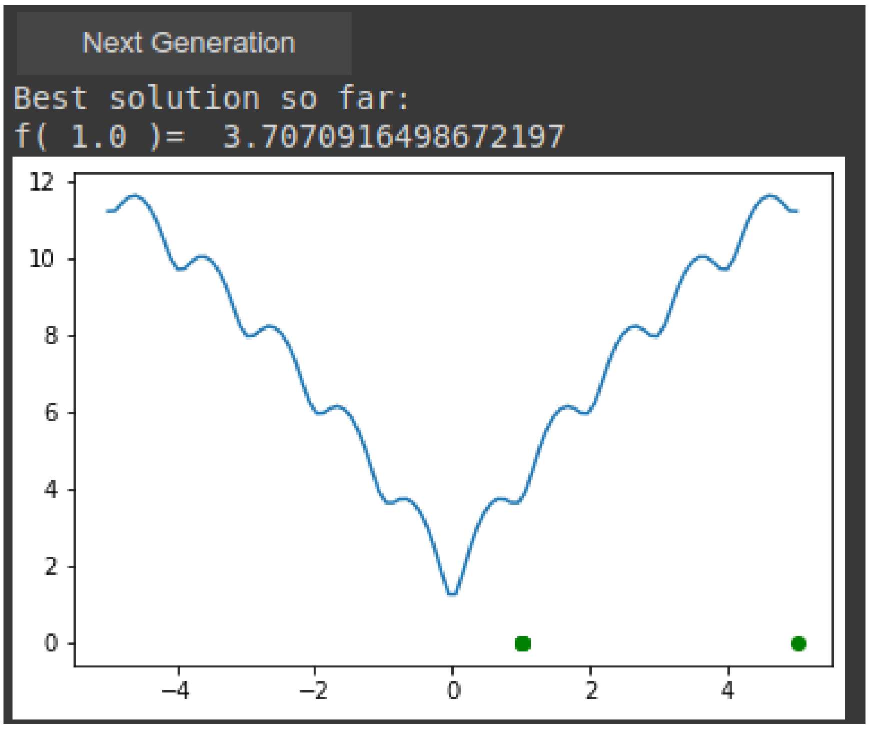


Figure 13: First run of the program.



```
Best solution so far:
f( 2.3333333333333333 )=  7.560764306673599
```

Figure 14: Second run of the program.



```
Best solution so far:
f( 1.0 )=  3.7070916498672197
```

Figure 15: Nineth run of the program a solution has been found.

4 Results and Recommendations

This practice was challenging, as I had never heard of genetic algorithms, so it was a bit difficult for me to understand the program. Otherwise, it is quite difficult once you understand it.

The recommendation I will give it's that the Rastrigin functions work with the same x and y, and that's like cheating, so I'll change it later.

References

[1] Mathew, T. V. ”Genetic algorithm”, IIT Bombay, 2012. [Accessed 8 October 2021].

[2] ”Test functions for optimization - Wikipedia”, En.wikipedia.org, 2021. [Online]. Available: `https://en.wikipedia.org/wiki/Test_functions_for_optimization`. [Accessed: 08- Oct- 2021].

[3] ”Rastrigin function - Wikipedia”, En.wikipedia.org, 2021. [Online]. Available: `https://en.wikipedia.org/wiki/Rastrigin_function`. [Accessed: 08- Oct- 2021].

[4] ”Ackley function - Wikipedia”, En.wikipedia.org, 2021. [Online]. Available: `https://en.wikipedia.org/wiki/Ackley_function`. [Accessed: 08- Oct- 2021].