Evolution Algorithm

Student no. 23019866

1 INTRODUCTION

Given two minimization functions,

$$f(\mathbf{x}) = (x_1 - 1)^2 + \sum_{i=2}^{d} i\left(2x_i^2 - x_{i-1}\right)^2$$

where $-10 \le x \le 10$, start with $d=20$

and

$$f(\mathbf{x}) = \sum_{i=1}^{d} x_i^2 + \left(\sum_{i=1}^{d} 0.5ix_i\right)^2 + \left(\sum_{i=1}^{d} 0.5ix_i\right)^4$$

where $-5 \le x \le 10$, start with $d=20$

each function is implemented in an evolutionary algorithm that produces the lowest bestfitness depending on the parameters used. With my algorithm, I created functions, test_function() where the minimization function is implemented, fits() function to calculate the fitness using the test_function() function, selection() function where the parents are randomly chosen from the population and their genes and fitness are copied into the offspring, crossovers() function where a random point is chosen in the offspring genes and swapped, and the mutation() function where the offspring genes are mutated. Various lists are created to save the fitness and genes of the population. I create a new variable G, which is the generation and a list Gen, using a loop, the functions created earlier are called and used to calculate the bestfiness of the generation. The size of the generation changes depending on the number assigned to G, for each generation the bestfitness and average fitness are calculated and the smallest bestfitness value is chosen. The values of the bestfitness are what I used for this experiment to determine what parameters produce the smallest bestfitness values.

2 EXPERIMENTATION

The aim of my Evolution Algorithm is to calculate the bestfitness value of each generation and find the smallest bestfitness. To calculate the bestfitness function, there are parameters included, MIN, MAX, MUTSTEP (mutation step), MUTRATE (mutation rate), Population (P), Generation (G), Number of genes (N), and Crosspoints. MIN, MAX, and N stay constant, their constants differ with each function. The Number of genes are randomly generated float numbers between MAX and MIN, which are generated bit-by-bit. Within this algorithm there's a class (individual) within this class is a function that initializes gene and fitness. The test_function() is where the minimization function is implemented, this function is used with the fits() function to calculate the fitness of the population and within the mutation function to calculate the new fitness value after the mutation. Within the selection() function two parents are selected and two offspring are created using the parents' genes and fitness, whichever offspring has the smaller fitness values is saved into the offspring list. In the mutation function a random number is generated and if that number is less than the MUTRATE another variable is created, the variable is a random number generated between -MUTSTEP and MUSTEP, and then added to the gene, if the gene is greater than MAX the gene becomes MAX, if its less than gene the gene becomes MIN, this process is done bit-by-bit and then the new individual is appended to the offspring list.

These functions are used in a loop to calculate the fitness values of the generation created and the values are added to the generation list.

Starting with the first function,

$$f(\mathbf{x}) = (x_1 - 1)^2 + \sum_{i=2}^{d} i\left(2x_i^2 - x_{i-1}\right)^2$$

where $-10 \le x \le 10$, start with $d=20$

Function 1

the constant parameters are N = 20, MIN = -10, and MAX = 10

Population P =100		
Generation GEN = 100		
Crosspoints = 1		
MUTRATE	0.02	
MUTSTEP	1.00	
	(Bestfitness) low	Average fitness
	671.27	2019.94
	635.05	1493.71
	3908.69	7110.84
	919.12	2222.65
	417.66	1354.62
Average	1310.36	2840.35
Population = 50		
	16638.84	25938.07
	27910.95	44164.79
	25279.75	39608.17
	54678.42	71097.08
	5475.68	8938.21
Average	25996.73	37949.27
Population = 10		
	1616142.45	1733639.53
	6919409.05	7762348.00
	3370397.60	3370397.60
	3772577.06	4384646.73
	1290890.95	1496242.88
Average	3393883.42	3749454.95

Table 1. Population testing

With a decrease in population, the bestfitness values increase—the higher the population the higher the bestfitness values.

Population P =500						
Crosspoints = 1						
MUTRATE = 0.009						
MUTSTEP = 2.00						
Generation	60.00		100.00		200.00	
	(Bestfitness) low	Average fitness	(Bestfitness) med	Average fitness	(Bestfitness) high	Average fitness
	1008.32	4842.55	18.50	544.33	0.12	615.14
	415.45	1937.79	17.29	1037.21	0.04	526.78
	515.52	2675.59	8.75	639.13	0.19	658.02
	1843.49	5741.52	11.97	844.84	0.14	632.88
	563.00	2738.95	10.82	817.07	0.07	553.05
Average	869.16	3587.28	13.47	776.52	0.11	597.17

Table 2. Generation testing

Same with the generation, the higher the generation size the lower the bestfitness values.
Giving the MUTRATE a starting value that is 1/N, I started testing out different values slightly higher or lower than 1/N which is 0.05.

Population P =500						
Generation GEN = 100						
Crosspoints = 1						
MUTRATE	0.05		0.07		0.10	
MUTSTEP	2.00		2.00		2.00	
	(Bestfitness) low	Average fitness	(Bestfitness) med	Average fitness	(Bestfitness) high	Average fitness
	150.50	10520.95	898.84	23819.65	817.65	43462.93
	231.23	11076.04	375.45	22901.85	1732.72	47299.01
	293.65	10532.90	422.09	20317.83	471.97	53154.50
	103.65	10830.01	578.86	24006.51	2283.98	48858.83
	145.49	10310.99	455.19	20432.35	1447.09	51175.49
Average	184.90	10654.18	546.09	22295.64	1350.68	48790.15
MUTRATE	0.01		0.008		0.002	
MUTSTEP	2.00		2.00		2.00	
	26.07	1128.33	4.82	713.60	92.87	219.79
	55.00	969.10	18.13	790.67	214.94	379.33
	20.79	1044.25	12.56	596.37	81.25	182.31
	7.20	1103.45	18.38	554.12	71.64	126.31
	15.33	917.22	9.70	710.83	191.55	267.86
Average	24.88	1032.47	12.72	673.12	130.45	235.12
MUTRATE	0.001		0.005		0.009	
MUTSTEP	2.00		2.00		2.00	
	62.68	272.79	10.93	366.60	11.72	913.83
	210.87	563.32	24.99	531.35	10.97	633.35
	118.62	216.31	68.53	554.16	11.17	785.89
	174.01	670.36	29.65	554.16	14.27	775.32
	523.24	687.78	7.52	342.44	7.38	564.37
Average	217.88	482.11	28.32	469.74	11.10	734.55

Table 3. MUTRATE testing

Tested numbers higher than 0.05 and noticed the values increased the higher the MUTRATE values, then I tested lower values which resulted in smaller bestfitness values. The values increased when the bestfitness values were 0.002 and 0.001 which are numbers lower than 0.05, I concluded that MUTRATE values that are 0.01 or approximately 0.01 produce smaller bestfitness values.

For the MUTSTEP, my values for testing MUTSTEP centered around 10% of N, which is 2.

Population = 500						
Generation GEN = 100						
Crosspoints = 1						
MUTRATE	0.008		0.008		0.008	
MUTSTEP	0.10		0.50		1.00	
	(Bestfitness) low	Average fitness	(Bestfitness) med	Average fitness	(Bestfitness) high	Average fitness
	2842.02	3326.13	53.07	115.95	14.10	87.18
	5010.18	5894.64	46.49	93.30	13.53	70.68
	2136.22	2616.85	93.41	386.65	34.42	157.22
	3713.19	4636.63	2.42	7.45	14.50	96.89
	17058.84	19292.79	151.32	404.34	24.18	76.76
Average	6152.09	7153.41	69.34	201.54	20.15	97.74
MUTRATE	0.008		0.008		0.008	
MUTSTEP	2.00		3.00		10.00	
	13.31	804.49	14.02	3642.47	80.89	379586.95
	35.20	807.88	10.41	2788.89	201.67	297632.57
	16.58	639.18	45.76	2788.50	360.51	328402.08
	12.51	573.50	19.73	3656.74	300.71	284838.84
	20.46	614.38	33.60	3214.61	150.29	346797.10
Average	19.61	687.88	24.71	3218.24	218.81	327451.51
MUTRATE	0.008		0.008		0.008	
MUTSTEP	5.00		7.00		8.00	
	55.77	27141.37	225.56	81360.64	117.37	162961.23
	19.32	29625.29	63.16	91897.50	152.86	151812.21
	88.90	21717.04	369.76	68685.15	208.54	194772.14
	100.11	24807.02	65.63	92365.94	177.37	130952.55
	85.56	23947.32	83.37	105577.41	146.63	128481.19
Average	69.93	25447.61	161.50	87977.33	160.55	153795.86

Table 4. MUTSTEP testing

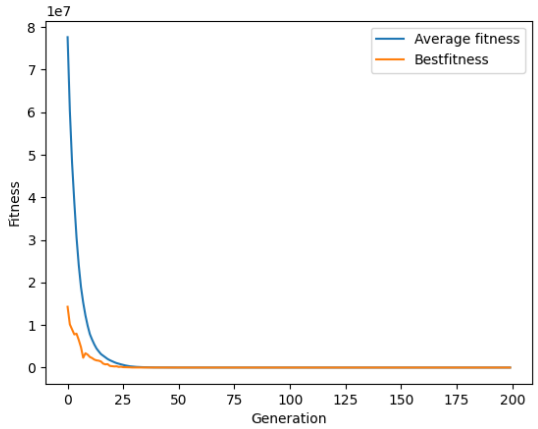
Numbers greater than 2 produced higher bestfitness values, the higher than 2 produced smaller fitness values but the further the values are from 2 the higher the bestfitness values. The smaller fitness values were produced when the MUTSTEP was 0.50, 1.00, 2.00, and 3.00. From that, I concluded the smaller fitness values are produced between + or – 1.00 from 10% of N, and the numbers approximately equal to them.

The next parameter is the Crosspoints, comparing the algorithm results when it has a single Crosspoint and when it has two Crosspoints,

Population P =500						
Crosspoints = 1						
MUTRATE = 0.009						
MUTSTEP = 2.00						
Generation	60.00		100.00		200.00	
	(Bestfitness) low	Average fitness	(Bestfitness) med	Average fitness	(Bestfitness) high	Average fitness
	1008.32	4842.55	18.50	544.33	0.12	615.14
	415.45	1937.79	17.29	1037.21	0.04	526.78
	515.52	2675.59	8.75	639.13	0.19	658.02
	1843.49	5741.52	11.97	844.84	0.14	632.88
	563.00	2738.95	10.82	817.07	0.07	553.05
Average	869.16	3587.28	13.47	776.52	0.11	597.17
Crosspoint = 2						
	484.03	2399.01	8.92	774.15	0.10	606.56
	337.65	2152.59	6.34	833.82	0.16	594.60
	295.91	2546.84	25.88	758.15	0.10	614.97
	519.54	2927.06	21.85	964.50	0.26	684.18
	477.42	3597.24	3.31	735.27	0.04	672.32
Average	422.91	2724.55	13.26	813.18	0.13	634.53

Table 5. Crosspoint Testing

The difference between them is very little, based on the table above, the lower the generation, two Crosspoints produced a smaller bestfitness value, but the higher the generation got a single Crosspoint produced smaller fitness values, although the differences are very small, and could be insignificant differences.



Graph 1. Generation = 200, Population = 500, MUTRATE = 0.009, MUTSTEP= 2.00

The graph above shows the fitness values using the best parameters.

The MUTRATE and MUTSTEP are applied in the mutation function. Starting with the MUTRATE, if the mutprob variable (which is a random number generated between 1 and 0) is less than the MUTRATE, the alter variable (which is a number generated between -MUTSTEP and MUTSTEP) is added to the gene of the offspring. Many numbers are generated depending on the population and generation. Within the generation loop, multiple values of the mutprob are generated depending on the size of the population and generation, that's why a higher population and generation size is better. The more numbers are generated the higher the probability of the number being smaller than the MUTRATE. If the mutprob isn't less than the MUTRATE, the gene isn't added to the alter variable and gives us a smaller fitness value. If the mutprob is smaller than the MUTRATE, the alter value is generated between -MUTSTEP and MUTSTEP and is added to the gene, depending on the new value of the gene (if it is greater than MAX or less than MIN) the gene value is changed.

So why do MUTRATE numbers approximately close to 0.01 produce smaller fitness values? The mutprob is generated randomly there is no predicting what number would be generated but according to the data presented in Table 3, with numbers approximately 0.01, there's a higher possibility of the mutprob being greater than it.

For the MUTSTEP, values + or – 1.00 from 2.00 and their approximate produces best values. These numbers gave a larger range of options for alter variables compared to smaller numbers like 0.30. This range offers a varied list of possibilities, but within a boundary that isn't too small. Since the aim is to produce smaller fitness values, we would want our gene to be equal to MIN(-10), meaning the gene after being added to the alter variable has to be less than MIN.

The second function

$$f(\mathbf{x}) = \sum_{i=1}^{d} x_i^2 + \left(\sum_{i=1}^{d} 0.5ix_i\right)^2 + \left(\sum_{i=1}^{d} 0.5ix_i\right)^4$$

where $-5 \le x \le 10$, start with d=20

Function 2

The constant parameters are N =50, MIN =-5 and MAX = 10

Generation GEN = 100						
Crosspoints = 1						
MUTRATE = 0.008						
MUTSTEP= 1.00						
Population	100.00		300.00		500.00	
	(Bestfitness) low	Average fitness	(Bestfitness) med	Average fitness	(Bestfitness) high	Average fitness
	158.91	205.74	123.88	2438.31	69.85	1579577.27
	240.85	307.69	74.65	3398.76	151.39	9940.86
	187.35	204.73	105.11	673.99	124.31	74541.54
	195.94	243.80	119.86	719.54	74.82	97424.06
	143.11	156.25	119.83	16985.42	45.71	3040.37
Average	185.23	223.64	108.66	4843.20	93.21	352904.82
Population = 500						
Generation	100.00		300.00		500.00	
	93.62	55805.60	14.71	47.34	21.77	62.01
	89.02	125856.18	24.85	72.65	6.74	43.42
	49.29	160060.99	18.12	58.49	7.34	43.02
	104.49	69237.78	32.11	71.06	9.82	53.64
	57.89	3612.67	42.92	76.94	17.18	52.89
Average	78.86	82914.64	26.54	65.30	12.57	50.99
Crosspoints = 2						
	56.46	652975.10	21.63	97.63	20.73	48.60
	49.00	220955.69	47.54	222.87	20.77	69.88
	35.41	88884.98	28.62	84.64	17.56	69.72
	79.96	126312.46	34.52	98.58	13.85	61.33
	77.30	582601.61	64.94	229.28	24.92	57.02
Average	59.63	334345.97	39.45	146.60	19.57	61.31

Table 6. Population, Generation, and crosspoint testing

The larger the population and generation size the smaller the fitness value which can be seen in the

table above. A single crosspoint compared to a double crosspoint shows very little difference, according to the table the fitness values are lower at double crosspoints when the population and generation size is a hundred.

For the MUTSTEP, my testing values centered around 10% of N, which is 2.

Population P =100						
Generation GEN = 100						
Crosspoints = 1						
MUTRATE	0.05		0.05		0.05	
MUTSTEP	0.50		2.00		4.00	
	(Bestfitness) low	Average fitness	(Bestfitness) med	Average fitness	(Bestfitness) high	Average fitness
	181.06	647.96	145.91	123702.70	165.60	743868.68
	108.30	313.43	140.33	62111.11	150.70	404208.69
	109.44	439.56	216.69	92885.30	106.15	774395.33
	143.29	305.84	121.00	68484.96	152.97	1060117.79
	158.42	690.29	182.85	40437.93	153.95	1019498.24
Average	140.10	479.41	161.36	77524.40	145.87	800417.75
MUTRATE	0.05		0.05		0.05	
MUTSTEP	0.30		0.50		1.00	
	108.84	138.50	216.53	904.10	99.74	4940.60
	179.62	235.89	187.38	647.46	191.98	7076.57
	90.81	164.85	137.27	364.08	142.54	3274.54
	216.36	277.60	184.87	398.56	169.11	5166.59
	195.79	274.71	219.35	982.78	88.04	8939.17
Average	158.28	218.31	189.08	659.39	138.28	5879.49
MUTRATE	0.05		0.05		0.05	
MUTSTEP	0.70		1.00		1.50	
	230.98	2119.98	136.56	6774.56	103.07	19798.27
	136.79	1491.89	129.80	12813.48	237.99	38459.21
	116.13	2248.91	118.42	22210.08	207.94	56215.96
	195.32	1106.14	158.26	2980.28	107.43	19301.23
	183.16	1201.65	176.92	9223.59	193.45	18735.95
Average	172.47	1633.71	143.99	10800.40	169.98	30502.13

Table 7. MUTSTEP testing

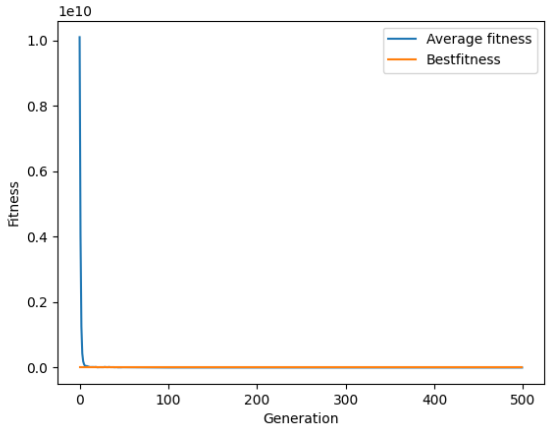
Numbers larger than 2 produced larger fitness values. Values producing small fitness values were 1 and numbers approximately equal to 1.

For the MUTRATE, values centered around 1/N, which is 0.05.

Population P =100						
Generation GEN = 100						
Crosspoints = 1						
MUTRATE	0.01		0.05		0.09	
MUTSTEP	1.00		1.00		1.00	
	(Bestfitness) low	Average fitness	(Bestfitness) med	Average fitness	(Bestfitness) high	Average fitness
	122.01	178.77	149.92	6838.38	166.92	26374.53
	164.34	196.31	111.82	3530.29	152.04	53800.71
	158.35	190.61	119.11	3023.07	122.95	21290.23
	129.28	159.23	212.11	8985.58	76.27	24963.73
	138.65	190.34	125.51	20411.76	217.88	27486.30
Average	142.52	183.05	143.69	8557.82	147.21	30783.10
MUTRATE	0.008		0.01		0.04	
MUTSTEP	1.00		1.00		1.00	
	107.85	129.11	173.54	265.04	239.10	2615.71
	163.13	222.66	196.22	265.71	107.12	3682.71
	133.04	144.44	177.74	208.04	163.92	3286.15
	201.22	238.49	136.92	186.16	123.19	2106.78
	158.52	208.64	118.85	152.76	147.71	3838.84
Average	152.75	183.67	160.65	215.54	156.21	3106.04
MUTRATE	0.004		0.008		0.009	
MUTSTEP	1.00		1.00		1.00	
	145.64	187.46	109.75	120.21	134.80	154.37
	108.49	211.30	160.81	268.56	162.62	198.04
	131.16	149.62	175.58	201.94	84.79	111.79
	153.95	204.67	58.62	70.00	227.92	264.23
	169.24	172.70	152.01	181.98	215.17	256.20
Average	141.70	185.15	131.36	168.54	165.06	196.93

Table 8. MUTRATE testing

Numbers smaller than 0.01 produced smaller fitness values. The smaller fitness values were produced when the number was approximately equal to 0.01.



Graph. 2 Generation = 500, Population = 500, MUTRATE = 0.008, MUTSTEP= 1.00

The graph above shows the fitness values using the best parameters.

The application of the MUTSTEP, MUTRATE, MIN, and MAX are the same as in function 1. Same as the first function, the mutprob is generated randomly there is no predicting what number would be generated but according to the data presented in Table 8, with numbers approximately 0.01, there's a higher possibility of the mutprob being greater than it.

The smaller the number of the MUTSTEP, the less the range for the alter values, but when we have numbers like 0.30 as the MUTSTEP, the range of the values is from -0.30 to 0.30 produces a much smaller range of possibilities for the alter variable. Based on the data collected in Table 7, a small range of possibilities is the best approach compared to a smaller range of possibilities, this is because since we want to produce small fitness values, the genes should be equal to the MIN(-5), meaning the gene after added to the alter variable should be less than MIN.

3 COMPARISON

The Hillclimber algorithm can be used to find the solution to a given optimization problem. This algorithm can be used to solve the functions given. "Hill climbing is the simplest type of local search-based method where the search nature only accepts the 'downhill' move." (Al-betar, M.A., 2017). "It only accepts the downhill moves where the quality of the neighboring solution should be better than the current one." (Al-betar, M.A., 2017). The main problem with using a hillclimber algorithm is that it could get stuck

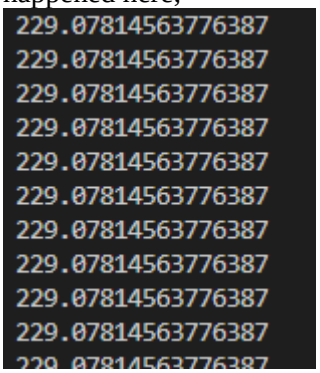
in a local optima. “However, it can easily get stuck in local optima which in most cases is not satisfactory.” (Al-betar, M.A., 2017). A hill-climbing algorithm tries to find a better set of solutions based using the set of previous solutions. “This procedure tries to find the improved neighboring solution from the set of the neighbors using any adopted acceptance rules such as first improvement, best improvement, random walk, or sidewalk.” (Al-betar, M.A., 2017). The evolutionary algorithm implemented in this code randomly searches for the best solution (the lowest bestfitness value), using a hill climber to run the functions I got these results.

First Function	Second Function
369.05	257.65
406.02	240.83
319.27	281.59
397.22	213.83
352.05	229.08
368.72	244.60

Table 9. Tables generated using Hillclimber

Compared to the fitness values generated using the evolutionary algorithm, these values produced are large. The aim of this was to find the lowest fitness values, while these are the lowest fitness values produced using the hillclimber algorithm it is much larger than those produced using the evolutionary algorithm.

A problem with the hillclimber is that it can get stuck in a local optima which I believe is what happened here,



Multiple numbers of the same values were produced during each run, which clearly shows that the program got stuck in a local optima. Compared to the evolutionary algorithm that searches the range of spaces available, the hillclimber searches downward from the initial solution. The search movement of the evolutionary algorithm made way for different probabilities but due to the search nature of a hillclimber, once stuck in a local optima its search stops there.

4 CONCLUSIONS

The larger the population and generation size the larger the search space, these give way for larger probabilities to be generated from the program. To produce even smaller fitness values a larger search space needs to be created. For these functions, I noted that the MUTRATE and the MUTSTEP only produce small fitness values within a certain range regardless of the generation and population size, outside the range it produces large fitness values that stray from the aim of the experiment, the range that produces the bestfitness values based on the experiments are ranges that are neither too large nor too small. From this experiment, I noticed that even with the parameters that give better results, due to the random selection of numbers the fitness values do not always stay within the desired range. To improve this algorithm, I would implement another search algorithm within the evolutionary algorithm to gain better control of the random selection of numbers made by the program, this way I can direct the program to produce smaller bestfitness values and improve the effectiveness of the program.

REFERENCES

Al-betar, M.A. (2017) Hill Climbing: An Exploratory Local Search. *Neural Comput & Applic* [online]. 28, pp. 153-168. [Accessed 06 December 2023].

APPENDIX

First Function:

```
import random
import copy
import matplotlib.pyplot as plt
import numpy as np
import math

#random. randint(0, 1)
#number of genes
N= 20
#population size
```

```
P= 500
G = 200
MUTRATE = 0.009
MUTSTEP = 2.00
#first worksheet, xi ranges from -5.12 to 5.12, in this
   one x ranges -10 to 10, how does
#this apply in the code, max and min is used in
   mutation, mutation step?
MIN = -10
MAX = 10



class individual:
   def __init__(self):
      self.gene = [0]*N
      self.fitness=0

#edit this part
def test_function( ind ):
   a = 0
   b=0

   for i in range (1,N):
      b = b + (i*(2*ind.gene[i])**2 - (ind.gene[i-
   1]))**2

   a = a + ((ind.gene[0]-1)**2)
   utility = a + b
   return utility



population = []
#initializing population


#list of total fitness
def fits():
   for x in range(0, P):
      total_fit=[]
      for y in range(0,N):
         total_fit.append(random.uniform(MIN,
   MAX))
      newind = individual()
      newind.gene=total_fit.copy()

         # calculate the fitness using the test function
      # newind.fitness = sum(total_fit)
      newind.fitness = test_function(newind)
      population.append(newind)
   return total_fit

#append total fitness to plot

#call the fitness function
fits()
#make functions
#selection func
offspring = []
def selection():
   #print(len(population))
   for i in range (0,P):
      parent1= random.randint(0, P-1)
      off1 = copy.deepcopy(population[parent1])
      parent2 = random.randint(0, P-1)
      off2 = copy.deepcopy(population[parent2])
      if off1.fitness < off2.fitness:
         offspring.append(off1)
      else:
         offspring.append(off2)
   return offspring

#crossover function
def crossover():
   toff1 = individual()
   toff2 = individual()
   temp = individual()
   for i in range (0, P, 2):
      toff1 = copy.deepcopy(offspring[i])
      toff2 = copy.deepcopy(offspring[i+1])
      temp = copy.deepcopy(offspring[i])
      crosspoint= random.randint(1,N)
      crosspoint2 = random.randint(crosspoint, N)
      for j in range(crosspoint, crosspoint2):
         toff1.gene[j] = toff2.gene[j]
         toff2.gene[j] = temp.gene[j]
      offspring[i] = copy.deepcopy(toff1)
      offspring[i+1]= copy.deepcopy(toff2)
   return offspring

#mutation func
#mutation function is edited too, we work with floats
   now do random.uniform
def mutation():
   for i in range (0, P):
      newind = individual()
      newind.gene=[]
      for j in range (0, N):
         gene = offspring[i].gene[j]
```

```
        mutprob = random.random()
        #print(mutprob)
        if mutprob < MUTRATE:
            alter = random.uniform(-
    MUTSTEP,MUTSTEP)
            gene = gene + alter
            if gene > MAX:
                gene = MAX
            if gene < MIN:
                gene = MIN
            #if(gene==1):
                #gene = 0
            #else:
                #gene = 1
        newind.gene.append(gene)
      newind.fitness = test_function(newind)
       #you must now append the new individual or
    overwite offspring
      offspring[i]=copy.deepcopy(newind)
    return newind

#generation loop
#for each time the mutation loops runs and creates
    new mutations, the generation increases, so count?

Gen = [] #a list of the generation, to plot against the
    fitness of the generation
avrgen= []

for x in range(0, G):
    offspring = [] #added
    selection() #withing the loop, select new parents
    and create offsprings
    crossover()
    mutation()
    #fits()#fitness to calculate the fitness of each
    generation made within the loop
    bestfit = population[0].fitness
    for i in range(0, P):
        fitnessi = copy.deepcopy(population[i].fitness)
        #print(fitnessi)
        if fitnessi < bestfit:
            bestfit = fitnessi
    #print(bestfit)

    total = 0
    for i in range(P):
      total = (total + population[i].fitness)
    #print(f"avg: {total/P}")
    avrgen.append(total/ P)
    Gen.append(bestfit)#append into the list
    population= offspring

print(f"Minimum bestfitness in the generations:
    {min(Gen)}")
print(f"Lowest average in the generations:
    {min(avrgen)}")
plt.plot(avrgen, label = "Average fitness")
plt.plot(Gen, label = "Bestfitness")
plt.xlabel('Generation')
plt.ylabel('Fitness')
plt.legend()
plt.show()
#for each generations fitness plot against generation
```

Second Function:

```
import random
import copy
import matplotlib.pyplot as plt
import numpy as np
import math

#random. randint(0, 1)
#number of genes
N= 20
#population size
P= 500
G = 500
MUTRATE = 0.008
MUTSTEP = 1.00
MIN = -5
MAX = 10


class individual:
    def __init__(self):
        self.gene = [0]*N
        self.fitness=0

#edit this part
def test_function( ind ):
    utility = 0
    a=0
    b=0
    c=0
    for i in range (0,N):
        #print(i)
        a = a + ind.gene[i]**2
```

```
        b = b + (0.5*(i+1)*ind.gene[i])
        c = c + (0.5*(i+1)*ind.gene[i])

    utility= a + b**2 + c**4
    return utility


population = []
#initializing population


#list of total fitness
def fits():
    for x in range(0, P):
        total_fit=[]
        for y in range(0,N):
            total_fit.append(random.uniform(MIN,
    MAX))
        newind = individual()
        newind.gene=total_fit.copy()

            # calculate the fitness using the test function
        #newind.fitness = sum(total_fit)
        newind.fitness = test_function(newind)
        population.append(newind)
    return total_fit
#append total fitness to plot

#call the fitness function
fits()
#make functions
#selection func
offspring = []
def selection():
    #print(len(population))
    for i in range (0,P):
        parent1= random.randint(0, P-1)
        off1 = copy.deepcopy(population[parent1])
        parent2 = random.randint(0, P-1)
        off2 = copy.deepcopy(population[parent2])
        if off1.fitness < off2.fitness:
            offspring.append(off1)
        else:
            offspring.append(off2)
    return offspring

#crossover function
def crossover():
    toff1 = individual()
    toff2 = individual()
    temp = individual()
    for i in range (0, P, 2):
        toff1 = copy.deepcopy(offspring[i])
        toff2 = copy.deepcopy(offspring[i+1])
        temp = copy.deepcopy(offspring[i])
        crosspoint= random.randint(1,N)
        crosspoint2 = random.randint(crosspoint, N)
        for j in range(crosspoint, crosspoint2):
            toff1.gene[j] = toff2.gene[j]
            toff2.gene[j] = temp.gene[j]
        offspring[i] = copy.deepcopy(toff1)
        offspring[i+1]= copy.deepcopy(toff2)
    return offspring

#mutation func
#mutation function is edited too, we work with floats
    now do random.uniform
def mutation():
    for i in range (0, P):
        newind = individual()
        newind.gene=[]
        for j in range (0, N):
            gene = offspring[i].gene[j]
            mutprob = random.random()

            if mutprob < MUTRATE:
                alter = random.uniform(-
    MUTSTEP,MUTSTEP)
                gene = gene + alter
                if gene > MAX:
                    gene = MAX
                if gene < MIN:
                    gene = MIN
                #if(gene==1):
                    #gene = 0
                #else:
                    #gene = 1
            newind.gene.append(gene)
        newind.fitness = test_function(newind)
         #you must now append the new individual or
    overwite offspring
        offspring[i]=copy.deepcopy(newind)
    return newind

#generation loop

Gen = [] #a list of the generation, to plot against the
    fitness of the generation
avrgen= []
for x in range(0, G):
```

```
    offspring = [] #added
    selection() #withing the loop, select new parents and create offsprings
    crossover()
    mutation()
    #fits()#fitness to calculate the fitness of each generation made within the loop
    bestfit = population[0].fitness
    for i in range(0, P):
        fitnessi = copy.deepcopy(population[i].fitness)
        #print(fitnessi)
        if fitnessi < bestfit:
            bestfit = fitnessi
    #print(bestfit)

    total = 0
    for i in range(P):
      total = (total + population[i].fitness)

    avrgen.append(total/ P)
    Gen.append(bestfit)#append into the list
    population= copy.deepcopy(offspring)

print(f"Minimum bestfitness in the generations: {min(Gen)}")
print(f"Lowest average in the generations: {min(avrgen)}")
plt.plot(avrgen, label = "Average fitness")
plt.plot(Gen, label = "Bestfitness")
plt.xlabel('Generation')
plt.ylabel('Fitness')
plt.legend()
plt.show()
#for each generations fitness plot against generation
```

Hillclimber for first function:

```
import random
import math

MIN=-10
MAX=10
N=20
LOOPS = 500*200
class solution:
    variable = [0]*N
    utility = 0
individual = solution()

for j in range (N):
    individual.variable[j] = random.uniform(MIN,MAX)
individual.utility = 0

def test_function( ind ):
    utility = 0
    b=0
    for i in range (N):
      b = b + ( ind.variable[i]**2 - (10* math.cos((2*math.pi)*ind.variable[i])))
    a = 10*N
    utility = a + b
    return utility


individual.utility= test_function(individual)
newind = solution()
for x in range (LOOPS):
    for i in range(N):
        newind.variable[i] = individual.variable[i]
    change_point = random.randint(0, N-1)
    newind.variable[change_point] = random.uniform(MIN,MAX)
    newind.utility = test_function( newind )
    if individual.utility >= newind.utility:
        individual.variable[change_point] = newind.variable[change_point]
        individual.utility = newind.utility
    print(individual.utility)
```

Hillclimber for second function:

```
import random
import math

MIN=-10
MAX=10
N=20
LOOPS = 500*500
class solution:
    variable = [0]*N
    utility = 0
individual = solution()

for j in range (N):
    individual.variable[j] = random.uniform(MIN,MAX)
individual.utility = 0

def test_function( ind ):
    utility = 0
    a=0
```

```
    b=0
    c=0
    for i in range (0,N):
        a = a + ind.variable[i]**2
        b = b + (0.5*(i+1)*ind.variable[i])
        c = c + (0.5*(i+1)*ind.variable[i])

    utility= a + b**2 + c**4
    return utility


individual.utility= test_function(individual)
newind = solution()
for x in range (LOOPS):
    for i in range(N):
        newind.variable[i] = individual.variable[i]
    change_point = random.randint(0, N-1)
    newind.variable[change_point]                    =
    random.uniform(MIN,MAX)
    newind.utility = test_function( newind )
    if individual.utility >= newind.utility:
        individual.variable[change_point]            =
    newind.variable[change_point]
        individual.utility = newind.utility
    print(individual.utility)
```