RAMAIAH UNIVERSITY
OF APPLIED SCIENCES

ASSIGNMENT

Course Code	CSC401A
Course Name	COMPUTATIONAL INTELLIGENCE
Programme	B.Tech
Department	Computer Science
Faculty	FET

Name of the Student	SHUBHAM AGARWAL
Reg. No	17ETCS002175
Semester/Year	7th /4th year
Course Leader/s	Mr. Sagar U

Declaration Sheet

Student Name	SHUBHAM AGARWAL		
Reg. No	17ETCS002175		
Programme	B.Tech	Semester/Year	7th /4th year
Course Code	CSC401A		
Course Title	Computational Intelligence		
Course Date	****	to	****
Course Leader	Mr. Sagar U		

Declaration

The assignment submitted herewith is a result of my own investigations and that I have conformed to the guidelines against plagiarism as laid out in the Student Handbook. All sections of the text and results, which have been obtained from other sources, are fully referenced. I understand that cheating and plagiarism constitute a breach of University regulations and will be dealt with accordingly.

Signature of the Student		Date	05/12/2020
Submission date stamp (by Examination & Assessment Section)			

Signature of the Course Leader and date	Signature of the Reviewer and date

Faculty of Engineering and Technology			
Ramaiah University of Applied Sciences			
Department	Computer Science and Engineering	Programme	B. Tech. in CSE
Semester/Batch	7/2017		
Course Code	CSC401A	Course Title	Computational Intelligence
Course Leader	Dr. Vaishali R. Kulkarni/Prof. Prabhakar/Mr. Sagar U.		

Assignment-01			
Reg.No.	**17ETCS002175**	**Name of Student**	**SHUBHAM AGARWAL**

Sections	**Marking Scheme**		**Mark**		
			Max Marks	**First Examiner Marks**	**Moderator**
Part A					
	A.1.1	**Pitfalls in traditional AI**	**02**		
	A.1.2	**Synergism of CI tools**	**03**		
		Part-A Max Marks	**5**		
Part B.1					
	B.1.1	**Key ideas in hill-climbing approach**	**04**		
	B.1.2	**Key ideas in alternative CI approach**	**04**		
	B.1.3	**Python Program demonstration**	**02**		
		B.1 Max Marks	**10**		
Part B.2					
	B.2.1	**Discussion on genetic algorithm and benchmark functions**	**04**		
	B.2.2	**Python program showing the minimized values**	**04**		
	B.2.3	**Comparison with any other heuristic algorithm**	**02**		
		B.2 Max Marks	**10**		
		Total Assignment Marks	**25**		

Course Marks Tabulation				
Component-1 (B) Assignment	**First Examiner**	**Remarks**	**Moderator**	**Remarks**
A				
B.1				
B.2				
Marks (out of 25)				

Contents

List of Figures

1. Part A

Solution to Part A Question No 1:

The relationship shared by traditional AI and CI.

Machine learning algorithms can train by teaching themselves as new data comes in. This activity is taking place today, but problems already have arisen that bode ill for future applications lacking appropriate measures. Machine learning algorithms can be deceived by adversaries in ways that affect the fidelity of the information gleaned from the data, AI can be self-deceiving. “Any algorithm that can change itself can corrupt itself”.

A.1.1 Pitfalls in traditional AI

No one could argue that life hasn’t been more convenient since the initial breakthrough of artificial intelligence (AI) in 2012. The world is moving faster and faster, driven by technology and innovation. However, with the hype and noise around AI, it’s easy for companies to take missteps along the way when trying to take advantage of the technology. As it’s always a good idea to exercise caution when deploying an AI-driven service, here are a few common pitfalls to watch out for:

1. Being Shielded by the Hype:

Adopters need to look beyond the hype to accurately judge AI's benefits, as it’s far too easy for organizations to underestimate the time, knowledge, and data required to effectively implement AI systems.

Organizations give authorities of decision-making power to AI just after implementation. It’s common to be blinded by the hype and jump fully into AI, but at the same time one needs to give time for AI to learn and grow into its own state. AI success purely depends on trusts to make decisions and take actions.

2. Lacking IT Team to Manage AI Effectively:

Organizations with AI which has capabilities and insights, but without proper team, the utility and management of AI resources cannot be handled properly hence organizations will be less likely to take full advantage of AI.

Current IT Organizations need huge automation to compete with current trends, growth. But this does not mean that one needs complex algorithm to solve and solution for the given situation. Having effective data collectors that feed into a condition system is a great way to get good data into the system.

The key throughout all of this, however, will be ensuring organizations have an IT team that can manage these AI systems and bring insights to the information gathered.

3. Trying to Keep Up With Google

Organizations also can’t fall into the trap of comparing themselves to Google. Google used the DeepMind AI engine to make their data centers more efficient. They use a system of neural networks, with a firm grasp of mechanics. One needs to have high training and huge test sets to validate the data. Developing and utilizing neural networks the correct way requires a ton of experts and computing resources, which Google clearly has and cannot be compared.

4. Using AI to Answer All of an Organization’s Problems

It's helpful to utilize alert while conveying an AI-driven assistance. While there is this idea that AI is here to make all the difference, it positively helps — yet isn't the end-all. There is no point-and-click, off-the-shelf AI programming that "makes my server farm work better." Organizations that share this idea will be set up for disappointment.

5. Beware of human bias:

Yes, AI and machine learning can help mitigate or remove human bias, but they’re not foolproof, especially if the software team doesn’t address the bias issue when designing and training models.

There have been more than a few high-profile AI fails that have caused companies to take a step back and think about how bias is introduced into the data. These types of bias could cause consumers to distrust AI systems permanently, potentially impacting the technology’s future uses.

To avoid these scenarios, developers should carefully consider the kinds of data used to train algorithms and where that data comes from. It is important to create specific rules about data selection to provide guidelines for developers.

6. Create a feedback loop.

Communication is key for AI and machine learning development, yet too many teams get stuck in silos and fail to integrate a consistent feedback loop into their processes. With a proper feedback loop, the team stays up to speed on all issues, including what’s going right and what can be improved. Initially, you might start with a communications sequence limited to subject matter experts, but as machine learning models evolve, it is important to broaden the feedback loop to include other stakeholders in the process. Ultimately, you want the model to be able to collect feedback on its own in an automated fashion.

7. Avoid analysis paralysis.

It happens to the best of us — getting stuck on a machine model for too long with the quest for perfection that causes delays in the delivery cycle. AI and machine learning are all about iteration and continuous improvement, so the work is never really done. The more data fed to the hungry beast, the better the model will be. At some point, however, you need to draw the line and get the product out the door.

A.1.2 Synergism of CI tools

Computational intelligence is a group of computational models and tools that encompass elements of learning, adaptation and heuristic optimization. It is used to help study problems that are difficult to solve using conventional computational algorithms.

The technology of Computational Intelligence (CI) intensively exploits various mechanisms of interaction with humans and processes domain knowledge with intent of building intelligent systems. As commonly perceived, CI dwells on three highly synergistic technologies of neural networks, fuzzy sets or granular computing and evolutionary optimization. The currently available tools of computational intelligence include fuzzy sets, neural networks, genetic algorithms, probabilistic reasoning, and some aspects of chaos theory and computational learning theory.

Neural Network:

Artificial Neural Networks (ANN) is a supervised learning system built of a large number of simple elements, called neurons or perceptrons. Each neuron can make simple decisions, and feeds those decisions to other neurons, organized in interconnected layers. Together, the neural network can emulate almost any function, and answer practically any question, given enough training samples and computing power.

The idea of ANNs is based on the belief that working of human brain by making the right connections, can be imitated using silicon and wires as living neurons and dendrites. ANNs are composed of multiple nodes. The neurons are connected by links and they interact with each other. The output at each node is called its activation or node value.

Fuzzy Logic:

Fuzzy Logic (FL) is a method of reasoning that resembles human reasoning. The approach of FL imitates the way of decision making in humans that involves all intermediate possibilities between digital values YES and NO.

The conventional logic block that a computer can understand takes precise input and produces a definite output as TRUE or FALSE, which is equivalent to human’s YES or NO.

Fuzzy logic helps to deal with the uncertainty in engineering. Fuzzy logic is a generalization of standard logic, in which a concept can possess a degree of truth anywhere between 0.0 and 1.0. Fuzzy logic is a technique for representing and manipulating uncertain information.

Swarm Intelligence:

Swarm intelligence is an emerging field of biologically-inspired artificial intelligence based on the behavioral models of social insects such as ants, bees, wasps, termites etc. Swarm Intelligence is the Complex Collective, Self-Organized, Coordinated, Flexible and Robust Behavior of a group following the simple rules. Agents interacts locally with each other and the environment.

Advantages:

Flexible: The colony respond to internal disturbances and external challenges.

Robust: Tasks are completed even if some agents fail.

Scalable: From a few agents to millions

Self-organized: The solutions are emergent rather than pre-defined.

Adaptation: The swarm system can not only adjust to predetermined stimuli but also to new stimuli.

Speed: Changes in the network can be propagated very fast.

Modularity: Agents act independently of other network layers.

Evolutionary computational:

Evolutionary computation techniques are stochastic algorithms whose search methods model some natural phenomena: genetic inheritance and Darwinian strive for survival. The best known algorithms in this class include genetic algorithms, evolutionary programming, evolution strategies, and genetic programming. The idea behind evolutionary algorithms is to do what nature does.

Evolutionary algorithms are a group of algorithms which use their special operators as mutation, crossover, and others to find an ideal solution. Possible candidates are defined by a cost function in which arguments are values of each solution. The best one is in the global extreme—maximum or minimum.

Ant colony:

Ant colony optimization (ACO) is a Probabilistic technique and population-based metaheuristic that can be used to find approximate solutions to difficult optimization problems. Searching for optimal path in the graph-based method is behavior of ants seeking a path between their colony and source of food. Each ant moves at random. Shortest path is discovered via pheromone trails. The Ant colony is efficient for Traveling Salesman Problem and similar problems. ACO Can be used in dynamic applications (adapts to changes such as new distances, etc).

2. Part B

Solution to Part B Question No 1:

B.1.1 Key ideas in hill-climbing approach

- Looks one step ahead to determine if any successor is better than the current state; if there is, move to the best successor.
- Rule:

If there exists a successor s for the current state n such that

- h(s) < h(n) and
- h(s) ≤ h(t) for all the successors t of n,

then move from n to s. Otherwise, halt at n.

- Similar to Greedy search in that it uses h(), but does not allow backtracking or jumping to an alternative path since it doesn't "remember" where it has been.
- Corresponds to Beam search with a beam width of 1 (i.e., the maximum size of the nodes list is 1).
- Not complete since the search will terminate at "local minima," "plateaus," and "ridges."
- Hill climbing is also helpful to solve pure optimization problems where the objective is to find the best state according to the objective function.
- Optimal if the space to be searched is convex and Terminates when a peak is reached.
- Does not look ahead of the immediate neighbors of the current state.
- A heuristic function is a function that will rank all the possible alternatives at any branching step in search algorithm based on the available information. It helps the algorithm to select the best route out of possible routes.

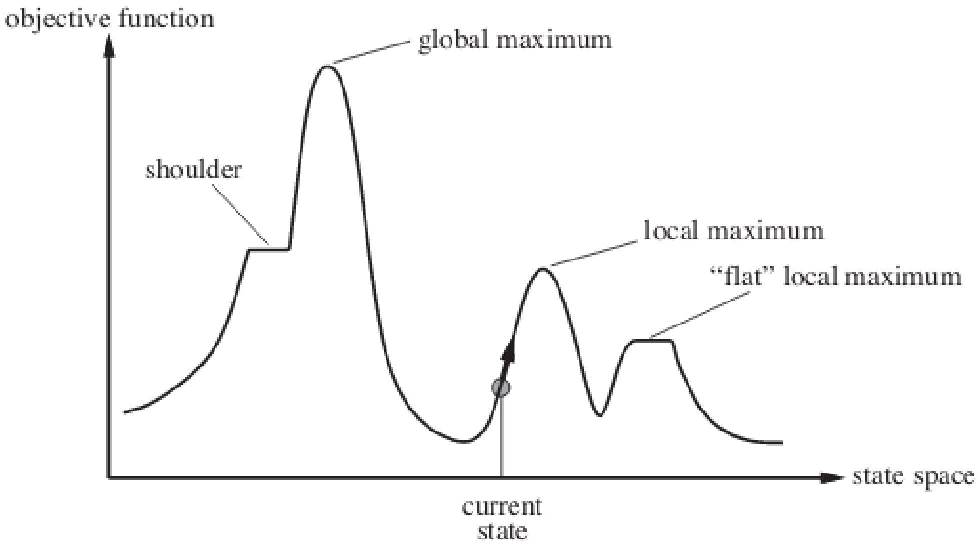


Figure 1: Hill climbing

Hill Climbing suffers from the following problems-

1. **Local Maxima:**

It is a state which is better than all of its neighbours but isn't better than some other states which are farther away. At local maxima, all moves appear to make the things worse. They are sometimes frustrating also as they often occur almost within slight of a solution. So, it is also called as Foot-Hills.

2. Plateau:

It is a flat area of the search space in which a whole set of neighboring states(nodes) have the same order. On a plateau, it is not possible to determine the best direction in which to move by making local comparisons. Actually, a plateau is an area of the state space landscape where the evaluation function is flat. It can be a flat local maximum or a shoulder from which it is possible to make progress.

3. Ridge:

It is a special kind of local maximum. it is an area of the search space which is higher than the surrounding areas and that itself has a slope. We can't travel the ridge by single moves as the orientation of the high region compared to the set of available moves makes it impossible.

B.1.2 Key ideas in alternative CI approach:

Simulated Annealing

- Simulated Annealing (SA) is an effective and general form of optimization. It is useful in finding global optima in the presence of large numbers of local optima
- Simulated annealing (SA) exploits an analogy between the way in which a metal cools and freezes into a minimum-energy crystalline structure (the annealing process) and the search for a minimum [or maximum] in a more general system.
- Simulated annealing (SA) is a probabilistic technique for approximating the global optimum of a given function. Specifically, it is a metaheuristic to approximate global optimization in a large search space for an optimization problem. It is often used when the search space is discrete (e.g., the traveling salesman problem). For problems where finding an approximate global optimum is more important than finding a precise local optimum in a fixed amount of time, simulated annealing may be preferable to exact algorithms such as gradient descent, Branch and Bound.
- Simulated annealing (SA) is based on the similarity between the solid annealing process and solving combinatorial optimization problems [50]. SA consists of several decreasing temperatures. Each temperature has a few iterations. First, the beginning temperature is selected, and an initial solution is randomly chosen. The value of the cost function based on the current solution (i.e., the initial solution in this case) will then be calculated. The goal is to minimize the cost function. Afterwards, a new solution from the neighborhood of the current solution will be generated. The new value of the cost function

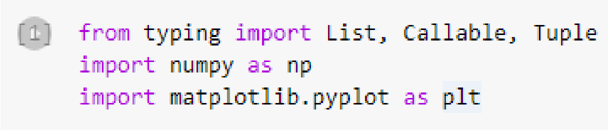
based on the new solution will be calculated and compared to the current cost function value. If the new cost function value is less than the current value, it will be accepted. Otherwise, the new value would be accepted.

- SA can avoid becoming trapped at local minima. SA uses a random search that accepts changes that increase objective function f, as well as some that decrease it.
- SA uses a control parameter T, which by analogy with the original application is known as the system "temperature." T starts out high and gradually decreases toward 0.
- A "bad" move from A to B is accepted with a probability P(moveA→B) = $e^{\frac{(f(B)-f(A))}{T}}$
- Designing a neighbor function is quite tricky and must be done on a case by case basis, but below are some ideas for finding neighbors in locational optimization problems.
  - Move all points 0 or 1 units in a random direction
  - Shift input elements randomly
  - Swap random elements in input sequence
  - Permute input sequence
  - Partition input sequence into a random number of segments and permute segments

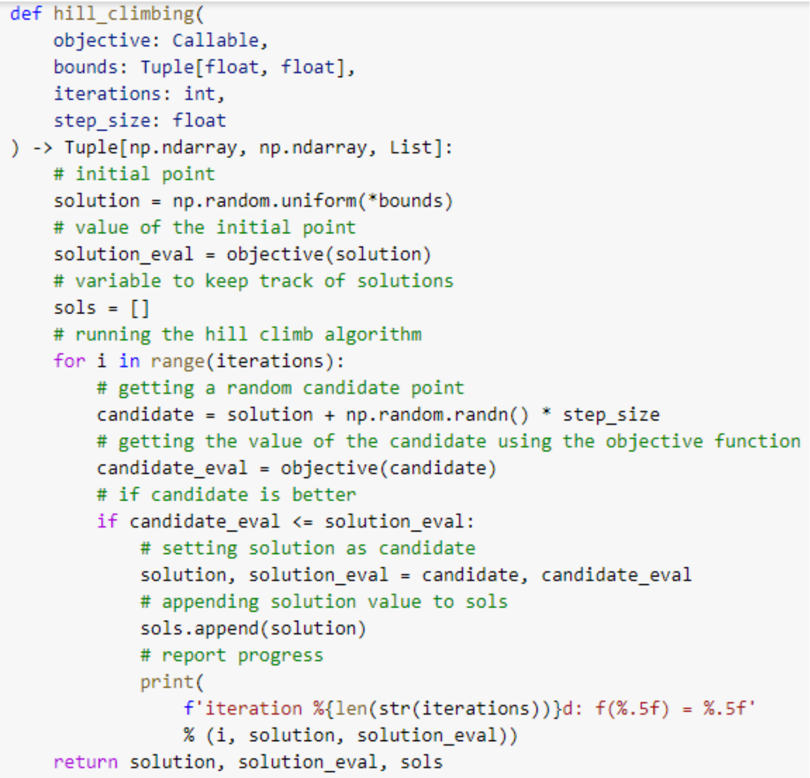
B.1.3 Python Program Demonstration of any test case using both approaches

Hill Climbing approach:

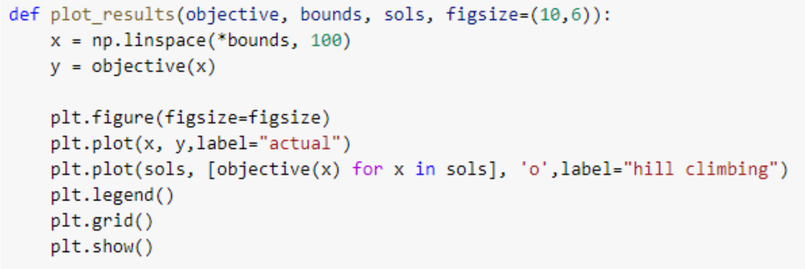
The objective function is $X^2 + 9$ with a bound of [-5,5] and number of iterations are set to 1000, with step size of 0.1. the python code is given below along with the output:



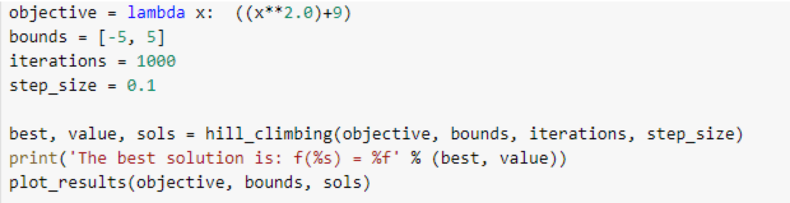
```
from typing import List, Callable, Tuple
import numpy as np
import matplotlib.pyplot as plt
```



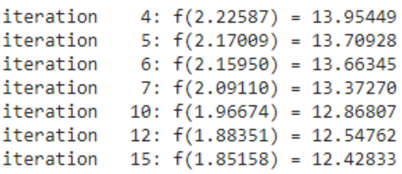
```
def hill_climbing(
    objective: Callable,
    bounds: Tuple[float, float],
    iterations: int,
    step_size: float
) -> Tuple[np.ndarray, np.ndarray, List]:
    # initial point
    solution = np.random.uniform(*bounds)
    # value of the initial point
    solution_eval = objective(solution)
    # variable to keep track of solutions
    sols = []
    # running the hill climb algorithm
    for i in range(iterations):
        # getting a random candidate point
        candidate = solution + np.random.randn() * step_size
        # getting the value of the candidate using the objective function
        candidate_eval = objective(candidate)
        # if candidate is better
        if candidate_eval <= solution_eval:
            # setting solution as candidate
            solution, solution_eval = candidate, candidate_eval
            # appending solution value to sols
            sols.append(solution)
            # report progress
            print(
                f'iteration %{len(str(iterations))}d: f(%.5f) = %.5f'
                % (i, solution, solution_eval))
    return solution, solution_eval, sols
```



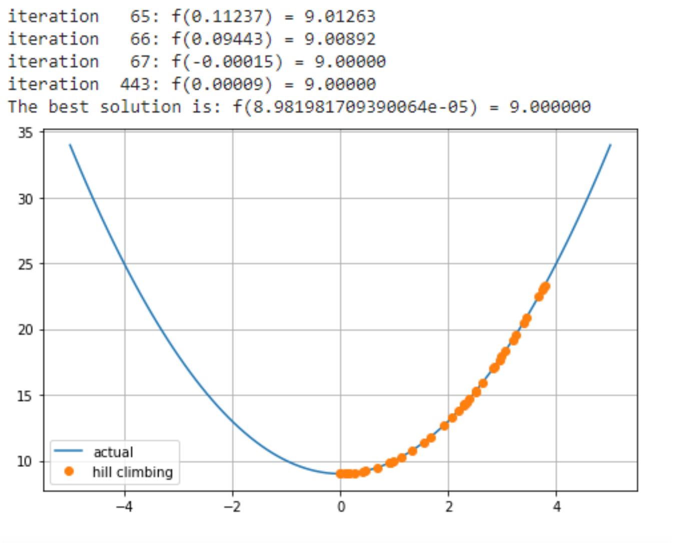
```
def plot_results(objective, bounds, sols, figsize=(10,6)):
    x = np.linspace(*bounds, 100)
    y = objective(x)

    plt.figure(figsize=figsize)
    plt.plot(x, y,label="actual")
    plt.plot(sols, [objective(x) for x in sols], 'o',label="hill climbing")
    plt.legend()
    plt.grid()
    plt.show()
```



```
objective = lambda x:  ((x**2.0)+9)
bounds = [-5, 5]
iterations = 1000
step_size = 0.1

best, value, sols = hill_climbing(objective, bounds, iterations, step_size)
print('The best solution is: f(%s) = %f' % (best, value))
plot_results(objective, bounds, sols)
```



```
iteration    4: f(2.22587) = 13.95449
iteration    5: f(2.17009) = 13.70928
iteration    6: f(2.15950) = 13.66345
iteration    7: f(2.09110) = 13.37270
iteration   10: f(1.96674) = 12.86807
iteration   12: f(1.88351) = 12.54762
iteration   15: f(1.85158) = 12.42833
```

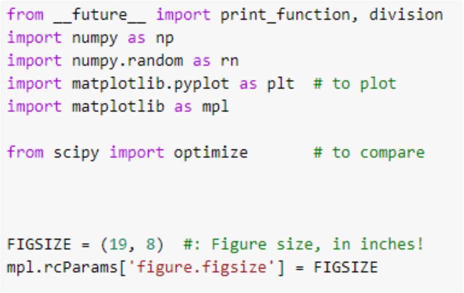


```
iteration   65: f(0.11237) = 9.01263
iteration   66: f(0.09443) = 9.00892
iteration   67: f(-0.00015) = 9.00000
iteration  443: f(0.00009) = 9.00000
The best solution is: f(8.981981709390064e-05) = 9.000000
```

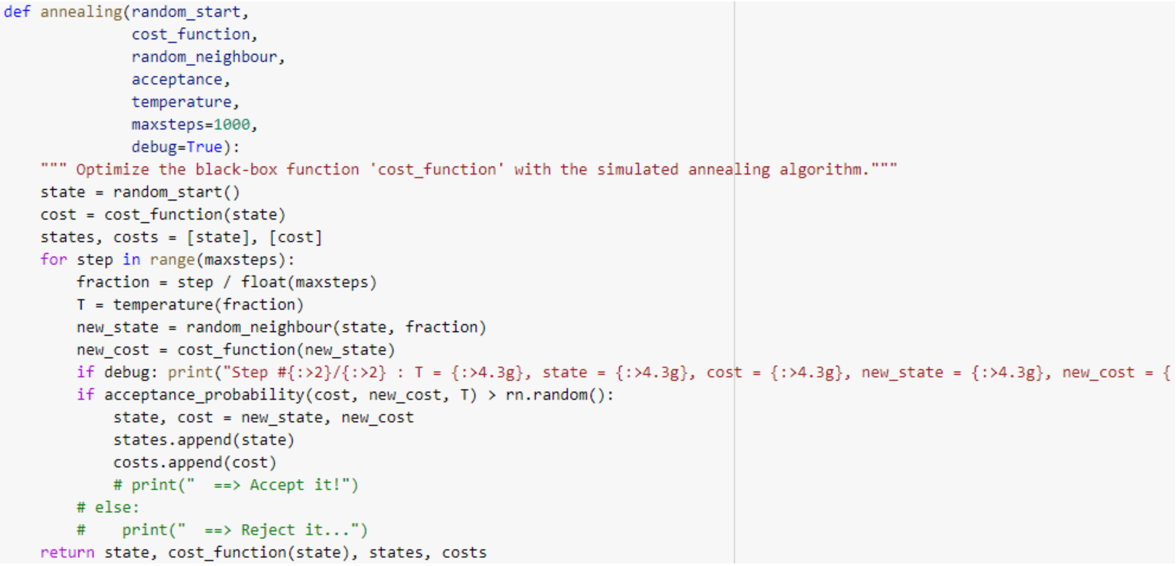
Simulated annealing:

The objective function is $X^2 + 9$ with a bound of [-10,10] and number of iterations are set to 1000, with step size of 0.1. the python code is given below along with the output:

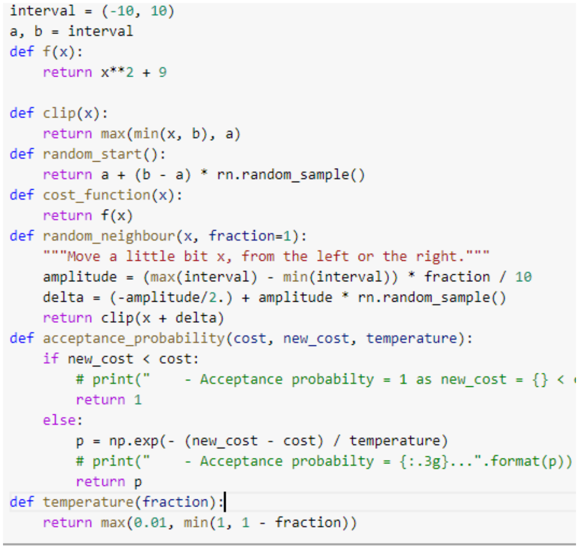
```
Algorithm
Create random initial solution γ
 Eold=cost( γ);
  for(temp=tempmax; temp>=tempmin;temp=next_temp(temp) ) {
          for(i=0;i<imax;i++)
                    successorfunc(γ);
                    Enew= cost(γ)
                    Delta =Enew- Eold
                    If (delta > 0):
                              if(random() >= exp(-delta/K*temp);
                                        undofunc(γ)
                              else Eold= Enew
                    else Eold = Enew
          }
}
```



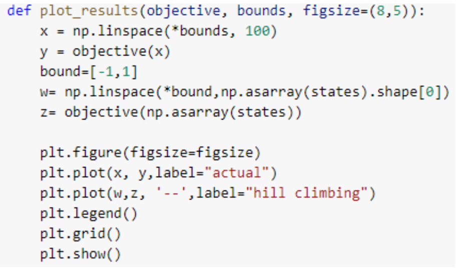
```python
from __future__ import print_function, division
import numpy as np
import numpy.random as rn
import matplotlib.pyplot as plt  # to plot
import matplotlib as mpl

from scipy import optimize       # to compare



FIGSIZE = (19, 8)  #: Figure size, in inches!
mpl.rcParams['figure.figsize'] = FIGSIZE
```



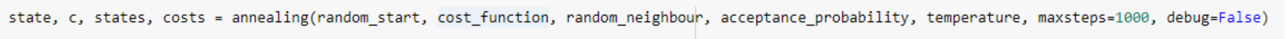
```
def annealing(random_start,
              cost_function,
              random_neighbour,
              acceptance,
              temperature,
              maxsteps=1000,
              debug=True):
    """ Optimize the black-box function 'cost_function' with the simulated annealing algorithm."""
    state = random_start()
    cost = cost_function(state)
    states, costs = [state], [cost]
    for step in range(maxsteps):
        fraction = step / float(maxsteps)
        T = temperature(fraction)
        new_state = random_neighbour(state, fraction)
        new_cost = cost_function(new_state)
        if debug: print("Step #{:>2}/{:>2} : T = {:>4.3g}, state = {:>4.3g}, cost = {:>4.3g}, new_state = {:>4.3g}, new_cost = {
        if acceptance_probability(cost, new_cost, T) > rn.random():
            state, cost = new_state, new_cost
            states.append(state)
            costs.append(cost)
            # print("  ==> Accept it!")
        # else:
        #    print("  ==> Reject it...")
    return state, cost_function(state), states, costs
```



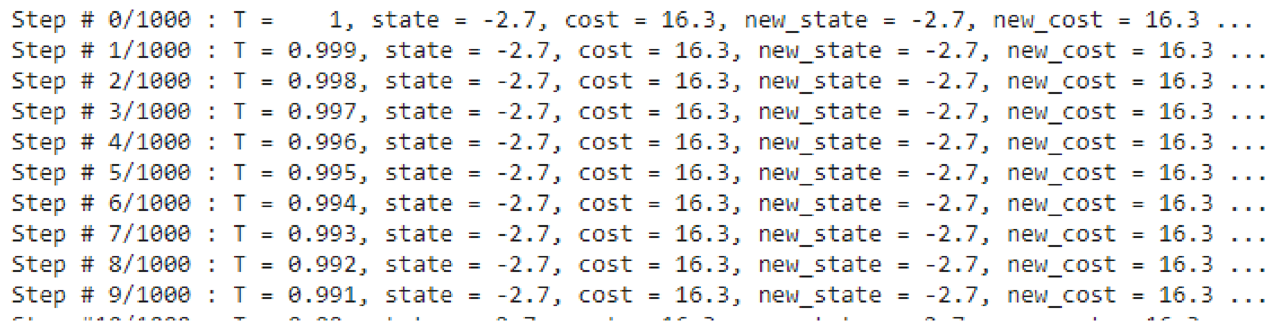
```
interval = (-10, 10)
a, b = interval
def f(x):
    return x**2 + 9

def clip(x):
    return max(min(x, b), a)
def random_start():
    return a + (b - a) * rn.random_sample()
def cost_function(x):
    return f(x)
def random_neighbour(x, fraction=1):
    """Move a little bit x, from the left or the right."""
    amplitude = (max(interval) - min(interval)) * fraction / 10
    delta = (-amplitude/2.) + amplitude * rn.random_sample()
    return clip(x + delta)
def acceptance_probability(cost, new_cost, temperature):
    if new_cost < cost:
        # print("    - Acceptance probabilty = 1 as new_cost = {} < 
        return 1
    else:
        p = np.exp(- (new_cost - cost) / temperature)
        # print("    - Acceptance probabilty = {:.3g}...".format(p))
        return p
def temperature(fraction):
    return max(0.01, min(1, 1 - fraction))
```



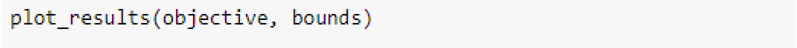
```
def plot_results(objective, bounds, figsize=(8,5)):
    x = np.linspace(*bounds, 100)
    y = objective(x)
    bound=[-1,1]
    w= np.linspace(*bound,np.asarray(states).shape[0])
    z= objective(np.asarray(states))

    plt.figure(figsize=figsize)
    plt.plot(x, y,label="actual")
    plt.plot(w,z, '--',label="hill climbing")
    plt.legend()
    plt.grid()
    plt.show()
```



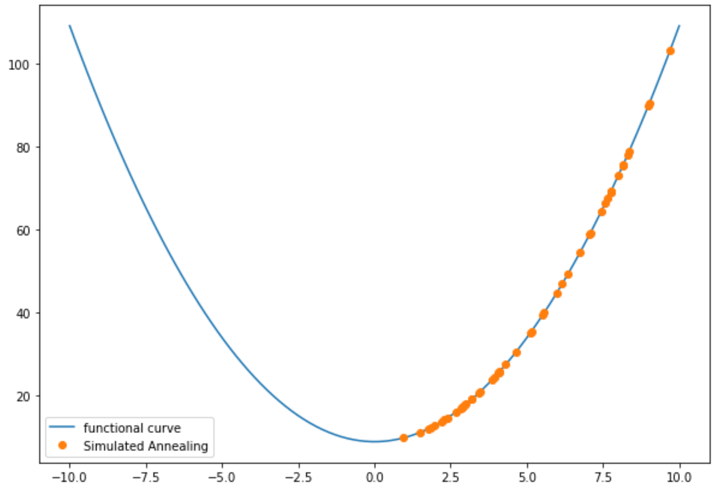
```
state, c, states, costs = annealing(random_start, cost_function, random_neighbour, acceptance_probability, temperature, maxsteps=1000, debug=False)
```



```
Step # 0/1000 : T =     1, state = -2.7, cost = 16.3, new_state = -2.7, new_cost = 16.3 ...
Step # 1/1000 : T = 0.999, state = -2.7, cost = 16.3, new_state = -2.7, new_cost = 16.3 ...
Step # 2/1000 : T = 0.998, state = -2.7, cost = 16.3, new_state = -2.7, new_cost = 16.3 ...
Step # 3/1000 : T = 0.997, state = -2.7, cost = 16.3, new_state = -2.7, new_cost = 16.3 ...
Step # 4/1000 : T = 0.996, state = -2.7, cost = 16.3, new_state = -2.7, new_cost = 16.3 ...
Step # 5/1000 : T = 0.995, state = -2.7, cost = 16.3, new_state = -2.7, new_cost = 16.3 ...
Step # 6/1000 : T = 0.994, state = -2.7, cost = 16.3, new_state = -2.7, new_cost = 16.3 ...
Step # 7/1000 : T = 0.993, state = -2.7, cost = 16.3, new_state = -2.7, new_cost = 16.3 ...
Step # 8/1000 : T = 0.992, state = -2.7, cost = 16.3, new_state = -2.7, new_cost = 16.3 ...
Step # 9/1000 : T = 0.991, state = -2.7, cost = 16.3, new_state = -2.7, new_cost = 16.3 ...
```



```
plot_results(objective, bounds)
```



Simulated Annealing is an algorithm that never makes a move towards lower esteem destined to be incomplete that it can stall out on a nearby extreme.

A similar procedure is utilized in reenacted toughening in which the calculation picks an arbitrary move, rather than picking the best move. In the event that the irregular move improves the state, at that point, it follows a similar way. Something else, the calculation follows the way which has a likelihood of under 1 or it moves downhill and picks another way.

3. Part B

Solution to Part B Question No 2:

B.2.1 Discussion on genetic algorithm and benchmark functions:

Genetic Algorithms are a class of probabilistic algorithm that are loosely based on biological evolution. The genetic algorithm is a method for solving both constrained and unconstrained optimization problems that is based on natural selection. The genetic algorithm repeatedly modifies a population of individual solutions. At each step, the genetic algorithm selects individuals at random from the current population to be parents and uses them to produce the children for the next generation. Over successive generations, the population "evolves" toward an optimal solution. The genetic algorithm can address problems of mixed integer programming, where some components are restricted to be integer-valued.

Genetic Algorithm (GA) is a search-based optimization technique based on the principles of Genetics and Natural Selection. It is frequently used to find optimal or near-optimal solutions to difficult problems which otherwise would take a lifetime to solve. It is frequently used to solve optimization problems, in research, and in machine learning.

In GAs, we have a pool or a population of possible solutions to the given problem. These solutions then undergo recombination and mutation (like in natural genetics), producing new children, and the process is repeated over various generations. Each individual (or candidate solution) is assigned a fitness value (based on its objective function value) and the fitter individuals are given a higher chance to mate and yield more "fitter" individuals.

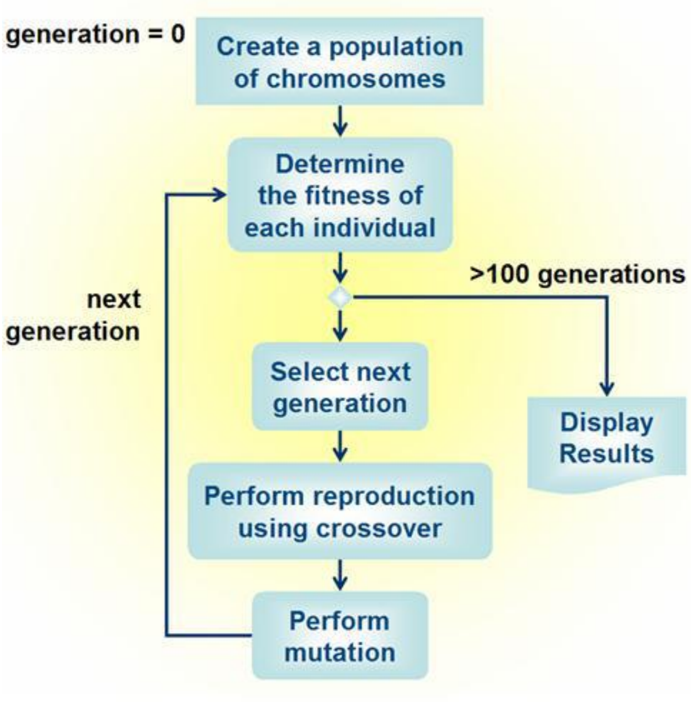

generation = 0
Create a population of chromosomes
Determine the fitness of each individual
>100 generations
next generation
Select next generation
Display Results
Perform reproduction using crossover
Perform mutation

Figure 2: Flowchart of Genetic Algorithm

Advantages of GA are:

- Does not require any derivative information (which may not be available for many real-world problems). And GA Is faster and more efficient as compared to the traditional methods. Also GA Has very good parallel capabilities.
- Optimizes both continuous and discrete functions and also multi-objective problems.
- Provides a list of “good” solutions and not just a single solution.
- Always gets an answer to the problem, which gets better over the time.
- Useful when the search space is very large and there are a large number of parameters involved.

Disadvantages of GA:

- Fitness value is calculated repeatedly which might be computationally expensive for some problems.
- Being stochastic, there are no guarantees on the optimality or the quality of the solution.
- If not implemented properly, the GA may not converge to the optimal solution.

Benchmark Function:

The function, which can be used to test performance of any optimization approach and the related problem. In computing, a benchmark is the act of running a computer program, a set of programs, or other operations, in order to assess the relative performance of an object, normally by running a number of standard tests and trials against it. Benchmarks provide a method of comparing the performance of various subsystems across different chip/system architectures.

Benchmarking is not easy and often involves several iterative rounds in order to arrive at predictable, useful conclusions. Interpretation of benchmarking data is also extraordinarily difficult.

There are seven vital characteristics for benchmarks. These key properties are:

1. Relevance: Benchmarks should measure relatively vital features.
2. Representativeness: Benchmark performance metrics should be broadly accepted by system.
3. Equity: All systems should be fairly compared.
4. Repeatability: Benchmark results can be verified.
5. Cost-effectiveness: Benchmark tests are economical.
6. Scalability: Benchmark tests should work across systems possessing a range of resources.
7. Transparency: Benchmark metrics should be easy to understand.

Different types of Benchmark function:

There are many different benchmark function which can be used to test the performance and optimize the solution. These functions are selected based on convergence , continuity , differentiability and separable.

Some of the Benchmark functions are stated: Sphere Function, High Conditioned Elliptic Function, Discus Function, Rosenbrock Function, Ackley Function, Weierstrass Function, Himmelblau's function, Booth function and many more.

Sphere Function:

The Benchmark Functions are used to compute the performance of Genetic algorithm. Its optimization is compared with different probability of crossover and mutation.

The **sphere function** is a quadratic function by nature and has properties as follows:

- The function is continuous.
- The function is convex.
- The function can be defined on n-dimensional space.
- The function is differentiable.
- The function is separable.
- The function is unimodal.

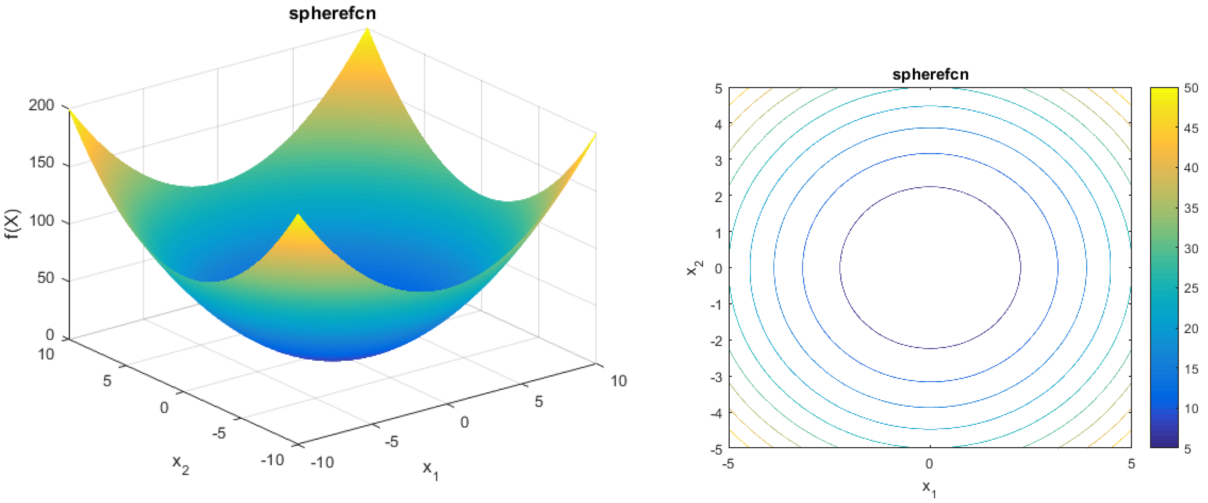


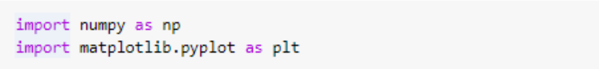
Figure 3: spherical plot of sphere function, contour plot of sphere function.

$$f(x) = \sum_{i=1}^{d} x_i^2$$

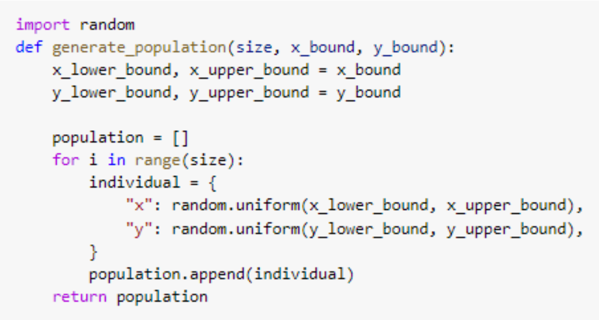
The Sphere function has d local minima except for the global one. It is continuous, convex and unimodal. The plot shows its two-dimensional form. The function is usually evaluated on the hypercube $x_i \in [-5.12, 5.12]$, for all i = 1, …, d. The global minima is $f(x_1 \ldots x_n) = f(0, \ldots, 0) = 0$.

B.2.2 Python program showing the minimized values:

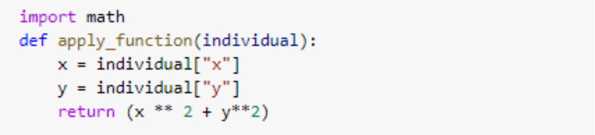
The below Python code for Genetic Algorithm with Sphere Benchmark Function is implemented and the output along with the graph is tabulated.



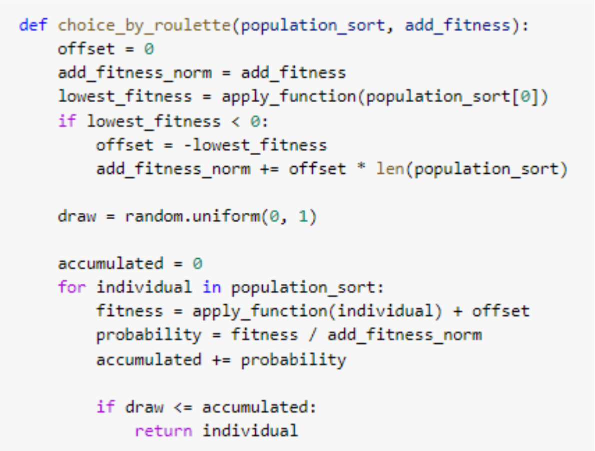
```
import numpy as np
import matplotlib.pyplot as plt
```



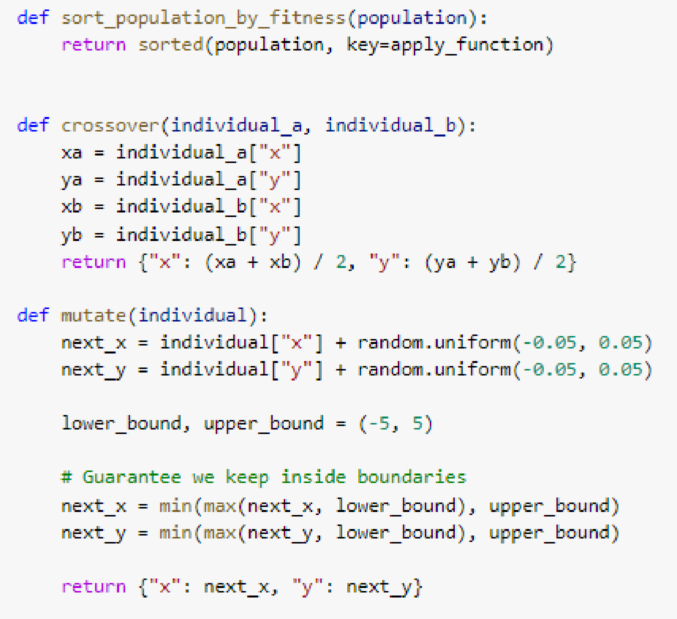
```
import random
def generate_population(size, x_bound, y_bound):
    x_lower_bound, x_upper_bound = x_bound
    y_lower_bound, y_upper_bound = y_bound

    population = []
    for i in range(size):
        individual = {
            "x": random.uniform(x_lower_bound, x_upper_bound),
            "y": random.uniform(y_lower_bound, y_upper_bound),
        }
        population.append(individual)
    return population
```



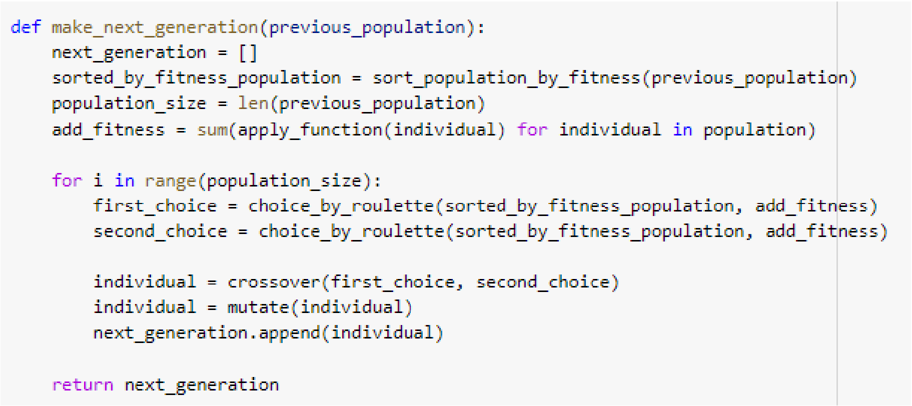
```
import math
def apply_function(individual):
    x = individual["x"]
    y = individual["y"]
    return (x ** 2 + y**2)
```



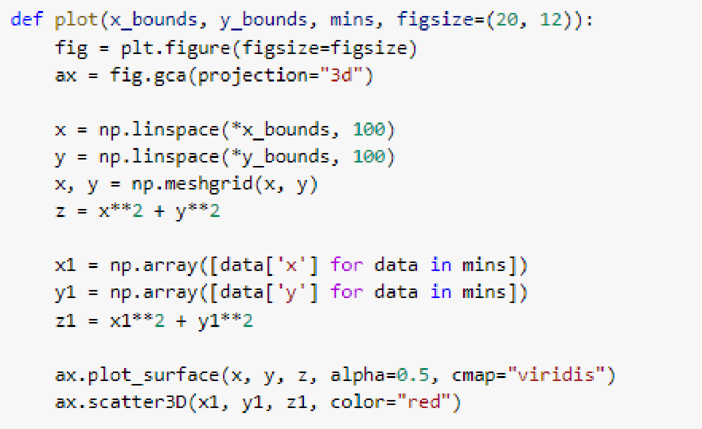
```
def choice_by_roulette(population_sort, add_fitness):
    offset = 0
    add_fitness_norm = add_fitness
    lowest_fitness = apply_function(population_sort[0])
    if lowest_fitness < 0:
        offset = -lowest_fitness
        add_fitness_norm += offset * len(population_sort)

    draw = random.uniform(0, 1)

    accumulated = 0
    for individual in population_sort:
        fitness = apply_function(individual) + offset
        probability = fitness / add_fitness_norm
        accumulated += probability

        if draw <= accumulated:
            return individual
```



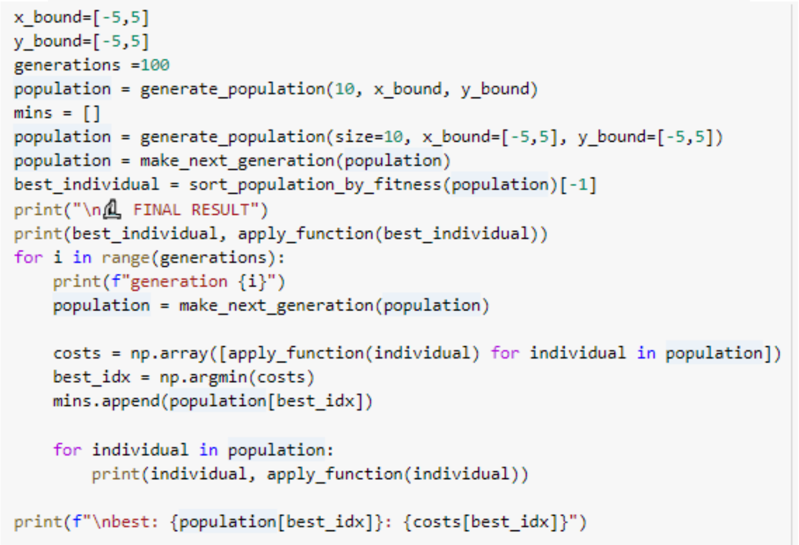
```
def sort_population_by_fitness(population):
    return sorted(population, key=apply_function)


def crossover(individual_a, individual_b):
    xa = individual_a["x"]
    ya = individual_a["y"]
    xb = individual_b["x"]
    yb = individual_b["y"]
    return {"x": (xa + xb) / 2, "y": (ya + yb) / 2}

def mutate(individual):
    next_x = individual["x"] + random.uniform(-0.05, 0.05)
    next_y = individual["y"] + random.uniform(-0.05, 0.05)

    lower_bound, upper_bound = (-5, 5)

    # Guarantee we keep inside boundaries
    next_x = min(max(next_x, lower_bound), upper_bound)
    next_y = min(max(next_y, lower_bound), upper_bound)

    return {"x": next_x, "y": next_y}
```



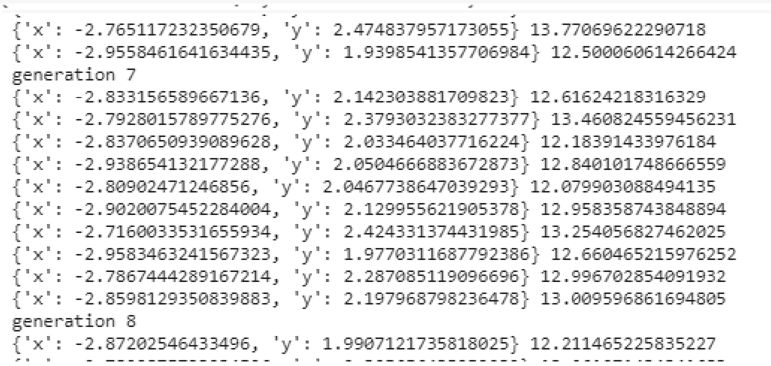
```
def make_next_generation(previous_population):
    next_generation = []
    sorted_by_fitness_population = sort_population_by_fitness(previous_population)
    population_size = len(previous_population)
    add_fitness = sum(apply_function(individual) for individual in population)

    for i in range(population_size):
        first_choice = choice_by_roulette(sorted_by_fitness_population, add_fitness)
        second_choice = choice_by_roulette(sorted_by_fitness_population, add_fitness)

        individual = crossover(first_choice, second_choice)
        individual = mutate(individual)
        next_generation.append(individual)

    return next_generation
```



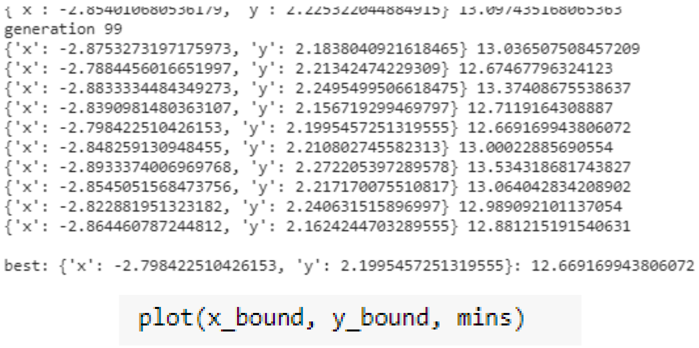
```
def plot(x_bounds, y_bounds, mins, figsize=(20, 12)):
    fig = plt.figure(figsize=figsize)
    ax = fig.gca(projection="3d")

    x = np.linspace(*x_bounds, 100)
    y = np.linspace(*y_bounds, 100)
    x, y = np.meshgrid(x, y)
    z = x**2 + y**2

    x1 = np.array([data['x'] for data in mins])
    y1 = np.array([data['y'] for data in mins])
    z1 = x1**2 + y1**2

    ax.plot_surface(x, y, z, alpha=0.5, cmap="viridis")
    ax.scatter3D(x1, y1, z1, color="red")
```



```
x_bound=[-5,5]
y_bound=[-5,5]
generations =100
population = generate_population(10, x_bound, y_bound)
mins = []
population = generate_population(size=10, x_bound=[-5,5], y_bound=[-5,5])
population = make_next_generation(population)
best_individual = sort_population_by_fitness(population)[-1]
print("\n🏆 FINAL RESULT")
print(best_individual, apply_function(best_individual))
for i in range(generations):
    print(f"generation {i}")
    population = make_next_generation(population)

    costs = np.array([apply_function(individual) for individual in population])
    best_idx = np.argmin(costs)
    mins.append(population[best_idx])

    for individual in population:
        print(individual, apply_function(individual))

print(f"\nbest: {population[best_idx]}: {costs[best_idx]}")
```

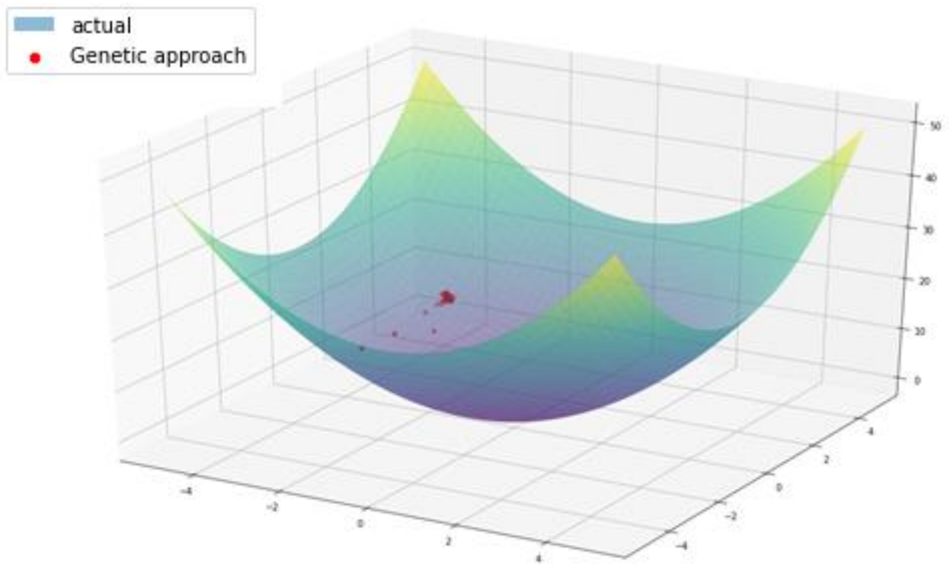


```
{'x': -2.765117232350679, 'y': 2.474837957173055} 13.77069622290718
{'x': -2.9558461641634435, 'y': 1.9398541357706984} 12.500060614266424
generation 7
{'x': -2.833156589667136, 'y': 2.142303881709823} 12.61624218316329
{'x': -2.7928015789775276, 'y': 2.3793032383277377} 13.460824559456231
{'x': -2.8370650939089628, 'y': 2.033464037716224} 12.18391433976184
{'x': -2.938654132177288, 'y': 2.0504666883672873} 12.840101748666559
{'x': -2.80902471246856, 'y': 2.0467738647039293} 12.079903088494135
{'x': -2.9020075452284004, 'y': 2.129955621905378} 12.958358743848894
{'x': -2.7160033531655934, 'y': 2.424331374431985} 13.254056827462025
{'x': -2.9583463241567323, 'y': 1.977031168779238} 12.660465215976252
{'x': -2.7867444289167214, 'y': 2.287085119096696} 12.996702854091932
{'x': -2.8598129350839883, 'y': 2.197968798236478} 13.009596861694805
generation 8
{'x': -2.87202546433496, 'y': 1.9907121735818025} 12.211465225835227
```



```
{'x': -2.854010680536179, 'y': 2.225322044884915} 13.097435168065363
generation 99
{'x': -2.8753273197175973, 'y': 2.1838040921618465} 13.036507508457209
{'x': -2.7884456016651997, 'y': 2.21342474229309} 12.67467796324123
{'x': -2.8833334484349273, 'y': 2.2495499506618475} 13.37408675538637
{'x': -2.8390981480363107, 'y': 2.156719299469797} 12.7119164308887
{'x': -2.798422510426153, 'y': 2.1995457251319555} 12.669169943806072
{'x': -2.848259130948455, 'y': 2.210802745582313} 13.00022885690554
{'x': -2.8933374006969768, 'y': 2.272205397289578} 13.534318681743827
{'x': -2.8545051568473756, 'y': 2.217170075510817} 13.064042834208902
{'x': -2.822881951323182, 'y': 2.240631515896997} 12.989092101137054
{'x': -2.864460787244812, 'y': 2.1624244703289555} 12.881215191540631

best: {'x': -2.798422510426153, 'y': 2.1995457251319555}: 12.669169943806072
```

```
plot(x_bound, y_bound, mins)
```



As seen above, the solutions have been minimized, when compared to the zeroth generation and the ninety-ninth generation. The minima achieved was 12.6691, which is close to the actual global minima. Plotting the objective function landscape and the best individual in each generation.

B.2.3 Comparison with any other heuristic algorithm

Particle Swarm Optimization characterized into the domain of Artificial Intelligence. The term ‘Artificial Intelligence’ or ‘Artificial Life‘ refers to the theory of simulating human behavior through computation. It involves designing such computer systems which are able to execute tasks which require human intelligence.

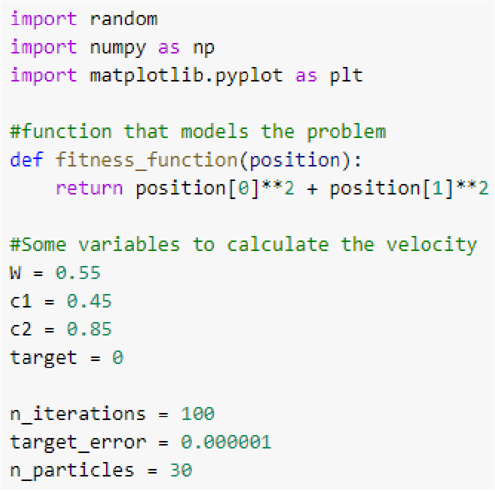
Swarm intelligence is an emerging field of biologically-inspired artificial intelligence based on the behavioral models of social insects such as ants, bees, wasps, termites etc. Swarm Intelligence is the Complex Collective, Self-Organized, Coordinated, Flexible and Robust Behavior of a group following the simple rules. Agents interacts locally with each other and the environment.

Particle Swarm Optimization (PSO), a population based technique for stochastic search in a multidimensional space, has so far been employed successfully for solving a variety of optimization problems including many multifaceted problems, where other popular methods like steepest descent, gradient descent, conjugate gradient, Newton method, etc. do not give satisfactory results. Thus Particle Swarm Optimization Technique is said to be inspired by a swarm of birds or a school of fish.

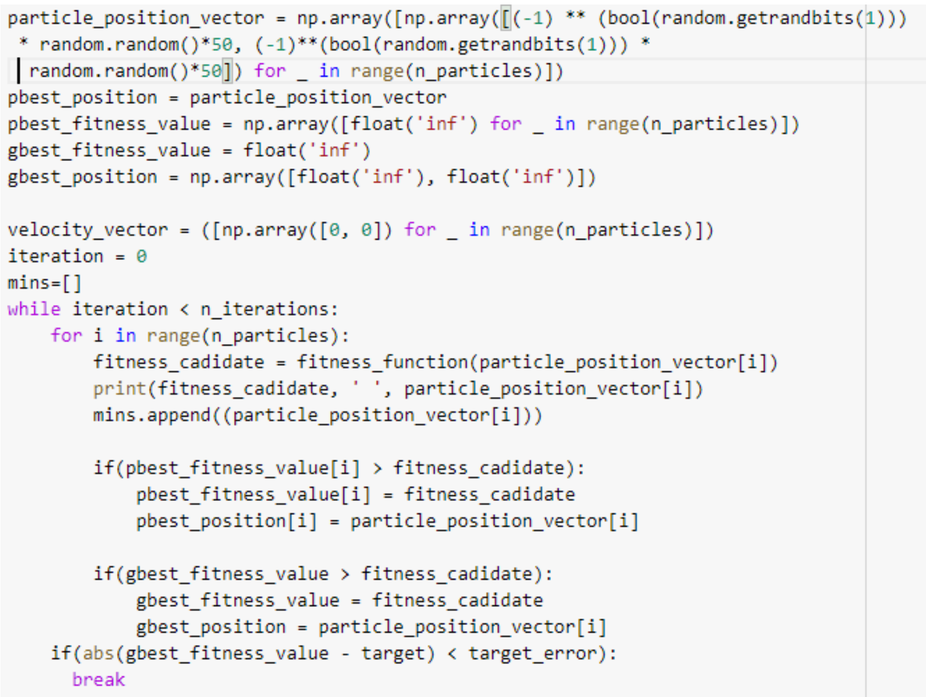
Comparison of Performances of PSO and GA

- The most important distinction between PSO with GA is the sharing of information. In GA, chromosomes share information with each other, whereas in PSO the best particle informs the others and the information of variables is stored in small memory. Again, PSO search for the global best solution is unidirectional, while GA follows the parallel searching process.
- In contrast to GA, PSO does not use any genetic kind of operator, i.e., crossover and mutation, and the internal velocity leads the particle to the next better place.
- PSO implementation is more simple and easier than GA as it deals with few parameters (like position and velocity only).
- GA provides satisfactory results in case of combinatorial problems, PSO being less suitable there.
- PSO takes much less time to execute and the convergence rate is also faster than that of GA.

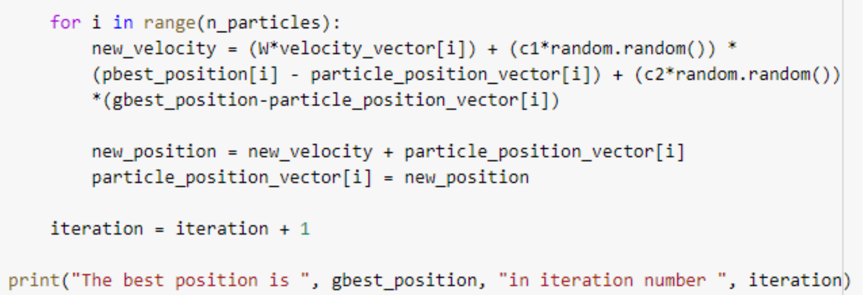
Code:



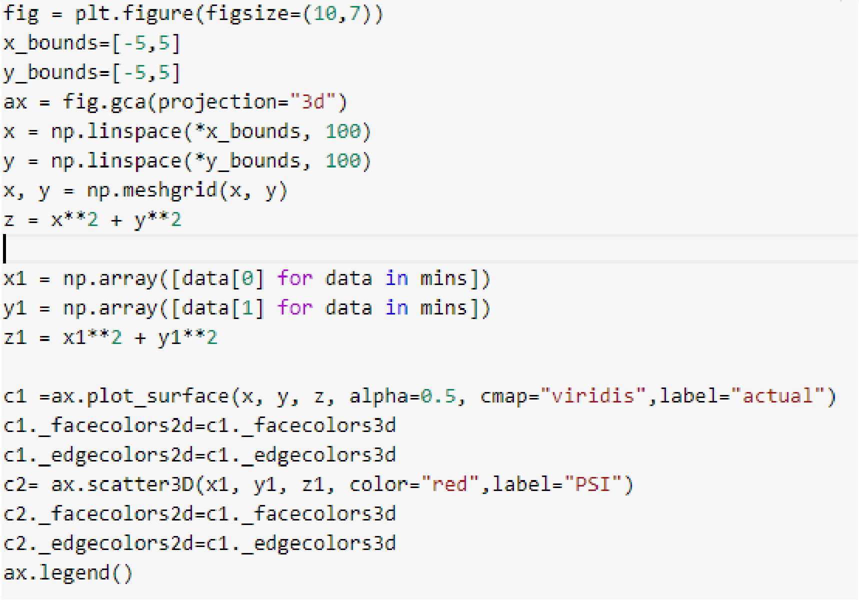
```python
import random
import numpy as np
import matplotlib.pyplot as plt

#function that models the problem
def fitness_function(position):
    return position[0]**2 + position[1]**2

#Some variables to calculate the velocity
W = 0.55
c1 = 0.45
c2 = 0.85
target = 0

n_iterations = 100
target_error = 0.000001
n_particles = 30
```



```
particle_position_vector = np.array([np.array([(-1) ** (bool(random.getrandbits(1)))
 * random.random()*50, (-1)**(bool(random.getrandbits(1))) *
 random.random()*50]) for _ in range(n_particles)])
pbest_position = particle_position_vector
pbest_fitness_value = np.array([float('inf') for _ in range(n_particles)])
gbest_fitness_value = float('inf')
gbest_position = np.array([float('inf'), float('inf')])

velocity_vector = ([np.array([0, 0]) for _ in range(n_particles)])
iteration = 0
mins=[]
while iteration < n_iterations:
    for i in range(n_particles):
        fitness_cadidate = fitness_function(particle_position_vector[i])
        print(fitness_cadidate, ' ', particle_position_vector[i])
        mins.append((particle_position_vector[i]))

        if(pbest_fitness_value[i] > fitness_cadidate):
            pbest_fitness_value[i] = fitness_cadidate
            pbest_position[i] = particle_position_vector[i]

        if(gbest_fitness_value > fitness_cadidate):
            gbest_fitness_value = fitness_cadidate
            gbest_position = particle_position_vector[i]
    if(abs(gbest_fitness_value - target) < target_error):
      break
```

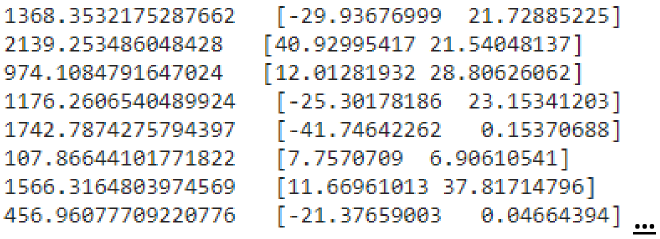


```
    for i in range(n_particles):
        new_velocity = (W*velocity_vector[i]) + (c1*random.random()) *
        (pbest_position[i] - particle_position_vector[i]) + (c2*random.random())
        *(gbest_position-particle_position_vector[i])

        new_position = new_velocity + particle_position_vector[i]
        particle_position_vector[i] = new_position

    iteration = iteration + 1

print("The best position is ", gbest_position, "in iteration number ", iteration)
```

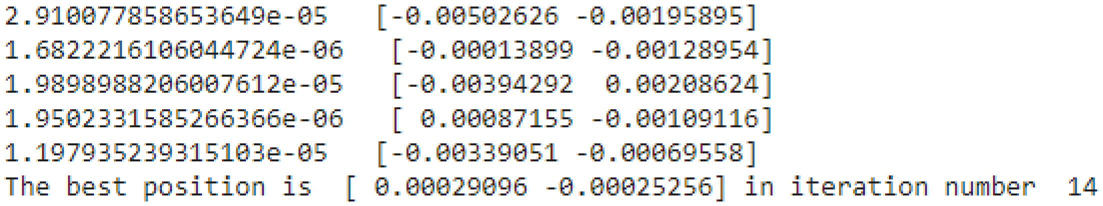


```
fig = plt.figure(figsize=(10,7))
x_bounds=[-5,5]
y_bounds=[-5,5]
ax = fig.gca(projection="3d")
x = np.linspace(*x_bounds, 100)
y = np.linspace(*y_bounds, 100)
x, y = np.meshgrid(x, y)
z = x**2 + y**2

x1 = np.array([data[0] for data in mins])
y1 = np.array([data[1] for data in mins])
z1 = x1**2 + y1**2

c1 =ax.plot_surface(x, y, z, alpha=0.5, cmap="viridis",label="actual")
c1._facecolors2d=c1._facecolors3d
c1._edgecolors2d=c1._edgecolors3d
c2= ax.scatter3D(x1, y1, z1, color="red",label="PSI")
c2._facecolors2d=c1._facecolors3d
c2._edgecolors2d=c1._edgecolors3d
ax.legend()
```

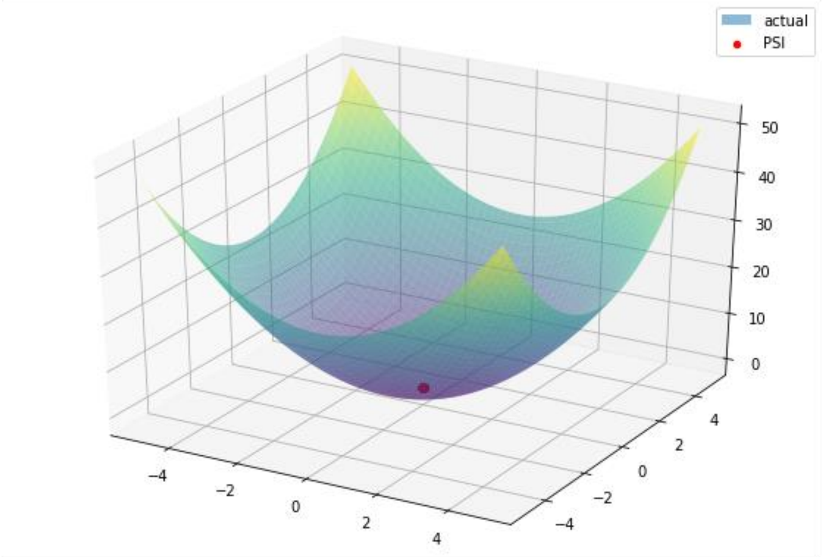
Output:



```
1368.3532175287662   [-29.93676999  21.72885225]
2139.253486048428   [40.92995417 21.54048137]
974.1084791647024   [12.01281932 28.80626062]
1176.2606540489924   [-25.30178186  23.15341203]
1742.7874275794397   [-41.74642262   0.15370688]
107.86644101771822   [7.7570709  6.90610541]
1566.3164803974569   [11.66961013 37.81714796]
456.96077709220776   [-21.37659003   0.04664394] …
```



```
2.910077858653649e-05   [-0.00502626 -0.00195895]
1.6822216106044724e-06   [-0.00013899 -0.00128954]
1.9898988206007612e-05   [-0.00394292  0.00208624]
1.9502331585266366e-06   [ 0.00087155 -0.00109116]
1.197935239315103e-05   [-0.00339051 -0.00069558]
The best position is  [ 0.00029096 -0.00025256] in iteration number  14
```



4. Bibliography

- https://towardsdatascience.com/traditional-ai-vs-modern-ai-5117b469a0c9
- https://dzone.com/articles/4-artificial-intelligence-pitfalls
- https://www.forbes.com/sites/forbestechcouncil/2019/09/26/avoid-these-pitfalls-when-investing-in-ai/?sh=31a9265352d5
- Artificial Intelligence and Marketing: Pitfalls and Opportunities- A survey paper.
- https://cs.brynmawr.edu/Courses/cs372/spring2012/slides/05_OptimizationGAs.pdf
- https://www.seas.upenn.edu/~cis391/Lectures/informed-search-II.pdf
- http://www.cs.cmu.edu/afs/cs.cmu.edu/project/learn-43/lib/photoz/.g/web/glossary/anneal.html
- https://www.edureka.co/blog/a-search-algorithm/
- https://www.educative.io/edpresso/what-is-the-a-star-algorithm
- https://www.sfu.ca/~ssurjano/spheref.html
- http://ijarcet.org/wp-content/uploads/IJARCET-VOL-5-ISSUE-1-90-94.pdf
- http://benchmarkfcns.xyz/benchmarkfcns/spherefcn.html
- https://machinelearningmastery.com/stochastic-hill-climbing-in-python-from-scratch/
- https://hackernoon.com/genetic-algorithms-explained-a-python-implementation-sd4w374i
- https://missinglink.ai/guides/neural-network-concepts/complete-guide-artificial-neural-networks/
- https://www.tutorialspoint.com/artificial_intelligence/artificial_intelligence_neural_networks.htm
- https://www.scientificamerican.com/article/what-is-fuzzy-logic-are-t/
- https://plato.stanford.edu/entries/logic-fuzzy/
- http://www.techferry.com/articles/swarm-intelligence.html#
- https://www.hindawi.com/journals/mse/2011/496732/
- https://cs.adelaide.edu.au/~zbyszek/Papers/p20.pdf
- http://mat.uab.cat/~alseda/MasterOpt/ACO_Intro.pdf
- http://www.scholarpedia.org/article/Ant_colony_optimization

5. Appendix

The implementation of various algorithm was done on google colaboratory. The code for the Hill Climbing, Simulated Annealing, Genetic Algorithm, and Particle Swarm Intelligence can be found in below links:

- https://colab.research.google.com/drive/17ef_U5Gll9nv72nN_Sl1W8QT5I80wIvm?usp=sharing
- https://colab.research.google.com/drive/1ijPJ85d4b4M55sUWTM1zOK9CA3BxjKjk?usp=sharing
- https://colab.research.google.com/drive/1ohhXSbPpWkcLEiSRZv6GeoGGfblWDBsy?usp=sharing
- https://colab.research.google.com/drive/1WnUS7BltDS8wRVNzo8-hEGuAzs5w2Obp?usp=sharing