RAMAIAH UNIVERSITY OF APPLIED SCIENCES

# Assignment

| | |
|---|---|
| **Course Code** | CSC401A |
| **Course Name** | Computational Intelligence |
| **Programme** | B.Tech |
| **Department** | CSE |
| **Faculty** | FET |


| | |
|---|---|
| **Name of the Student** | Satyajit Ghana |
| **Reg. No.** | 17ETCS002159 |
| **Semester/Year** | 07/2020 |
| **Course Leader(s)** | Mr. Sagar U. |

# Declaration Sheet

| | | | |
|---|---|---|---|
| Student Name | Satyajit Ghana | | |
| Reg. No | 17ETCS002159 | | |
| Programme | B.Tech | Semester/Year | 07/2020 |
| 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 | |
| Submission date stamp (by Examination & Assessment Section) | | | |

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

# Contents

## List of Figures

# 1 Question 1

Solution to Question No. 1 Part A

## 1.1 Pitfalls in Traditional AI

Traditional problem-solving methods in AI are concerned with representation of the problem states by symbols, and construction of a set of rules to describe the transitions in problem states. The states of the problem are then matched against the IF part of the IF-THEN rules, and on successful matching the selected rule is fired causing a transition of the existing state to a new state as obtained from the THEN part of the fired rule. To keep the rules firable until the goal is found, the knowledge base, however, calls for more search time and thus is responsible for degradation in efficiency of a reasoning system. One approach to circumvent this problem is to organize the knowledge base with fewer rules but to allow partial matching of the problem states with the IF-part of the rules. The *logic of fuzzy sets* is capable of such partial matching.

Traditional AI is very good in inductive and analogy-based learning but it is inefficient to realize supervised learning. In supervised learning, the trainer provides a number of training input/output instances for the learning system. The learning system has to adapt its internal parameters so as to generate the correct output instance in response to a given input instance. Neural Networks can do this task really well.

Except the heuristic search algorithm, traditional AI is not much competent to handle real world optimization problems. But the need for optimization of system parameters and resources in design, synthesis, diagnosis and scheduling problems, for example, is gradually increasing. Genetic Algorithm, fortunately is a new tool that has a good potential in optimizing the parameters of intelligent systems.

Decision making in real-time system largely depends on the available facts, their level of precision, the knowledge base and its soundness. A sound knowledge base should not generate

false inferences, but the degree of precision of the inferences may degrade when the input facts are less precise. However, when the facts are available from multiple sources, their level of precision can be improved through the process of data fusion. Traditional AI is not concerned with data fusion. Fortunately, there is a vast contemporary literature on data fusion technology. The classical *Bayesian statistics, Dempster-Shafer theory, Pearl's belief networks, Kalman filtering,* and *neural network* based methods are some of the well-known techniques of data fusion. These tools and techniques have successfully been used in many industrial autonomous systems. For example, noisy sensory data received by various transducers of a mobile robot are fused to take decisions about is direction of motion in a constrained obstacle-map. The other application of data fusion technique include object recognition from multi-sensory noisy data, prediction of earthquake, storm or heavy rain from multi-sensory measurements and automated diagnosis of a complex system from noisy measurements.

Traditional AI was incompetent to serve the increasing demand of search, optimization and machine learning in i) information systems with large biological and commercial databases and ii) factory automation for steel, aerospace, power and pharmaceutical industries. The failure of classical AI opened up new avenues for the non-conventional models in various engineering applications. These computational tools gave rise to a new discipline called **computational intelligence**. (A. Konar, 2006)

> *"A system is computationally intelligent when it: deals with only numerical (low level) data, has pattern recognition components, does not use knowledge in the AI sense; and additionally when it (begins to) exhibit i) computational adaptivity, ii) computational fault tolerance, iii) speed approaching human-like turnaround and iv) error rates that approximate human performance." – Prof. James (Jim) Bezdek*

## 1.2 Synergism of CI Tools

To understand and elaborate on the Synergism of CI Tools we'll start by defining CI, In an attempt to define computational intelligence, Robert J. Marks clearly mentions the name of the constituent members of the family. According to him,

> *"... neural networks, genetic algorithms, fuzzy systems, evolutionary programming and artificial life are the building blocks of computational intelligence."*

Out of these 5 members the last 2 emerged very recently, roughly speaking, and evolutionary programing is a new programming paradigm where a number of individual computer programs evolve using the concepts of genetic algorithm.

For the sake of brevity, we'll try to understand neural networks and genetic algorithms as our CI tools.

### 1.2.1 Computational Models of Neural Nets

Neurons are the fundamental building blocks of a biological nervous system. Typical estimates of the number of neurons in the human brain are in the order of 10 to 500 billion. Neurons are arranged into about 1,000 main modules, each with about 500 neural networks. Each network has on the order of 100,000 neurons. The axon of each neuron connects to anywhere from hundreds to thousands of other neurons; the value varies greatly from neuron to neuron and from neuron type to neuron type.

While the processing element in an artificial neural network (ANN) is generally considered to be the very roughly analogous to a biological neuron, the cell body in an ANN is modeled by a linear activation function. The activation function in general, attempts to enhance the signal contribution received through different dendrons.

But how do we know that our human neuron system uses a firing function to start the neural process? There was this Retinotopic Mapping performed on a Cat, basically a cat was strapped

to a place, and the brain was injected with a radioactive fluid, now an image was shown to the cat, and it was killed as soon as the cat saw the image, the cat's brain was laid out flat and X-Ray was taken of it, to a surprise the image had a complete print over the brain, as shown below,

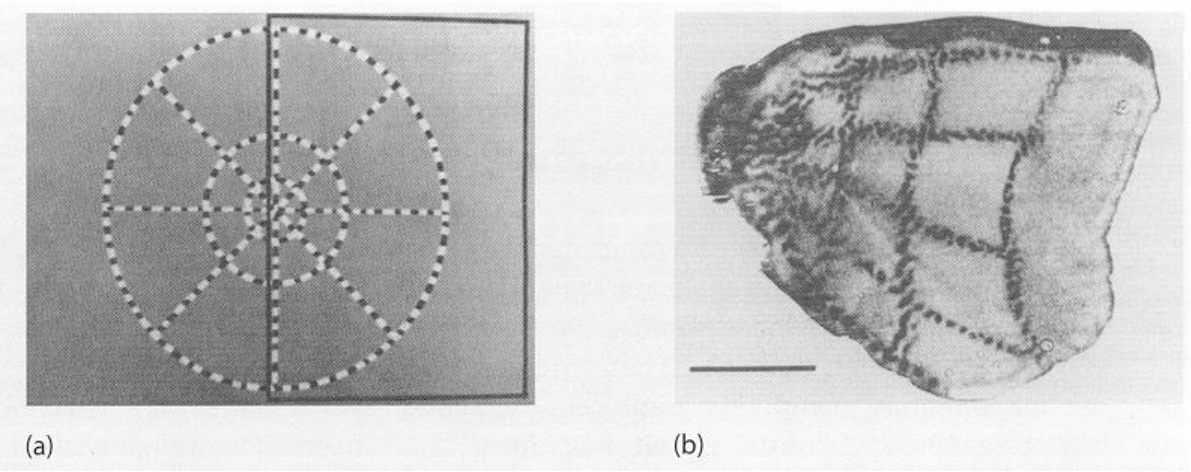



Figure 1-1 Retinotopic Mapping of a Cat's Cortex

This experiment paved the way that Neural Networks would be the future of Artificial Intelligence, even in our modern neural network architectures for computer vision specifically, the entire input image is fed to the network, and several convolutions are performed to make sense of the input and get some meaningful output.

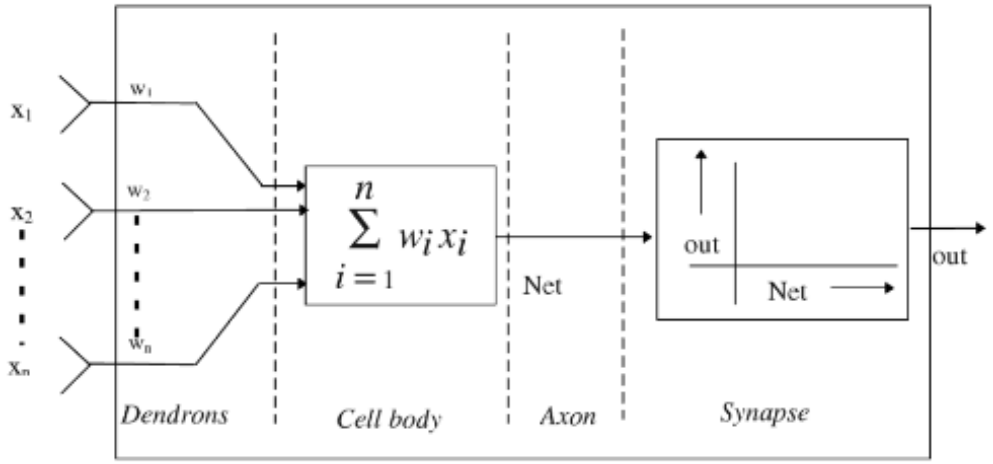



Figure 1-2 An electrical equivalent of the biological neuron

ANNs can learn facts (representation by patterns) and determine the inter-relationship among the patterns. The above diagram is a simple representation of a ANN Neuron, which is very much equivalent to that of a human neuron.

### 1.2.2 Genetic Algorithms

Genetic algorithm (GA) is a stochastic algorithm that models the evolutionary process of biological species through natural selection. Proposed by Holland in early 60's, this algorithm is gaining its importance for its wide acceptance in solving three classical problems, such as learning, searching and optimizing.

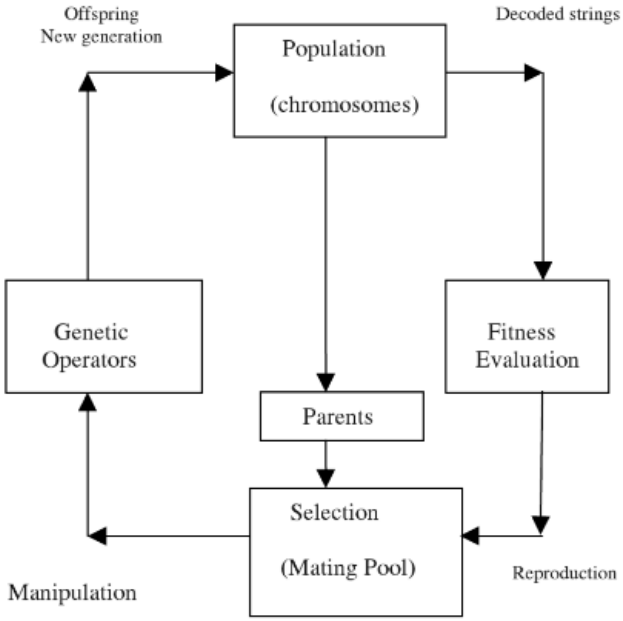



Figure 1-3 The cycle of genetic algorithms

A GA operated through a simple cycle of stages:

i) Creation of a "population" of strings

ii) Evaluation of each string

iii) Selection of best strings and

iv) Genetic manipulation to create new population of strings

Each cycle in GA produces a new generation of possible solutions for a given problem. In the first phase, an initial population, describing representatives of the potential solution, is created to initialize the search process. The elements of the population are encoded into bit strings, called chromosomes. The performance of the strings, often called fitness, is then evaluated. Depending upon the fitness of the chromosomes, they are selected for a genetic manipulation process. In the first step, crossover operation combines two individuals about a crossover point, basically their chromosomes are swapped and two new individuals are created. The crossover point is chosen at random. The next step is mutation, in which random places in the offsprings's chromosome are mutated or flipped.

While doing this assignment, I came across this new kind of algorithm, The Genetic Flock Algorithm, this algorithm is a type of hybrid of a Genetic Algorithm and a Particle Swarm Optimization Algorithm. There is an obvious correspondence between a GA and a PSO that makes a hybrid algorithm possible. They both consist of groups of potential solutions to the problem of interest. Usually chromosomes are bit strings; however, they can be arrays of real numbers. A particle in a PSO, on the other hand, has a location in a high dimensional space, which is usually a collection of real numbers. The Genetic Flock works by executing the Genetic Algorithm phase for a fixed number of iterations and then executing the Particle Swarm Optimization phase for a fixed number of iterations. The number of iterations in each phase may be adjusted, but the default is 50 for each. This process is repeated until the maximum number of iterations has been reached.

The GA phase uses tournament selection with single point crossover. Mutations are additive and are chosen from a Gaussian distribution with standard deviation a fixed fraction (typically .01) of the range over which optimization is to occur. The default mutation rate is 0.10.

In the PSO phase, particles are grouped together into collections called neighborhoods with a default size of 7. Particles move to a new location based on inertia, a randomized spring-like force towards the best location the particle visited, and a similar force toward the best location visited by any other member of its neighborhood. Each time the particle moves from one

location to another, the spring-like force constants involving the neighborhood and personal best locations are modified by random percentages. A particle's maximum velocity is not allowed to exceed one half the distance between the particles' boundaries; that is to say, that a particle is not allow to take a single step of a size that was bigger than one half of the entire space being searched. The algorithm can be set to run for a fixed number of generations or until there is insufficient progress. (Brooks J et. al, 2013)

# 2 Question 2

Solution to Question No. 1 Part B

## 2.1 Key ideas in hill-climbing approach

The hill-climbing search algorithm (steepest-ascent version), is simply a loop that continually moves in the direction of increasing value—that is, uphill. It terminates when it reaches a "peak" where no neighbor has a higher value. The algorithm does not maintain a search tree, so the data structure for the current node need only record the state and the value of the objective function. Hill climbing does not look ahead beyond the immediate neighbors of the current state. This resembles trying to find the top of Mount Everest in a thick fog while suffering from amnesia

Hill climbing is sometimes called *greedy local search* because it grabs a good neighbor state without thinking ahead about where to go next. Although greed is considered one of the seven deadly sins, it turns out that greedy algorithms often perform quite well. Hill climbing often makes rapid progress toward a solution because it is usually quite easy to improve a bad state.

```
function HILL-CLIMBING(problem) returns a state that is a local minimum
current ← MAKE-NODE(problem.INITIAL-STATE)
loop do
    neighbour ← a highest-values successor of current
    if neighbour.VALUE <= current.VALUE then return current.STATE
    current ← neighbour
```

Some key things that Stochastic Hill Climbing does, but explained like a layman

- close your eyes, and just move, if you moved and can feel that you are at a higher step, then stay there
- if you are stuck in a local-minima you are stuck there forever, period.
- you can try to take longer steps to get out of that minima, but that will make you move randomly and you will end up brute-forcing the entire search space

- if the search space has a single global-minima, you'll be able to reach there pretty quickly, imagine like running down a mountain, its easy right?

## 2.2 Key ideas in alternative CI approach

A quick and dirty fix to the pitfalls of hill climbing is Simulated Annealing, A hill-climbing algorithm that never makes "downhill" moves toward states with lower value (or higher cost) is guaranteed to be incomplete, because it can get stuck on a local maximum. In contrast, a purely random walk—that is, moving to a successor chosen uniformly at random from the set of successors—is complete but extremely inefficient. Therefore, it seems reasonable to try to combine hill climbing with a random walk in some way that yields both efficiency and completeness. Simulated annealing is such an algorithm.

In metallurgy, annealing is the process used to temper or harden metals and glass by heating them to a high temperature and then gradually cooling them, thus allowing the material to reach a low-energy crystalline state. To explain simulated annealing, we switch our point of view from hill climbing to gradient descent (i.e., minimizing cost) and imagine the task of getting a ping-pong ball into the deepest crevice in a bumpy surface. If we just let the ball roll, it will come to rest at a local minimum. If we shake the surface, we can bounce the ball out of the local minimum. The trick is to shake just hard enough to bounce the ball out of local minima but not hard enough to dislodge it from the global minimum. The simulated-annealing solution is to start by shaking hard (i.e., at a high temperature) and then gradually reduce the intensity of the shaking (i.e., lower the temperature).

```
function SIMULATED-ANNEALING(problem, schedule) returns a solution state
inputs: problem, a problem
     schedule, a mapping from time to "temperature"
current ← MAKE-NODE(problem.INITIAL-STATE)
for t = 1 to ∞ do
    T ← schedule(t)
    if T = 0 then return current
    next ← a randomly selected successor of current
```

```
ΔE ← next.VALUE - current.VALUE
if ΔE > 0 then current ← next
else current ← next only with probability e^(ΔE/T)
```

## 2.3 Python program demonstration

In order to test the algorithm's optimization capability, they were subjected to two types of functions, one having a single global minima, and one with many local minima but with a single global minima.

Why? Because real world problems don't necessarily are smooth and have a single minima, instead its highly nonlinear with many local minima.

What about maxima finding? Well, all maxima finding optimizations can be converted to minima finding by simply negating the value.

Benchmark Function 1:

$$f\ X,Y\ = X^2 + Y^2$$

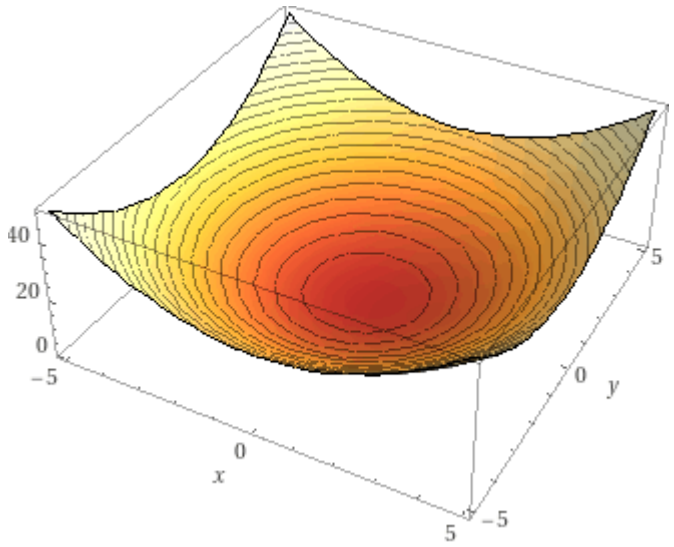



Figure 2-1 environment function x^2+y^2

Benchmark Function 2:

$$f\ X,Y\ = 5 \times \sin(X^2 + Y^2) + X^2 + Y^2$$

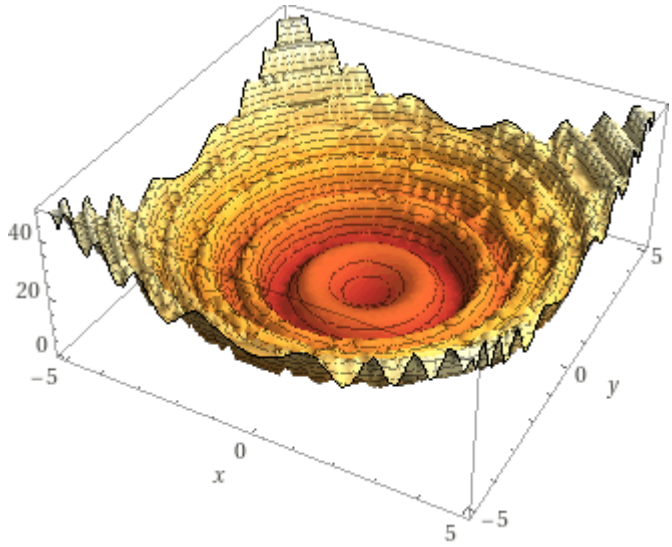



Figure 2-2 environment function 5*sin(x^2+y^2)+x^2+y^2

Source code for Hill Climbing Algorithm:

NOTE: francium is a self-made library for testing ai algorithms given an environment and an agent, the details to which can be found in Appendix.

Refer to Appendex for Jupyter Notebooks

**agent.py**

```
from francium.algorithms.hill_climbing.environment import Environment
from francium.core import BaseAgent, State, BaseEnvironment

import numpy as np

class Agent(BaseAgent):
    """Hill Climbing AI Agent"""

    def __init__(self, step_size: float = 1e-2):
        self.step_size = step_size

    def act(self, state: State, env: Environment):
        # get the current state, act upon it and give a new state
        new_state = State({"x": state["x"], "y": state["y"]})
```

```
        new_state["x"] += np.random.randn() * self.step_size
        new_state["y"] += np.random.randn() * self.step_size

        # check if you are in bounds of the environment
        new_state["x"] = min(max(new_state["x"], env.x_bounds[0]), env.x_bounds[1])
        new_state["y"] = min(max(new_state["y"], env.y_bounds[0]), env.y_bounds[1])

        return new_state
```

**environment.py**

```
from typing import Tuple, Optional

from francium.core import BaseEnvironment, State

from francium.core.eval_functions import *

class Environment(BaseEnvironment):
    def __init__(
        self,
        x_bounds: Tuple[float, float],
        y_bounds: Tuple[float, float],
        goal_val: Optional[State] = None,
        tolerance: Optional[float] = 1e-4,
        eval_func=convex_x_square,
    ):

        BaseEnvironment.__init__(
            self, x_bounds, y_bounds, goal_val, tolerance, eval_func=eval_func
        )

    def evaluate_state(self, state: State) -> Tuple[float, bool]:
        eval_val = self.evaluation_func(state["x"], state["y"])

        if self.goal_val:
            is_done = (
                True
                if np.abs(eval_val - self.goal_val["z"]) < self.tolerance
                else False
            )
        else:
            is_done = False

        return eval_val, is_done
```

**solver.py**

```
from typing import Optional

from francium.algorithms.hill_climbing.agent import Agent
from francium.algorithms.hill_climbing.environment import Environment
```

```python
from francium.core import BaseSolver, State, setup_logger

logger = setup_logger(__name__)

class Solver(BaseSolver):
    def __init__(self, agent: Agent, environment: Environment):
        BaseSolver.__init__(self, agent, environment)
        self.initialized: bool = False

    def init_solver(self, init_state: Optional[State] = None):
        if init_state:
            self.memory.add_episode(init_state)
        else:
            init_state = self.env.get_random_init_position()

        logger.info(f"=> Initialized Solver with State: {init_state}")

        self.memory.add_episode(init_state)

        self.initialized = True

    def train_step(self) -> bool:

        if not self.initialized:
            logger.error("=> Solver not initialized !")
            raise Exception("Initialize the solver `solver.init_solver()`")

        curr_state = self.memory.get_curr_state()

        new_state = self.agent(curr_state, self.env)

        eval_val, is_done = self.env.evaluate_state(new_state)

        if is_done:
            logger.info(
                f"=> training is done ! best state: {self.memory.get_curr_state()}"
            )
            return False

        if eval_val < curr_state["z"]:
            # logger.info(f"z: f(x = {new_state['x']}, y = {new_state['y']}) = {eval_val}")
            new_state["z"] = eval_val
            self.memory.add_episode(new_state)

        return True
```

Running the Hill Climbing Algorithm

Hill Climbing: Environment I

```
import francium.algorithms.hill_climbing as hc
import francium.core.eval_functions as eval_functions
from francium.core import State

agent = hc.Agent(step_size=1e-1)
env = hc.Environment(x_bounds=(-5.0, 5.0), y_bounds=(-
5.0, 5.0), eval_func=eval_functions.convex_x_square)
solver = hc.Solver(agent=agent, environment=env)

solver.init_solver(
    init_state=State({
        'x': 4.0,
        'y': 2.0,
        'z': env.evaluation_func(4.0, 2.0)
    })
)
for episode in range(1000):
    trainable = solver.train_step()
    if not trainable:
        break

solver.memory.best_episode
```

```
>>> {'x': -0.0003257200454512485, 'y': 0.006713925485853664, 'z': 4.518288897760412e-05}
```

```
solver.plot_history()

>>>
```

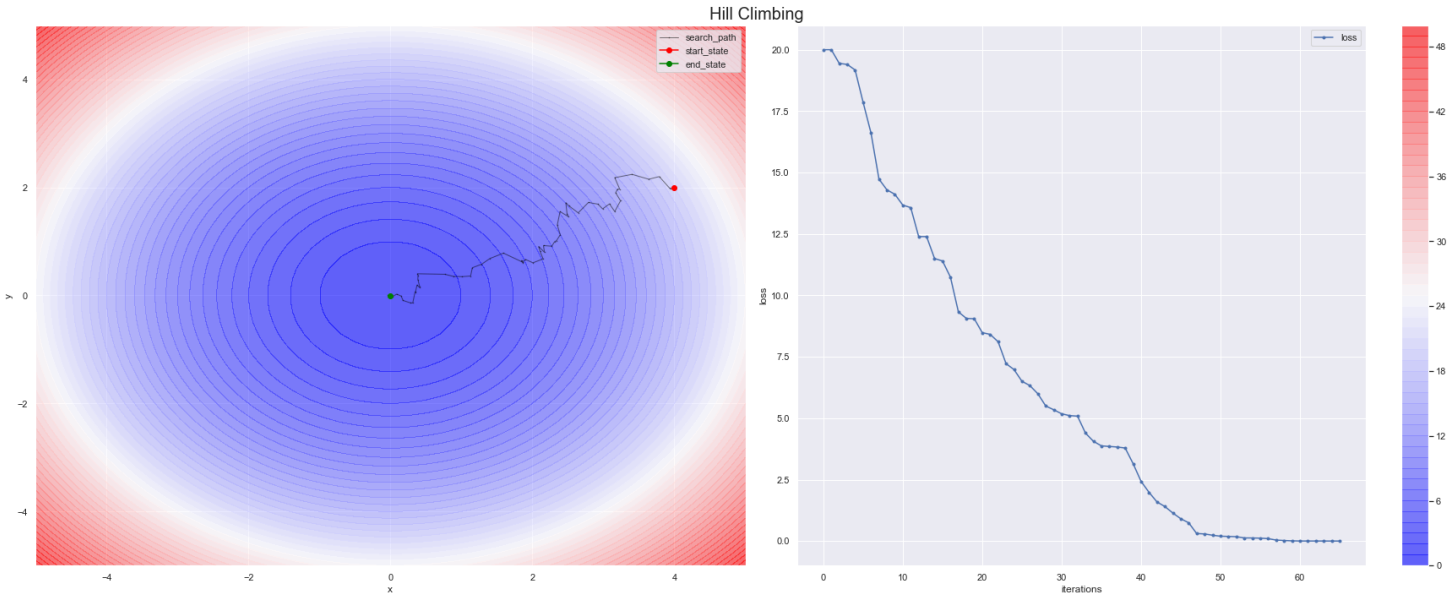



Figure 2-3 hill-climbing solver on environment-I

```
env.plot_environment()

>>>
```

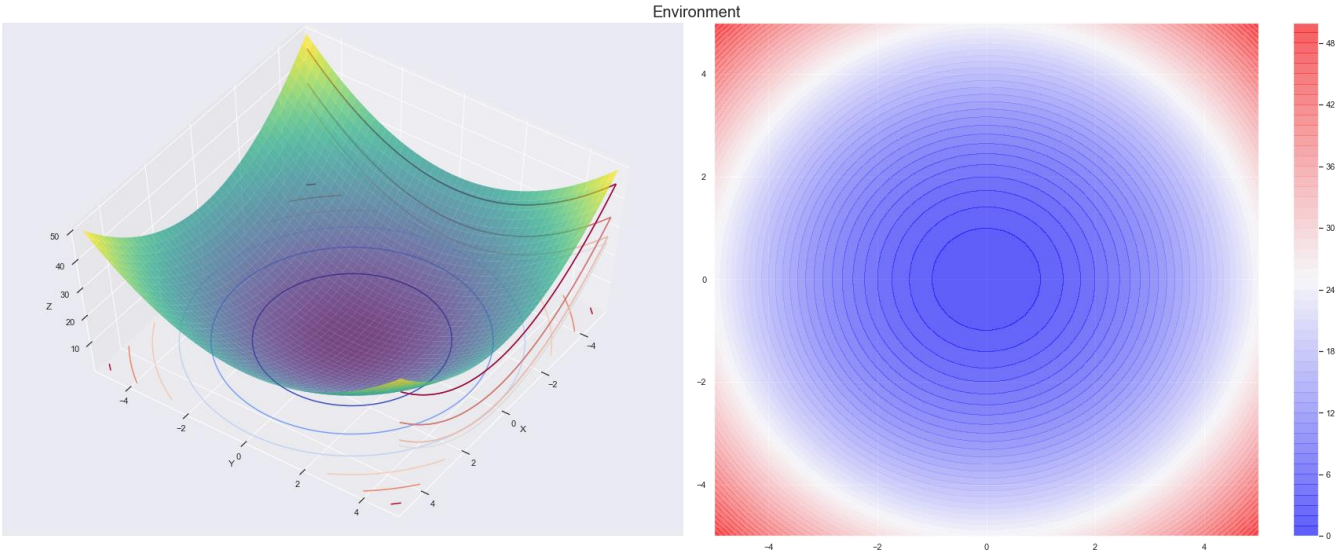



Figure 2-4 hill-climbing environment-I

Hill Climbing: Environment II

```
agent = hc.Agent(step_size=1e-1)
env = hc.Environment(x_bounds=(-5.0, 5.0), y_bounds=(-
5.0, 5.0), eval_func=eval_functions.sinx_plus_x)
solver = hc.Solver(agent=agent, environment=env)

solver.init_solver(
    init_state=State({
        'x': 4.0,
        'y': 2.0,
        'z': env.evaluation_func(4.0, 2.0)
    })
)

for episode in range(1000):
    trainable = solver.train_step()
    if not trainable:
        break

solver.memory.best_episode

>>> {'x': 3.4188550057829175, 'y': 2.321218803102678, 'z': 12.178423661021002}


solver.plot_history()
>>>
```

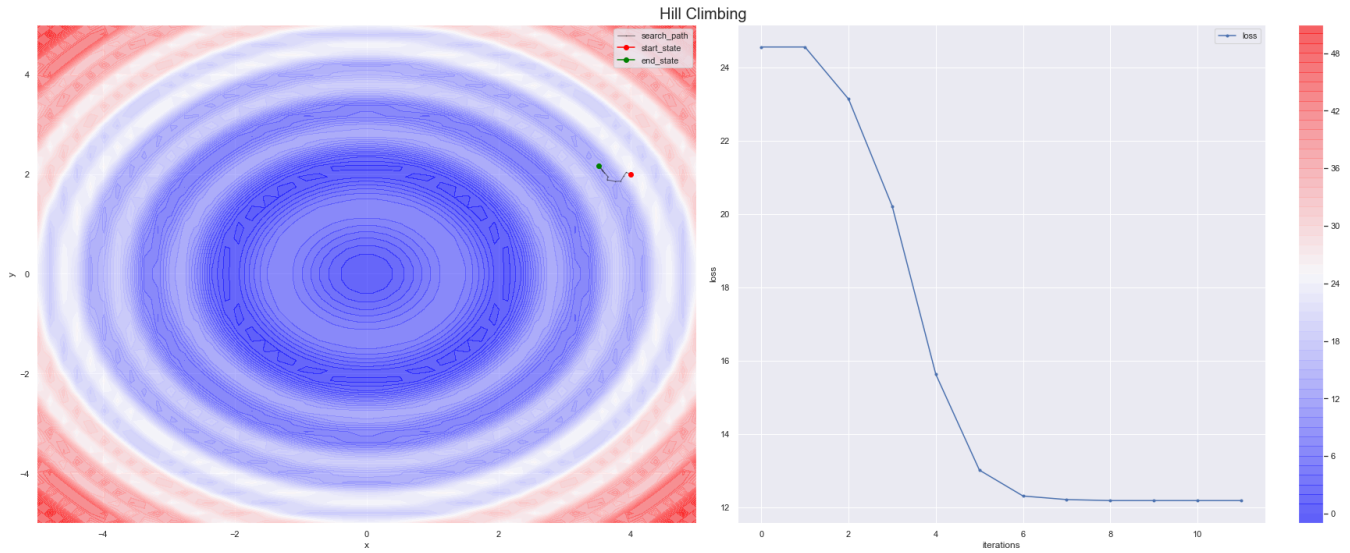



Figure 2-5 hill-climbing solver in environment-II

```
env.plot_environment()
```

```
>>>
```

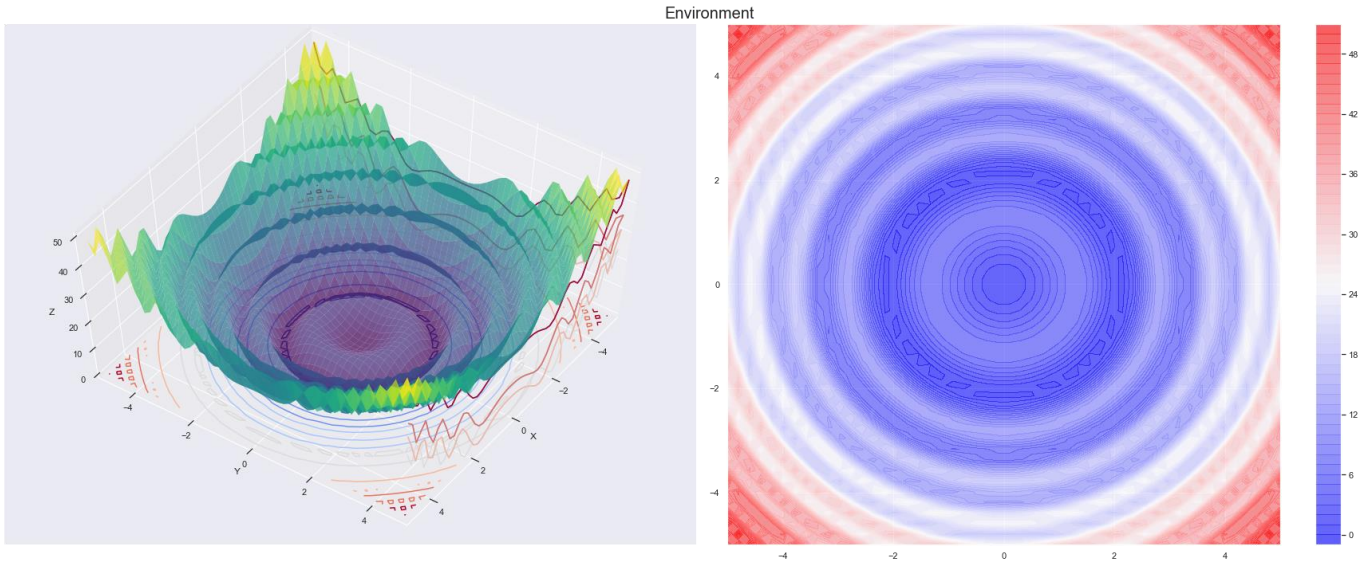



Figure 2-6 hill-climbing environment-II

### 2.3.1 Alternate CI Approach (Simulated Annealing)

**solver.py**

```python
from typing import Optional

from francium.algorithms.simulated_annealing.agent import Agent
from francium.algorithms.simulated_annealing.environment import Environment
from francium.core import BaseSolver, setup_logger, State

import numpy as np

logger = setup_logger(__name__)

class Solver(BaseSolver):
    def __init__(
        self,
        agent: Agent,
        environment: Environment,
        initial_temp: float,
        final_temp: float,
        iters_per_temp: int = 100,
        temp_reduction: Optional[str] = None,
        alpha: float = 10.0,
        beta: float = 5.0,
    ):
        BaseSolver.__init__(self, agent, environment, "Simulated Annealing")
        self.initialized: bool = False
        self.curr_temp = initial_temp
        self.final_temp = final_temp
        self.iters_per_temp = iters_per_temp
        self.alpha = alpha
        self.beta = beta

        if temp_reduction == "linear":
            self.temp_decrement = self.linear_temp_reduction
        elif temp_reduction == "geometric":
            self.temp_decrement = self.geometric_temp_reduction
        elif temp_reduction == "slow_decrease":
            self.temp_decrement = self.slow_decrease_temp_reduction
        else:
            logger.info("=> Using linear_temp_reduction")
            self.temp_decrement = self.linear_temp_reduction

    def linear_temp_reduction(self):
        self.curr_temp -= self.alpha

    def geometric_temp_reduction(self):
        self.curr_temp *= 1 / self.alpha

    def slow_decrease_temp_reduction(self):
        self.curr_temp = self.curr_temp / (1 + self.beta * self.curr_temp)
```

```
    def init_solver(self, init_state: Optional[State] = None):
        if init_state:
            self.memory.add_episode(init_state)
        else:
            init_state = self.env.get_random_init_position()

        logger.info(f"=> Initialized Solver with State: {init_state}")

        self.memory.add_episode(init_state)
        self.initialized = True

    def train_step(self) -> bool:

        if not self.initialized:
            logger.error("=> Solver not initialized !")
            raise Exception("Initialize the solver `solver.init_solver()`")

        for iter in range(self.iters_per_temp):

            if self.curr_temp <= self.final_temp:
                logger.warning(
                    f"=> curr_temp {self.curr_temp} <= final_temp {self.final_temp} ! cannot ann
eal further"
                )
                return False

            curr_state: State = self.memory.get_curr_state()
            new_state: State = self.agent(curr_state, self.env)

            eval_val, is_done = self.env.evaluate_state(new_state)

            if is_done:
                logger.info(
                    f"=> training is done ! best state: {self.memory.get_curr_state()}"
                )
                return False

            cost: float = curr_state["z"] - eval_val

            # check if we can update state based on annealing temp.
            can_anneal: bool = np.random.uniform(0, 1) < np.exp(cost / self.curr_temp)

            if cost >= 0 or can_anneal:
                new_state["z"] = eval_val
                self.memory.add_episode(new_state)

        # reduce the temperature
        self.temp_decrement()

        return True
```

Running the Simulated Annealing Algorithm

Simulated Annealing: Environment I

```
import francium.algorithms.simulated_annealing as sa
import francium.core.eval_functions as eval_functions
from francium.core import State

agent = sa.Agent(step_size=1e-1)
env = sa.Environment(x_bounds=(-5.0, 5.0), y_bounds=(-
5.0, 5.0), eval_func=eval_functions.convex_x_square)
solver = sa.Solver(agent=agent, environment=env, initial_temp=100.0, final_temp=0
.0, iters_per_temp=100, temp_reduction="linear")

solver.init_solver(
    init_state=State({
        'x': 4.0,
        'y': 2.0,
        'z': env.evaluation_func(4.0, 2.0)
    })
)

for episode in range(10):
    trainable = solver.train_step()
    if not trainable:
        break

solver.memory.best_episode
>>> {'x': 0.048191555495727595, 'y': 0.5528596995811312, 'z': 0.3079762734420365}

solver.plot_history()
>>>
```

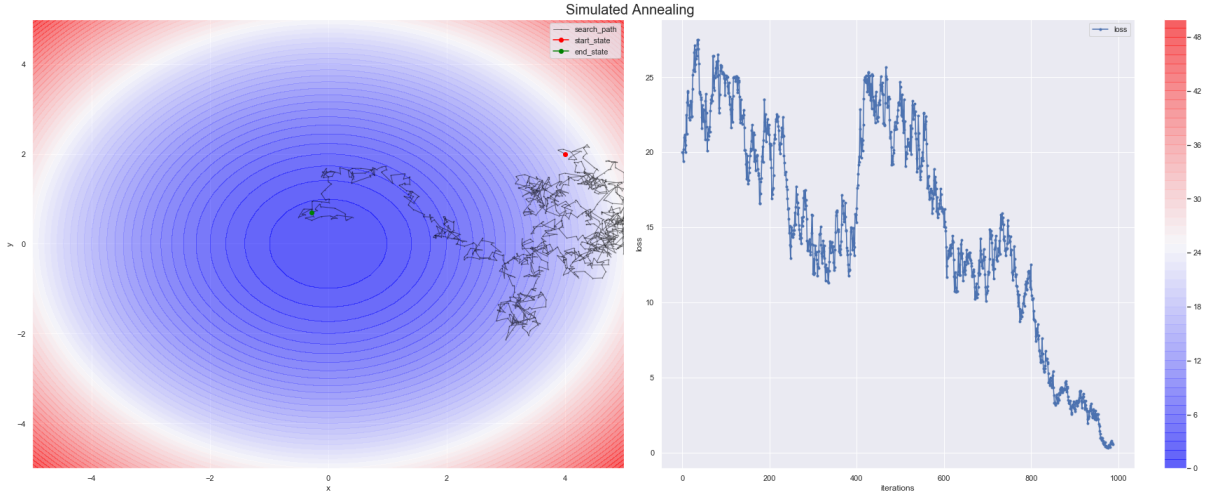



Figure 2-7 simulated-annealing solver in environment-I

```
env.plot_environment()
>>>
```

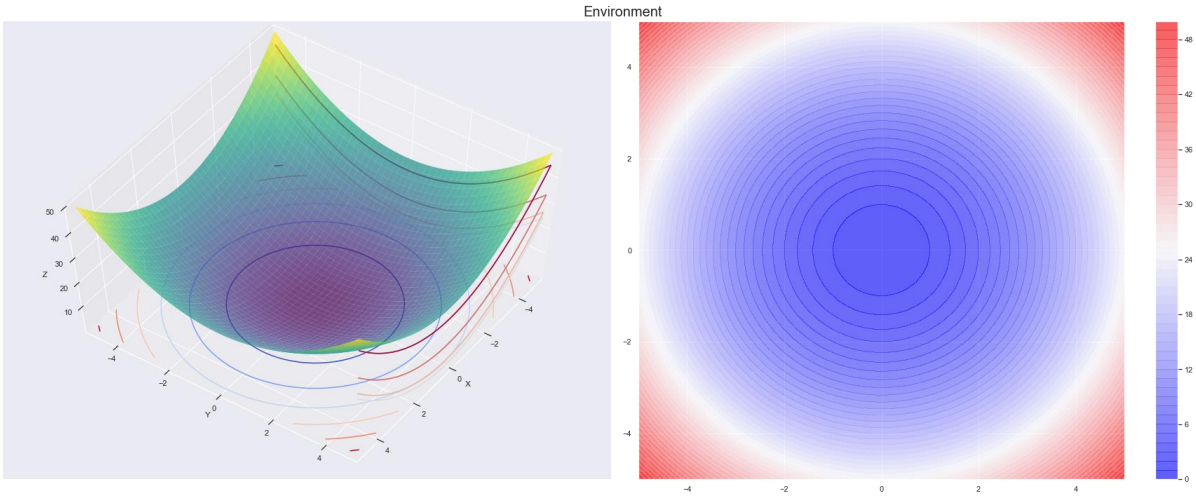



Figure 2-8 simulated-annealing environment-I

Simulated Annealing: Environment II

```
agent = sa.Agent(step_size=1e-1)
env = sa.Environment(x_bounds=(-5.0, 5.0), y_bounds=(-
5.0, 5.0), eval_func=eval_functions.sinx_plus_x)
solver = sa.Solver(agent=agent, environment=env, initial_temp=100.0, final_temp=0
.0, iters_per_temp=100, temp_reduction="linear")

solver.init_solver(
    init_state=State({
        'x': 4.0,
        'y': 2.0,
        'z': env.evaluation_func(4.0, 2.0)
    })
)


for episode in range(10):
    trainable = solver.train_step()
    if not trainable:
        break


solver.memory.best_episode

>>>
{'x': 1.8416088700771585, 'y': -1.0496854976975198, 'z': -0.38718472068411014}
```

```
solver.plot_history()
>>>
```

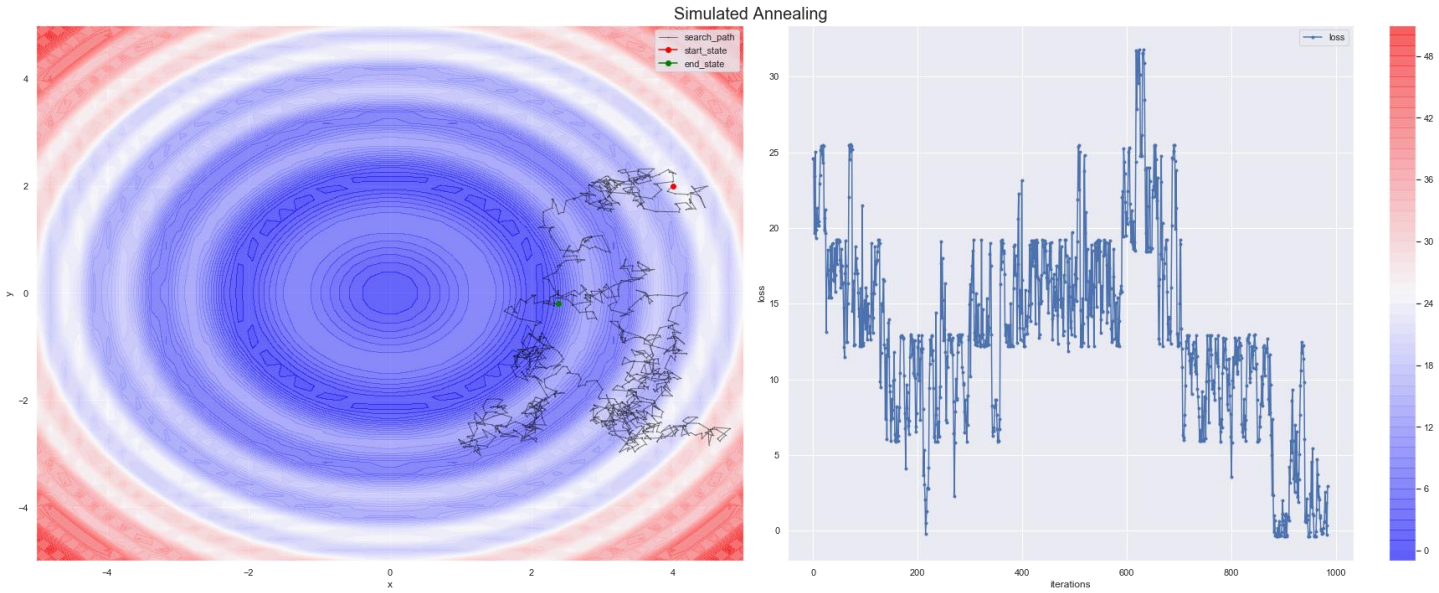



Figure 2-9 simulated-annealing solver in environment-II

```
env.plot_environment()
>>>
```

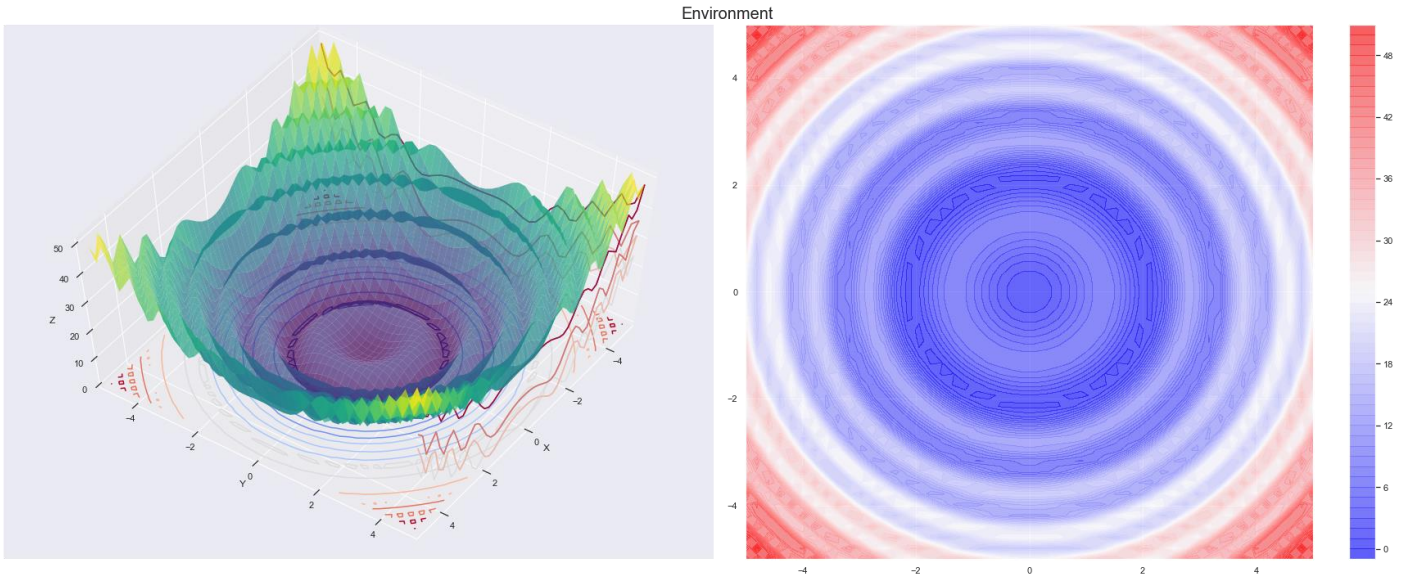



Figure 2-10 simulated-annealing environment-II

## 2.4 Conclusion:

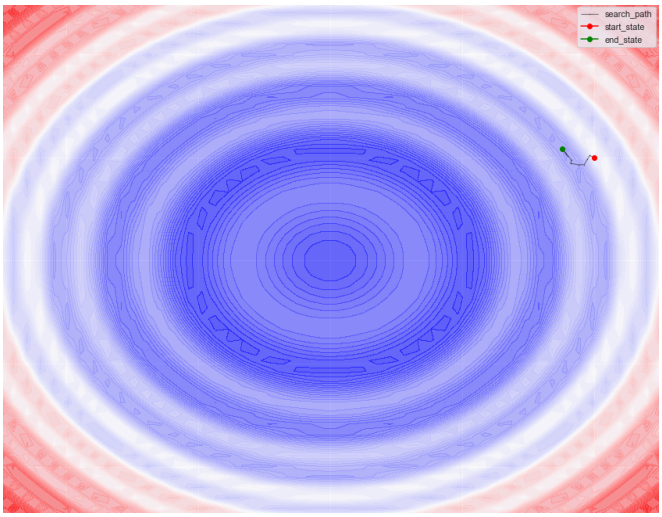
Results in Environment-I

| | X | Y | Z | ~computations |
|---|---|---|---|---|
| *Hill Climbing* | -0.00399 | -0.004717 | 0.00003825076 | 1000 |
| *Simulated Annealing* | 0.04819 | 0.5528 | 0.3079 | 1000 |

Results in Environment-II

| | X | Y | Z | ~computations |
|---|---|---|---|---|
| *Hill Climbing* | 3.5189 | 2.1670 | 12.1784 | 1000 |
| *Simulated Annealing* | 1.8416 | -1.0496 | -0.3871 | 1000 |

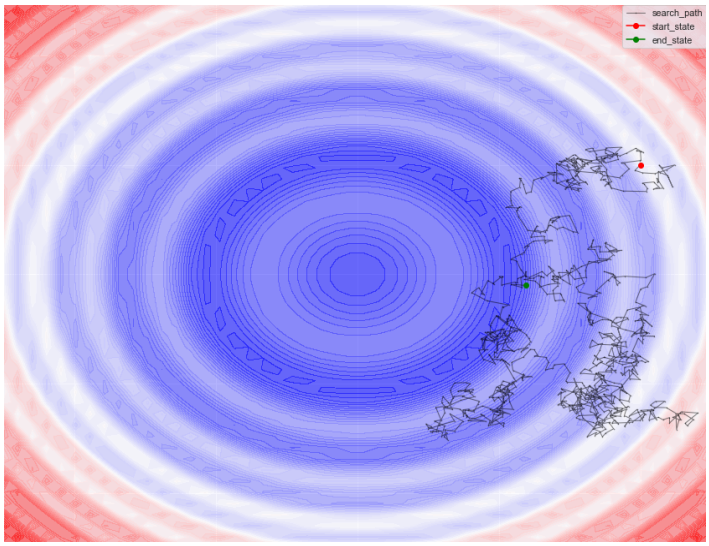
- Hill climbing is really good when it comes to an optimization problem that has a single optimum value, within 1000 iterations it was able to converge to the global minima in environment-I
- But Hill Climbing performs really worse when it comes to an environment with local




of local optimum points.

- Simulated annealing is a big improvement to the hill climbing, in the same 1000 iterations it was able to explore much more area than Hill Climbing and the great

thing is that it doesn't get stuck in local optimal points, because of the high temperature values, the steps are taken all around the search space, gradually as the temperature decreases based on the decrement function, it settles down to an optimal value, which is better compared to hill climbing in environment-II



# 3 Question 3

Solution to Question No. 2 Part B

## 3.1 Discussion on genetic algorithm and benchmark functions

A genetic algorithm (or GA) is a variant of stochastic beam search in which successor states are generated by combining two parent states rather than by modifying a single state. The analogy to natural selection is the same as in stochastic beam search, except that now we are dealing with sexual rather than asexual reproduction.

Like beam searches, GAs begin with a set of k randomly generated states, called the population. Each state, or individual, is represented as a string over a finite alphabet—most commonly, a string of 0s and 1s. Each state is rated by the objective function, or (in GA terminology) the fitness function. A fitness function should return higher values for better states. Two pairs are selected at random for reproduction, in accordance with the probabilities. For each pair to be mated, a crossover point is chosen randomly from the positions in the string. The crossover points are after the third digit in the first pair and after the fifth digit in the second pair. Finally, each location is subject to random mutation with a small independent probability.

Like stochastic beam search, genetic algorithms combine an uphill tendency with random exploration and exchange of information among parallel search threads. The primary advantage, if any, of genetic algorithms comes from the crossover operation. Yet it can be shown mathematically that, if the positions of the genetic code are permuted initially in a random order, crossover conveys no advantage. Intuitively, the advantage comes from the ability of crossover to combine large blocks of letters that have evolved independently to perform useful functions, thus raising the level of granularity at which the search operates.

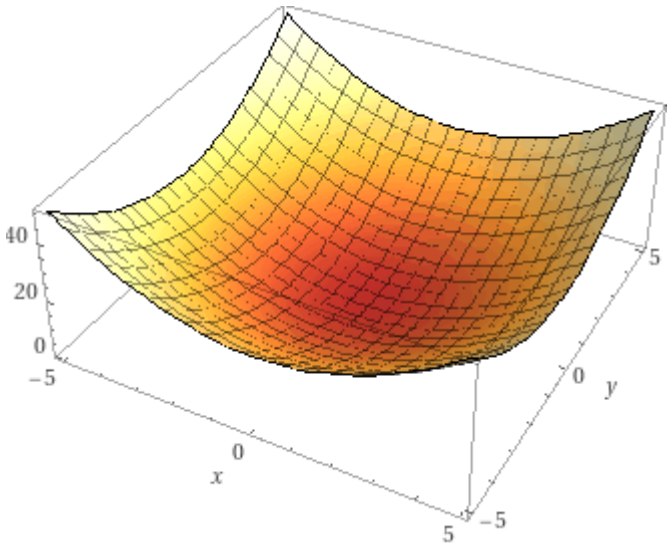
### 3.1.1 Benchmark functions

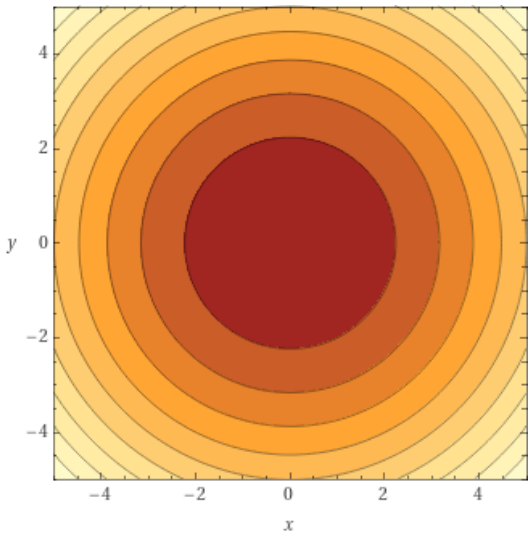
To benchmark GA, or any search algorithm, usually an N-Dimensional space is taken in which the algorithm is made to search for an optimal point.

1. One simple example is the sphere function, which is,

$$f\ X_1, X_2, X_3 \ldots X_n\ = \sum_{i=1}^{n} X_i^2$$

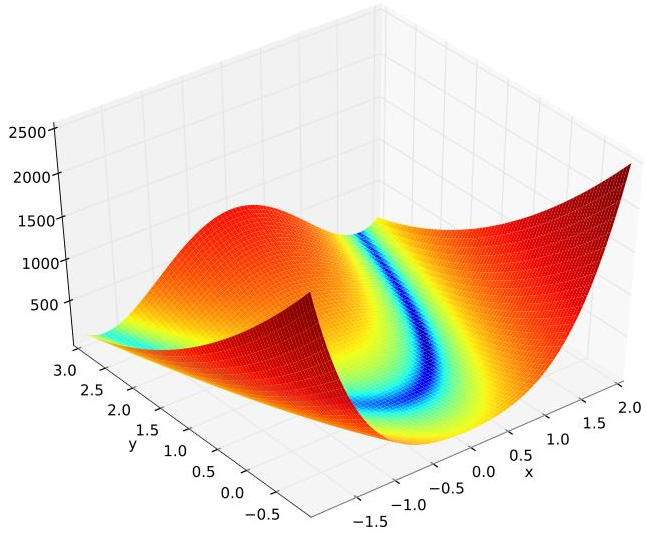
In a 3D space, this would look like, it has a single optimal value at $X_1 = X_2 \ldots = 0$



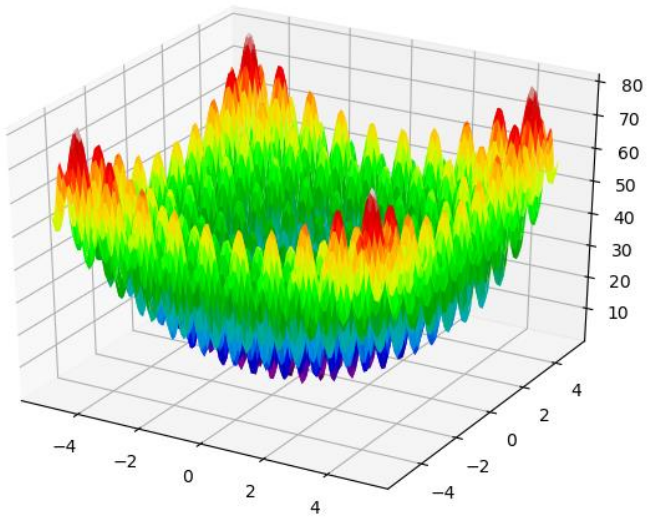

2. Another good benchmark function, is the Rosenbrock function,

$$f\ x, y\ =\ 1 - x^2\ + 100\ y - x^2\ ^2$$



3. Rastrigin function is another, which is really complicated to look at, it has a local minimas all around, and with a single global minima, this is a great example of a benchmark function, mathematically it can be represented as,

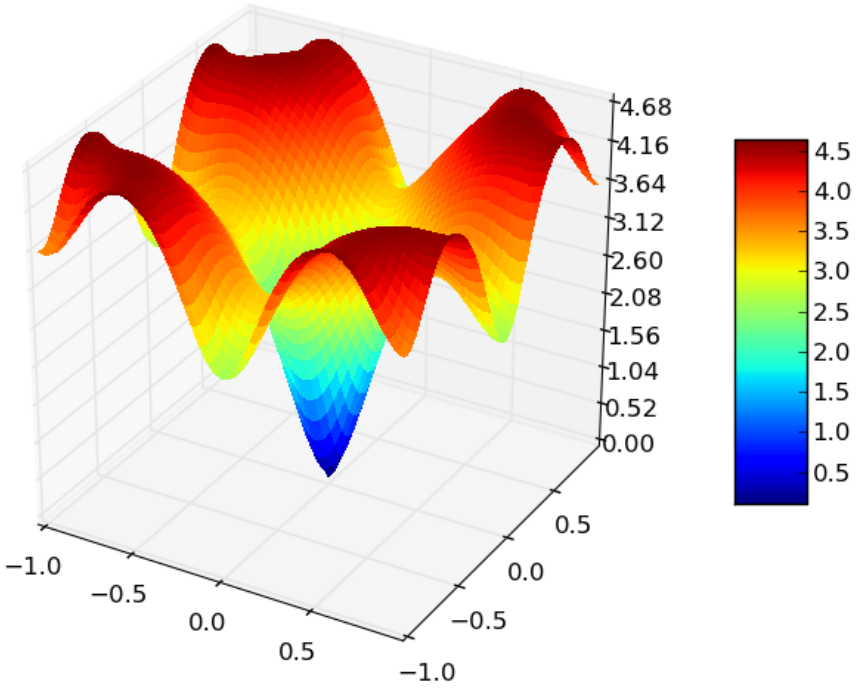
$$f\ x, y\ = x^2 - 10 \times \cos\ 2\pi x\ + y^2 - 10 \times \cos(2\pi y) + 20$$




Or in Python,

```
Z = (X**2 -10*np.cos(2*np.pi*X))
  + (Y**2 -10*np.cos(2*np.pi*Y)) + 20
```

4. Ackley Function is a hard single objective optimization problem. It has a large funnel shape, which on a large scale looks quite simple. However, the surface of the funnel has a very rough texture with many local minima.




In summary we have, Sphere, Rosenbrock, Step, Quartic with noise, Foxhole, Rastrigin, Schwefel, Griewangk, Ackley, Easom, Schwefel's Double Sum, Royal Road, Goldberg and Whitley benchmark functions.

## 3.2 Python program showing the minimized values

**agent.py**

```python
from typing import Tuple

from francium.algorithms.genetic_algorithm.environment import Environment
from francium.core import State, setup_logger, BaseAgent

import numpy as np

logger = setup_logger(__name__)

class Agent(BaseAgent):
    """Genetic Algorithm Agent"""

    def __init__(self):
        self.population = []
        self.initialized = False

    def init_agent(
        self,
        pop_size: int,
        x_bounds: Tuple[float, float],
        y_bounds: Tuple[float, float],
    ):
        x_min, x_max = x_bounds
        y_min, y_max = y_bounds

        population = []
        for _ in range(pop_size):
            individual = State(
                {
                    "x": np.random.uniform(x_min, x_max),
                    "y": np.random.uniform(y_min, y_max),
                }
            )

            population.append(individual)

        self.population = population

        self.initialized = True

    def sort_population_by_fitness(self, fitness_func):
        return sorted(self.population, key=fitness_func)

    @staticmethod
    def choice_by_roulette(sorted_population, fitness_sum, fitness_func):
        offset = 0
        normalized_fitness_sum = fitness_sum
```

```
        lowest_fitness = fitness_func(sorted_population[0])
        if lowest_fitness < 0:
            offset = -lowest_fitness
            normalized_fitness_sum += offset * len(sorted_population)

        draw = np.random.uniform(0, 1)

        accumulated = 0
        for individual in sorted_population:
            fitness = fitness_func(individual) + offset
            probability = fitness / normalized_fitness_sum
            accumulated += probability

            if draw <= accumulated:
                return individual

    @staticmethod
    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}

    @staticmethod
    def mutate(individual, x_bounds, y_bounds):
        x_min, x_max = x_bounds
        y_min, y_max = y_bounds

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

        # Guarantee we keep inside boundaries
        next_x = min(max(next_x, x_min), x_max)
        next_y = min(max(next_y, y_min), y_max)

        return State({"x": next_x, "y": next_y})

    def act(self, env: Environment):

        if not self.initialized:
            logger.error(
                "=> initialize the agent: `agent.init_agent(pop_size, x_bounds, y_bounds)`"
            )
            raise Exception("Agent not Initialized")

        next_generation = []
        sorted_by_fitness_population = self.sort_population_by_fitness(env.fitness_func)
        population_size = len(self.population)
```

```
        fitness_sum = sum(
            env.fitness_func(individual) for individual in self.population
        )

        for i in range(population_size):
            first_choice = self.choice_by_roulette(
                sorted_by_fitness_population, fitness_sum, env.fitness_func
            )
            second_choice = self.choice_by_roulette(
                sorted_by_fitness_population, fitness_sum, env.fitness_func
            )

            individual = self.crossover(first_choice, second_choice)
            individual = self.mutate(individual, env.x_bounds, env.y_bounds)
            next_generation.append(individual)

        self.population = next_generation

        best_individual = self.sort_population_by_fitness(env.fitness_func)[-1]

        # best individual is the new state
        new_state = State({"x": best_individual["x"], "y": best_individual["y"]})

        return new_state
```

**environment.py**

```
from typing import Tuple, Optional

from francium.core import State, BaseEnvironment
from francium.core.eval_functions import *

import numpy as np

class Environment(BaseEnvironment):
    def __init__(
        self,
        x_bounds: Tuple[float, float],
        y_bounds: Tuple[float, float],
        goal_val: Optional[State] = None,
        tolerance: Optional[float] = 1e-4,
        eval_func=convex_x_square,
    ):

        BaseEnvironment.__init__(
            self, x_bounds, y_bounds, goal_val, tolerance, eval_func=eval_func
        )

    def evaluate_state(self, state: State) -> Tuple[float, bool]:
        eval_val = self.evaluation_func(state["x"], state["y"])
```

```
        if self.goal_val:
            is_done = (
                True
                if np.abs(eval_val - self.goal_val["z"]) < self.tolerance
                else False
            )
        else:
            is_done = False

        return eval_val, is_done
```

**solver.py**

```
from francium.algorithms.genetic_algorithm.agent import Agent
from francium.algorithms.genetic_algorithm.environment import Environment
from francium.core import BaseSolver, setup_logger

logger = setup_logger(__name__)

class Solver(BaseSolver):
    def __init__(self, agent: Agent, environment: Environment, pop_size: int = 100):
        BaseSolver.__init__(self, agent=agent, environment=environment, solver_type="Genetic A
lgorithm")
        self.initialized: bool = False
        self.pop_size = pop_size

    def init_solver(self):
        self.agent.init_agent(self.pop_size, self.env.x_bounds, self.env.y_bounds)

        logger.info(f"=> Initialized Agent !")

        self.initialized = True

    def train_step(self) -> bool:

        if not self.initialized:
            logger.error("=> Solver not initialized !")
            raise Exception("Initialize the solver `solver.init_solver()`")

        new_state = self.agent(self.env)

        eval_val, is_done = self.env.evaluate_state(new_state)

        if is_done:
            logger.info(
                f"=> training is done ! best state: {self.memory.get_curr_state()}"
            )
            return False

        # logger.info(f"z: f(x = {new_state['x']}, y = {new_state['y']}) = {eval_val}")
        new_state["z"] = eval_val
```

```
        self.memory.add_episode(new_state)

        return True
```

Running the Genetic Algorithm

Genetic Algorithm: Environment I

```
import francium.algorithms.genetic_algorithm as ga
import francium.core.eval_functions as eval_functions

agent = ga.Agent()
env = ga.Environment(x_bounds=(-5.0, 5.0), y_bounds=(-
5.0, 5.0), eval_func=eval_functions.convex_x_square)
solver = ga.Solver(agent=agent, environment=env, pop_size=100)

solver.init_solver()

for episode in range(50):
    trainable = solver.train_step()
    if not trainable:
        break

solver.memory.best_episode
>>>
{'x': 2.5146588957311944, 'y': -2.1445624334917754, 'z': 10.922657393024195}

solver.plot_history()
>>>
```

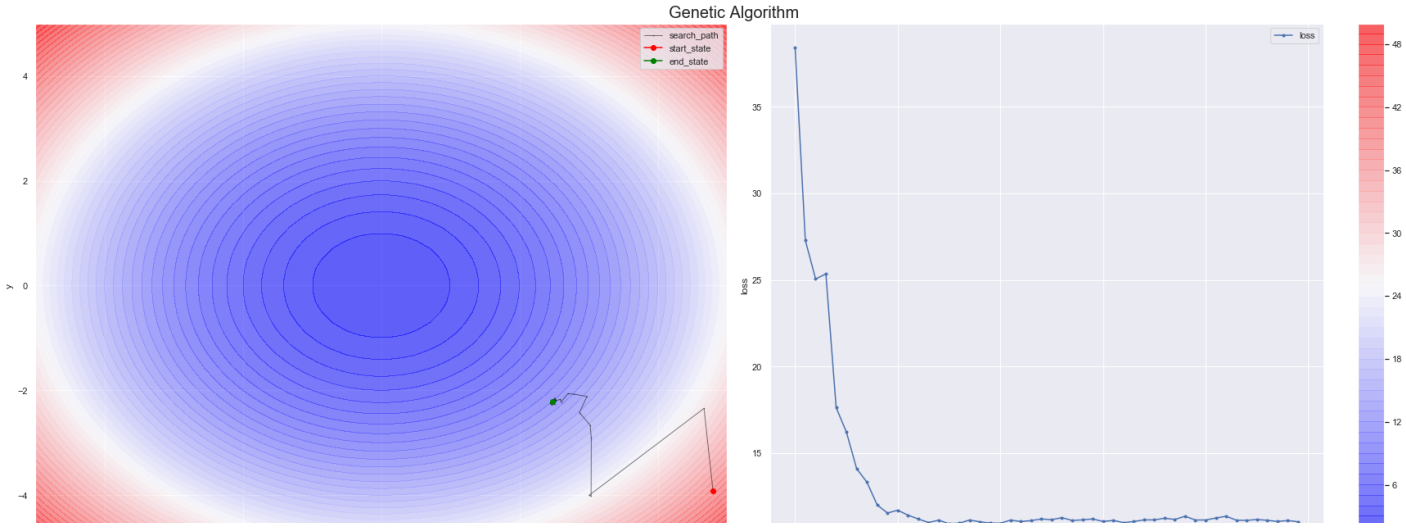



Figure 3-1 genetic-algorithm solve in environment-I

```
env.plot_environment()
>>>
```

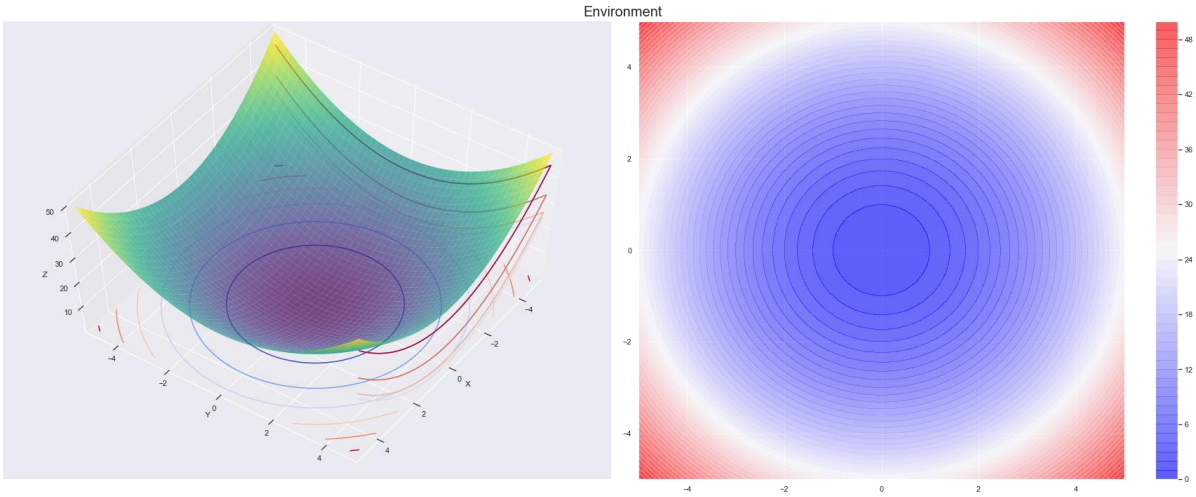



Figure 3-2 genetic-algorithm environment-I

Genetic Algorithm: Environment II

```
agent = ga.Agent()
env = ga.Environment(x_bounds=(-5.0, 5.0), y_bounds=(-
5.0, 5.0), eval_func=eval_functions.sinx_plus_x)
solver = ga.Solver(agent=agent, environment=env, pop_size=100)

solver.init_solver()

for episode in range(50):
    trainable = solver.train_step()
    if not trainable:
        break

solver.memory.best_episode

>>>
{'x': 2.7889714146335893, 'y': 0.15665519229271258, 'z': 12.796381098863423}

solver.plot_history()

>>>
```

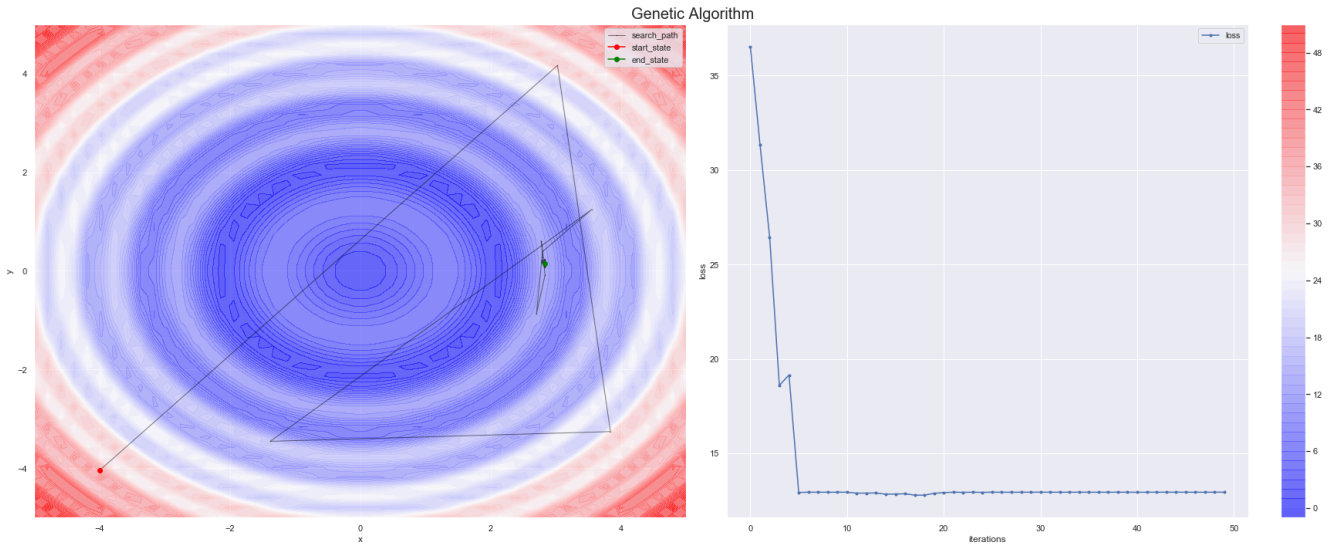



Figure 3-3 genetic-algorithm solver in environment-II

```
env.plot_environment()
```

```
>>>
```

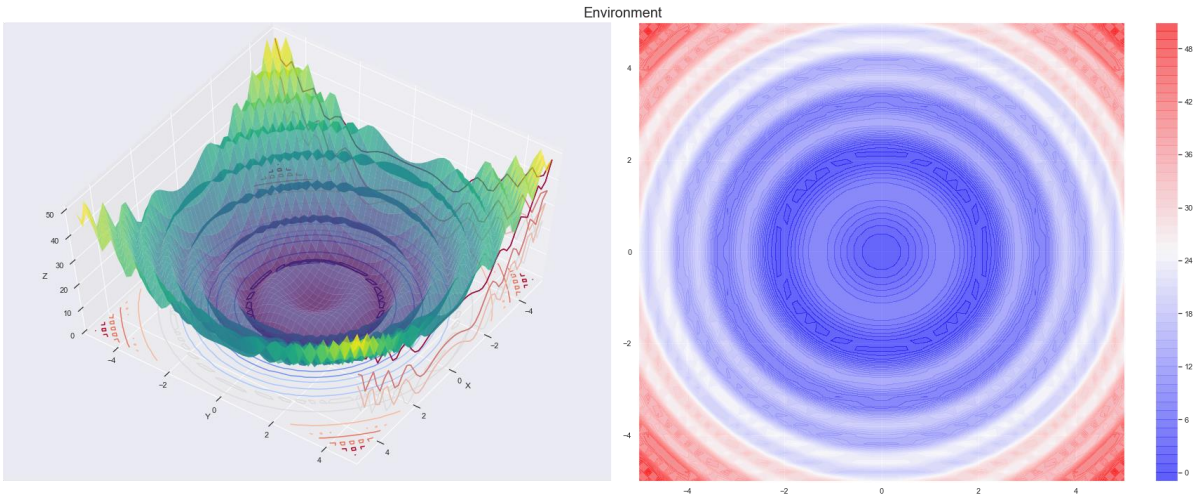



Figure 3-4 genetic-algorithm environment-II

### 3.3 Comparison with any other heuristic algorithm and Conclusion

A genetic algorithm is a stochastic hill-climbing search in which a large population of states is maintained. New states are generated by mutation and by crossover, which combines pairs of states from the population.

Results in Environment-I

| | X | Y | Z | ~computations |
|---|---|---|---|---|
| *Genetic Algorithm* | 2.5146 | -2.1445 | 10.9226 | 5000 |
| *Hill Climbing* | -0.00399 | -0.004717 | 0.00003825076 | 1000 |
| *Simulated Annealing* | 0.04819 | 0.5528 | 0.3079 | 1000 |

Results in Environment-II

| | X | Y | Z | ~computations |
|---|---|---|---|---|
| *Genetic Algorithm* | 2.7889 | 0.1566 | 12.7963 | 5000 |
| *Hill Climbing* | 3.5189 | 2.1670 | 12.1784 | 1000 |
| *Simulated Annealing* | 1.8416 | -1.0496 | -0.3871 | 1000 |

We'll try to compare Genetic Algorithm with Hill Climbing and Simulated Annealing, both of which are Heuristic algorithms, so basically Genetic Algorithm is like having bunch of inefficient stochastic hill climbing going on in place, but in practice we can definitely notice that GA is not good for optimization of *our selected kind of environments*. It performs pretty similar to hill climbing in environment-II and performs worse in environment-I, which was supposed to be a really easy task, this results can be because of the number of generations we are taking, in testing we have taken 50 generation, because of the fact that there are 100

individuals all around the search space at a given point of time, that's 5000 iterations in total ! which is 5X more iterations that other search algorithms we are comparing GA to.

So, it boils down to, if the search space has a single optimum, then choose hill climbing without any doubt, if it has many local optima, then try out simulated annealing, at the end, if the search space is large enough, probably GA? With my limited number of experiments, *I was not able to come to a good conclusion about the usage of GA.* A bigger search space with more complicated functions and more number of iterations would definitely yield much more comparable results than the ones shown in this assignment.

## Bibliography


1. Konar, A., 2006. Computational intelligence: principles, techniques and applications. Springer Science & Business Media.
2. Klir, G.J. and Yuan, B., 1996. Fuzzy sets and fuzzy logic: theory and applications. Possibility Theory versus Probab. Theory, 32(2), pp.207-208.
3. Anderson, J.A., 1972. A simple neural network generating an interactive memory. Mathematical biosciences, 14(3-4), pp.197-220.
4. Bender, E.A., 1996. Mathematical methods in artificial intelligence.
5. Vose, M.D. and Liepins, G.E., 1991. Punctuated equilibria in genetic search. Complex systems, 5(1), pp.31-44.
6. Vose, M.D., 1999. What are genetic algorithms? A mathematical prespective. In Evolutionary algorithms (pp. 251-276). Springer, New York, NY.
7. Russell, S. and Norvig, P., 2002. Artificial intelligence: a modern approach.
8. Brooks, J. and Hibler, D., 2013. The genetic flock algorithm. Procedia Computer Science, 20, pp.71-76.

## Appendix

ProjektFrancium


This assignment gave rise to ProjektFrancium, a tool for experimenting with AI Algorithms

**https://github.com/satyajitghana/ProjektFrancium**

```
import francium.algorithms.hill_climbing as hc
import francium.core.eval_functions as eval_functions
from francium.core import State

agent = hc.Agent(step_size=1e-1)
env = hc.Environment(x_bounds=(-5.0, 5.0), y_bounds=(-
5.0, 5.0), eval_func=eval_functions.sinx_plus_x)
solver = hc.Solver(agent=agent, environment=env)

solver.init_solver(
    init_state=State({
        'x': 4.0,
        'y': 2.0,
        'z': env.evaluation_func(4.0, 2.0)
    })
)

for episode in range(1000):
    trainable = solver.train_step()
    if not trainable:
        break

solver.plot_history()

env.plot_environment()
```