# Chapter 1

# Introduction

Nowadays every aspect of life is characterized by the need to make choices. Each choice is conditioned by a variable number of parameters. In many applicative domains (recommendation systems, medical analysis tools, real time game engines, speech recognizers...), this number could be very high. The possibility to manage in an efficient way such a number of parameters is guaranteed by a significant number of mathematical techniques increasingly performing and refined.

Every decisional problem can be translated into a *mathematical model* that represents the essence of the problem itself. A crucial step in formulating a model is the construction of the *objective function*. This requires developing a quantitative measure of performance relative to each of the decision maker's ultimate objectives that were identified while the problem was being defined. [HL01]

However we know that it is not always possible to define the objective function in advance. It is possible that objective function is unknown or, at least, partially unknown at the time the problem is dealt with. In these cases the optimization process of the objective function takes the name of *black-box optimization*.

Mathematically, we are considering the problem of finding a global maximizer (or minimizer) of an unknown objective function $f$ :

$$x^* = \arg\max_{x \in \mathcal{X}} f(x) \tag{1.1}$$

where $\mathcal{X}$ is some design space of interest. In global optimization, $\mathcal{X}$ is often a compact subset of $\mathbb{R}^d$ [SSW+16].

Bayesian optimization is one of the best known black-box optimization techniques. It is a powerful tool for the joint optimization of design choices that is gaining great popularity in recent years. It is a sequential model-based approach to solving problem [AS08]. We prescribe a prior beliefs over the possible objective functions and then sequentially refine this model as data are observed via Bayesian posterior updating. The Bayesian Posterior represents our updated beliefs -given data- on the likely objective function we are optimizing [SSW+16].

In this thesis I propose an innovative approach to black-box optimization based on the Reinforcement Learning (RL) technique.

Reinforcement Learning (RL) is the problem faced by a learner that must behaviour through trial-and-error interactions with a dynamic environment. It can be considered the problem of mapping situations to actions in order to maximize a numerical reward signal [KLM96].

In the first part of this work I will compare the state of the art performances of the Bayesian model with those obtained through the implementation of a Reinforcement Learning (RL) agent in the same order.

In the last part of the thesis I will explain how RL and Bayesian optimization can be joined to emulate human brain learning process.

# Chapter 2

# Background

## 2.1 Markov Decision Process

Markov Decision Processes (MDPs) are a classical formalization of sequential decision making under uncertainty, where actions influence not just immediate responses, but also subsequent situations. Hence an MDP can be described as a controlled *Markov chain*[1], where the control is given at each step by the chosen action. In this chapter we will present the structure of an MDP and many techniques to solve its.

**Markov Decision Process** Markov decision processes are defined as controlled stochastic processes satisfying the *Markov property*[2] and assigning reward values to state transitions [Put94]. Formally, they are described by the 5-tuple ($S$, $A$, $T$, $p$, $r$) where:

- $S$ is the state space in which the process' evolution takes place;
- $A$ is the set of all possible actions which control the state dynamics;
- $T$ is the set of time steps where decisions need to be made;
- $p()$ denotes the state transition probability function;
- $r()$ provides the reward function defined on state transitions.

[1] A sequence of random variables $X_0, X_1, ...$ with values in a countable set $S$ is a *Markov chain* if at any time $n$, the future states (or values) $X_{n+1}, X_{n+2}, ...$ depend on the history $X_0, ..., X_n$ only through the present state $X_n$ [Kon09] .

[2] A stochastic process has the *Markov property* if the probabilistic behaviour of the chain in the future depends only on its present value and discards its past behaviour.

**States** The set of environmental states $S$ is defined as the finite set $\{s_1, ..., s_N\}$ where the size of the state space is $N$, i.e. $|S| = N$ [WvO12]. Each state $s \in S$ is a vector of attributes (*state variables*) that describes the current configuration of the system [NFK06]. More specifically, a state variable is the minimally dimensioned function of history that is necessary and sufficient to compute the "state of knowledge" [Pow07].

**Actions** The set of actions $A$ is defined as the finite set $\{a_1, ..., a_K\}$ where the size of the action space is $K$, i.e. $|A| = K$. Actions can be used to control the system state. The set of actions that can be applied in some particular state $s \in S$, is denoted $A(s)$, where $A(s) \subseteq A$. In more structured representations the fact that some actions are not applicable in some states, is modelled by a precondition function : $S \times A \rightarrow true, false$, stating whether action $a \in A$ is applicable in state $s \in S$ [WvO12].

**Time Steps** The set of time steps $T$ is defined as the finite set $\{t_1, ..., t_M\}$ where the size of the action space is $M$, i.e. $|T| = M$. Speaking about time we have to distinguish between *epochs* and *episodes*. An episode is made of a fixed number $Z$ of epochs. An epoch is the smallest time unit in an MDP.

**Transition Probability** The transition probabilities $p()$ characterize the state dynamics of the system, i.e. indicate which states are likely to appear after the current state. For a given action $a$, $p(s'|s, a)$ represents the probability for the system to transit to state $s'$ after undertaking action $a$ in state $s$. This $p()$ function is usually represented in matrix form where we write $P_a$ the $|S|$ x $|S|$ matrix containing elements $\forall s, s', P_{\text{a, s, s'}} = p(s'|s, a)$. Since each line of these matrices sums to one, the $P_a$ are said to be stochastic matrices. The $p()$ probability distributions over the next state $s'$ follow the fundamental property which gives their name to Markov decision processes. If we write $h_t = (s_0, a_0, ..., s_{\text{t-1}}, a_{\text{a-1}}, s_t)$ the history of states and actions until time step $t$, then the probability of reaching state $s_{\text{t+1}}$ consecutively to action $a_t$ is only a function of $a_t$ and $s_t$, and not of the entire history $h_t$ [SB10]. We can resume this concept through the following equation :

$$\forall h_t, a_t, s_{\text{t+1}} \quad P(s_{\text{t+1}}|h_t, a_t) = P(s_{\text{t+1}}|s_t, a_t) = p(s_{\text{t+1}}|s_t, a_t) \tag{2.1}$$

**Reward Function** The reward function specifies rewards for being in a state, or doing some action in a state. The state reward function is defined as $R : S \rightarrow \mathbb{R}$, and it specifies the reward obtained in states. The

reward function is an important part of the MDP that specifies implicitly the *goal* of learning. Thus, the reward function is used to give direction in which way the system, i.e. the MDP, should be controlled [WvO12]. It is critical that the rewards we set up truly indicate what we want accomplished. If we reward the achievement of subgoals , then the agent might find a way to achieve them without achieving the real goal. Reward signal is our way of communicating the agent *what* we want to achieve, not *how* it achieved [SB18].

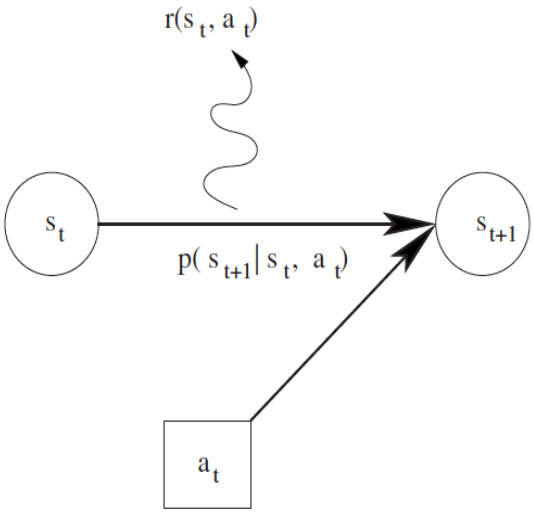


Figure 2.1: Markov decision process [SB10].

Markov decision processes allow us to model the state evolution dynamics of a stochastic system when this system is controlled by an agent choosing and applying the actions $a_t$ at every time step $t$. The procedure of choosing such actions is called an action policy, or strategy, and is written as $\pi$ [SB10] .

**Policy** Formally, given an MDP $\langle S, A, p(), r() \rangle$, a policy is a computable function that outputs for each state $s \in S$ an action $a \in A(s)$ [WvO12] A policy can decide deterministically upon the action to apply or can define a probability deistribution over the possible applicable actions. Then, a policy can be based on the whole history $h_t$ (history-dependent policy) or can only consider the current state $s_t$. Thus, we can obtain four main families policies, as shown in table 2.1.

| Policy $\pi_t$ | Deterministic | Stochastic |
|---|---|---|
| Markov | $s_t \to a_t$ | $a_t, s_t \to [0,1]$ |
| History-dependent | $h_t \to a_t$ | $h_t, s_t \to [0,1]$ |

Table 2.1: Different policy families for MDPs [SB10]

For a deterministic policy, $\pi_t(s_t)$ or $\pi_t(h_t)$ defines the chosen action $a_t$. For a stochastic policy, $\pi_t(a, s_t)$ or $\pi_t(a, h_t)$ represents the probability of selecting $a \in A$ for $a_t$ [SB10]. The sets so defined are included in each other, from the most general case of stochastic, history-dependent policies, to the very specific case of deterministic, Markov policies, as shown in figure 2.2.

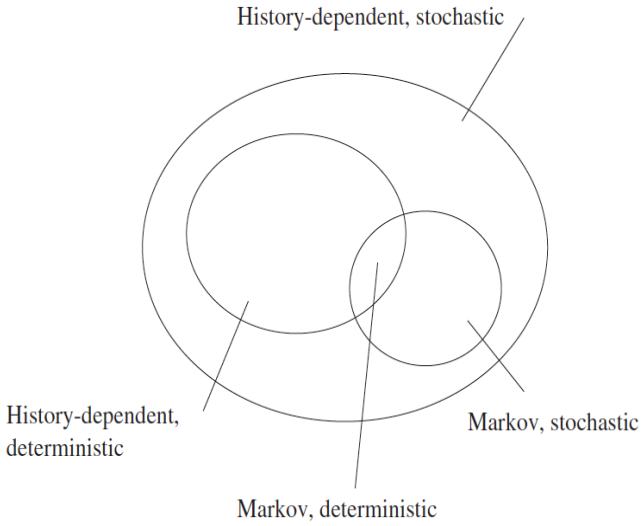


Figure 2.2: Relationship between the different sets of policies [SB10].

Application of a policy to an MDP is done in the following way. As shown in figure 2.3 each time an agent performs an action $a_\mathrm{t}$ in a state $s_\mathrm{t}$, it receives a real-valued reward $t_\mathrm{t}$ that indicates the immediate value of this state-action transition. This produces a sequence of states $s_\mathrm{i}$, actions $a_\mathrm{i}$, and immediate rewards $r_\mathrm{i}$ as shown in the figure. The agent's task is to learn a control policy, $\{\pi : S \longrightarrow A\}$, that maximizes the expected sum of these rewards, with future rewards discounted exponentially by their delay [Mit97].

As already said the goal of learning in an MDP is to gather rewards. There are several ways of taking into account the future in how to behave

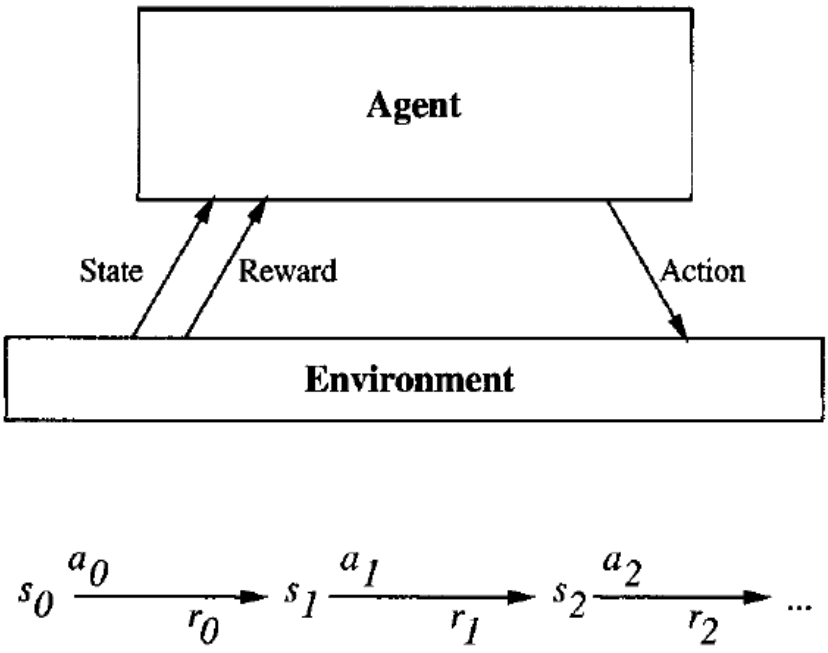


Figure 2.3: Policy application schema [SB18].

now. There are basically three models of optimality in the MDP, which are sufficient to cover most of the approaches in the literature :

$$E[\sum_{t=0}^{h} r_t] \qquad E[\sum_{t=0}^{\infty} \gamma^t r_t] \qquad \lim_{h\to\infty} E[1/h \sum_{t=0}^{h} r_t]$$

Figure 2.4: Optimality : **a)** finite horizon, **b)** discounted, infinite horizon, **c)** average reward

**Discount Factor** The *finite horizon* model simply takes a finite horizon of length $h$ and states that the agent should optimize its expected reward over this horizon. In the *infinite-horizon model*, the long-run reward is taken into account, but the rewards that are received in the future are discounted according to how far away in time they will be received. A

*discount factor* $\gamma$, with $0 \leq \gamma \leq 1$ is used for this. Note that in this discounted case, rewards obtained later are discounted more than rewards obtained earlier. Additionally, the discount factor, ensures that, even with infinite horizon, the sum of the rewards obtained is finite. In episodic tasks, i.e. in tasks where the horizon is finite, the discount factor is not needed or can equivalently be set to 1. If $\gamma = 0$ the agent is said to be myopic, which means that it is only concerned about immediate rewards. The last optimality model is *average-reward* model, maximizing the long-run *average-reward*. Sometimes this is called the *gain optimal* policy and in the limit, it is equal to the infinite-horizon discounted model [WvO12]

**Bellman Equation** The concept of *value function* is the link between optimality criteria and policies. Most learning algorithms for MDPs compute optimal policies by learning value functions. The value of a state $s$ under policy $\pi$, denoted $V^\pi(s)$ is the expected return when starting in $s$ and following $\pi$ thereafter. Using the infinite-horizon, discounted model :

$$V^\pi(s) = E_\pi\{\sum_{k=0}^{\infty} \gamma^k t_{\mathrm{t+k}} | s_t = s\} \tag{2.2}$$

A similar state-action value function : $Q : S \text{ x } A \rightarrow \mathbb{R}$ can be defined as the expected return starting from state $s$. taking action $a$ and thereafter following policy $\pi$ :

$$Q^\pi(s,a) = E_\pi\{\sum_{k=0}^{\infty} \gamma^k r_{\mathrm{t+k}} | s_t = s, a_t = a\} \tag{2.3}$$

For any policy $\pi$ and any state $s$ the expression 2.3 can recursively be defined in terms of a so-called *Bellman Equation* :

$$\begin{aligned} V^\pi(s) &= E_\pi\{r_t + \gamma_{\mathrm{t+1}} + \gamma^2 r_{\mathrm{t+2}} + ... | s_t = t\} \\ &= E_\pi\{r_t + \gamma V^\pi(s_{\mathrm{t+1}}) | s_t = s\} \\ &\sum_{s'} T(s, \pi(s), s')(R(s,a,s') + \gamma V^\pi(s')) \end{aligned} \tag{2.4}$$

It denotes that the expected value of state is defined in terms of immediate reward and values of possible next state weighted by the transition probabilities, and additionally a discount factor. Note that multiple policies can have the same value function, but for a given policy $\pi$, $V^\pi$ is unique. The goal for any MDP is to find a best policy, i.e. the policy that receives the

most reward. This means maximizing the value function of equation 2.3 for all states $s \in S$. An optimal policy, denoted $\pi^*$, is such that $V^{\pi^*}(s) \geq V^{\pi}(s)$ for all $s \in S$ and all policies $\pi$ :

$$V^*(s) = \max_{a \in A} \sum_{s'} T(s, \pi(s), s')(R(s, a, s') + \gamma V^{\pi}(s')) \tag{2.5}$$

This expression is called the *Bellman optimality equation*. It states that the value of a state under an optimal policy must be equal to the expected return for the best action in a state [WvO12].

**Value Iteration vs Policy Iteration** Before studying how to solve MDPs let's clarify main differences between two key concepts we will recursively find : *value iteration* and *policy iteration*.

Both value iteration and policy iteration compute the same thing (all optimal values), i.e. they work with Bellman updates. In Value iteration we start with a random value function and then we find a new (improved) value function in an iterative process, until reaching the optimal value function. Value iteration computation of Bellman Equation is:

$$V_{\mathrm{k+1}}(s) \leftarrow \max_{a} \sum_{s'} T(s, a, s')[R(s, a, s') + \gamma V_k(s')] \tag{2.6}$$

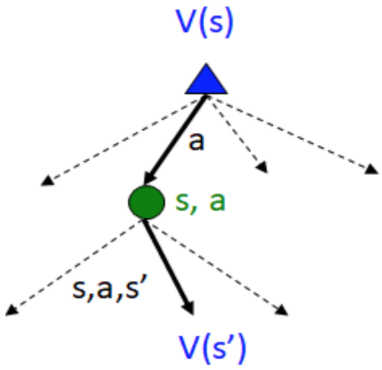


Figure 2.5: Value Iteration Schema

Value iteration has three disadvantages: it is slow ($O(S^2A)$ per iteration), the max at each state rerely changes, the policy often converges long before the values.

Instead, policy iteration computations of Bellman Equation are:

- Evaluation : for a fixed current policy $p$, find values with policy evaluation:

$$V^{\pi_i}_{k+1}(s) \leftarrow \sum_{s'} T(s, \pi_i(s), s')[R(s, \pi_i(s), s') + \gamma V^{\pi_i}_{\mathrm{k}}(s')] \tag{2.7}$$

- Improvement : for fixed values, get a better policy using policy extraction :

$$\pi_{\mathrm{i+1}}(s) = \arg\max_a \sum_{s'} T(s, a, s')[R(s, a, s') + \gamma V^{\pi_i}(s')] \tag{2.8}$$

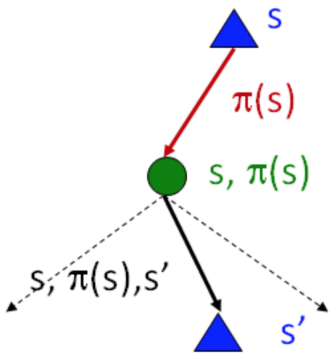


Figure 2.6: Policy Iteration Schema

Complexity of policy iteration is $O(S^2)$.

Summarizing what said until now we can write:

In value iteration :

- Every iteration updates both the values and implicitly the policy;
- We don't track the policy, but tacking the max over actions implicitly recomputes it.

In policy iteration :

- We do several passes that update utilities with fixed policy;

- After the policy is evaluated, a new policy is chosen;
- The new policy will be always better until the end.

## 2.2 Solving MDP

Now that we have defined MDPs, policies, optimality criteria and value functions, it is time to consider the question of how to solve an MDP computing an optimal policy $\pi^*$. Several dimensions exists along which algorithms have been developed for this purpose. The most important distinction is that between *model-based* and *model-free* algorithms [WvO12].

| | **Model-based algorithms** | **Model-free algorithms** |
|---|---|---|
| **General name** | DP | RL |
| **Basic assumption** | A model of the MDP is known beforehand, and can be used to compute value functions and policies using the Bellman equation. | Rely on interaction with the environment. Because a model of the MDP is not known, the agent has to explore the MDP to obtain information. |
| **Planning** | Yes | Yes |
| **Learning** | No | Yes |

Table 2.2: Main differences between model-based and model-free algorithms.

### 2.2.1 Dynamic Programming

The term Dynamic Programming (DP) refers to a class of algorithms that is able to compute optimal policies in the presence of a perfect model of the environment described as a Markov Decision Process. These algorithms are known as *planning algorithms*. In planning, the idea is that we are given some description of a starting state or states; a goal state or states; and some set of possible actions that the agent can take; we want to find the sequence of actions that get us from the start state to the goal state. We search through a tree that is the sequences of actions that we can take, and we try to find a nice short plan. In so doing two of the possible planning algorithms are `BFS algorithm` and `DFS algorithm`.

The method of Dynamic Programming systematically records solutions for all sub-problems of increasing lengths. Using this programming paradigm the optimal policy is defined through the step-by-step definition of optimal sub-policies. According to Bellman optimality principle, all optimal sub-policies of an optimal policy are optimal sub-policies. DP algorithms are

obtained by turning Bellman equations into update rules for improving approximations of the desired value functions.

Studying DP we can distinguish two different main methods. *Policy evaluation* refers to the typically iterative computation of the value functions for a given policy. *Policy improvement* refers to the computation of an improved policy given the value function for that policy. Either of these can be used to reliably compute optimal policies and value functions for finite MDPs given complete knowledge of the MDP.

Classical DP methods operate in sweeps through the state set, performing an *expected update* operation on each state. Each such operation updates the value of one state based on the values of all possible successor states and their probabilities of occurring. Expected updates are closely related to Bellman equations: they are little more than these equations turned into assignment statements. When the updates no longer result in any changes in value, convergence has occurred to values that satisfy the corresponding Bellman equation [SB18].

The assumption that a model is available will be hard to ensure for many applications, however DP algorithms are very relevant because they define fundamental computational mechanism which are also used when no model is available.

### 2.2.2 Reinforcement Learning

Dynamic Programming' s methods compute optimal policies for an MDP assuming that a perfect model is available. Reinforcement Learning (or *approximate dynamic programming*, or *neuro-dynamic programming*) is primarily concerned with how to obtain an optimal policy when such a model is not available. RL adds to MDPs a focus on approximation and incomplete information, and the need for sampling and exploration. In contrast with DP's algorithms, model-free methods do not rely on the availability of priori known transition and reward models. The lack of the model generates a need to *sample* the MDP to gather statistical knowledge about this unknown model. Many model-free RL techniques exist that probe the environment by doing actions, thereby estimating the same kind of state value and state-action value functions as model-based techniques [WvO12].

Roughly speaking Reinforcement Learning (RL) is the problem faced by a learner that must behaviour through trial-and-error interactions with a dynamic environment. It can be considered a problem of mapping situa-

tions to actions in order to maximize a numerical reward signal [KLM96].

In RL the learner must select an action to take in each time step: every choice done by the agent changes the environment in an unknown fashion and receives a reward which value is based on the consequences. The objective of the learner is to choose a sequence of actions based on observations of the current environment that maximizes cumulative reward or minimizes cumulative cost over all time steps [LM16]. The general class of algorithms that interact with the environment and update their estimates after each experience is called *online* RL.

**Algorithm 1:** A general algorithm for online RL [WvO12]

```
1  foreach episode do
2      s ∈ S is initialized as the starting state;
3      t := 0;
4      repeat
5          choose an action a ∈ A(s);
6          perform action a ;
7          observe the new state s' and received reward r update T̃, R̃, Q̃
             and/or Ṽ using the experience ⟨s, a, r, s'⟩;
8          s := s';
9      until s' is a goal state;
10 end
```

Studying RL, one of the first problems we have to face with is the distinction between *direct* and *indirect* Reinforcement Learning. We will explain the difference between these two approaches in figure 2.7.

| | **REINFORCEMENT LEARNING** | |
|---|---|---|
| | **Model Based / Indirect** | **Model Free / Direct** |
| **Basic Assumption** | First to learn the transition and reward model from interaction with the environment. After that, when the model is (approximately or sufficiently) correct, all the DP methods from the previous section apply. | Step right into estimating values for actions, without even estimating the model of the MDP. |

Figure 2.7: RL's approaches classification.

RL is different both from *supervised learning* and from *unsupervised learning*. It is not a sample of learning from a training set of labelled examples provided by a knowledgeable external supervisor (*supervised learning*) and it is not a sample of searching and finding structure hidden in a collection of unlabelled data (*unsupervised learning*). More specifically we can say that RL can be distinguished from other forms of learning based on the following characteristics :

- Reinforcement Learning deals with temporal sequences. In contrast with non-supervised learning problems where the order in which the examples are presented is not relevant, the choice of an action at a given time step will have consequences on the examples that are received at a subsequent time steps [SB10].

- In contrast with supervised learning, the environment does not tell the agent what would be the best possible action. Instead, the agent may just receive a scalar reward representing the value of its action and it must *explore* the possible alternative actions to determine whether its action was the best or not [SB10].

According to what just said, one of the challenges that arise in RL, and not in other kind of learning, is the trade-off between *exploration* and *exploitation*. To obtain an higher reward, an RL agent must prefer actions that it has tried in the past and found to be effective in producing reward. In order to discover such actions, it has to try actions that it has not selected before. The agent *exploits* what it has already experienced in order to obtain reward, but it has also to *explore* in order to eventually make better action selections in the future [SB18]. In other words *exploitation* consists of doing again actions which have proven fruitful in the past, whereas *exploration* consists of trying new actions, looking for a larger cumulated reward, but eventually leading to a worse performance. Dealing with the exploration/exploitation trade-off consists of determining how the agent should explore to get as fast as possible a policy that is optimal or close enough to the optimum. The most basic exploration strategy is the $\epsilon$-greedy policy, i.e. the learner takes its current best action with probability $(1 - \epsilon)$ and a (randomly selected) other action with probability $\epsilon$. [SB10].

**Monte Carlo Methods** Monte Carlo Methods (MC) represent a first instance of model-free RL's algorithms. They are one way solving the Reinforcement Learning problem based on averaging sample returns.

The Monte Carlo approach consists of performing a large number of trajectories from all states $s$ in $S$, and estimating $V(s)$ as an average of the cumulated rewards observed along these trajectories. In each trial, the agent records its transitions and rewards, and updates the estimates of the value of the encountered states according to a discounted reward scheme. The value of each state then converges to $V^{\pi}(s)$ for each $s$ if the agent follows policy $\pi$.

More formally, let $(s_0, s_1, ..., s_N)$ be a trajectory consistent with the policy $\pi$ and the unknown transition function $p()$, and let $(r_1, ..., r_N)$ be the rewards observed along this trajectory. In MC method, the $N$ values $V(s_t), \quad t = 0, ...., N-1$ are updated according to :

$$V(s_t) \leftarrow V(s_t) + \alpha(s_t)(r_{\mathrm{t}+1} + r_{\mathrm{t}+2} + ... + r_N - V(s_t)) \tag{2.9}$$

with the learning rates $\alpha(s_t)$ converging to 0 along the iterations [SB10] . MC algorithms treat the long-term reward as a random variable and take as its estimate the sampled mean. Because the sampling is dependent on the current policy $\pi$, only returns for actions suggested by $\pi$ are evaluated. Thus, *exploration* is of key importance here, just as in other model-free methods. One way of ensuring enough exploration is to use exploring starts, i.e. each state-action pair has a non-zero probability of being selected as the initial pair.

A distinction can be made between *every-visit* MC, which averages over all visits of a state $s \in S$ in all episodes, and *first-visit* MC, which averages over just the returns obtained from the first visit to a state $s \in S$ for all episodes. Both variants will converge to $V^{\pi}$ for the current policy $\pi$ over time [WvO12] .

Studying RL methods we have to distinguish between *on-policy methods* and *off-policy methods*. On-policy methods attempt to evaluate or improve the policy that is used to make decisions. Off-policy methods evaluate or improve a policy different from that used to generate the data. MC methods can be used for both on-policy and off-policy control, and the general pattern complies with the generalized policy iteration procedure.

**Temporal Difference Learning** An important problem faced by RL is the so called *temporal credit assignment problem*. In model-free contexts it is difficult to assess the utility of some action, if the real effects of this particular action can only be perceived much later. One possibility is to wait

| | **On-policy methods** | **Off-policy methods** |
|---|---|---|
| **Basic assumption** | They attempt to evaluate or improve the policy that is used to make decisions. | We do not need to follow any specific policy; our agent could even behave randomly and despite this it can still find the optimal policy. |

Table 2.3: Difference between on-policy and off-policy methods.

until the "end" (e.g. of an episode) and punish or reward specific actions along the path taken. However, this will take a lot of memory and often, with ongoing tasks, it is not known beforehand whether, or when, there will be an "end". Instead, one can use similar mechanisms as in value iteration to adjust the estimated value of a state based on the immediate reward and the estimated (discounted) value of the next state. This is generally called *temporal difference learning* which is a general mechanism underlying the model-free methods [WvO12].

We can formally represent this concept modifying equation 2.9 as follow:

$$V(s_t) \leftarrow V(s_t) + \alpha(s_t)(\delta_t + \delta_{\text{t+1}} + ... + \delta_{\text{N-1}}) \tag{2.10}$$

defining the temporal difference error $\delta_t$ by :

$$\delta_t = r_{\text{t+1}} + \gamma V(s_{\text{t+1}}) - V(s_t), \quad t = 0, ..., N-1. \tag{2.11}$$

The error $\delta_t$ must be interpreted in each state as a measure of the difference between the current estimation $V(s_t)$ and the correct estimation $r_{\text{t+1}} + V(s_{\text{t+1}})$.

Like Monte Carlo methods, TD ones performs estimation of action-value functions based on the experience of the agent and can do without a model of the underlying MDP; however, they combine this estimation using local estimation propagation mechanisms coming from dynamic programming, resulting in their incremental properties. Thus TD methods, which are at the heart of most reinforcement learning algorithms, are characterized by this combination of estimation methods with local updates incremental properties [SB10].

**Sarsa Algorithm : On-policy TD Control** Knowing the exact value of all states is not always enough to determine what to do. If the agent does not know which action results in reaching any particular state, i.e. if the agent does not have a model of the transition function, knowing $V$ does

not help it determine its policy. To solve this problem, Watkins [WD92] introduced the action-value function $Q$, whose knowledge is similar to the knowledge of $V$ when $p$ is known. The action-value function of a fixed policy $\pi$ whose value function is $V^\pi$ is :

$$\forall s \in S, a \in A, \quad Q^\pi(s,a) = r(s,a) + \gamma \sum_{s'} p(s'\ s,a) V^\pi(s'). \tag{2.12}$$

The value of $Q^\pi(s,a)$ is interpreted as the expected value when starting from $s$, executing $a$ and then following the policy $\pi$ afterwards. We have $V^\pi(x) = Q^\pi(x, \pi(x))$ and the corresponding Bellman equation is :

$$\forall s \in S, a \in A \quad Q^*(s,a) = r(s,a) + \gamma \sum_{s'} p(s'|s,a) \max_b Q^*(s',b) \tag{2.13}$$

Than we have :

$$\forall s \in S, \quad V^*(s) = \max_a Q^*(s,a), \quad \pi^*(s) = \arg\max_a Q^*(s,a). \tag{2.14}$$

The SARSA algorithm works on state-action pairs rather than on states. Its update equation is :

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha[r_{t+1} + \gamma Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t)]. \tag{2.15}$$

the information necessary to perform such an update is $(s_t, a_t, r_{t+1}, s_{t+1}, a_{t+1})$, hence the name of the algorithm : SARSA. This algorithm suffers from one conceptual drawback: performing the updates as stated above implies knowing in advance what will be the next action $a_{t+1}$ for any possible next state $s_{t+1}$. As a result, the learning process is tightly coubled to the current policy (the algorithm is called "on-policy") and this complicates the exploration process. As a result, proving the convergence of SARSA was more difficult than proving the convergence of "off-policy" algorithms such as Q-learning. Empirical studies often demonstrates the better performance of SARSA compared to Q-learning [SB10].

**Q-learning Algorithm : Off-policy TD Control** The Q-Learning Algorithm can be seen as a simplification of the algorithm, given that it is no

**Algorithm 2:** SARSA Algorithm : On-policy TD Control

Initialize $Q(s, a)$ arbitrarily;
Repeat the next cycle until $s$ is terminal;
**foreach** *epsiode* **do**
    Initalize $s$;
    Choose $a$ from $s$ using policy derived from $Q$ (e.g. $\epsilon$-greedy);
    **foreach** *step of episode* **do**
        Take action $a$, observer $r$, $s'$;
        Choose $a'$ from $s'$ using policy derived from $Q$ (e.g. $\epsilon$-greedy);
        $Q(s, a) \leftarrow Q(s, a) + \alpha[r + \gamma Q(s', a') - Q(s, a)]$ ;
        $s \leftarrow s'$ ;
        $a \leftarrow a'$ ;
    **end**
**end**

more necessary to determine the action at the next step to calculate updates. Its update equation is :

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha[r_{t+1} + \gamma \max_a Q(s_{t+1}, a) - Q(s_t, a_t)] \tag{2.16}$$

The main difference between SARSA and Q-Learning lies in the definition of the error term. The $Q(s_{t+1}, a_{t+1})$ term in the equation 2.15 is replaced by $\max_a Q(s_{t+1}, a)$ in equation 2.16. Updates are based on instantaneously available information. In this algorithm, the $T_{tot}$ parameter corresponds to the number of iterations. There is here one learning rate $\alpha_t(s, a)$ for each state-action pair, it decreases at each visit of the corresponding pair. The `Simulate` function returns a new state and the corresponding reward according to the dynamics of the system. The choice of the current state and of the executed action is performed by functions `ChooseState` and `ChooseAction`. The `Initialize` function initializes the $Q$ function with $Q_0$, which is often initialized with `null` values, whereas more adequate choices can highly improve the performance [SB10].

**SARSA($\lambda$)** Previous algorithms only perform one update per time step in the state that the agent is visiting. This update process is particularly slow. Indeed, an agent deprived of any information on the structure of the value function needs at least $n$ trials to propagate the immediate reward of a state to another state that is $n$ transitions away. Before this propagation

**Algorithm 3:** Q-learning Algorithm [SB10]

$\alpha_t$ is the learning rate ;
Initialize $(Q_0)$;
**for** $t = 0$ ; $t \leq T_{tot} - 1$ ; $t++$ **do**
    $s_t \leftarrow$ `ChooseState`;
    $a_t \leftarrow$ `ChooseAction`;
    $s_{t+1}, r_{t+1} \leftarrow$ `Simulate`$(s_t, a_t)$;
    update $Q_t$ ;
    **begin**
        $Q_{t+1} \leftarrow Q_t$;
        $\delta_t \leftarrow r_{t+1} + \gamma \max_b Q_t(s_{t+1}, b) - Q_t(s_t, a_t)$ ;
        $Q_{t+1}(s_t, a_t) \leftarrow Q_t(s_t, a_t) + \alpha_t(s_t, a_t)\delta_t$ ;
    **end**
**end**
**return** $Q_{Tot}$

is achieved, if the initial values are `null`, the agent performs a random walk in the state space, which means that it needs an exponential number of steps as a function of $n$ before reaching the reward "trail". A naive way to solve the problem consists of using a memory of trajectory and to propagate all the information backwards along the performed transitions each time a reward is reached. Such a memory of performed transitions is called an eligibility trace". A problem with this naive approach is that the required memory grows with length of trajectories, which is obviously not feasible in the infinite horizon context. In SARSA($\lambda$) algorithm a more sophisticated approach that addresses the infinite horizon context.

As we already saw Q-learning integrates the temporal difference error idea. With the update rule of Q-learning :

$$Q_{t+1}(s_t, a_t) = Q_t(s_t, a_t) + \alpha_t\{r_{t+1} + \gamma V_t(s_{t+1}) - Q_t(s_t, a_t)\} \tag{2.17}$$

for transition $s_t, a_t, s_{t+1}, r_{t+1})$, and in the case where action $a_t$ executed in state $s_t$ is the optima action for $Q_t$, then the error term is $r_{t+1} + \gamma V_t s_{t+1} - V_t(s_t)$. For a generic parameter $\lambda \in [0, 1]$ we can generalize as follow:

where $z_t(s, a)$ and is the eligibility trace and *absorbing state* means terminal state [SB10].

**Algorithm 4:** SARSA ($\lambda$) Algorithm [SB10]

/* $\alpha$ is a learning rate */ ;
`Initialize` $Q_0$ ;
$z_0 \leftarrow 0$ ;
$s_0 \leftarrow$ `ChooseState` ;
$a_0 \leftarrow$ `ChooseAction` ;
$t \leftarrow 0$ ;
**while** $t \leq T_{tot} - 1$ **do**
    $(s'_t, r_{t+1}) \leftarrow$ `Simulate` $(s_t, a_t)$ ;
    $a'_t \leftarrow$ `ChooseAction` ;
    update $Q_t$ and $z_t$ ;
    **begin**
        $\delta_t \leftarrow r_{t+1} + \gamma Q_t(s'_t, a'_t) - Q_t(s_t, a_t)$ ;
        $z_t(s_t, a_t) \leftarrow z_t(s_t, a_t) + 1$ ;
        **for** $s \in S, a \in A$ **do**
            $Q_{t+1}(s, a) \leftarrow Q_t(s, a) + \alpha_t(s, a) z_t(s, a) \delta_t$ ;
            $z_{t+1}(s, a) \leftarrow \gamma \lambda z_t(s, a)$
        **end**
        **if** $s'_t$ *non absorbing* **then**
            $s_{t+1} \leftarrow s'_t$ and $a_{t+1} \leftarrow a'_t$ ;
        **end**
        **else**
            $s_{t+1} \leftarrow$ `ChooseState` ;
            $a_{t+1} \leftarrow$ `ChooseAction` ;
        **end**
    **end**
**end**
**return** $Q_{Tot}$

# Chapter 3

# A Reinforcement Learning Approach to Black-Box Optimization

One of the principal challenges in optimization practice is the possibility to optimize in absence of an algebraic model of the system to be optimized. This kind of optimization is known as *black-box optimization*.

A black-box function $f(x) : \mathbb{R}^n \to \mathbb{R}$ is a function for which the analytic form is not known. Nowadays there are lots of mathematical models to succeed in optimize this kind of functions. One of the best known of these methods is the *Bayesian Optimization Model* (BO).

In this chapter we will first analyse the state of art of the most used black-box optimization model explaining how BO works and than we will propose an innovative, RL based approach to solve the same problem.

**Gaussian Processes** Before introducing BO we have to describe what *Gaussian Processes* (GPs) are. GPs are an alternative approach to regression problems. The GP approach is a *non-parametric* approach (we don't have a priori knowledge of how many parameters will be useful for our regression) to find a distribution over the possible functions $f(x)$ that are consistent with observed data. A GP is a generalization of the Gaussian probability distribution. Whereas a probability distribution describes random variables which are scalars or vectors (for multivariate distributions), a *stochastic process* governs the properties of functions.

GP is a convenient and powerful prior distribution on functions, which we will take here to be of the form

$$f : \mathcal{X} \leftarrow \mathbb{R}. \tag{3.1}$$

The GP is defined by the property that any finite set of $N$ points $\{x_n \in \mathcal{X}\}_{n=1}^{N}$ induces a multivariate Gaussian distribution on $\mathbb{R}^N$. The $n$th of these points is taken to be the function value $f(x_n)$ [SLA12]. The support and properties of the resulting distribution on functions are determined by a mean function

$$m : \mathcal{X} \leftarrow \mathbb{R} \tag{3.2}$$

and by a positive definite covariance function

$$k : \mathcal{X} \times \mathcal{X} \leftarrow \mathbb{R}. \tag{3.3}$$

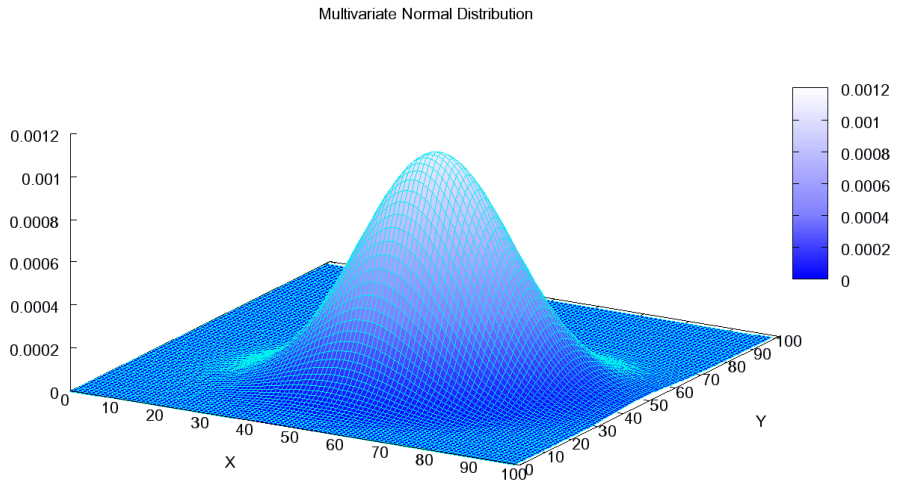


Figure 3.1: Multivariate Gaussian Distribution [MND]

**Acquisition Functions for Bayesian Optimization** Let's assume that the function $f(x)$ is drawn from a GP prior and that our observation are of the form $\{x_n \in \mathcal{X}\}_{n=1}^{N}$, where $y_n \sim \mathcal{N}(f(x_n), v)$ and $v$ is the variance of

noise introduced into the function observations. This prior and these data induce posterior over functions; the acquisition function, which we denote by

$$a : \mathcal{X} \leftarrow \mathbb{R}^+, \tag{3.4}$$

determines what point in $\mathcal{X}$ should be evaluated next via a proxy optimization

$$x_{\text{next}} = \arg\max_x a(x), \tag{3.5}$$

where several different functions have been proposed. There are several popular choices of acquisition functions. Under the Gaussian process prior, these functions depend on the model solely through its predictive mean function $\mu(x; \{x_n, y_n\})$ and predictive variance function $\sigma^2(x; \{x_n, y_n\})$ [SLA12].

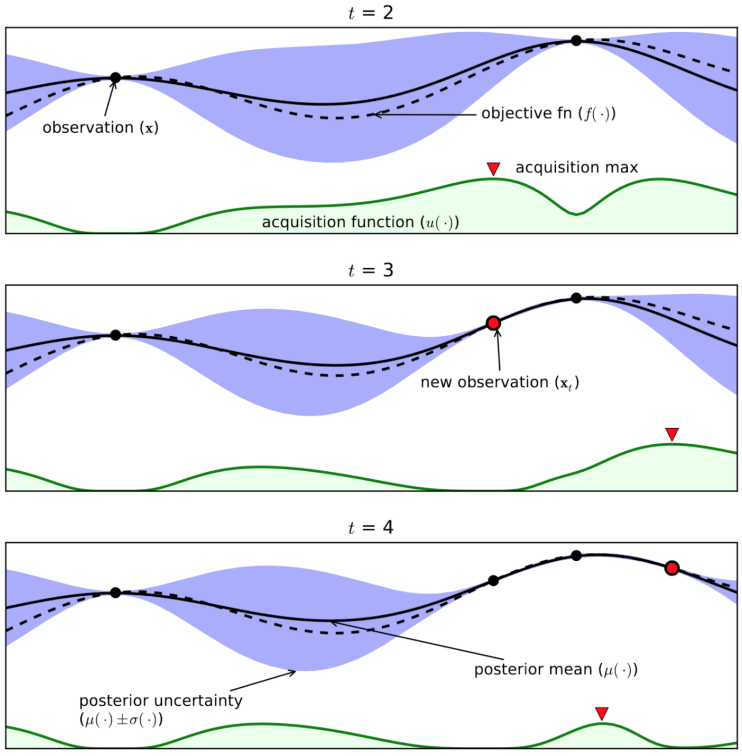


Figure 3.2: 2-*d* Bayesian Optimization Process Example [Bay]

**An RL Approach To Black-Box Optimization** As previously said, the aim of this thesis is to describe an innovative RL approach to the black-box function optimization problem. Let's assume to dispose of a $3d$-function $f(x, y)$. In this scenario :

- **Agent** : The agent has to maximize a black-box bivariate function. The function is continuously defined over a specific domain. The agent has to complete its job making exactly 150 epochs for each one of the 1000 episodes. In each epoch it has a position in space described through the two coordinates $(x, y)$. Each time the agent makes an action the angle between $(x, y)$ and $(x', y')$ and the value of the function $f(x', y')$ are computed.
- **State** : The state is represented by two lists: the first one contains the last two computed *angles* and the second one contains the correspondent last two *actions*.
- **Actions** : In each epoch the agent can make one of four different actions : *move north*, *move south*, *move east*, *move west*. Each time the agent moves itself of 40 pixels in one of the previously described directions. The resultant effective movement is computed as follow:

**Algorithm 5:** From pixels to real values

```
1  /* knowing pixelX and pixelY */;
2  /* knowing pixelXRange and pixelYRange */ ;
3  /* knowing function */;
4
5  domain = function.getDomain() ;
6  xRange = domain.maxX − domain.minX ;
7  yRange = domain.maxY − domain.minY ;
8
9  xReal = domain.minX + (pixelX ∗ xRange)/pixelXRange ;
10 yReal = domain.minY + (pixelY ∗ yRange)/pixelYRange ;
11
12 return xReal, yReal
```

- **Reward** : In this context we have decided to reward every action of the agent because a real terminal state doesn't exist. Cause we are working with a black-box function we cannot select two coordinate $(x, y)$ and a corresponding value function $z$ as a terminal state. The

simulation ends after 1000 episodes are done. We define a $\Delta$ equals to $\max f(z_n)$ minus $f(z)$ computed in the *current state*:

$$\Delta = maxf(x_n, y_n) - f(x, y) \tag{3.6}$$

Our reward at each epoch is equal to $\Delta$.

The movement the agent can make in each epoch can be of two different types : *linear movement* or *parametric movement*. If the movement is linear we compute the angle as follow :

**Algorithm 6:** Angle computation in linear movement case.

```
1  /* knowing (x, y) */ ;
2  /* knowing (x', y')*/ ;
3  /* movementAmount = M */ ;
4  /* currentMax = max f(x_n, y_n) */ ;
5
6  z = f(x, y) ;
7  z' = f(x', y');
8
9  δ = ((x' − x), (y' − y)) ;
10 α = arctan(δ / movementAmount)
11
12 Δ =z'−currentMax
```

We can explain this algorithm briefly recalling some trigonometry. Let's suppose to be in a simple 2d case like the one represented in figure 3.3.

Let's suppose that the starting point of our RL agent at epoch $e$ is the point $A$ with coordinates $(x, y)$. In the represented case the agent can make only one of two actions for each epoch : *move on* and *go back*. Making a linear movement of the type *move on* the arriving point at epoch *e'* is $B$ with coordinates $(x', y)$. Knowing that the objective function is $f(x) = \sin(2x)$, we can now compute $f(x')$. We now know point $C$ with coordinates $(x', f(x'))$. $u$ is the `movement amount` and $CB$ equals to $f(x') - f(x)$ is $\delta$. From trigonometry we know that

$$\tan\alpha = \frac{\delta}{\texttt{movementAmount}}, \tag{3.7}$$

so

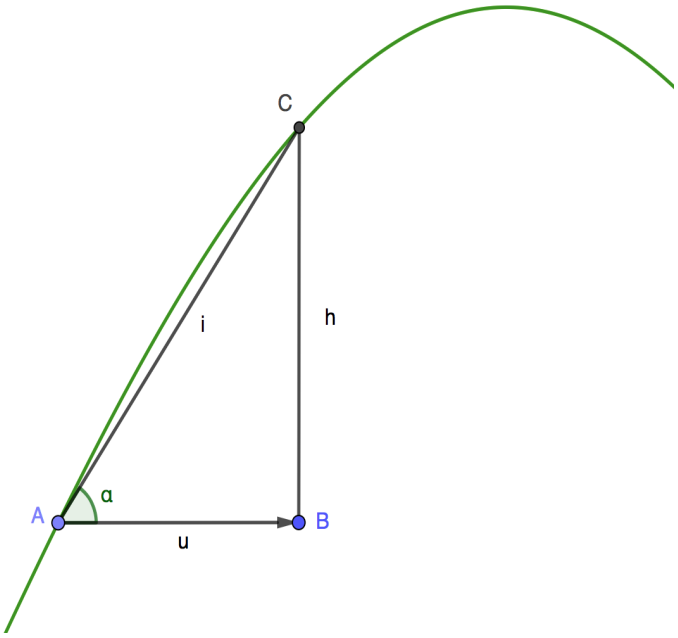


Figure 3.3: Linear movement computations.

$$\alpha = \arctan \frac{\delta}{movementAmount} \tag{3.8}$$

If the movement is parametric we want to better approximate the amount of movement over the function. We do this using the following algorithm :

We can easily explain this algorithm looking at figure 3.4.

As already previously done, let's suppose that the starting point of our RL agent at epoch $e$ is the point $A$ with coordinates $(x, y)$. Because of the two dimensions, even in this case the agent can make only one of two actions for each epoch : *move on* and *go back*. Making a linear movement of the type *move on* the arriving point at epoch *e'* is $B$ with coordinates $(x', y)$. Knowing that the objective function is $f(x) = \sin(2x)$, we can now compute $f(x')$. We now know point $C$ with coordinates $(x', f(x'))$. $u$ is the `movement amount` and $CB$ equals to $f(x') - f(x)$ is $\delta$. From trigonometry we know $\alpha$ but we are interested in knowing the real amount of movement done by the agent on the function or, at least, its approximation. We want to know how much is $\overline{AB}$. From the first theorem of Euclid we know that *in a right-angled triangle, the square constructed on a cathetus is equivalent to*

**Algorithm 7:** Angle computation in parametric movement case.

1 /* knowing $(x, y)$ */ ;
2 /* knowing $(x', y')$*/ ;
3 /* `movementAmount` $= M$ */ ;
4 /* `currentMax` $= \max f(x_n, y_n)$ */ ;
5
6 $z = f(x, y)$ ;
7 $z' = f(x', y')$;
8
9 $\delta = ((x' - x), (y' - y))$ ;
10 $\alpha = \arctan(\frac{\delta}{\texttt{movementAmount}})$ ;
11
12 $hypotenuse = \frac{\texttt{movementAmount}}{\cos \alpha}$ ;
13
14 $projectionOnHypotenuse = \frac{\texttt{movementAmount} * \texttt{movementAmount}}{hypotenuse}$ ;
15
16 $realMovementAmount = projectionOnHypotenuse * \cos \alpha$ ;
17
18 $\Delta$ =z'−`currentMax`;

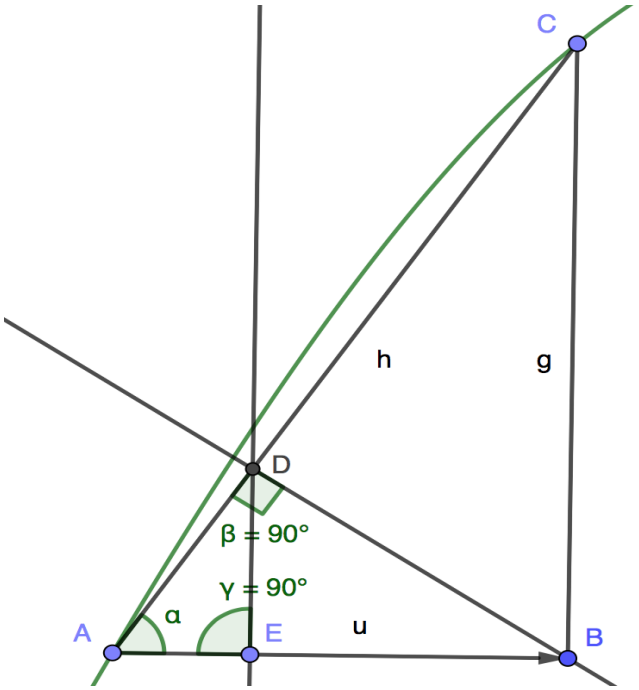


Figure 3.4: Parametric movement computations.

*the rectangle that has for dimensions the hypotenuse and the projection of the cathetus on the hypotenuse.* This means that in a right-angled triangle, the cathetus is proportional medium between the hypotenuse and its own projection on it. According to this we can write the following proportion :

$$h : u = u : AD. \tag{3.9}$$

So :

$$AD = \frac{u * u}{h}, \tag{3.10}$$

that is the same of :

$$pojectionOnHypotenuse = \frac{\texttt{movementAmount} * \texttt{movementAmount}}{hypotenuse} \tag{3.11}$$

of algorithm 7. Now we have all elements in order to compute the *real movement amount*. It is simply $\overline{AE}$ segment and it is computed as :

$$\overline{AE} = \overline{DA}\cos\alpha \tag{3.12}$$

The real question regards why select one method instead of the other one. The answer at this question is simple. Adopting the parametric approach the optimization process is longer but more accurate. In addition to this using the parametric approach we could ideally train our agent on a set of specific functions and it should do very good on a generic function using its knowledge about angles, actions and slope.

Adopting the linear approach the optimization process is slower but less accurate. In addition to this using the linear approach we couldn't abstract from the concept of function. This means that for every specific function we have to train our agent one more time.

### 3.0.1 Implementation

In order to formalize and solve our RL problem, we used the BURLAP (Brown-UMBC Reinforcement Learning and Planning) Java library developed and maintained by James MacGlashan. BURLAP uses a highly flexible system for defining states and and actions of nearly any kind of form, supporting discrete continuous, and relational domains. Planning and learning algorithms range from classic forward search planning to value function-based stochastic planning and learning algorithms [BUR].

In order to define specific MDPs, BURLAP offers a set of classes and interfaces (figure 3.5).

The main features offered by BURLAP are :

- **State** : implementing this interface we define the state variables of our MDP state space. An instance of this object will specify a single state from the state space [BUR] .
- **Action** : implementing this interface we define a possible action that the agent can select. If our MDP action set is discrete and unparametrized, we may consider using the provided concrete implementation `SimpleAction`, which defines an action entirely by a single string name [BUR].
- **SampleModel** : implementing this interface we define the model of our MDP. This interface only requires us to implement methods that can sample a transition: spit back out a possible next state and reward given a prior state and action taken [BUR].

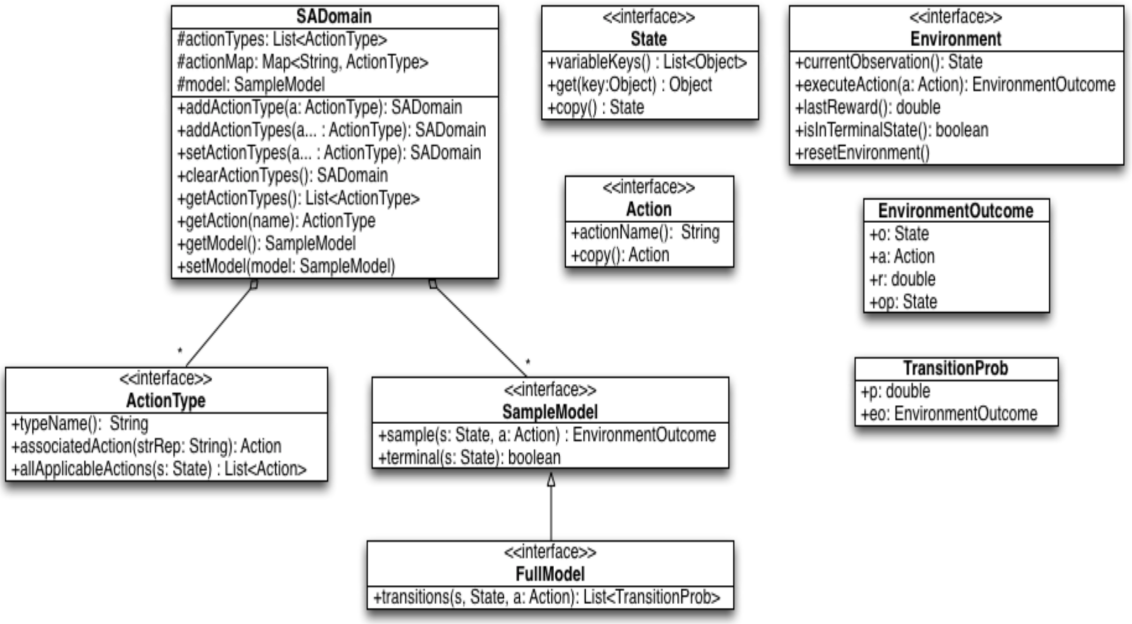


Figure 3.5: UML Digram of the Java interfaces/classes for an MDP definition.

- **Environment** : An MDP defines the nature of an environment, but ultimately, an agent will want to interact with an actual environment, either through learning or to execute a policy it computed from planning for the MDP. An environment has a specific state of the world that the agent can only modify by using the MDP actions. Implement this interface to provide an environment with which BURLAP agents can interact. If we define the MDP ourself, then we'll probably don't want to implement `Environment` ourself and instead use the provided concrete `SimulatedEnvironment` class, which takes an `SADomain` with a `SampleModel`, and simulates an environment for it [BUR].

An extended definition of classes and interfaces can be found at `http://burlap.cs.brown.edu/doc/index.html`.

Starting from features offered by BURLAP, we have extended interfaces and implemented abstract classes in order to modelling the problem described above as shown in figure 3.6. Because of the previous detailed explanation of choices made to model the problem, a deepened analysis of

architectural definition is left to the reader.

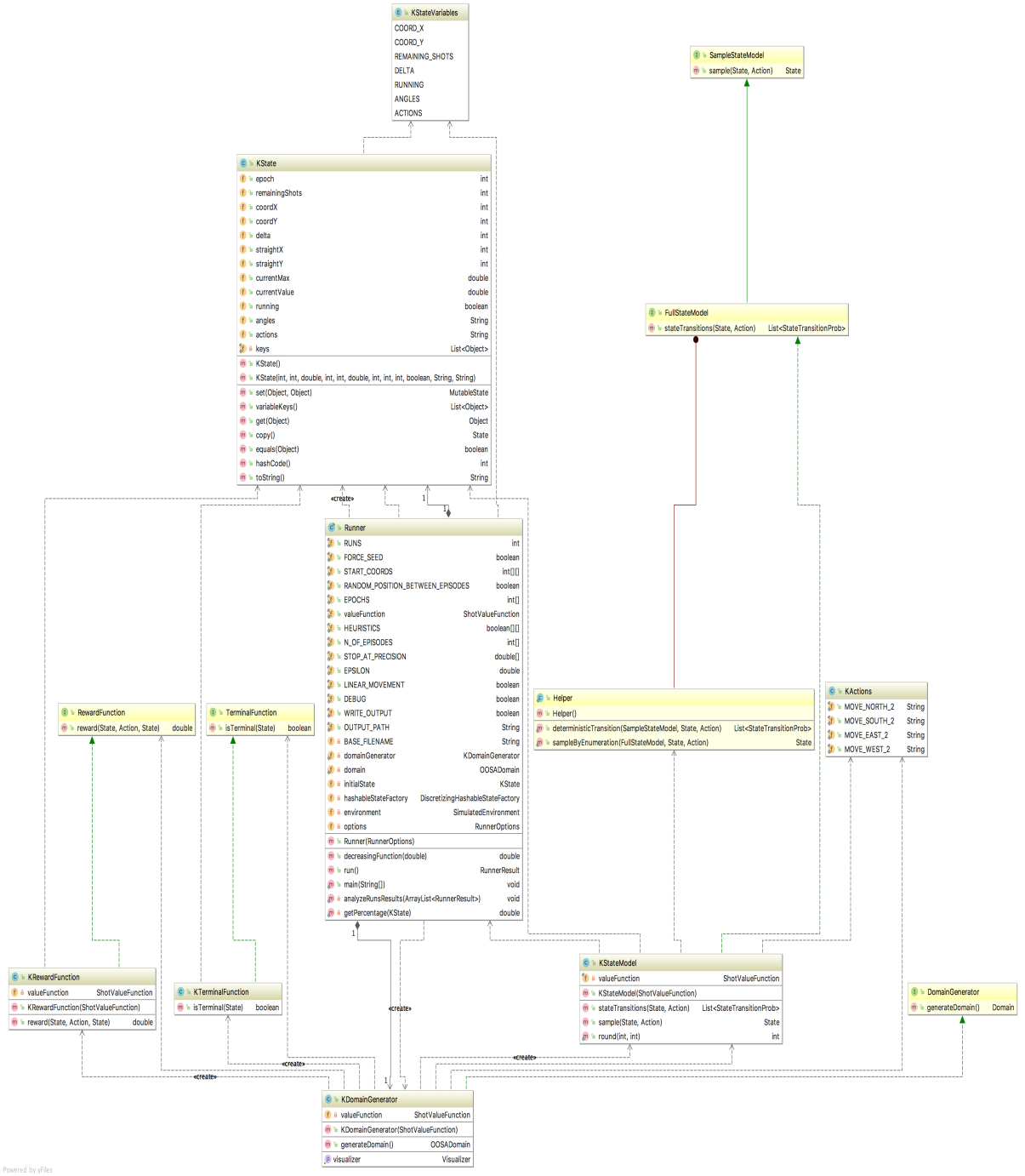


Figure 3.6: Class diagram of RL black-box optimization tool.

# Chapter 4

# Experimental Setting

The current chapter is divided into two sections. In the first one we will describe the benchmark of the experiment regarding the creation of the RL agent and its results. In the second one we will describe the same experiment applied to humans and its results.

## 4.1 Benchmark

**Test Functions** As already explained, the experiment described in this work is about maximize black-box functions adopting an RL based approach. The first important choice is about selecting suitable *test functions*. In applied mathematics, test functions, also known as *artificial landscapes*, are useful to evaluate characteristics of optimization algorithms. We have chosen four test functions for this work:

- Himmelblau' s Function;
- Paraboloid of Revolution;
- Beale Function;
- Styblinski-Tang' s Revised Function.

**Himmelblau' s Function** In mathematical optimization, Himmelblau' s function is a continuous, bivariate, multi-modal function introduced by David Mautner Himmelblau (1924–2011). The original function is defined by:

$$f(x, y) = (x^2 + y - 11)^2 + (x + y^2 - 7)^2 \tag{4.1}$$

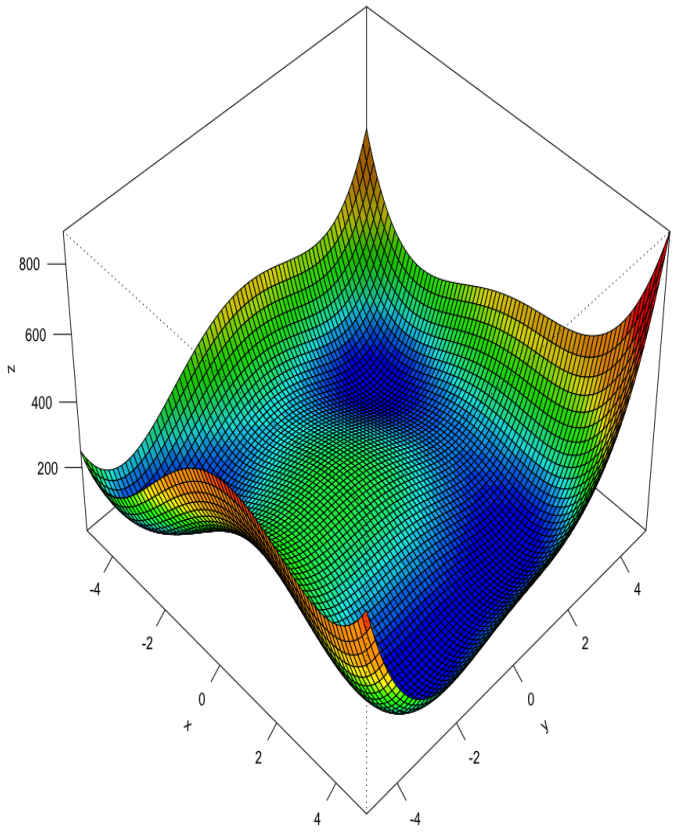


Figure 4.1: Original Himmelblau' s Function.

It has four local minima :

- $f(3.0, 2.0) = 0.0$;
- $f(-2.805118, 3.131312) = 0.0$;
- $f(-3.779310, -3.283186) = 0.0$;
- $f(3.584428, -1848126) = 0.0$.

The function can be defined on any input domain but it is usually evaluated on $x \in [-5, 5]$ and $y \in [-5, 5]$.

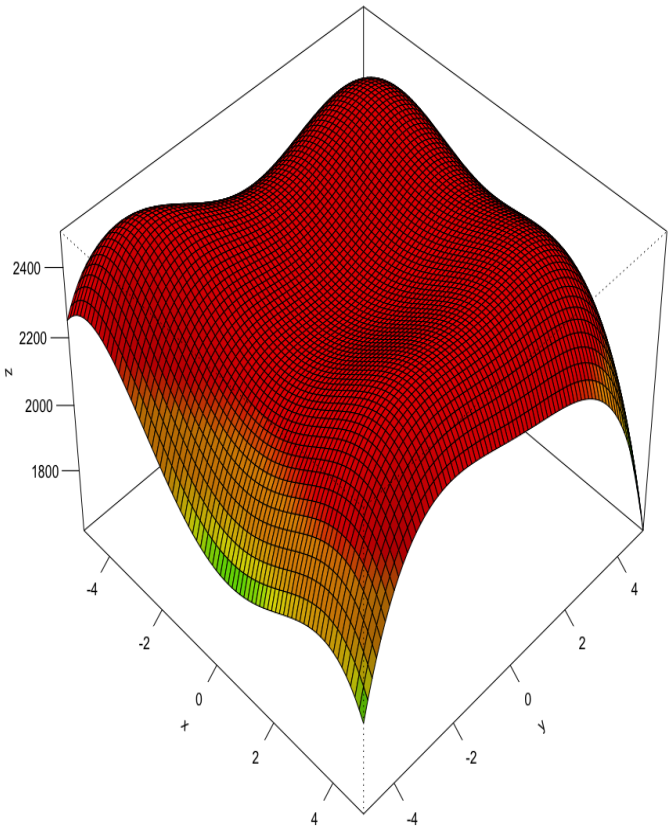


Figure 4.2: Customized Himmelblau' s Function.

Because of our aim to maximize we inverted the function as follow :

$$f(x, y) = -(x^2 + y - 11)^2 + (x + y^2 - 7)^2 \tag{4.2}$$

and we picked it up of 2500 units in order to have as less as possible negative values. So the final adopted function is :

$$f(x, y) = -(x^2 + y - 11)^2 + (x + y^2 - 7)^2 + 2500 \tag{4.3}$$

This function has its global maximum in $f(x, y) = 2500$.

In order to represent this customized version of Himmelblau' s Function using Java Graphical Environment, we mapped it in a space of $600 \times 600$ pixels and we properly rotated it. The resulting contour plot is the one represented in figure 4.3.

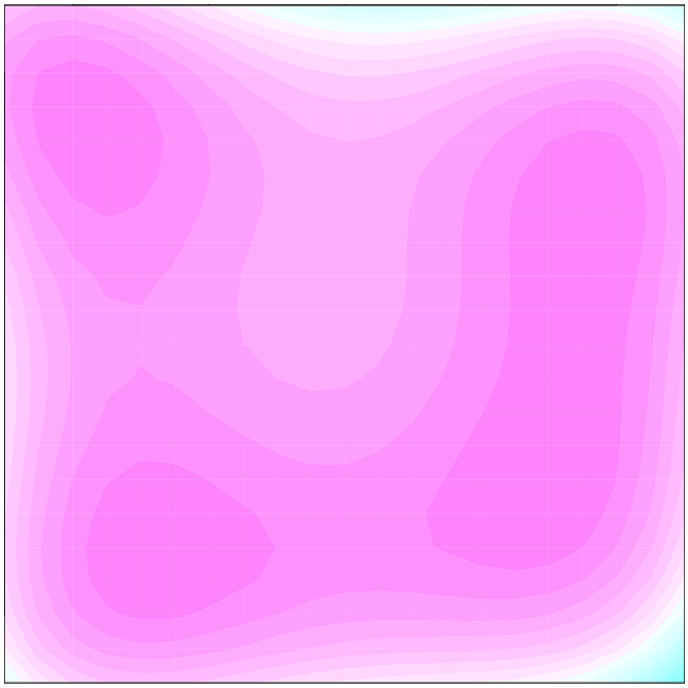

Figure 4.3: Contour plot of customized version of Himmelblau' s Function.

**Paraboloid of Revolution** In mathematical optimization, Paraboloid of Revolution is a continuous, bivariate, multi-modal function.

The original function is defined by:

$$f(x, y) = (x^2 + y^2) \tag{4.4}$$

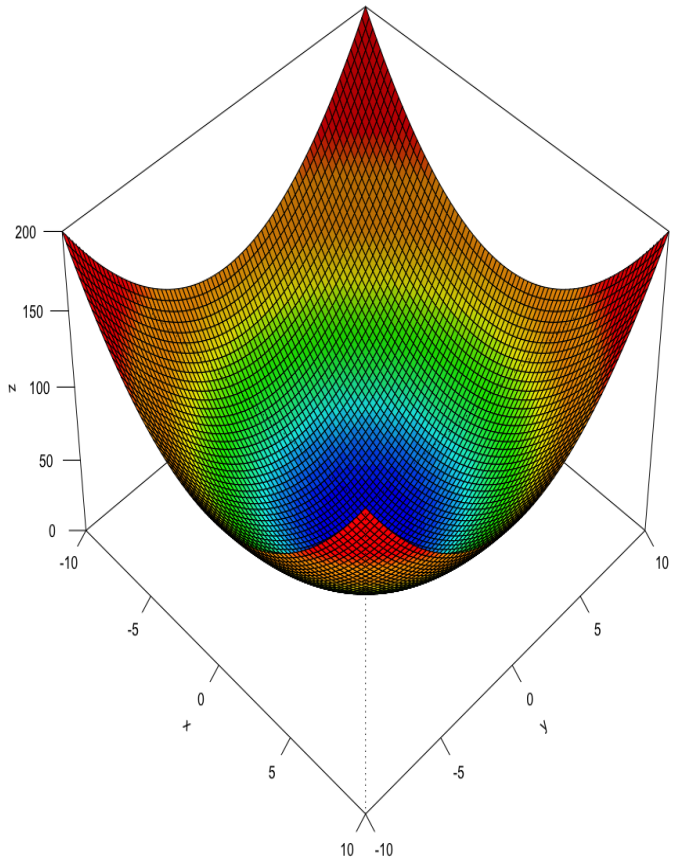


Figure 4.4: Original Paraboloid of Revolution.

It has a global minimum in $f(x, y) = 0$.

The function can be defined on any input domain but it is usually evaluated on $x \in [-10, 10]$ and $y \in [-10, 10]$.

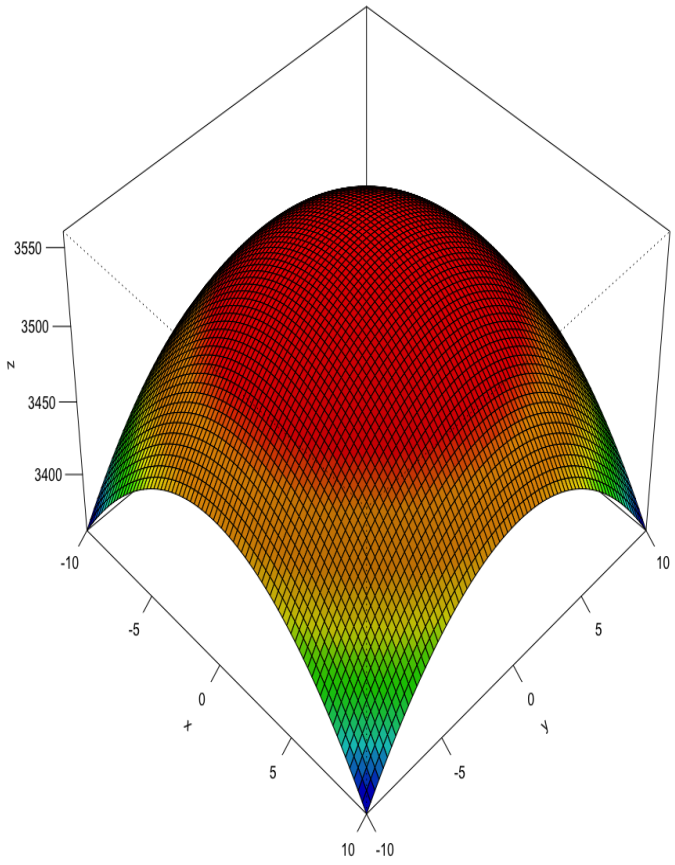


Figure 4.5: Customized Paraboloid of Revolution.

Because of our aim to maximize, we inverted the function as follow :

$$f(x, y) = -(x^2 + y^2) \tag{4.5}$$

and we picked it up of 3560 units in order to have as less as possible negative values. So the final adopted function is :

$$f(x, y) = -(x^2 + y^2) + 3560 \tag{4.6}$$

This function has its local maximum in $f(x, y) = 3560$.

In order to represent this customized version of Paraboloid of Revolution function using Java Graphical Environment, we mapped it in a space of $600 \times 600$ pixels and we properly rotated it. The resulting contour plot is the one represented in figure 4.6

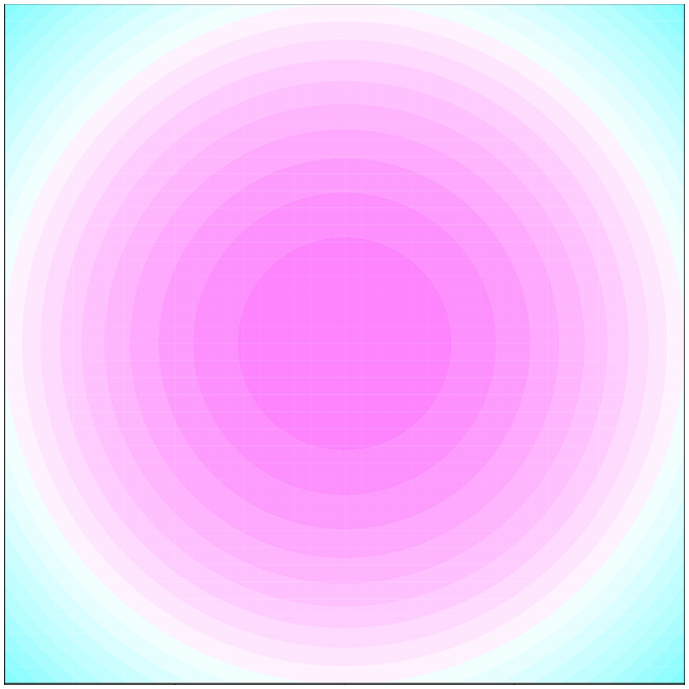

Figure 4.6: Contour plot of customized Parabolic Function.

**Beale Function** In mathematical optimization, Beale Function is a continuous, multi-modal function defined on a two-dimensional space. The function is defined by:

$$f(x, y) = (1.5 - x + xy)^2 + (2.25 - x + xy^2)^2 + (2.625 - x + xy^3)^2 \quad (4.7)$$

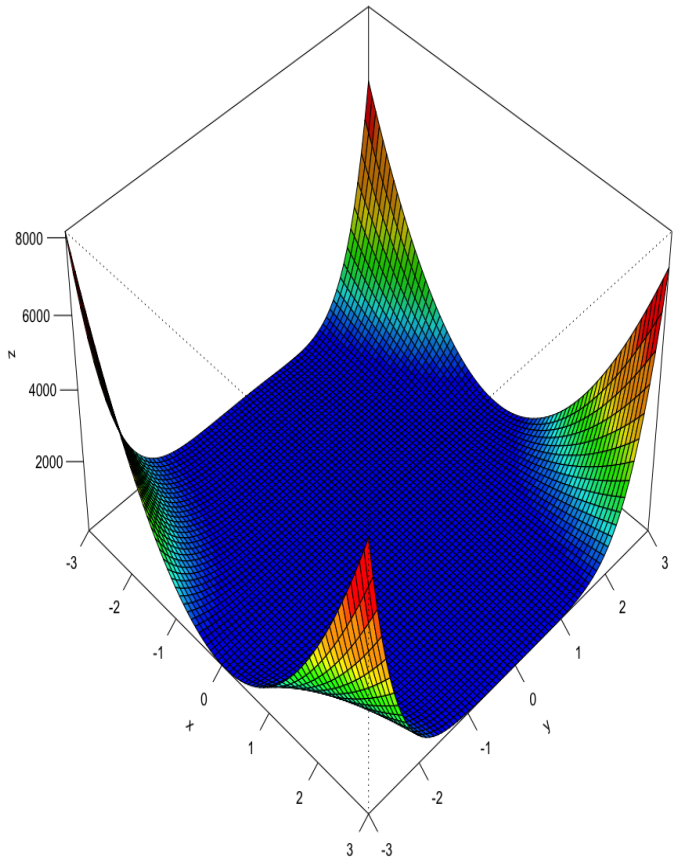


Figure 4.7: Original Beale Function.

The function can be defined on any input domain but it is usually evaluated on $x \in [-3, 3]$ and $y \in [-3, 3]$.

It has one global minimum at: $f(x, y) = 0$.

In this thesis our aim is to maximize. In order to do this we inverted the function as

$$f(x,y) = -((1.5 - x + xy)^2 + (2.25 - x + xy^2)^2 + (2.625 - x + xy^3)^2) \quad (4.8)$$

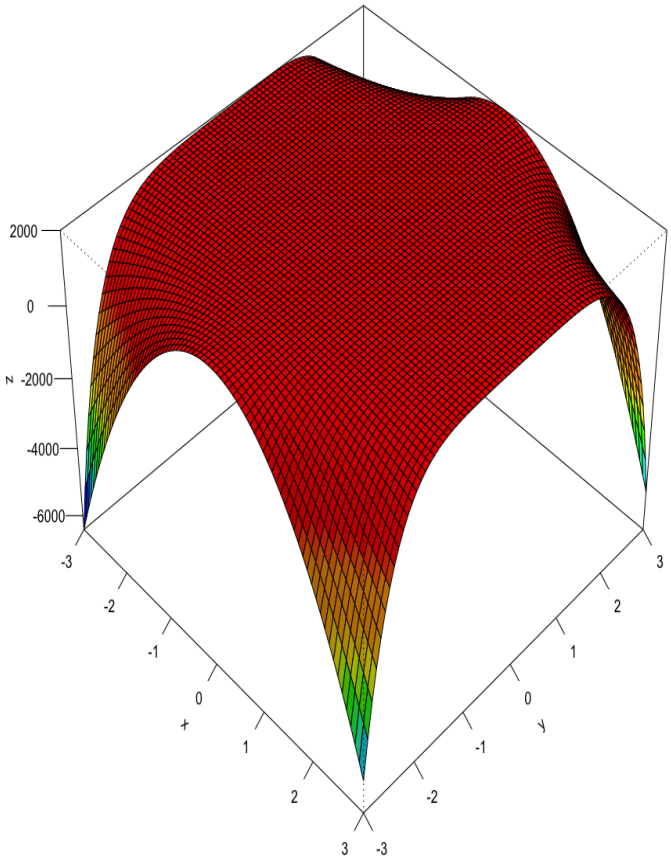


Figure 4.8: Customized Beale Function.

and we picked it up of 2000 units in order to have as less as possible negative values. So the final adopted function is :

$$f(x,y) = -((1.5-x+xy)^2+(2.25-x+xy^2)^2+(2.625-x+xy^3)^2)+2000 \quad (4.9)$$

This customized function has its global maximum in $f(x, y) = 1000$.

In order to represent this function using Java Graphical Environment we mapped it in a space of $600 \times 600$ pixels and we properly rotated it. The resulting contour plot is the one represented in figure 4.9

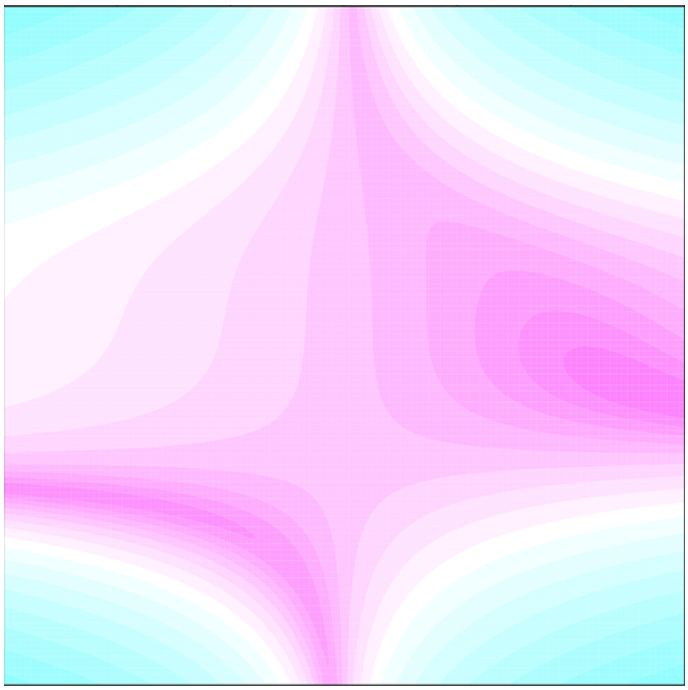

Figure 4.9: Contour plot of customized Beale Function.

**Styblinski-Tang Revised Function** In mathematical optimization, Styblinski-Tang Function is a continuous, multi-modal function defined on a multi-dimensional space. The original function is defined by :

$$f(x) = \frac{\sum_{i=1}^{n} x_i^4 - 16x_i^2 + 5x_i}{2} \tag{4.10}$$

In this thesis we consider the bivariate version of the original function multiplied by two in order to make it less flattened :

$$f(x, y) = (x^4 - 16x^2 + 5x) + (y^4 - 16y^2 + 5y). \tag{4.11}$$

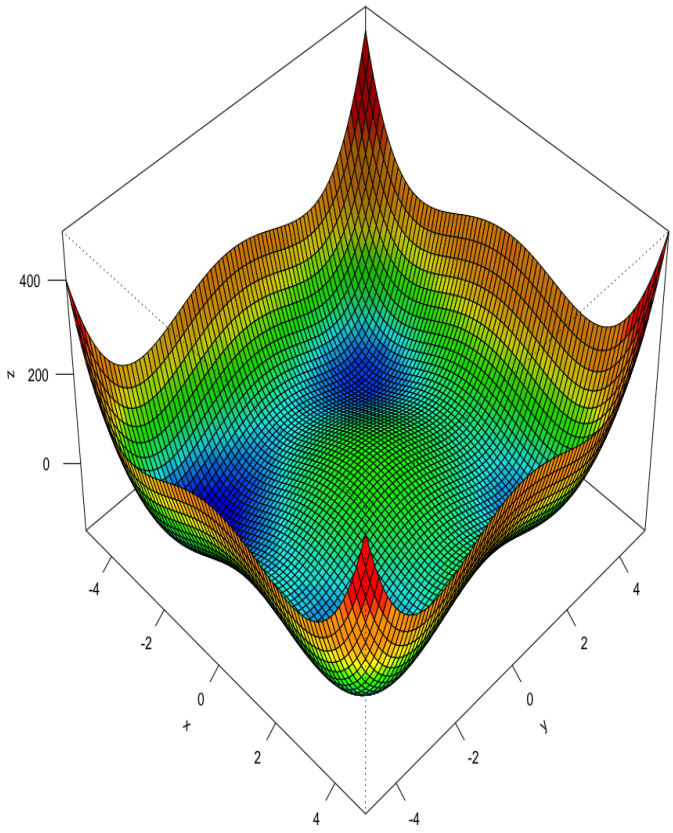


Figure 4.10: Original Styblinski Function.

The function can be defined on any input domain but it is usually evaluated on $x \in [-5, 5]$ and $y \in [-5, 5]$.

In this thesis our aim is to maximize. In order to do this we inverted the function as

$$f(x, y) = -((x^4 - 16 * x^2 + 5 * x) + (y^4 - 16 * y^2 + 5 * y)) \tag{4.12}$$

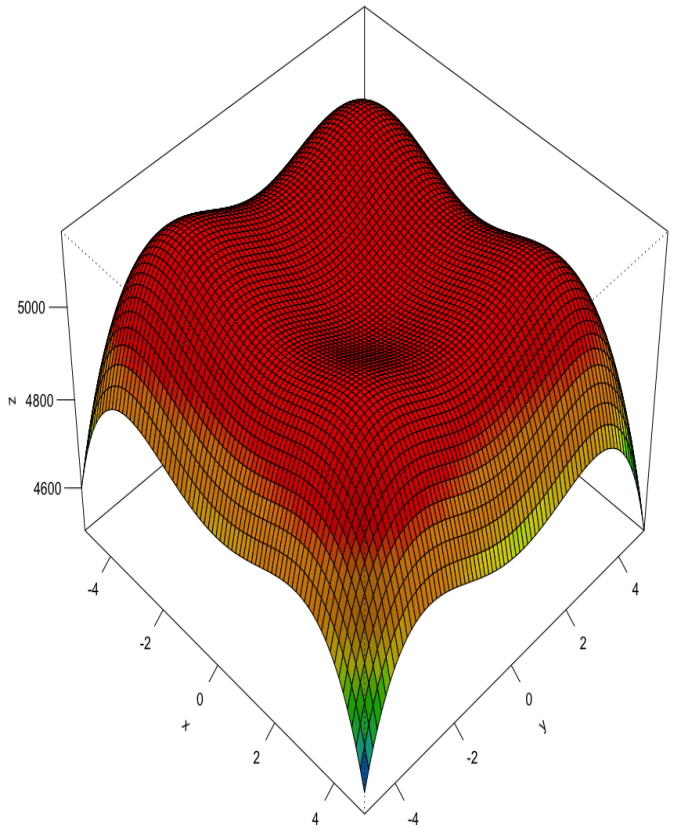


Figure 4.11: Customized Styblinski Function.

and we picked it up of 5000 units in order to has as less as possible negative values. So the final adopted function is:

$$f(x,y) = -((x^4 - 16 * x^2 + 5 * x) + (y^4 - 16 * y^2 + 5 * y)) + 5000 \quad (4.13)$$

This customized function has its global maximum in $f(x,y) = 5156.6638$.

In order to represent this function using Java Graphical Environment we mapped its in a space of $600 \times 600$ pixels and we properly rotated its. The

resulting contour plot is the one represented in figure 4.12.

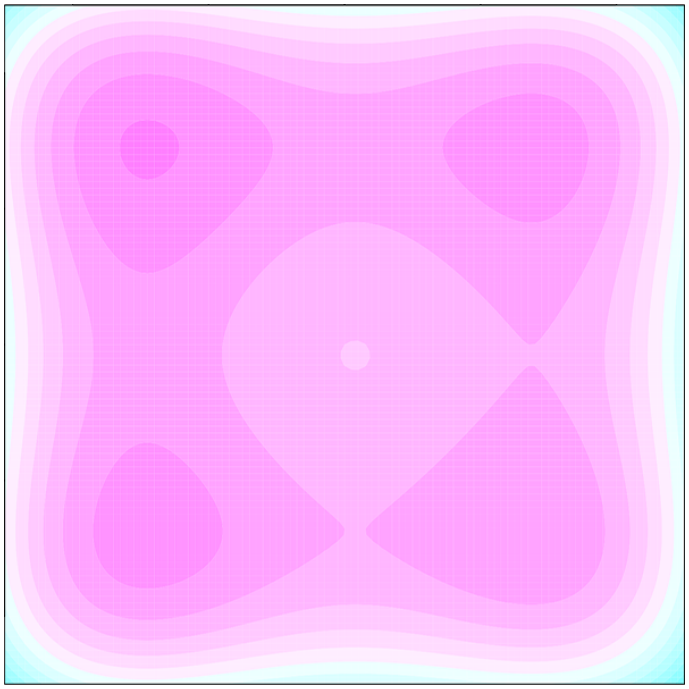

Figure 4.12: Contour plot of customized Styblinski Function.

# Bibliography


[AS08] Ryan P. Adams and Oliver Stegle. Gaussian process product models for nonparametric nonstationarity. In *ICML*, 2008.

[Bay] `https://towardsdatascience.com/shallow-understanding-on-bayesian-optimization-324b6c1f7083`. Accessed: 2018-05-20.

[BUR] `http://burlap.cs.brown.edu/tutorials/bd/p2.html`. Accessed: 2018-05-20.

[HL01] Frederick S. Hillier and Gerald J. Lieberman. *Introduction to Operations Research*. McGraw-Hill, New York, NY, USA, seventh edition, 2001.

[KLM96] Leslie Pack Kaelbling, Michael L. Littman, and Andrew W. Moore. Reinforcement learning: A survey. *CoRR*, cs.AI/9605103, 1996.

[Kon09] Takis Konstantopoulos. Markov chains and random walks. 2009.

[LM16] Ke Li and Jitendra Malik. Learning to optimize. *CoRR*, abs/1606.01885, 2016.

[Mit97] Tom M. Mitchell. *Machine Learning*. 1997.

[MND] `https://en.wikipedia.org/wiki/Multivariate_normal_distribution/media/File:Multivariate_Gaussian.png`. Accessed: 2018-05-20.

[NFK06] Yuriy Nevmyvaka, Yi Feng, and Michael Kearns. Reinforcement learning for optimized trade execution. In *Proceedings of the 23rd International Conference on Machine Learning*, ICML '06, pages 673–680, New York, NY, USA, 2006. ACM.

[Pow07] Warren B. Powell. *Approximate Dynamic Programming: Solving the Curses of Dimensionality (Wiley Series in Probability and Statistics)*. Wiley-Interscience, 2007.

[Put94] M. Puterman. *Markov Decision Processes: Discrete Stochastic Dynamic Programming*. John Wiley and Sons, 1994.

[SB10] Olivier Sigaud and Olivier Buffet. *Markov Decision Processes in Artificial Intelligence*. Wiley-IEEE Press, 2010.

[SB18] R. S. Sutton and A. G. Barto. *Reinforcement Learning : An Introduction*. 2018.

[SLA12] Jasper Snoek, Hugo Larochelle, and Ryan P Adams. Practical bayesian optimization of machine learning algorithms. In F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems 25*, pages 2951–2959. Curran Associates, Inc., 2012.

[SSW+16] Bobak Shahriari, Kevin Swersky, Ziyu Wang, Ryan P. Adams, and Nando de Freitas. Taking the human out of the loop: A review of bayesian optimization. *Proceedings of the IEEE*, 104(1):148–175, 2016.

[WD92] Christopher JCH Watkins and Peter Dayan. Q-learning. *Machine learning*, 8(3-4):279–292, 1992.

[WvO12] M. Wiering and M. van Otterlo. *Reinforcement Learning: State-of-the-Art*. Adaptation, Learning, and Optimization. Springer Berlin Heidelberg, 2012.