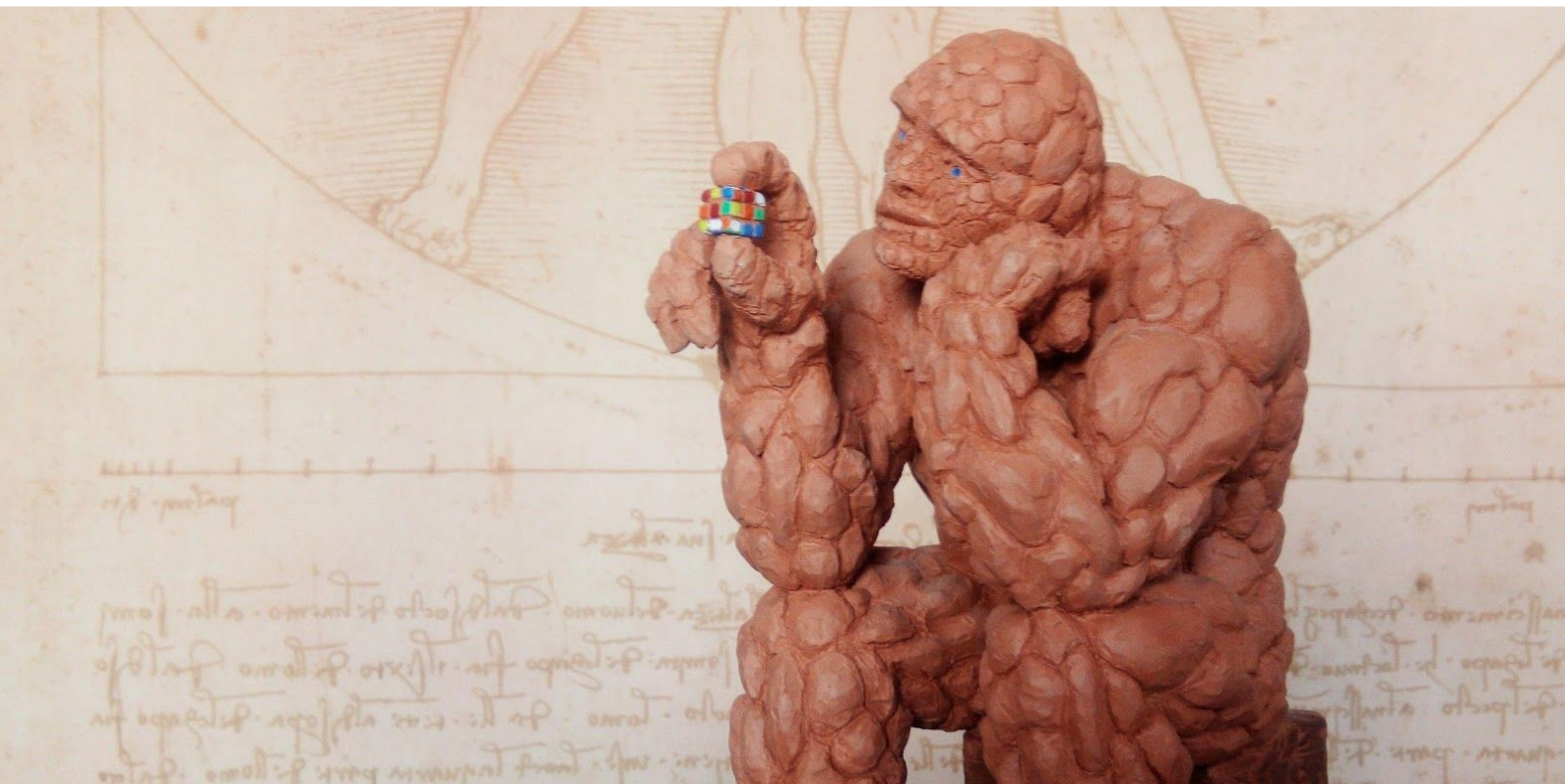


# Fractal AI

## A Fragile Theory of Intelligence

Sergio Hernández Cerezo  
Guillem Duran Ballester



# Fr{AGI}le

## BOOK #1

### “Forward Thinking”

Version: V1.0



**Main researchers**

Sergio Hernández Cerezo ([@EntropyFarmer](#))

Guillem Duran Ballester ([@Miau\\_DB](#))

**Reviewers**

Eiso Kant ([@EisoKant](#)), CEO at [source\[d\]](#)

José María Amigó García ([Elche University](#))

Roshawn Terrell ([@RoshawnTerrell](#))

Juan G. Cruz Ayoroa

Jesús P. Nieto ([@HedgeFair](#))

Aidan Rocke ([@AidanRocke](#))

**Special thanks**

Researchers' families for suffering us in all the 'eureka' moments.

[HCSoft](#) and [Source\[d\]](#) for their unconditional support.

## Contents

---

<b>1 - Introduction</b>	<b>5</b>
1.1 - The playground of intelligence	5
1.1.1 - Cart-pole example	6
1.1.2 - General strategy	6
1.1.2.1 - Forward vs Backward intelligence	7
1.1.2.2 - Scoring actions	7
<b>2 - Fundamental concepts</b>	<b>9</b>
2.1 - Causal Cones	9
2.1.1 - Causal Slices	10
2.1.2 - Conditional Causal Cones	10
2.2 - Reward function	11
2.2.1 - Dead vs Alive states	11
2.2.2 - Reward function properties	12
2.2.3 - Reward density over Causal Slices	12
2.3 - Policies: defining strategies	12
2.3.1 - Scanning policy	13
2.3.2 - Deciding policy	13
2.3.3 - Probability density due to policy over Causal Slices	14
2.4 - Divergence between distributions	14
<b>3 - Defining Intelligence</b>	<b>16</b>
3.1 - Scanning process	16
3.1.1 - Causal Slice divergence	16
3.1.2 - Intelligent Scanning	16
3.1.3 - Scanning sub-optimality coefficient	17
3.2 - Decision process	18
3.2.1 - Intelligent decision	18
3.2.2 - Decision sub-optimality coefficient	19
3.3 - Global sub-optimality	19
3.4 - Policy IQ	19
<b>4 - Fractal AI algorithm</b>	<b>20</b>
4.1 - Algorithm blueprints	21
4.1.1 - Starting at Monte Carlo	21
4.1.2 - Choosing the intelligence parameters	22
4.1.2.1 - Decisions per second	22
4.1.2.2 - Time horizon	23
4.1.1.3 - Number of walkers	23
4.1.3 - Simultaneous walks	23

4.1.4 - Probability densities	25
4.1.4.1 - Density of walkers	25
4.1.4.2 - Reward density	26
4.1.5 - Migratory flows	27
4.1.6 - Taking the decision	29
4.2 - The migratory process	30
4.2.1 - Virtual reward	31
4.2.2 - Simplifying the Virtual Reward	32
4.2.3 - Balancing exploitation and exploration	34
4.2.3.1 - The “Common Sense”	35
4.2.4 - Probability of cloning	35
4.3 - Pseudo-code	36
4.4 - Classifying the algorithm	37
4.4.1 - Monte Carlo Tree Search	37
4.4.2 - Swarm algorithm	38
4.4.3 - Evolutive algorithm	38
4.4.4 - Entropic algorithm	39
4.4.5 - Fractal algorithm	39
<b>5 - Experiments</b>	<b>40</b>
5.1 - Discrete case: Atari 2600 games	40
5.1.1 - RAM vs Images	40
5.1.2 - Results	40
5.1.2.1 - MCTS vs Fractal AI	42
5.1.2.2 - Human record vs Fractal AI	42
5.1.3 - Implementation details	43
5.1.3.1 - Auto-adjusting ‘Samples per step’	43
5.1.3.2 - Additional death condition	43
5.1.4 - Github repository	44
5.2 - Continuous case: Flying rocket	44
5.2.1 - Flying a chaotic attractor	44
5.2.2 - Results	45
5.2.3 - Implementation details	46
5.2.3.1 - Reward	46
5.2.3.2 - Distance	47
<b>6 - Research topics</b>	<b>48</b>
6.1 - Distributed computing	48
6.2 - Adding learning capabilities	48
6.2.1 - Using a DQN for learning	50
6.3 - Common Sense Assisted Control	50
6.4 - Consciousness	50
6.5 - Real-time decision-making	51

6.6 - Universality pattern	51
<b>7 - Conclusions</b>	<b>53</b>
<b>Bibliography</b>	<b>54</b>
ATARI experiment references	55

# 1 - Introduction

---

*“For instance, on the planet Earth, man had always assumed that he was more intelligent than dolphins because he had achieved so much—the wheel, New York, wars and so on—whilst all the dolphins had ever done was muck about in the water having a good time. But conversely, the dolphins had always believed that they were far more intelligent than man—for precisely the same reasons.”*

---

**Douglas Adams, *The Hitchhiker's Guide to the Galaxy***

One of the big obstacles in the field of artificial intelligence is not having a definition of intelligence based on solid mathematical and physical principles that could inspire the design and implementations of efficient intelligent algorithms.

For instance, consider the most widely accepted definition of intelligence, signed by 52 specialist on the field [2]:

*“A very general mental capability that, among other things, involves the ability to reason, plan, solve problems, think abstractly, comprehend complex ideas, learn quickly and learn from experience. It is not merely book learning, a narrow academic skill, or test-taking smarts. Rather, it reflects a broader and deeper capability for comprehending our surroundings...”*

A more recent definition [3] provided by Shane Legg, chief scientist of Deep Mind, and Marcus Hutter, founder of AIXI, is the following:

*“Intelligence measures an agent's ability to achieve goals in a wide range of environments.”*

Although there are many other definitions of intelligence, they are too fuzzy to help us develop a theory of intelligent behaviour or give us an insight on how a general, computable and efficient algorithm for generating intelligent behaviour should look like.

This document is an effort to present such a definition based on entropic principles deeply inspired by the concept of “Causal Entropic Forces” introduced by Alexander Wissner-Gross in 2013 [1] and to propose a generic implementation of those principles.

## 1.1 - The playground of intelligence

---

*“The most difficult thing is the decision to act, the rest is merely tenacity. The fears are paper tigers. You can do anything you*

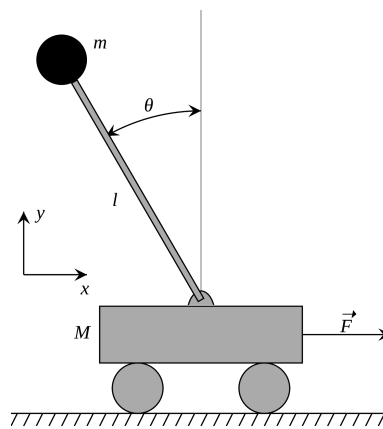
*decide to do. You can act to change and control your life; and  
the procedure, the process is its own reward"*

**Amelia Earhat**

As a first attempt in defining intelligence, we could say that intelligence works by taking decisions that directly affect the degrees of freedom of a system in such a way that its future evolution is biased toward rewarding futures.

### 1.1.1 - Cart-pole example

Let's suppose we are controlling a cart-pole that can move left or right by pushing one of the two available buttons. Our goal is to keep the pole standing up, so the ball at the tip of the cart-pole must be as high as possible.



In that case, the intelligence can affect the system evolution by pressing one of the two buttons at any time. The ultimate goal of the intelligence is then to continuously choose the actions that keeps the ball as high as possible.

In this example, the action-space is discrete, but it could also be continuous: instead of having two buttons to choose from, we may have a "joystick" we can push, so our available actions are real numbers in the range  $[-1, 1]$ . The simulation of the system can also be more or less deterministic, and the goals could be a combination of several sub-goals. None of this would change the problem except that they would require more computation.

### 1.1.2 - General strategy

When the intelligence is asked to choose between a discrete set of actions  $\{a_i\}$  it will internally score them accordingly to some metrics and then output an "intelligent decision" as being the action with the highest score or, in the continuous case, the average of a number of actions weighted by their normalised scores.

$$\text{Decision} = \sum(a_i * \text{Score}(a_i)) / \sum(\text{Score}(a_i))$$

### 1.1.2.1 - Forward vs Backward intelligence

---

*"You can know the past, but not control it. You can control the future, but have not knowledge of it."*

---

**Claude Shannon**

There are two main strategies used in an intelligent decision making process:

On one hand, we can use information from past events, along with the decisions that were taken and their corresponding outcomes, and eventually learn from that information, to influence future decisions. We will refer to this as 'backward-thinking', as we will only base our decisions on events from the past.

Such a backward-thinking process could in fact learn to predict the best action given an initial state, but it could also learn to predict the next state of the system, as it has access to pairs of initial states, actions, and final states. Predicting the final state from the initial one is equivalent to being able to internally simulate the system for relatively long periods by simulating one state after the other in small jumps.

At some point, evolution began to develop agents that were able to project their actual state into the future with relative accuracy. Enabling it to ponder about its available actions in terms not only of its past experiences, but by predicting the foreseeable future each action leads to and its consequences.

This 'forward-thinking' process will make better decisions as the simulation gets better, as opposed to learning-based strategies of back-thinking that depend on the accumulation of past experiences.

Both strategies are complementary. You need to learn how to simulate the system before you can start thinking forward, while forward-thinking can be used to detect and develop better decisions for situations where only repeating past strategies, is no longer viable.

This document will focus on forward-thinking, or the ability to make near-perfect intelligent decisions based on a near-perfect simulation of the system without the need of any previous learning. This is present in the human cognitive processes. But in contrast, is absent in current AI methods.

### 1.1.2.2 - Scoring actions

---

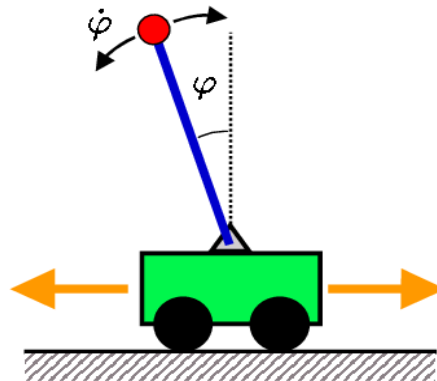


*"No sensible decision can be made any longer without taking into account not only the world as it is, but the world as it will be."*

---

Isaac Asimov

So what is the basic idea behind "scoring" an action? Imagine the cart holding the pole is in the situation shown in the image:



If we push the cart to the right, the pole will fall and the red ball will be at the lowest possible position, a single possible future state having a reward of zero. However, if we instead push it to the left, the pole will recover its up position, not only maximising the reward but also providing access to a greater number of future states.

The right decision here is then pushing to the left for two separated reasons:

1. It leads to a greater diversity of available future states.
2. It leads to future states with higher reward.

The process of scoring options needs then to be guided by a search for more possible future states, usually called 'exploration', while also taking into account how rewarding such future states are, i.e. 'exploitation'. Reaching and maintaining this fragile balance is the key idea behind any intelligent process.

We have nailed down the actual problem of intelligence to finding a way to scan the space of future states in such a way that exploration and exploitation are balanced during the process, and then make a decision about our next action based on the findings.

## 2 - Fundamental concepts

---

*"By far, the greatest danger of Artificial Intelligence is that people conclude too early that they understand it."*

---

**Eliezer Yudkowsky**

Building a detailed theory of intelligence based on these ideas requires fleshing out some fundamental concepts: the shape of this 'space of future states', what 'scanning' will mean to us, what 'balancing exploration and exploitation' means, and finally, how can we use information derived from the scanning process to make intelligent decisions over the available actions.

### 2.1 - Causal Cones

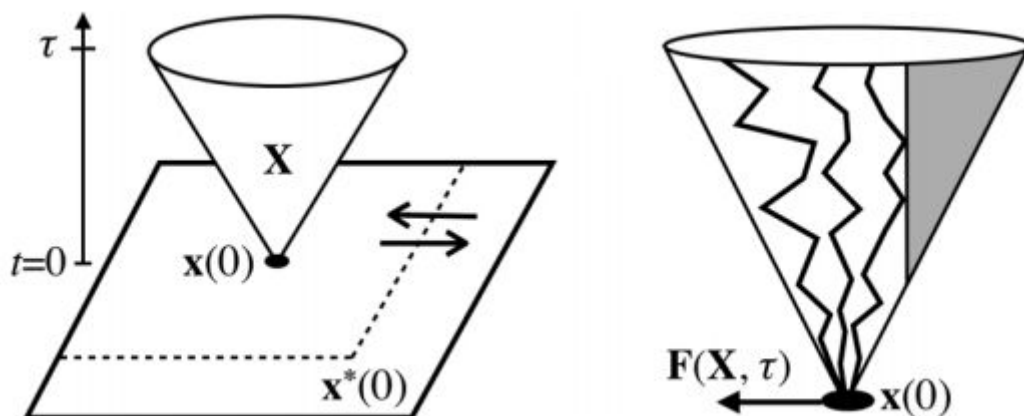
---

*"What is required is that the mind be prepared in advance, and be already stepping from thought to thought, so that it will not be too much held up when the path becomes slippery and treacherous."*

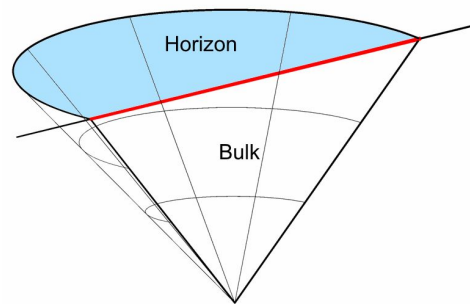
---

**Leibniz, on Rational Decision-Making**

In order to understand the 'space of future states' an intelligence will need to scan, we define a Causal Cone  $X(x_0, \tau)$  as the set of all the paths the system can take starting from an initial state  $x_0$  if allowed to evolve over a time interval of length  $\tau$ , the 'time horizon' of the cone.



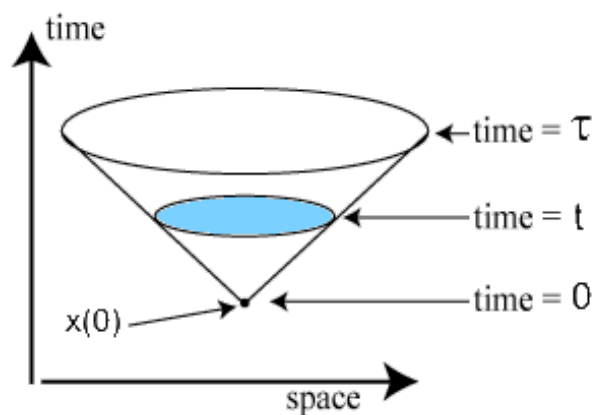
Causal cones are usually divided into two parts: the cone's 'horizon', formed by the final states ( $t = \tau$ ) for all the possible paths, and the rest of the cone ( $t < \tau$ ) usually referred to as the cone's 'bulk'.



### 2.1.1 - Causal Slices

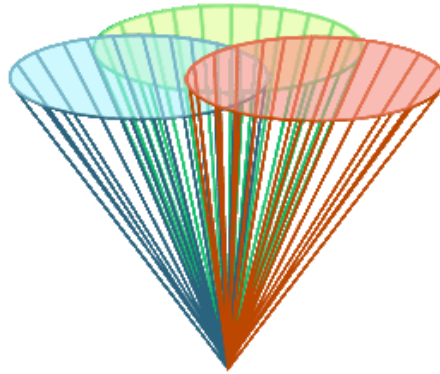
A Causal Slice of the cone  $X(x_0, \tau)$  at time  $t \in [0, \tau]$  is the horizon of the cone  $X(x_0, t)$  which may be denoted by  $X_H(x_0, t)$ . Meanwhile, it is important to note that causal slices consist of a set of states, unlike causal cones that are formed by full paths.

We can imagine these causal slices as being the set of all future states the system can evolve to in a given time  $t$ , starting from  $x_0$ .



### 2.1.2 - Conditional Causal Cones

Given that the initial state  $x_0$  contains specific information concerning the system's degrees of freedom, and given that the intelligence can alter those values by taking an initial action, all the paths forming the Causal Cone can then be partitioned based on this initial action taken.



We define the Conditional Causal Cone associated with an action  $a \in A$ ,  $X(x_0, \tau|a)$ , as the cone formed by all the paths that start by taking the option  $a$ . The conditional cones are then a partition of the original cone:

$$X(x_0, \tau) = \bigcup_{a \in A} X(x_0, \tau|a)$$

## 2.2 - Reward function

---

*"To employ the art of consequences, we need an art of bringing things to mind, another of estimating probabilities and, in addition, knowledge of how to evaluate goods and ills."*

---

**Leibniz, on Rational Decision-Making**

In our setup, we will assume a reward function  $R(x)$  is defined over the state space. This reward function must follow some basic rules in order to be useful to the intelligence.

### 2.2.1 - Dead vs Alive states

---

*"The reports of my death have been greatly exaggerated"*

---

Mark Twain

We will say an state of the system is 'alive' from the intelligence standpoint when:

- It is a feasible state, meaning the system dynamics allow it.
- Modifying the degrees of freedom causes the system evolution to be affected.

We will then say the intelligence is 'alive' when it is in a alive state, and 'dead' in the other cases.

## 2.2.2 - Reward function properties

---

*"I came to see that there is a species of mathematics in estimating reasons, where they sometimes have to be added, sometimes multiplied together in order to get the sum. This has not yet been noted by the logicians."*

---

**Leibniz, on Rational Decision-Making**

A function  $R(x)$  defined over the state space of the system can be considered a reward function for an intelligent agent when it meets the following criteria:

1. The reward is positive for all alive states of the system.
2. The reward is zero for all dead states of the system.
3. States with higher rewards are considered better for the agent.

As an example, consider  $R(x)$  as being the battery level of an electric powered remote controlled agent, then:

1. When  $R(x)$  is positive, the batteries are working and the intelligence can take decisions that will affect the evolution of the system.
2. if  $R(x)$  is zero, the agent is out of batteries and any decision the intelligence could take will not affect the evolution of the system.
3. The higher the energy, the better for the agent.

## 2.2.3 - Reward density over Causal Slices

For every slice  $X_H(x_0, t)$  of the causal cone, we can calculate the total reward  $R_{TOT}(x_0, t)$  of the slice as the integral of the reward over the slice. We may then convert the reward into a probability density  $P_R$  over the slice as follows:

$$P_R(x|x_0, t) = R(x) / R_{TOT}(x_0, t)$$

## 2.3 - Policies: defining strategies

---

*"You have brains in your head. You have feet in your shoes. You can steer yourself any direction you choose. You're on your own. And you know what you know. And YOU are the one who'll decide where to go..."*

---

**Dr. Seuss, Oh, The Places You'll Go!**

A policy  $\pi$  is a set of two functions that completely define the strategy an intelligent system would follow when scanning the future consequences of its actions and deciding on the next action to take.

$$\pi = \{\pi_s, \pi_D\}$$

### 2.3.1 - Scanning policy

The scanning policy  $\pi_s$  is a function that, given a state  $x \in E$  of the system, outputs a probability distribution  $P$  over the available actions:

$$\pi_s : E \rightarrow P$$

When considering the probability of choosing a particular action  $a$ , we will use the conditional notation  $\pi_s(a|x)$ .

A special case of scanning policy is the random policy  $\pi_s^{\text{RND}}$  that would assign a uniform distribution over the actions regardless of the state  $x$ , so all actions are equally probable and  $\pi_s^{\text{RND}}(a_i|x) = \pi_s^{\text{RND}}(a_j|x)$ ,  $\forall a_i, a_j \in A$ :

$$\pi_s^{\text{RND}} : E \rightarrow P \text{ with } P \text{ a uniform distribution.}$$

### 2.3.2 - Deciding policy

---

*"It is the mark of a truly intelligent person to be moved by statistics."*

---

**George Bernard Shaw**

A policy must also include a mechanism to score the available actions and assign them a probability of being chosen after the scanning phase is finished. This deciding probability distribution shall be denoted by:

$$\pi_D : A \rightarrow [0, 1]$$

There is also a special case where the decision is taken in a random manner so each action is equiprobable and  $\pi_D^{\text{RND}}(a_i) = \pi_D^{\text{RND}}(a_j)$ ,  $\forall a_i, a_j \in A$ :

$$\pi_D^{\text{RND}} : A \rightarrow [0, 1] \text{ is a uniform distribution.}$$

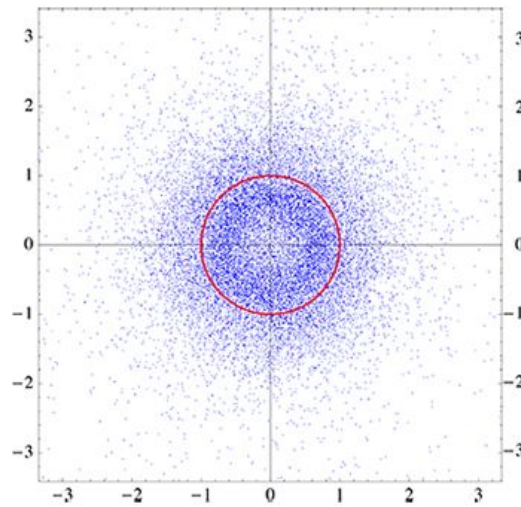
### 2.3.3 - Probability density due to policy over Causal Slices

Once a scanning policy  $\pi_s$  is defined, we may use it to calculate the probability of the system evolving to a given state, as the policy serves as a transition probability which, as in a Markov chain, allows us to project the evolution of the probability density over time.

For instance, in the initial causal slice for  $t = 0$ , the density is concentrated on the initial state  $x_0$ , as the slice  $X_H(x_0, t = 0)$ , only contain this state. However, as time increases, the causal slice volume will grow and the distribution will evolve.

We can then define this 'scanning probability density'  $P_s$  as being a function defined over the slice  $X_H(x_0, t)$ , with parameters  $x_0$ ,  $t$  and  $\pi_s$ :

$$P_s(x|x_0, t, \pi_s)$$



It is important to note that, given a scanning policy  $\pi_s$ , some of the states in a slice could be unreachable. For instance, if a policy chooses the same action all the time with probability 1, then  $P_s(x|x_0, t, \pi_s)$  is not guaranteed to be non zero for all  $x$  in the slice, as it occurs with the random policy  $\pi_s^{\text{RND}}$ .

## 2.4 - Divergence between distributions

We will need to define a reliable measure of how similar two probability distributions are. This is usually done using the Kullback-Leibler divergence as follows:

$$D_{\text{KL}}(p || q) = -\sum(p_i \log(p_i/q_i)) \geq 0$$

This formulation requires  $q_i > 0$  whenever  $p_i > 0$ . In our case, where  $p = P_R(x|x_0, t)$  and  $q_i = P_s(x|x_0, t, \pi_s)$ , this is not guaranteed to be true as we noted before, but we could use Gibbs' theorem to find a divergence formulation which doesn't suffer from this weakness.

**Gibbs' Theorem 1.** Let  $P, Q \in \mathbf{P}_n = \{p \in \mathbb{R}^n : p_i \geq 0, \sum p_i = 1\}$ , then  $\Pi(q_i^{p_i})$  is maximized by  $\Pi(p_i^{p_i})$ .

Using this theorem we may define a different divergence of two distributions as follows:

$$D_H(P \parallel Q) = \text{Log}(\Pi(p_i^{p_i}) / \Pi(q_i^{p_i}))$$

This divergence is well defined for any possible distributions  $p$  and  $q$ , including the problematic case when  $(p_i > 0, q_i = 0)$  and it satisfies the main properties of the KL-divergence we need:

1.  $D_H(P \parallel Q)$  is well defined for any pair of distributions.
2.  $D_H(P \parallel Q) \geq 0$  for any pair of distributions.
3.  $D_H(P \parallel Q) = 0$  if and only if  $p = q$ .



## 3 - Defining Intelligence

---

*"There is always another way to say the same thing that doesn't look at all like the way you said it before. I don't know what the reason for this is. I think it is somehow a representation of the simplicity of nature."*

---

**Richard P. Feynman**

We will define the intelligence as the ability to minimize a 'sub-optimality' coefficient based on the similitude of two pairs of probability distributions obtained during the processes involved: scanning and the decision-making.

### 3.1 - Scanning process

---

*"Intelligence is not to make no mistakes, but quickly to see how to make them good."*

---

**Bertolt Brecht**

In the scanning phase, the possible future outcomes of the initial actions are sampled using the scanning policy  $\pi_s$ .

#### 3.1.1 - Causal Slice divergence

Given a slice of the causal cone  $X_H(x_0, t)$  and a scanning policy  $\pi_s$  we previously defined two probability distributions over it: the scanning density distribution  $P_s(x|x_0, t, \pi_s)$ , and the reward distribution  $P_R(x|x_0, t)$ .

The reward distribution is provided by the environment in an objective manner so the policy can not change it, while the scanning density distribution is directly dependent on  $\pi_s$ , so it makes sense to use the divergence  $D_H(P_R, P_s)$  as a measure of how well the scanning policy leads the agent to rewarding states.

#### 3.1.2 - Intelligent Scanning

---

*"...in order to decide what we ought to do to obtain some good or avoid some harm, it is necessary to consider not only the good or harm in itself, but also the probability that it will or will*

*not occur, and to view geometrically the proportion all these things have when taken together."*

---

**Leibniz, on Rational Decision-Making**

We will define the 'Intelligent Scanning' as the optimal policy  $\pi_s^{\text{OPT}}$  that produces a scanning density distribution that is proportional to the reward for every slice of the causal cone:

$$P_s(x|x_0, t, \pi_s) \propto R(x), \forall x \in X_H(x_0, t)$$

This implies that both probability densities are coincident:

$$P_s(x|x_0, t, \pi_s) = P_R(x|x_0, t), \forall t \in [0, \tau], \forall x \in X_H(x_0, t)$$

Equivalently, we may define  $\pi_s^{\text{OPT}}$  as the policy that makes the causal slide divergence to equal zero for any  $t \in [0, \tau]$ :

$$D_H(P_R(x|x_0, t), P_s(x|x_0, t, \pi_s)) = 0, \forall t \in [0, \tau], \forall x \in X_H(x_0, t)$$

The idea behind this definition is that the optimal way of scanning a space is to make the probability of searching on a particular zone to be proportional to the expected reward: should you be searching for gold over a wide landscape, it would make sense to adjust the density of gold-miners in different zones to be proportional to the density -probability of finding- gold.

### 3.1.3 - Scanning sub-optimality coefficient

The more similar the two distributions are, the more intelligent and efficient the scanning will be, and, in the limit when the policy is optimal, we would reach an equilibrium where the divergence between both distributions is exactly zero.

Given that real-world scanning policies will not produce scanning probability densities exactly proportional to the rewards, this divergence will generally not be exactly zero over the different slices  $X_H(x_0, t)$ , so integrating the divergences over time can provide us with a measure of the sub-optimality of the scanning policy:

$$Scan(\pi_s|x_0, \tau) = \int_{t=0}^{\tau} D_H(P_R(x|x_0, t), P_s(x|x_0, t, \pi_s)) dt$$

In order to scale this coefficient into a more sensible figure, we may define the 'unit of sub-optimality' to be the sub-optimality associated with the random scanning policy  $\pi_s^{\text{RND}}$ .

$$\text{Random Scan Sub-optimality} = Scan(\pi_s^{\text{RND}}|x_0, \tau)$$

We may now divide the previous coefficient into this sub-optimality unit, so only policies that are 'better than random' will score below 1, while 'worse than random' ones will score above 1.

$$\text{Scan Sub-Optimality}(\pi_s | x_0, \tau) = \text{Scan}(\pi_s | x_0, \tau) / \text{Scan}(\pi_s^{\text{RND}} | x_0, \tau)$$

## 3.2 - Decision process

---

*"There is hardly anyone who could work out the entire table of pros and cons in any deliberation, that is, who could not only enumerate the expedient and inexpedient aspects but also weigh them rightly. Now, however, our characteristic will reduce the whole to numbers, so that reasons can also be weighed, as if by a kind of statics."*

---

**Leibniz, on Rational Decision-Making**

As the second part of the process, the information obtained by the scanning phase is used to decide which action to take or, more generally, the probability  $\pi_D(a)$  of each action to be taken.

### 3.2.1 - Intelligent decision

We will say a decision, defined as a probability distribution  $\pi_D(a)$  over all possible actions  $a \in A$ , is the 'intelligent decision'  $ID(a | \pi_s, \tau)$  for policy  $\pi_s$  and time horizon  $\tau$ , when the probabilities are proportional to the entropy of the conditional scanning probability densities for the actions over the last slice of the cone,  $X_H(x_0, \tau)$ .

$$ID(a | \pi_s, \tau) \propto \mathcal{H}(P_s(x | x_0, \tau, \pi_s, a))$$

Using that the conditional probabilities  $P_s$  for different actions  $a \in A$  are independent and that the corresponding conditional causal cones form a partition of the causal cone, we have that:

$$\mathcal{H}(P_s(x | x_0, \tau, \pi_s)) = \sum_{a \in A} \mathcal{H}(P_s(x | x_0, \tau, \pi_s, a))$$

So we may obtain the intelligent decision as a probability density over the actions as follows:

$$ID(a | \pi_s, \tau) = \mathcal{H}(P_s(x | x_0, \tau, \pi_s, a)) / \mathcal{H}(P_s(x | x_0, \tau, \pi_s))$$

This formulation implies that the intelligent decision in the discrete and the continuous cases are:

Discrete case	Intelligent decision = $\arg \max ID(a   \pi_s, \tau)$
---------------	--

Continuous case	Intelligent decision = $\int_{a \in A} a \cdot ID(a \pi_s, \tau) da$
-----------------	--

### 3.2.2 - Decision sub-optimality coefficient

Given a policy  $\pi = \{\pi_s, \pi_D\}$  that generates a probability distribution  $\pi_D(a)$  over the actions, we can define its sub-optimality as the divergence with the ideal distribution  $ID(a)$ :

$$\text{Decision sub-optimality}(\pi|x_0, \tau) \propto D_H(ID(a|\pi_s, \tau), \pi_D(a))$$

As we want this coefficient to be 1 for the random policy, we can define our unit of sub-optimality as the sub-optimality of the random decision policy  $\pi_D^{\text{RND}}$ , the uniform distribution:

$$\text{Decision sub-optimality}(\pi|x_0, \tau) = D_H(ID(a|\pi_s, \tau), \pi_D) / D_H(ID(a|\pi_s, \tau), \pi_D^{\text{RND}})$$

## 3.3 - Global sub-optimality

Given a policy  $\pi$  responsible for both scanning and deciding, we can define its global sub-optimality as the average of both sub-optimality coefficients:

$$\text{Sub-optimality}(\pi|x_0, \tau) = (\text{Scan sub-optimality}(\pi_s|x_0, \tau) + \text{Decision sub-optimality}(\pi|x_0, \tau)) / 2$$

This global sub-optimality coefficient allows us to determine which policies are approximately random ( $\approx 1$ ) and which are nearly optimal ( $\approx 0$ ).

## 3.4 - Policy IQ

Given the definition of sub-optimality of a policy, we can define the IQ of a given policy as follows:

$$IQ(\pi|x_0, \tau) = 1 / \text{sub-optimality}(\pi|x_0, \tau)$$

By using a moving average of this IQ over time as the system evolves, we can build a reliable real-time measurement of the intelligence of a particular system evolution.

## 4 - Fractal AI algorithm

---

*"Intelligence is the ability to avoid doing work, yet getting the work done."*

---

**Linus Torvalds**

Once we have a sensible definition of intelligence, the next step is use it to define a practical and efficient intelligent decision-taking algorithm, or more precisely:

"Given a system with some degrees of freedom that can be controlled, an informative simulation of the system's dynamics (not necessarily a perfect simulation nor a deterministic one), and a reward function defined over the state space, find an algorithm that use this information to push the degrees of freedom in such a way that the system behaves -or evolves- intelligently".

We will guide our search in two ambitious "design principles":

1. An algorithm with a sub-optimality coefficient tending to zero.
2. An algorithm with the lowest time-complexity possible.

The algorithm presented here is not the only possible one, nor it is a complete implementation of all the potential uses of the previous theory. It must instead be considered as one direct and intuitive use of the concepts presented in order to generate intelligent behaviour from the raw simulation of a system.

## 4.1 - Algorithm blueprints

---

*“You can recognize truth by its beauty and simplicity. When you get it right, it is obvious that it is right—at least if you have any experience—because usually what happens is that more comes out than goes in. (...) the truth always turns out to be simpler than you thought.”*

---

**Richard P. Feynman**

Before presenting the pseudo-code of the algorithm, we will introduce the ideas behind it and how its design aims to meet the previously defined theory in the lowest computational time complexity.

### 4.1.1 - Starting at Monte Carlo

---

*“Any one who considers arithmetical methods of producing random digits is, of course, in a state of sin.”*

---

**John von Neumann**

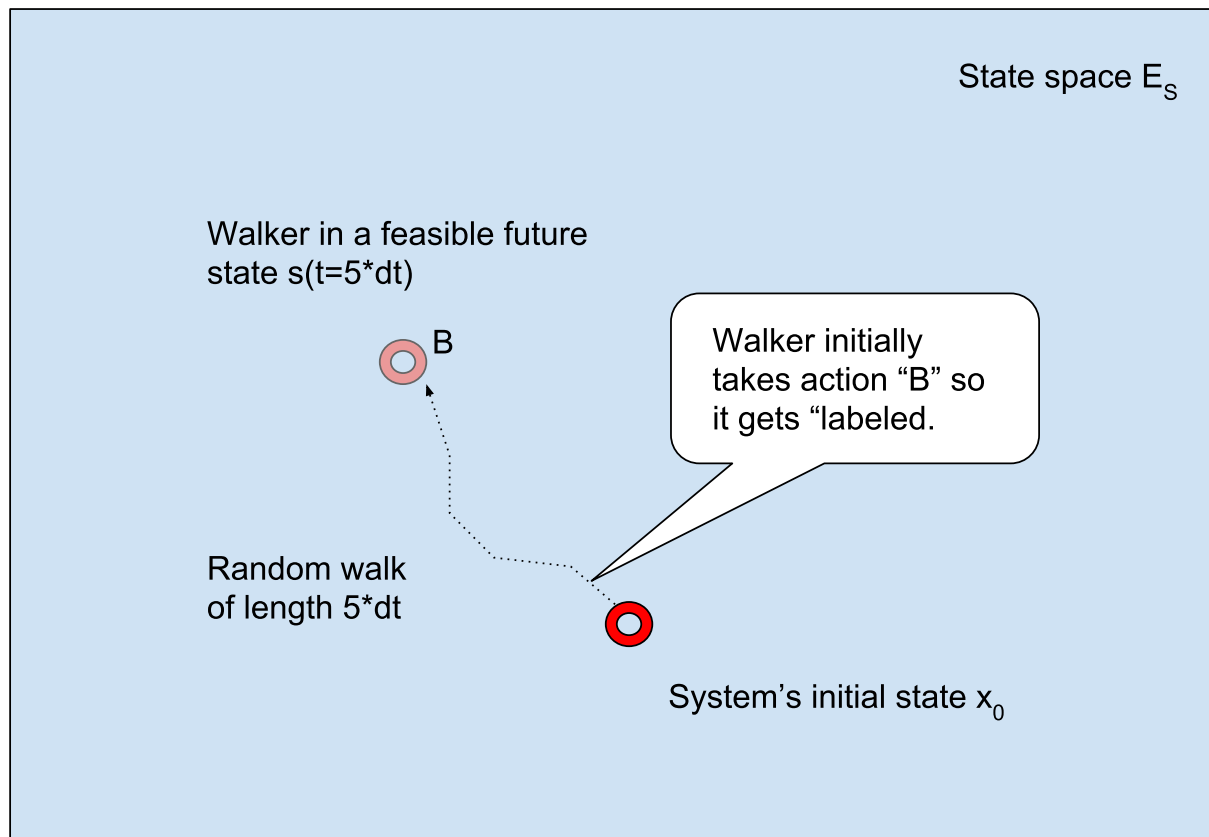
We will start our implementation by considering the random policy or, equivalently, using the standard Monte Carlo- approach, where a set of independent random walks are simulated over a number of discrete time steps to actually build a collection of paths in the causal cone considered in the theory.

In practical terms, we will need a informative simulation function -not necessary perfect nor deterministic- that, given a system state -that includes the positions of the different degrees of freedom the AI can modify- and a small delta of time  $dt$ , outputs the expected next state of the system:

$$x(t+dt) = \text{Simulation}(x(t), dt)$$

Starting at the actual system's state  $x_0$  and iterating the process of taking a random decision over the degrees of freedom and then simulating the next state for a fixed number  $T$  of ticks, we can build a random walk of length  $\tau = T \cdot dt$ .

In the image below, two different actions -pushing buttons “A” or “B”- are available. In the continuous case, actions would be real vectors representing the velocities of the degrees of freedom, or how strong it pushes each free param, and in which direction.



As we are looking for a decision-taking algorithm, the first decision -or action- taken during the walk, "B" in this example, will be an important piece of information, so we will keep it in the walker's internal state for later use.

By building one path after another we can actually build a set of  $N$  'feasible random walks' starting at  $x_0$  and ending at one of the possible system future states.

### 4.1.2 - Choosing the intelligence parameters

At this point, some practical question like how many random walks we will be using, the time horizon those walks will explore, and the number of ticks we will divide this time into will probably arise.

Before going any deeper in the algorithm it is worth spending some lines addressing those question as they are the main parameters related to the CPU you will need and the quality of the results.

#### 4.1.2.1 - Decisions per second

The first parameter to be chosen is about the number of decisions per second the agent will be taking, or the FPS (frames per second) of the algorithm. We are basically deciding here on the length of the  $dt$  used in the simulations, our tick length.

Basically, this length depends on the system reaction times. If our agent is modeling a fly we will need a faster decision taking, so a lower tick length. If it were modeling a spaceship traveling to andromeda, we could safely take one decision each month or even year.

As a guiding number, human brains are considered to run at about 12 decisions per second, so if we were to mimic some human behaviour, a dt of about 0.1 seconds (or 10 FPS) is a nice starting point.

Setting a FPS higher than the reaction time will not really improve the results, while CPU time will grow proportionally to FPS, so just keep this figure around the sweet point.

#### 4.1.2.2 - Time horizon

The time horizon dictates how far into the future will the walkers explore the consequences of their initial actions.

The idea here is to try to scan long enough to detect the problems before you can not avoid them and, again, it depends naturally on the task:

- A F1 driver, deciding on the driving wheel and pedals, needs to foresee where the car will be in about 5-10 seconds in order to properly drive a race.
- A spaceship traveling to andromeda needs to foresee some years to know if pushing the thruster now will lead you to andromeda or not.

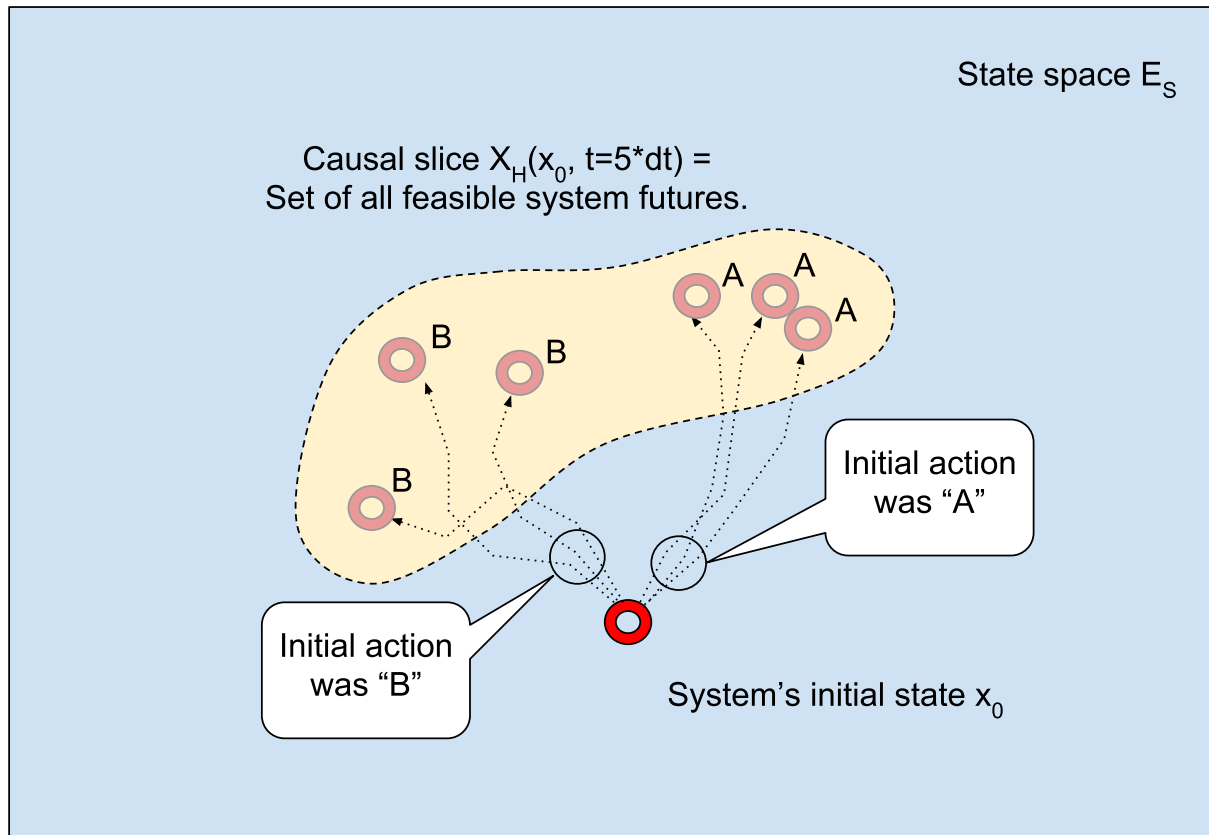
#### 4.1.1.3 - Number of walkers

This one is easy: the more the better. Of course it will use CPU linearly, but the most efficient way to improve the agent behaviour is using more walkers.

#### 4.1.3 - Simultaneous walks

As the theory mandates that the 'density of scanning' -or 'density of walkers'- must be kept proportional to some reward density, we will need to build all the random walks simultaneously, so a density of walkers can be defined.





Before going any further, we will show a simple pseudo-code that would generate this set of walks and finally decide based on the highest reward found at the final states of each path: if the path leading to the most rewarding state found started with the action "A", then it will actually take action "A".

---

```

// INITIALIZATION:
// Create N walkers with copies of the system's state:
FOR i:= 1 TO N DO BEGIN
    // Walkers start at the system's initial state:
    Walker(i).State:= System.State
    // Take walker's initial decision:
    Walker(i).Initial_decision:= random values
END
// SCANNING PHASE:
// Evolve walkers from time=t to t+Tau in M ticks:
FOR t:= 1 TO M DO BEGIN
    // PERTURBATION:
    FOR i:= 1 TO N DO BEGIN
        // First tick use the stored initial decision
        IF (t=1) THEN
            Walker(i).Degrees_of_freedom:= Walker(i).Initial_decision
        ELSE
            Walker(i).Degrees_of_freedom:= random values
        // Use the simulation to fill the other state's component:
        Walker(i).State:= Simulation(Walker(i).State, dt:= Tau/M)
    END
END

```

```
END
END
// DECIDING PHASE:
Best:= ArgMax(Reward(Walker(i).State))
Decision:= Walker(Best).Initial_decision
```

---

Being this code just a simple starting point, it already meets some of our design goals:

1. The more accurate the simulation is, and the smaller that  $dt$  is made, the more compatible with the system's dynamic the generated paths are.
2. The bigger the number of walkers used, the bigger portion of the causal cone will be scanned.

#### 4.1.4 - Probability densities

---

*"In applying dynamical principles to the motion of immense numbers of atoms, the limitation of our faculties forces us to abandon the attempt to express the exact history of each atom, and to be content with estimating the average condition of a group of atoms large enough to be visible. This method... which I may call the statistical method, and which in the present state of our knowledge is the only available method of studying the properties of real bodies, involves an abandonment of strict dynamical principles, and an adoption of the mathematical methods belonging to the theory of probability."*

---

**James Clerk Maxwell**

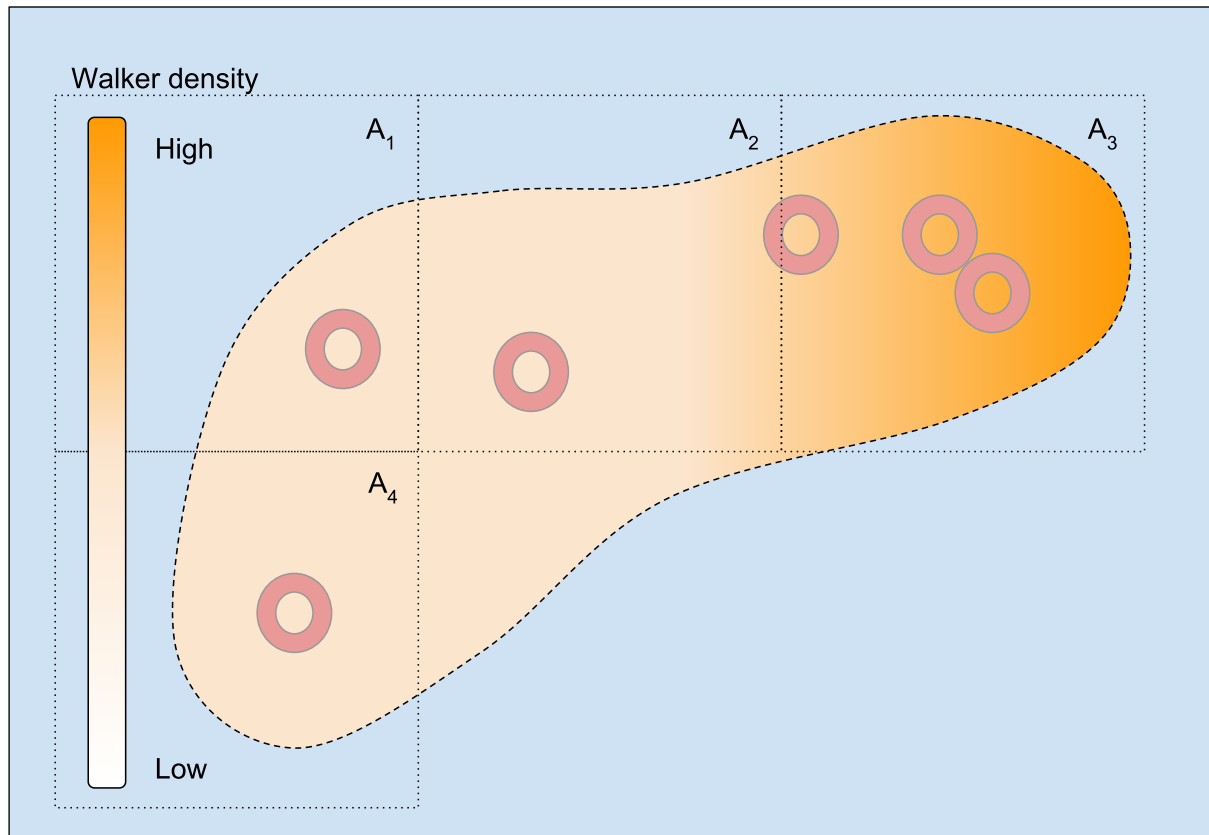
Accordingly to theory, no matter which partition  $\{A_1, \dots, A_N\}$  of the causal slice you consider, the walker density  $D_i$  on each part  $A_i$  should be made proportional to the density of reward  $R_i$

Our next step will be properly defining both densities.

##### 4.1.4.1 - Density of walkers

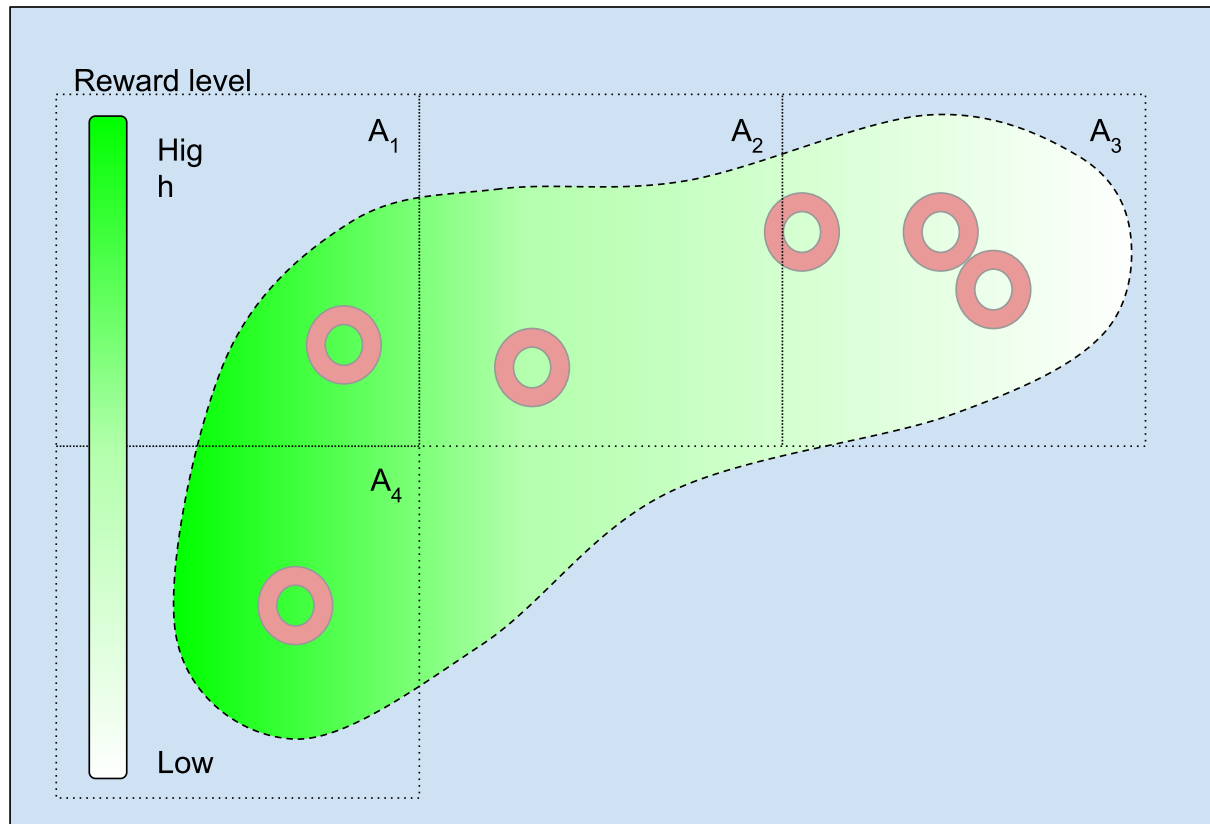
In the next image we have partitioned the space with boxes of the same size. Each one of the four populated boxes  $A_1$  to  $A_4$  have a number of walkers  $W_i$  inside it from a total of six walkers considered, so the walker's density at  $A_i$  will be  $D_i = W_i/6$ .

Please note that we didn't need it to be a partition, the different zones may overlap, so we are actually using a covering of the set of all the walker's positions.



#### 4.1.4.2 - Reward density

At the same time, a reward value is defined over the state space, so we can assign a reward value to each box  $A_i$  by averaging the rewards at the positions of the walkers in  $A_i$ .



#### 4.1.5 - Migratory flows

---

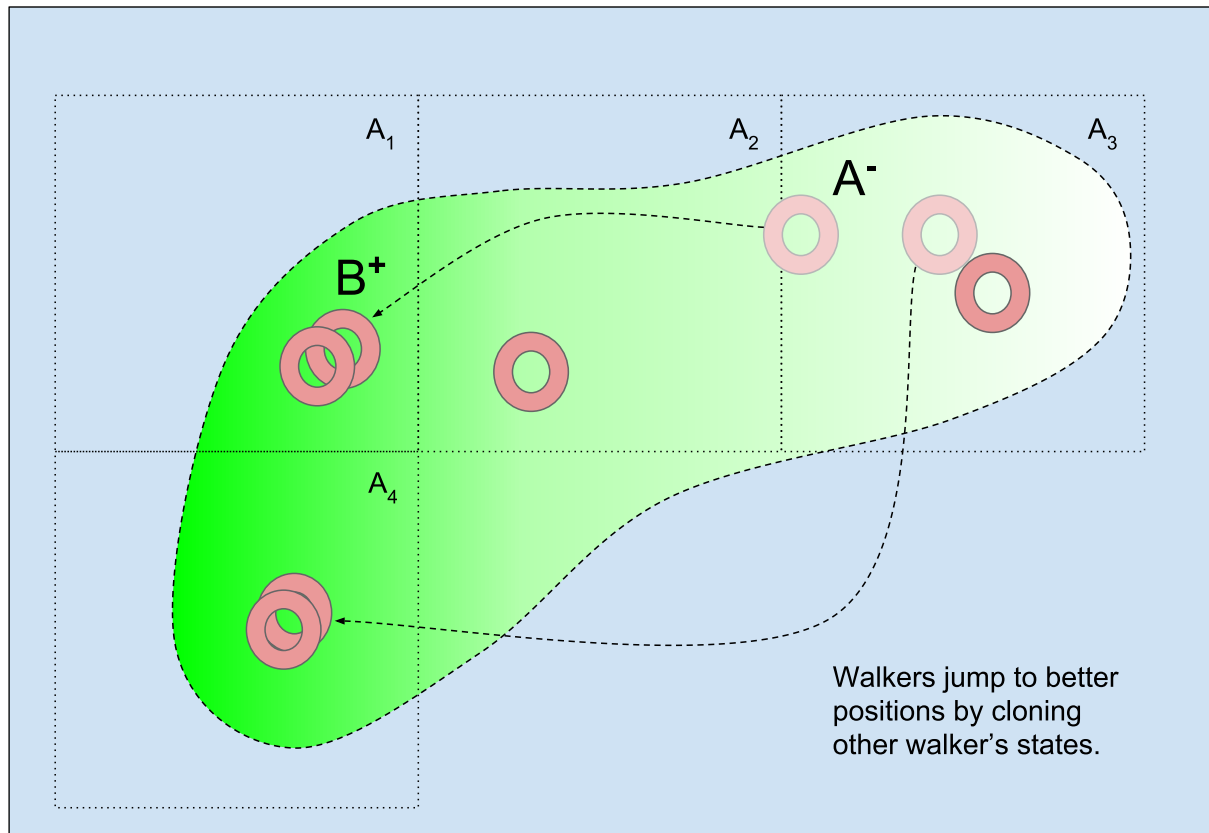
*"Failure is simply the opportunity to begin again,  
this time more intelligently."*

---

**Henry Ford**

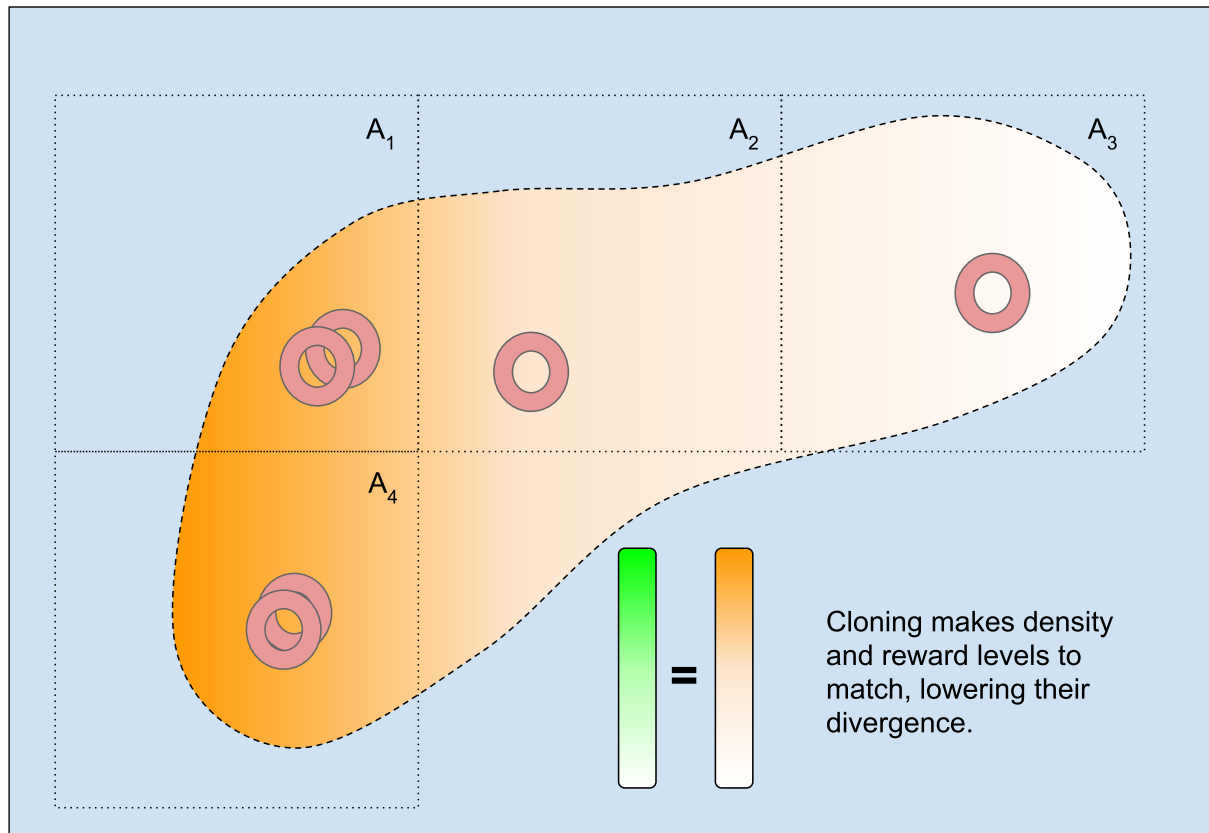
Our final goal is to make both densities proportional, or, equivalently, we need  $R_i/D_i$  to be as constant as possible.

Basically,  $R_i/D_i$  is a measure of the "reward per capita" that a walkers in the box  $A_i$  will receive, and the idea of making this a constant could be interpreted as the need of a fair wealth distribution over the walkers: if zone  $A_i$  has a much higher reward per capita than zona  $A_j$  then walkers in  $A_j$  will likely prefer to move to  $A_i$ . As we aim to make both densities to be proportional, we need to define a 'migratory flow' in order to balance the reward per capita.



This migratory flow will need to detect zones where walker density is higher-than-proportional to the reward (zones with a low 'reward per capita') and move some walkers from there to other zones with higher 'reward per capita'.

In order to move a walker  $W_1$  from zone  $A_i$  to  $A_j$  we could just select a random walker  $W_2$  at  $A_j$  and copy its state,  $W_1.state \leftarrow W_2.state$ . We say that  $W_1$  'cloned' into  $W_2$  position.



Here we just sketched a naïve way to choose which walkers needs to be cloned into which others by using boxes. In the final algorithm this idea will be replaced with a more general solution.

#### 4.1.6 - Taking the decision

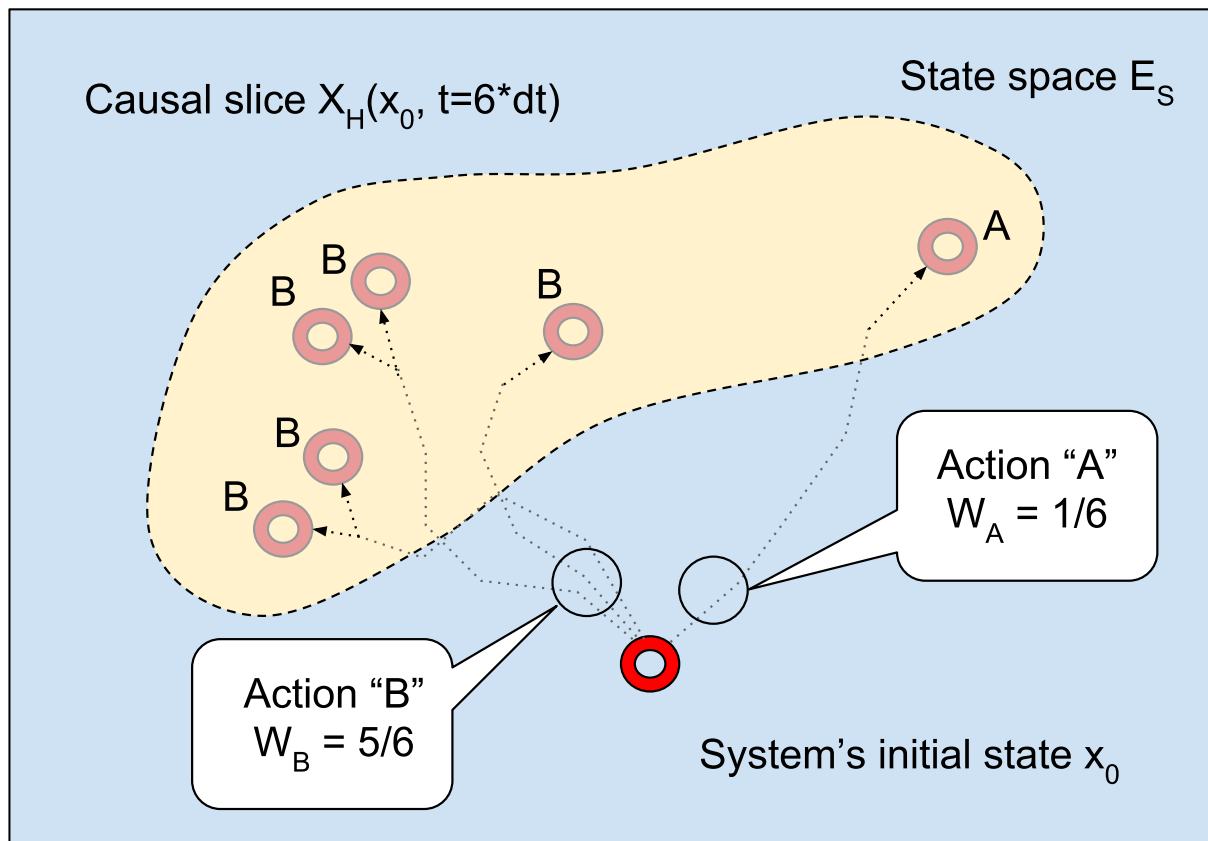
---

*"Deliberation is nothing else but a weighing, as it were on scales, the conveniences and inconveniences of the fact we are attempting."*

---

**Leibniz, on Rational Decision-Making**

In the previous example, when a walker labeled with the first option "A" is cloned to the position of a walker labeled "B", it not only clone its position but also its label, so after the clone, the initial action "A" will have one 'follower' less, while action "B" will gain one. After some ticks are performed, the distribution of the initial actions in the population of walkers will vary.



In the example above, both actions start having a proportion of  $3/6 = 0.5$  of the walkers, but after the migratory flow takes place, "B" population grows to  $5/6$  while "A" drops to  $1/6$ .

If the process was to be stop here -time horizon  $\tau$  was 6 ticks of length  $dt$ - then we would use these proportions to build our decision.

In the discrete case, the AI would take its decision by sampling an action from the probability distribution ( $1/6, 5/6$ ) of the actions, so the most probable action would be "B".

In the continuous case, where actions are real vectors and there is no finite list of available actions, the decision is formed by averaging the initial decisions of all the walkers.

## 4.2 - The migratory process

We commented on the need of forcing migratory flows from areas where reward was low or population density high, but defining a density is a tricky thing that can also be computationally demanding.

The idea of using boxes -as in the previous introduction- was just a sketch of the idea, we didn't define a method to effectively choose which walkers will clone and, more importantly, which walkers will they copy from.

### 4.2.1 - Virtual reward

---

*"Presuming that a man has wisdom of the third degree and power in the fourth, his total estimation would be twelve and not seven, since wisdom be of assistance to power."*

---

**Leibniz, on Rational Decision-Making**

The 'Virtual reward' is a generalization of the 'reward per capita' concept introduced before, but instead of using an externally defined partition, we will build one with a different zone per walker, so we can obtain a 'personalized' reward value that will tell a walker how 'lucky' he is as compared to other walkers: the higher the reward is, and the fewer the walkers around you, the better.

Choosing one partition per walker to account for the number of walkers inside each part is actually just a way to approximate the concept of 'density of walkers' around a position. At his time we will just assume we have defined some general measure of such density around the position of each walker:

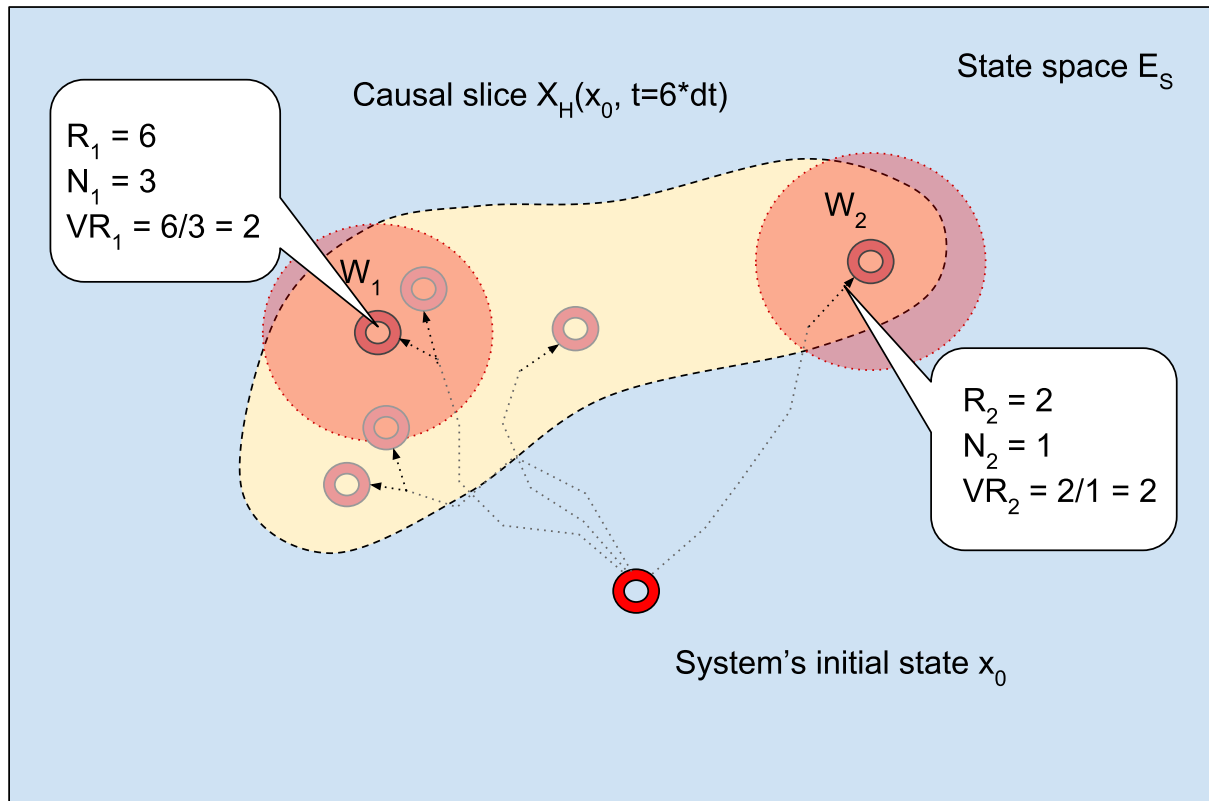
$D_i$  =Density of Walkers around  $W_i$  position/state.

We will then define the 'virtual reward'  $VR_i$  at a walker's  $W_i$  position, where the reward value is  $R_i$ , as being:

$$VR_i = R_i / D_i$$

One simple way to define a density of walkers around a given position is to consider our covering as being formed by a set of spheres of a fixed radius in the state space, each centered at one walker position, and counting for the number of walkers inside each one to get a density.





Please note that, as we will only focus on the proportionality of those densities -and not in their actual values- we can safely use the number  $N_i$  of walkers inside the ball as a density, without dividing it by  $N$ , as it will not alter the proportions.

Then, by looking at the image, we can say that at walker  $W_1$  position, there is a reward of  $R_1 = 6$  and a walker density of  $N_1 = 3$ , so we will calculate its virtual reward as:

$$VR_1 = R_1 / N_1 = 6 / 3 = 2$$

If we compare it with the virtual reward for walker  $W_2$ ,  $VR_2 = R_2 / N_2 = 2 / 1 = 2$ , we find that both positions are equally 'appealing' as the number of walkers is kept proportional to the reward.

#### 4.2.2 - Simplifying the Virtual Reward

---

*"Everything should be made as simple as possible,  
but not simpler."*

---

**Albert Einstein**

Using densities around the walker positions may not be an ideal approach for some reasons:

1. The algorithm would need to check the distances between all possible pairs of walkers, making the computational time-complexity to be, at least, of order  $O(n^2)$ , where  $n$  accounts for the number of walkers times the number of ticks you divide your time horizon in.
2. We would need to externally set a radius that makes sense and, probably, adjust it dynamically so it doesn't get too big or small during the process. You could, for instance, force the average number of walkers per ball to be between some reasonable values, let's say in  $(1.5, 3)$ , so if radius is so small that there is only one walker at each ball then, as  $1 < 1.5$ , you would need to increase the radius.

Our first simplification will eliminate the need of a externally defined radius, by far the smaller of our two concerns.

The key idea we will use here is that walker density  $D_i$  defined over a sphere centered at the walker  $W_i$  position, is roughly -the relation may not be strictly linear- inversely proportional to the average distance from  $W_i$  to the other walkers  $W_j$ .

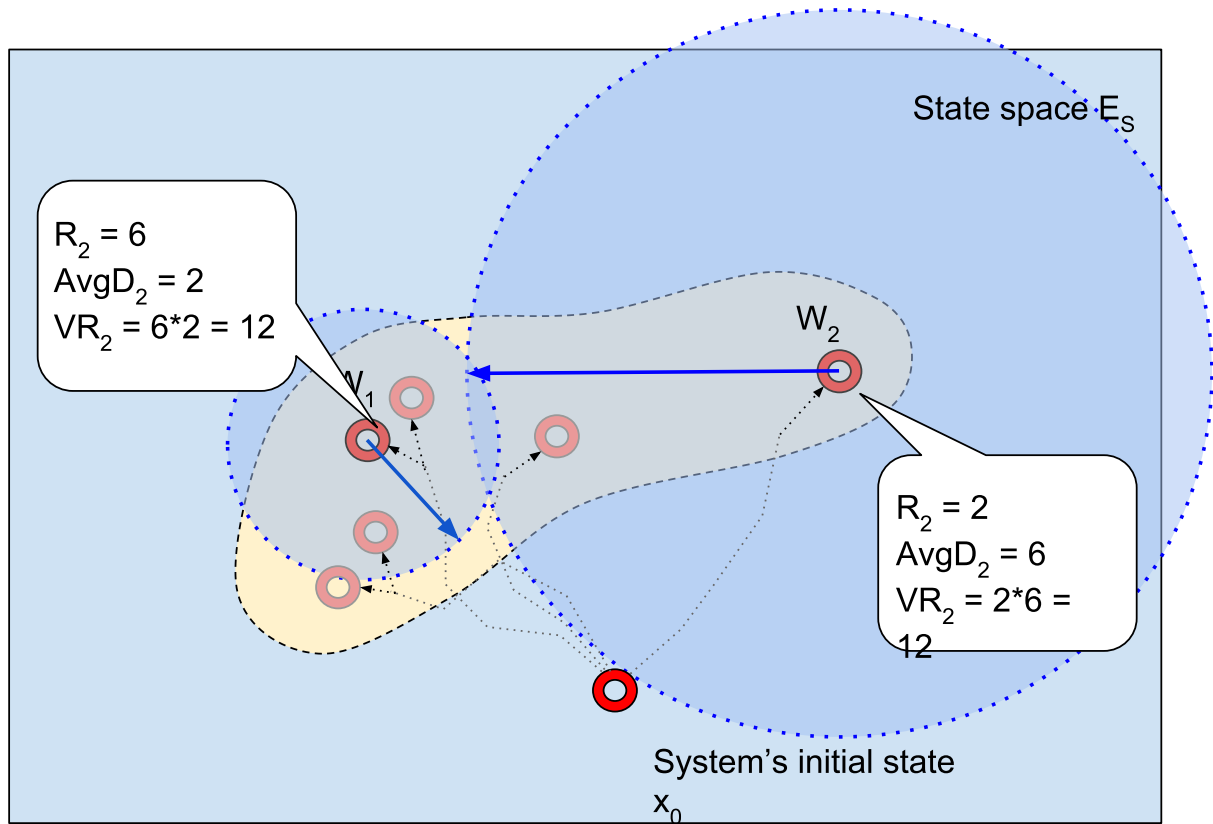
$$D_i \propto 1 / (\sum \text{Dist}(W_i, W_j) / N)$$

Again,  $N$  can be eliminated from the equation as we only need to compare proportions, so:

$$D_i \propto 1 / \sum \text{Dist}(W_i, W_j)$$

By replacing  $D_i \propto 1/N_i$  in the previous formula with this new version of  $D_i$  we obtain:

$$VR_i \propto R_i * \sum \text{Dist}(W_i, W_j)$$



The second simplification we will introduce is quite a dramatic one and it may initially sound like a really bad idea: we will replace the average distance from walker  $W_i$  to all the other walkers  $W_j$  with just one of those distances, randomly chosen:

$$D_i \propto 1 / \text{Dist}(W_i, W_j) \text{ with } j \text{ randomly chosen, } j < > i$$

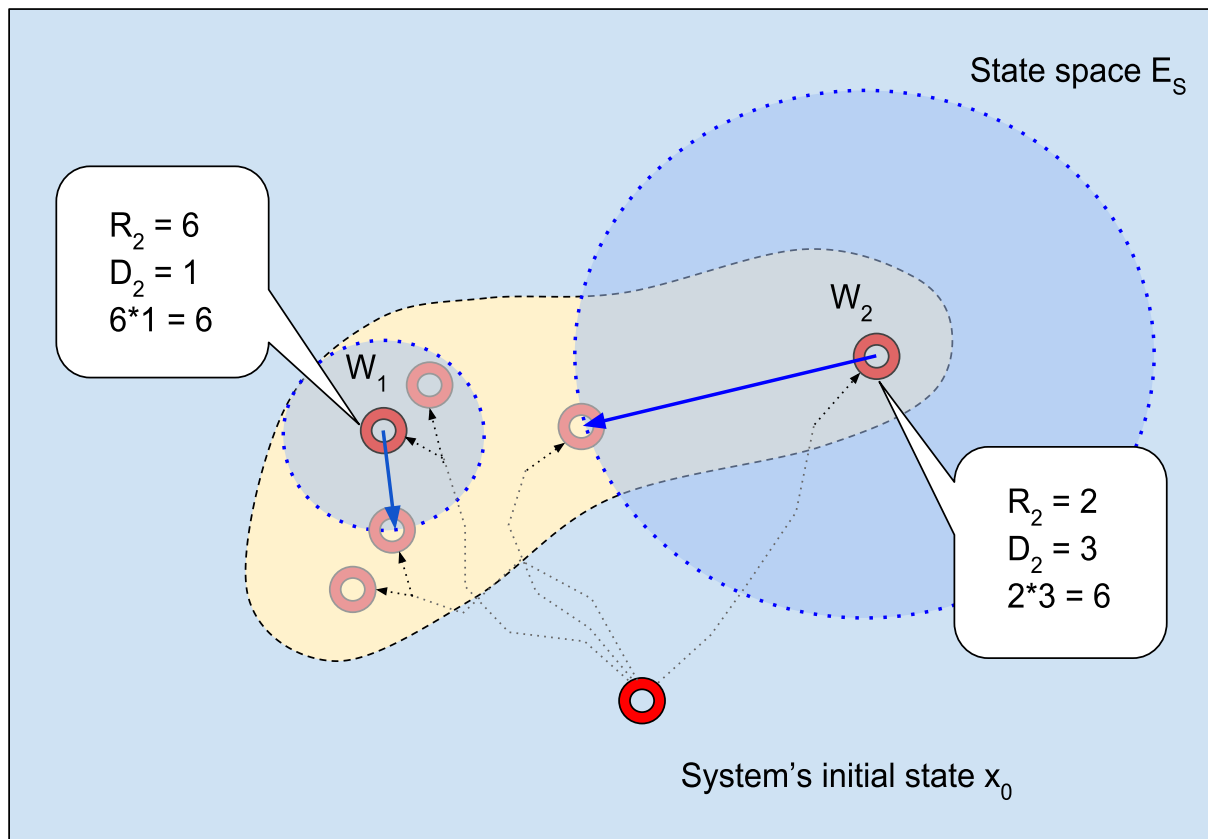
The resulting virtual reward formulation have a time-complexity of only  $O(n)$  while making it to be a highly stochastic function:

$$VR_i \propto R_i / D_i \propto R_i * \text{Dist}(W_i, W_j) \text{ with } j \neq i, \text{ randomly chosen.}$$

As our only purpose is to compare virtual rewards, being proportional allows us to safely define:

$$VR_i = R_i * \text{Dist}(W_i, W_j) \text{ with } j \neq i, \text{ randomly chosen.}$$

Using this stochastic version of the density actually do a better job than using the standard average distance, and at a much lower computational cost.



#### 4.2.3 - Balancing exploitation and exploration

---

*"Nobody ever figures out what life is all about, and it doesn't matter. Explore the world. Nearly everything is really interesting if you go into it deeply enough."*

---

**Richard Feynman**

A more general formula for the virtual reward can be considered:

$$VR_i = R_i^\alpha * \text{Dist}^\beta(W_i, W_j) \text{ with } j \neq i, \text{ randomly chosen.}$$

By default, both  $\alpha$  and  $\beta$  are set to 1, meaning that, as far as the the reward is properly scaled so its rate of change is roughly linear (and thus in the same order of magnitudes as for the distance changes), exploitation and exploration are balanced.

The value of  $\beta$  is considered fixed at 1 and highly dependent on the metric used, while  $\alpha$  is a parameter we can freely change in a standard range from 0 up to 2 or more, actually pushing this balance from equilibrium ( $\alpha = \beta$ ) toward an exploration-only mode ( $\alpha = 0$ ) or toward an aggressive search for reward ( $\alpha > \beta$ ).

#### 4.2.3.1 - The “Common Sense”

By manually setting  $\alpha = 0$ , the behaviour changes to a goal-less intelligence, where the decisions are taken in order to increase the number of different reachable futures, regardless of how rewarding they are. The effect is a very clever autopilot that can keep a plane flying around avoiding dangerous paths almost indefinitely.

Please note this “Common sense” mode is nearly equivalent to the idea of maximizing the empowerment [6] of the agent as an intrinsic goal. In fact, in this case the agent is maximizing a intrinsic reward represented by the entropy of the available futures after your decision.

Inversely, if  $\alpha > \beta$  the agent will be somehow blinded by the reward and will not care much about safety, driving the agent to dangerous situations where the reward is particularly high. In our system can be in death states, if our agent can die, then keeping  $\alpha$  low is mandatory.

We can then conclude that the value of  $\alpha$  roughly represents how safe is it to focus on exploitation in detriment of exploration, or how ‘safe’ the environment is for the agent.

#### 4.2.4 - Probability of cloning

---

*“Often one postulates that a priori, all states are equally probable. This is not true in the world as we see it. This world is not correctly described by the physics which assumes this postulate.”*

---

**Richard P. Feynman**

Once we defined a simple yet informative value for the virtual reward of a walker, it is time to get back to the migratory process we outlined before and try to draw a simple method to choose which walkers will be cloning which.

In the process of calculating the virtual reward, a walker must obtain the state of another randomly chosen walker in order to compare positions and obtain a distance. We will also start the cloning process by choosing another random walker in order to compare virtual rewards and obtain a probability of cloning.

Once walker  $W_i$  choose a second random walker  $W_k$  it will compare virtual rewards and, should he find his to be lower, he could decide to jump to  $W_k$  position by cloning its state.

We will define the probability of walker  $W_i$  with virtual reward  $VR_i$  cloning to  $W_k$  state, with virtual reward  $VR_k$ , as:

- Prob = 1                      If  $VR_i = 0$
- Prob = 0                      If  $VR_i < VR_k$

- $\text{Prob} = (\text{VR}_K - \text{VR}_i) / \text{VR}_i$       If  $\text{VR}_i \geq \text{VR}_K$

## 4.3 - Pseudo-code

We present a pseudo-code implementation to serve as a base code. The implementation presented here is aimed at simplicity and readability, not efficiency or scalability.

---

```

// [0] INITIALIZATION PHASE:
// Define dt for a time horizon Tau, divided on M ticks:
dt:= Tau/M
// Create N walkers with copies of the system's state:
FOR i:= 1 TO N DO BEGIN
    // Walkers start at the system's initial state:
    Walker(i).State:= System.State
    // Walkers take an initial decision:
    Walker(i).Ini_Dec:= random values
    // Walkers simulate their next states:
    Walker(i).State:= Simulation(Walker(i).State, dt)
END
// [1] SCANNING PHASE:
FOR t:= 1 TO M DO BEGIN
    // [1.1] CALCULATE VIRTUAL REWARD:
    FOR i:= 1 TO N DO BEGIN
        // Choose a random walker:
        j:= random index from 1 to n, j<>i
        // Use reward and distance to define virtual reward:
        d:= Distance(Walker(i), Walker(j))
        Walker(i).VR:= d * Reward(Walker(i).State)
    END
    // [1.2] PERTURB STATE:
    FOR i:= 1 TO N DO BEGIN
        // [1.2.1] Probability of cloning to random walker's position:
        j:= random index from 1 to n, j<>i
        IF (Walker(i).VR=0) THEN
            Prob:= 1
        ELSE IF (Walker(i).VR < Walker(j).VR) THEN
            Prob:= 0
        ELSE
            Prob:= (Walker(j).VR - Walker(i).VR) / Walker(i).VR
        // [1.2.2] Perturb state by Cloning or Simulation:
        IF (random < Prob) THEN BEGIN
            // [1.2.2a] Cloning:
            Walker(i).State:= Walker(j).State
            Walker(i).Ini_Dec:= Walker(j).Ini_Dec
        END ELSE BEGIN
            // [1.2.2b] Simulating:
            // Start by perturbing degrees of freedom:
            Walker(i).Degrees_of_freedom:= random values
        END
    END
END

```

```

        // Simulate the other state's component:
        Walker(i).State:= Simulation(Walker(i).State, dt)
    END
END
END
// [2] DECIDING PHASE:
Decision:= Sum(Walker(i).Ini_Dec) / N

```

---

## 4.4 - Classifying the algorithm

The algorithm presented has an algorithmic time-complexity of  $O(n)$ , where  $n$  is the number of walkers used times the number of ticks we divide the time horizon interval.

The algorithm may actually fit in many of the categories used in the field of algorithmics, making it difficult to properly classify it. Instead, we will name and comment on the categories where it could eventually fit.

### 4.4.1 - Monte Carlo Tree Search

---

*"We shall not cease from exploration  
And the end of all our exploring  
Will be to arrive where we started  
And know the place for the first time."*

---

**T.S. Eliot, Four Quartets**

One of the most evident classification of the algorithm is as a version of the Monte Carlo Tree Search ([MCTS](#)), where the different futures a system can visit are scanned up to some depth level (a time horizon) while forcing the less appealing branches to be rejected in the process in order to focus on the most promising ones.

The main differences between Fractal AI and the many MCTS variants [4] are:

1. MCTS only deal with discrete decision spaces, while Fractal AI deals with both cases.
2. MCTS build the decision tree one branch -or random walk- at a time, while Fractal AI builds a big number of branches simultaneously, interacting one with each other.

### 4.4.2 - Swarm algorithm

---

*“There is nothing that living things do that cannot be understood from the point of view that they are made of atoms acting according to the laws of physics.”*

---

**Richard P. Feynman**

Fractal AI is also a [swarm intelligence](#) algorithm where the collective behaviour of a pool of decentralized, self-organized agents, is solely based on the are used to take decisions.

Please note that, although in the pseudo-code the walkers are moved and cloned in a synchronized way for the sake of clarity, it is not mandatory at all, walkers could communicate via asynchronous messages and evolve in a asynchronous and isolated way, making it extremely easy to parallelize or distribute the algorithm in order to scale it. Adding a linear time-complexity makes the algorithm highly scalable.

#### 4.4.3 - Evolutive algorithm

---

*“There's a theory that says that life is based on a competition and the struggle and the fight for survival, and it's interesting because when you look at the fractal character of evolution, it's totally different. It's based on cooperation among the elements in the geometry and not competition.”*

---

**Bruce Lipton**

Fractal AI is a population-based [evolutive algorithm](#) where walkers are divided into population groups (based on their initial actions) that compete for success by cloning the best fits (highest virtual reward) overwriting the worst fits, and then mutating them by randomly changing their states on the simulating phase.

#### 4.4.4 - Entropic algorithm

---

*“Only entropy comes easy.”*

---

**Anton Chekhov**

There is not a real category of “entropic” algorithms, but it should. An algorithm is entropic when it is driven by the maximization/minimization of some kind of entropy, cross-entropy, or divergence.

Gradient optimization of a loss function like cross-entropy or divergence of two distributions, as used in deep learning, is a clear example of an entropic algorithm.



Fractal AI is clearly an entropic algorithm, as it is based on minimizing the divergence -or equivalently a cross-entropy- of two distributions: reward and walker probability distributions.

#### 4.4.5 - Fractal algorithm

---

*"All created forms are fractal, as is their purpose, use, and allotted time for existence."*

---

**Guy Finley**

In a Monte Carlo Tree Search, where actions are discrete, the graph of the states we visited in the search process is a tree. Fractal AI generate the same kind of trees in the same conditions.

When the decision and the state spaces are both continuous, then the distances between walkers and the time step  $dt$  we use for the simulation can be made as small as we want making the resulting tree to be formed by smaller and smaller pieces.

In the limit when both the number of ticks used to divide the time horizon and the number of walkers tend to infinity, the graph morph from a finite tree to a fractal tree.

There is a recursive version of the same algorithm -not covered in this document- that could be labeled as using "fractal recursivity".

## 5 - Experiments

We have run several experiments in order to show the algorithm strong points and to compare Fractal AI with other similar algorithms.

### 5.1 - Discrete case: Atari 2600 games

One of the most widely used benchmarks for AI methods are the [Atari 2600 games](#) from OpenAI gym. They all have a discrete decision space of six on/off buttons and well defined scorings allowing fair comparison between algorithms.

#### 5.1.1 - RAM vs Images

Playing any of those games involves a continuous two steps loop:

1. The game sends us its actual state.
2. AI answer with a decision vector of six booleans.

A game can send us its actual state in two flavours:

1. The screen image as a RGB array.
2. The internal RAM state as a byte array.

Fractal AI base its decisions on the entropic properties of the pool of walker states. Being this an intrinsic goal, fractal AI can be equally applied to RAM or image based games with very similar results.

As most of the methods we will be comparing with can only be played on image-based games, we can only compare Image vs RAM-based games solved with fractal AI using the same parameters, but in this case, there is no significant change in the scorings but takes less computation, as RAM size is smaller than image size on Atari-2600.

#### 5.1.2 - Results

Fractal AI is a forward-thinking algorithm -without any kind of learning- like a Monte Carlo Tree Search family algorithms, so it makes perfect sense to compare it with MCTS algorithms like UCT [7], as both measure efficiency in “samples per step”, where “samples” stand for the number of simulations being used in the internal process of making one decision.

Most of the games have never been played with a MCTS approach, mainly because they are too complex to be played with just a planning algorithm. Most of the methods make a heavy use of learning (DQN, A3C, etc.) plus, in some cases, a MCTS to help in the decisions.

Comparing Fractal AI with neural network based algorithms [8] [9] it is not that easy. Learning based strategies measured its efficiency in “number of samples used for learning”, a figure several orders of magnitude bigger that “samples per step”.

In general, learning based methods are very slow at learning (in the order of a billion samples) but, once they have learn, they can be very fast at making a decision. In contrast, Monte Carlo methods, like Fractal AI, doesn't need a previous learning process but use a lot more time to make a decision.

That said, we have compared Fractal AI with a number of methods in the literature (including the average human level after 2 hours of playing) focusing in the scores each method get on a representative number of atari games.

Game	Human Level	Best AI	Fractal AI (*)	FAI vs AI	Samples per step	Best AI method
<i>Allien</i>	7,128	5,899	<b>19,380</b>	329%	2,700	NoiseNet-A3C
<i>Amidar</i>	2,354	2,215	<b>4,306</b>	182%	1,222	NoiseNet-A3C
<i>Asteroids</i>	47,389	26,380	<b>76,274</b>	289%	2,733	NoiseNet-A3C
<i>Boxing</i>	12	99.4	<b>100</b>	100.6%	77.8	Dueling
<i>Centipede</i>	12,017	25,275	<b>1,222,000</b>	4,835%	1960	HiperNEAT
<i>Crazy climber</i>	35,829	179,877	<b>238,300</b>	132%	1,243	C51 DQN
<i>Double dunk</i>	-15.5	5	<b>20</b>	400%	5,327	Dueling
<i>Enduro</i>	860	<b>3,454</b>	800	23%	826	C51 DQN
<i>Ice hockey</i>	1	10.6	<b>52</b>	603%	12,158	Dueling
<i>Montezuma</i>	4,753	53	<b>2,500</b>	4,717%	5,175	A3C
<i>Ms Pacman</i>	15,693	6,283	<b>58,521</b>	931%	5,129	Dueling
<i>Phoenix</i>	7,243	<b>70,324</b>	11,930	17%	1,289	NoiseNet-A3C
<i>Q-Bert</i>	13,455	23,784	<b>35,750</b>	150%	2,728	MCTS (UCT)
<i>Seaquest</i>	<b>42,055</b>	266,434	5,220	42%	6,149	C51 DQN
<i>Space invaders</i>	1,669	<b>15,312</b>	3,605	24%	4,261	Dueling
<i>Tennis</i>	-8	23.1	<b>24</b>	104%	6,454	DQN
<i>Tutankham</i>	168	<b>321</b>	223	69%	3,023	A3C
<i>Video pinball</i>	17,668	949,604	<b>604,043</b>	64%	?	C51 DQN
<i>Wizard of wor</i>	4,756	12,352	<b>93,090</b>	753%	2,229	Dueling

Please note that:

- Most of the games have been played only 2 or 3 times with somehow standard parameters, and the best result used.
- Many games supported by the gym environment have not being tested at all.
- Some games were tested but didn't score so well. The most simple games (the easiest for neural networks) like Pong and Breakout, were the most difficult for FAI, surely because the game screen is so simple that the entropy gradient is not very significant, so learning is more efficient than planning here.

#### 5.1.2.1 - MCTS vs Fractal AI

Fractal AI can be considered in the family of MCTS algorithms and fairly compared with other state-of-the-art Monte Carlo methods like the UCT algorithm used in AlphaZero.

In the literature [7] we only found seven Atari games solved with the UCT algorithm. They are all simple games where DQN are good at (the main experiment were DQN + MCTS) but Fractal AI is not, at least using 1000 times fewer samples per step.

Game	Scores			Samples per step		
	UCT	Fractal AI	%	UCT	Fractal AI	%
<i>Beam rider</i>	<b>7,233</b>	2,160	30%	3,000,000	4,052	0,14%
<i>Breakout</i>	<b>406</b>	36	9%	3,000,000	5,309	0,18%
<i>Enduro</i>	<b>788</b>	800	102%	4,000,000	826	0,03%
<i>Pong</i>	<b>21</b>	---	---	150,000	---	---
<i>Q-bert</i>	18,850	<b>35,750</b>	189%	3,000,000	2,728	0.09%
<i>Seaquest</i>	3,257	<b>5,220</b>	160%	3,000,000	2,933	0.21%
<i>Space invaders</i>	2,354	<b>3,605</b>	153%	3,000,000	4,261	0.14%

#### 5.1.2.2 - Human record vs Fractal AI

---

*"To boldly play where no man has played before!"*

---

**William Shatner (almost)**

Human level stated in the previous table refers to the highest score obtained by an average human after 2 hours of playing. Being this a fair baseline to compare with, the figures are far from what a well trained human can get [10].

Beating humans records is much harder than beating the best actual AI methods, getting some of the records imply playing at a really high level and for several hours in clock time (Centipede human record took 12 hours). It can take days for a computer to play such a game so, even if seems to be theoretically feasible, we haven't focused on that.

On the other hand, some games are faster and based on reaction time and spatial coordination. In such games, Fractal AI can be on par with existing human absolute records.

Game	AI record	Human record	Fractal AI
<i>Ice hockey</i>	0	36	<b>64</b>
<i>Boxing</i>	99.4	99	<b>100</b>
<i>Centipede</i>	8,704	<b>1,301,709</b>	1,222,000 <sup>(*)</sup>

(\*) When this doc was petrified into a PDF, AI was still playing after 5 days with 1 extra life.

### 5.1.3 - Implementation details

The Atari games experiment was implemented in python following the ideas in this document but not the pseudo-code.

#### 5.1.3.1 - Auto-adjusting 'Samples per step'

---

*"With four parameters I can fit an elephant, and with five I can make him wiggle his trunk."*

---

**John von Neumann**

In the Atari implementation we focused on auto-adjusting the main parameters on-the-fly, so we could run the same code on a variety of Atari games.

In the code, instead of defining a fixed number of paths (walkers) and ticks (time depth) as in the included pseudo-code, we limit the number of samples (simulations) per step to be below a maximum, so the stop condition is just based on samples used so far.

The actual depth and number of walkers used at each step of the game is dynamically changed depending on the difficulty of the situation, allowing high "samples per step" when needed while keeping the average very low.

#### 5.1.3.2 - Additional death condition

Some games -like 'Tennis' and 'Ice hockey'- build their scorings by adding +1 when the player wins a ball, and -1 when it loses, so on those cases, your score could eventually get smaller over time.

In those cases, each ball played can be seen as a small game that ends when you gain or lose the point, and it makes sense to consider losing the point as a local 'death condition' that will triggered, in general, whenever your score get lower.

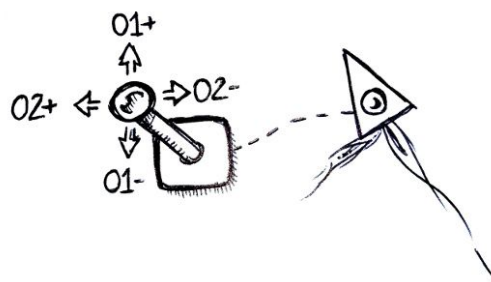
Although the algorithm can play any game without this additional death condition, it greatly helps improving the efficiency. In general, adding sensible death conditions is a powerful method to speed up fractal AI.

#### 5.1.4 - Github repository

You can find the full python code for the Atari experiment at <https://github.com/FragileTheory/FractalAI>

### 5.2 - Continuous case: Flying rocket

In this experiment we control a 2D rocket flying inside a closed environment. The rocket has two continuous degrees of freedom corresponding to the trust levels for the main rocket and a secondary one used for rotating.



At each step we define the force applied to each of the degrees of freedom, so our decision space is a continuous 2D space. In these cases we will not be choosing an action from a list, instead we will be deciding on a force as a 2D vector.

The rocket is a very interesting toy-system, it is very dynamic as equilibrium is never reached and making fast decisions is critic. When Fractal AI is used in a continuous decision system like this, it performs even better than it did in the discrete case. Being the decision an average of many decisions, it is more naturally defined in the continuous case than in the discrete one.

#### 5.2.1 - Flying a chaotic attractor

---

*"Chaos was the law of nature; Order was the dream of man."*

---

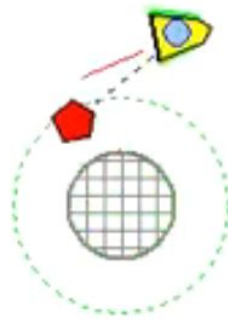
Henry Adams

As flying a rocket was not a big problem for fractal AI, we designed an special environment where the goals were almost impossible to get.

First, a hock was attached to the rocket using a rubber band, forming a chaotic oscillator where the final position of the hook is highly sensitive to small changes in the initial conditions, making it extremely difficult to sample the state space and find the low probability paths that define the right decision.



Secondly, we will define a ridiculous difficult goal for the system: drive the rocket in such a way that the chaotic hook picks a falling rock, take it into a distant deploy zone (crossed area), release it and wait until it leaves the dotted circle, and repeat it as many times as you can.



To define this 'hook reward' we just have to convert it into a reward function.

1. For every walker, the hook defines its target, being it the nearest rock if the hook is empty, and the deploy area when it is holding a rock.
2. For every walker, the hook calculates the distance  $D$  to the target.
3. Now the walker compares it with the same distance calculated at the initial state (the agent position).
4. If distance is 0, you did it!      Hook Reward = 100
5. If distance is bigger:      Hook Reward =  $10e-10$
6. If distance is 20% smaller:      Hook Reward = 0.2

Only this reward function was added, the rest of the code was just standard: the standard Fractal AI code, and a homebrew physic simulator of the system.

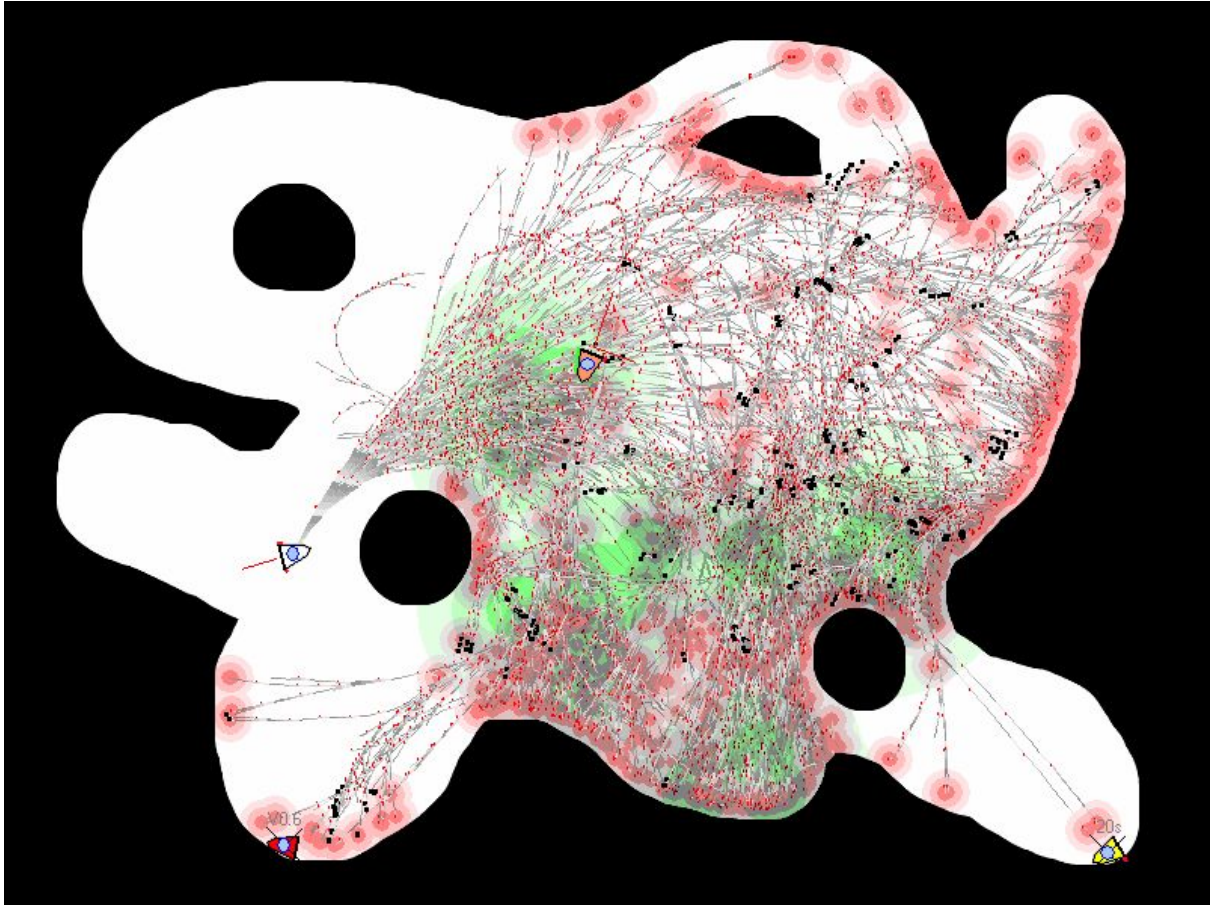
### 5.2.2 - Results

Fractal AI was able to solve this problem using only 300 walkers (paths) and 200 samples per walker (a time horizon of 2 seconds at steps of 0.1 second) for a total of 60,000 samples per decision.

The agent was able to chain several goals in a row, and also could recover the rock when it fell down the ground, totally solving the task with extremely low computational resources.

Video 1: [Solving the task](https://youtu.be/HLbThk624jl) (<https://youtu.be/HLbThk624jl>)

This experiment also allows us to watch the fractal paths generated in the process.



Video 2: [Visualizing the decision process](https://youtu.be/cyibNzyU4ug) (<https://youtu.be/cyibNzyU4ug>)

### 5.2.3 - Implementation details

This experiment used an early implementation of the algorithm that was 99% the same as in the pseudo-code. It is coded in object-pascal (delphi 7) and includes a complete add-hoc physical simulator.

The main differences with the general case came from the definition of distance between two states and the reward function used.



### 5.2.3.1 - Reward

The reward was the composition of two goals: keep alive -that correspond to the rocket health level (the simulator used the energy of the collisions to take health from the agent)- and the already commented 'hook reward', so the actual reward was defined as:

$$\text{Reward} = \text{Health\_Level} * \text{Hook\_Reward}$$

### 5.2.3.2 - Distance

Any informative distance would have made the trick, but as this was a very physical example, we decided to focus only on the position and momentum of the rocket to define its position, thus the distance used was:

$$\text{Dist}(A, B) = \text{Sqrt}((A.x-B.x)^2 + (A.vx-B.vx)^2 + (A.y-B.y)^2 + (A.vy-B.vy)^2)$$

Adding the rest of the coordinates to the distance formula resulted in a lower performance. In general, using a distance that makes sense in the problem helps. For the general case, using an embedding of the state can reduce its dimensionality and, as in any other method, helps to improve efficiency.

## 6 - Research topics

We will briefly comment on some of the research topic we find worth exploring.

### 6.1 - Distributed computing

The algorithm time complexity is  $O(n)$  where  $n = \text{num. walkers} \times \text{num. ticks}$ , but walkers work almost independently -except for some inter-walker communication- and even asynchronously, so a parallel implementation of the algorithm, assigning one core per walker can, in principle, lower the time complexity near to  $O(\text{num. ticks})$ .

Thus, adding more CPU can scale up the algorithm almost linearly but, eventually, the inter-processes communication overhead of sending system states from one walker to another will impose a practical limit dependent, in practical terms, on the size of the system state.

Reached this point, a second distributed strategy is launching several fractal AI processes -workers- in a distributed environment, each one using a smaller number of walkers to output a decision vector that will be averaged by the agent in order to make its decision. This 'clustered' decision is almost as reliable as using the sum of all walkers in a single fractal AI.

By adding the capability to reshuffle the states of all the walkers among the cluster of workers every  $m$  steps, you can continuously change from totally isolated workers ( $m = \text{number of total ticks}$ , no communication overhead) or totally connected workers ( $m = 1$ , overhead depending on state size and simulation time).

Please note that the implementation included in the [github repository](#) at the [experiment section](#) includes this parallel capabilities as an extra parameter.

### 6.2 - Adding learning capabilities

---

*"One remembers the past to imagine the future"*

---

**Daniel L. Schacter**

As we noted before, one of the limitations of the algorithm was dealing only with the forward-thinking process, so one natural research topic is mixing forward and backward-thinking in a single process.

The presented algorithm scans all available actions at any given state with an uniform probability distribution, it doesn't have any 'a priori' preference, so the walks produced are truly random.

Fractal AI algorithm could be used to feed examples of correct decisions -rollouts- to a neural network that, in turn, would train itself to predict the intelligent decision -a probability distribution over the actions- as a function of the state.

We can then use this as the 'a priori' distribution: when a walkers needs to randomly choose an action before simulating a delta of time, instead of choosing a random action, we can now sample it from this distribution.

A mechanism like this could give the walkers a natural tendency to repeat actions that worked well in the past on similar situations. Usually it means that, with the same number of walkers, you can get better decisions or, inversely, that you can safely lower the number of walkers needed to decide on situations that are familiar to the agent.

In the extreme case where the neural network can be considered well trained, you can completely disable walkers and decide by choosing the most probable action in the NN output.

By comparing the decision suggested by such a memory system with the one generated by the fractal AI algorithm -for the same initial state- we can estimate how accurate our a priori distribution is, enabling us to dynamically adjust the 'credibility' of this distribution.

To get a simplified sketch of the idea we could:

- 1) Define a function of the state that outputs a distribution over the actions:  $N(\text{State}) = (N_i)$ .
- 2) When the algorithm ends, we were already getting a distribution  $(P_i)$  over the available actions (the proportion of walkers associated with each action).
- 3) We can know how similar they are:  $\text{Credibility} = D_H(P_i, N_i) / D_H(P_i, \text{Uniform})$ .
- 4) In the next iteration we can reduce the number of walkers to:  $\text{Max\_Walkers} * (1 - \text{Credibility})$
- 5) In this next iteration, when walkers are building the random walks, they first get the 'a priori' distribution  $(N_i)$  associated to its actual state, and mix it with the uniform distribution  $(U_i) = (1/N)$ :  $(X_i) = (N_i) * \text{Credibility} + (U_i) * (1 - \text{Credibility})$ .

This mechanism opens the possibility of mixing fractal AI with any general learning method -like neural networks- in such a way that it can detect when the memory-based backward-thinking "fast" decisions are good enough to replace the more expensive -in terms of CPU usage- forward-thinking decisions, allowing us to:

1. Dynamically reduce the number of walkers used as the credibility of the distribution gets higher to speed up the process.
2. Get better results over time using the same number of walkers.
3. Generate a stand-alone fast policy (a standard neural network) to make decisions in absence of walkers (backward-thinking only).

## 6.2.1 - Using a DQN for learning

DQN is a model of deep learning designed to learn the probability of the actions -expressed as reward expectation- as a function of the system state and as such it is a perfect match for Fractal AI.

In one hand, Fractal AI is able to generate good quality game rollouts -sequences of pairs state-action from previous games- without the need of a priori density of probabilities for actions, so it is able to feed the learning process of the DQN with meaningful game sequences instead of random played ones, boosting the learning performance to some degree.

In the other hand, once the DQN has learned from a dataset of initial rollouts, Fractal AI can use it output as a priori for randomly choosing an action at each walker step, making the fractal AI to scan more deeply those actions suggested by the DQN and thus, making fractal AI more efficient and capable as the DQN learns to make better predictions.

This kind of combination is not new in the literature [7] but replacing UCT (a state-of-the-art implementation of MCTS) with Fractal AI could improve the efficiency of the combo. A fair comparison between both methods is presented in the [experiments section](#) showing that fractal AI can outperform UCT using about 0.01% to 0.1% of the samples per step.

## 6.3 - Common Sense Assisted Control

Imagine a drone driven with Fractal AI, following the only goal of not crashing into anything. Now add a remote control that, when pushed forward, send an a priori probability over the actions where going forward is much more probable than all the other actions, in the same way the DQN would be doing.

The Fractal AI will try to follow this direction, but only if this doesn't go against its first goal of not crashing into anything. The resulting drone will follow the control orders without crashing, allowing to fly it on difficult scenarios with ease and safety.

## 6.4 - Consciousness

---

*"In any decision for action, when you have to make up your mind what to do, there is always a 'should' involved, and this cannot be worked out from, 'If I do this, what will happen?' alone."*

---

**Richard P. Feynman**

When the reward function is a composition of several goals  $\{G_i\}$  we can assign a relative importance  $\{K_i\}$  for of each goal, having  $K_i \geq 0$  and  $\sum (K_i)=1$ , so our reward function would looks like this:

$$\text{Reward}(X) = \prod (G_i(X)^{K_i})$$

We can consider the vector  $\{K_i\}$  as being the “mental state” -as opposed to the physical state- of the agent. Any mechanism that could automatically adjust those coefficients in order to make better decisions can be considered as a conscious mechanism.

If we consider those  $\{K_i\}$  as being a second-level agent state, we can use them as both state component and degrees of freedom, and apply the same Fractal AI algorithm to intelligently adjust them, as far as we can define a sensible reward associated with a mental state  $\{K_i\}$ .

## 6.5 - Real-time decision-making

---

*“Life is the continuous adjustment of internal relations to external relations.”*

---

**Herbert Spencer**

The need in the present implementation of resetting all the walkers to the new agent position after a step is made, force us to erase valuable information, making the samples per step to be higher than it could be.

A slightly different implementation could lead to a continuous decision algorithm where a new decision is generated for each tick of the algorithm -as opposed to each step of the agent- in a totally continuous process, allowing to sample the decision at any time the agent needs it.

In this new implementation, each continuous decision would come with a measure of the average delay between the agent time when the walkers started its path and the actual agent time when the decision is used. A faster CPU would allow the algorithm to have a smaller delay, use more walkers and paths, or think at a longer time horizon. A new parameter would be needed to limit this delay below a desired value.

## 6.6 - Universality pattern

---

*“Find beauty not only in the thing itself, but also in the pattern of the shadows, the light and dark which that thing provides”*

---

**Junichiro Tanizaki**

It would be interesting to connect the distances between walkers at any moment with the 'Universality pattern" found in the eigenvalues of random matrices. This pattern seems to be universal to any system where the parts are heavily correlated. In the pool of walkers it is the case, so if this algorithm is some form of universal complex system solver, it makes sense to try to connect both ideas.

## 7 - Conclusions

The theory of intelligence introduced allowed us to build a very efficient agent that, in the discrete decision case, not only outperforms actual implementations of MTCS in at least three orders of magnitude, but can also beat state-of-the-art learning based algorithms, like DQN or A3C neural networks, without the need of any previous learning or understanding of the system, by just inspecting the tree of decisions using entropy-based principles that boost exploration while producing a exploitation-exploration balanced decision in the process.

The algorithm can also be directly applied to continuous decision spaces, where it proved to be highly efficiently, introducing Monte Carlo methods into a new spectrum of possible uses like driving vehicles or controlling robots.

The algorithm naturally allows working with neural network in a close simbiosis where the NN learns from the actions taken by the intelligence to produce a better a priori distribution over the actions, that is then used by the intelligence to get better over time, that are learned by the NN closing a virtuous cycle. By doing so, actual reinforced learning methods could be reshaped into a simpler and more efficient form of supervised learning.

## Bibliography

- [1] Wissner-Gross, A. D., and C. E. Freer. "[Causal Entropic Forces](#)". Physical Review Letters 110.16 (2013). © 2013 American Physical Society.
- [2] Gottfredson, Linda S. (1997). "[Mainstream Science on Intelligence: An Editorial With 52 Signatories, History, and Bibliography](#)". Intelligence. 24: 13–23. ISSN 0160-2896. doi:10.1016/s0160-2896(97)90011-8.
- [3] Shane Legg, Marcus Hutter. "A Collection of Definitions of Intelligence". Frontiers in Artificial Intelligence and Applications, Vol.157 (2007) 17-24, [arXiv:0706.3639](#) [cs.AI].
- [4] Cameron Browne, Edward Powley, Daniel Whitehouse, Simon Lucas, Peter I. Cowling, Philipp Rohlfshagen, Stephen Tavener, Diego Perez, Spyridon Samothrakis and Simon Colton. "[A Survey of Monte Carlo Tree Search Methods](#)". IEEE Transactions on computational intelligence and AI in games, Vol. 4, No. 1, March 2012.
- [5] Timothy Yee, Viliam Lisý, Michael Bowling, "[Monte Carlo Tree Search in Continuous Action Spaces with Execution Uncertainty](#)". Proceedings of the Twenty-Fifth International Joint Conference on Artificial Intelligence (IJCAI-16).
- [6] Christoph Salge, Cornelius Glackin, Daniel Polani (2013) "Empowerment -- an Introduction", [arXiv:1310.1863](#) [cs.AI].
- [7] Xiaoxiao Guo, Satinder Singh, Honglak Lee, Richard Lewis, Xiaoshi Wang, "[Deep learning for real-time Atari game play using offline Monte-Carlo tree search planning](#)", Advances in Neural Information Processing Systems 27 (NIPS 2014).
- [8] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A. Rusu, Joel Veness, Marc G. Bellemare, Alex Graves, Martin Riedmiller, Andreas K. Fidjeland, Georg Ostrovski, Stig Petersen, Charles Beattie, Amir Sadik, Ioannis Antonoglou, Helen King, Dhharshan Kumaran, Daan Wierstra, Shane Legg & Demis Hassabis (DeepMind Technologies), "[Human-level control through deep reinforcement learning](#)", Nature volume 518, pages 529–533 (26 February 2015), doi:10.1038/nature14236.
- [9] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, Martin Riedmiller (DeepMind Technologies), "Playing Atari with Deep Reinforcement Learning", [arXiv:1312.5602](#) [cs.LG].
- [10] "[Atari compendium](#)" (human world record list).



## ATARI experiment references

- [A1] Guo, Xiaoxiao and Singh, Satinder and Lee, Honglak and Lewis, Richard L and Wang, Xiaoshi. **Deep Learning for Real-Time Atari Game Play Using Offline Monte-Carlo Tree Search Planning**. [NIPS2014 5421](#), 2014.
- [A2] Greg Brockman and Vicki Cheung and Ludwig Pettersson and Jonas Schneider and John Schulman and Jie Tang and Wojciech Zaremba. **OpenAI Gym**. [arXiv:1606.01540](#), 2016.
- [A3] Marc G. Bellemare, Will Dabney Rémi Munos. **A Distributional Perspective on Reinforcement Learning**. [arXiv:1707.06887](#), 2017.
- [A4] Meire Fortunato, Mohammad Gheshlaghi Azar, Bilal Piot, Jacob Menick, Matteo Hessel, Ian Osband, Alex Graves, Vlad Mnih, Remi Munos, Demis Hassabis, Olivier Pietquin, Charles Blundell, Shane Legg. **Noisy networks for exploration**. [arXiv:1706.10295](#), 2018.
- [A5] Volodymyr Mnih & others. **Human-level control through deep reinforcement learning**. [doi:10.1038/nature14236](#), 2015.
- [A6] Matthias Plappert, Rein Houthooft, Prafulla Dhariwal, Szymon Sidor, Richard Y. Chen, Xi Chen, Tamim Asfour, Pieter Abbeel, Marcin Andrychowicz. **Parameter Space Noise for Exploration**. [arXiv:1706.01905](#), 2017.
- [A7] Tim Salimans, Jonathan Ho, Xi Chen, Szymon Sidor, Ilya Sutskever. **Evolution Strategies as a Scalable Alternative to Reinforcement Learning**. [arXiv:1703.03864](#), 2017.