Bernhard Ramsauer, BSc

Autonomous Nanocars based on Reinforcement Learning

MASTER'S THESIS

to achieve the university degree of
Diplom-Ingenieur

Master's degree programme:
Advanced Materials Science

submitted to
Graz University of Technology

Supervisor
Assoc.Prof. Dipl.-Ing. Dr.techn. Oliver Hofmann

Institute of Solid State Physics

Graz, November 2020

AFFIDAVIT

I declare that I have authored this thesis independently, that I have not used other than the declared sources/resources, and that I have explicitly indicated all material which has been quoted either literally or by content from the sources used. The text document uploaded to TUGRAZonline is identical to the present master's thesis.

Date, Signature

Acknowledgment

I would like to express my sincere gratitude to my supervisor Oliver T. Hofmann for the continuous support, his motivation, enthusiasm, and knowledge. His guidance helped me in becoming a better scientist and his style of leadership also immensely assisted my personal development immensely. I would also like to thank my team for all the animated discussions, brainstorming sessions and the inspiring ideas, which have made the development of my thesis so much easier.

I would like to express my immeasurable appreciation and deepest gratitude to Grant Simpson, who not only prepared the sample for me but also assisted me for two weeks straight with carrying out the experiment, measurements, additional specimen preparation and the experimental setup.

Furthermore, I also want to thank Leonhard Grill, Nacci Christophe and Monika Schied for giving me the opportunity to carry out the experiment on their scanning tunnelling microscope but also for the opportunity to test the code in advance whenever there was a time window.

Last but not least, I would like to wholeheartedly thank my partner Sarah for her support and patience. She made it possible for me to completely focus and be on top of things while giving me joy and love. In the end, I would like to express my cordial gratitude to my parents Elfriede and Alfred for giving me encouragement as well as their financial and mental support.

Abstract

Autonomous Nanocars based on Reinforcement Learning

Bernhard Ramsauer
Institute of Solid State Physics, Graz University of Technology

In April 2017, the Rice-Graz team, named after their Universities, with pilot Grant Simpson (Graz), participated at the world's first race of nanocars at the Center for Materials Development and Structure Studies (CEMES-CNRS) in Toulouse, France. At this race, participants had to direct a nanocar across a "racetrack" [6], which is 100 nm long for gold and 150 nm for silver, including two 45 ° turns and is set on a metallic substrate. In order to control their nanocar, they had to pull it via an STM-tip, but without being in direct contact with the nanocar.

The nanocars can be readily synthesized by using different shapes and properties. The physics that govern the molecule's movement and rotation is complex and involves the interaction between the molecule and the tip as well as the molecule and the substrate [8]. Therefore, it requires some expertise for humans to manoeuvre the nanocar and predict the outcome of a performed action.

This can be seen by taking the race from Toulouse as an example. Although the Rice-Graz team finished in first place by solving the 150 nm in 1.33 h, which gives an average speed of 112 nm/h and was much faster than anyone else, the rate of successful manouevres shows that there is a lot of room for improvement. Over the course of the race, the yield of successful pulling actions was about 54% and therefore only slightly better than predicting a coin flip. Thus, the idea of an artificial intelligence (AI)-controlled nanocar arose, which is the topic of this master thesis.

Here, we show how an artificial intelligence based on reinforcement learning can be implemented to manipulate single molecules. The AI is implemented in the form of an off-policy reinforcement learning algorithm, known as the Q-Learning algorithm. Being off-policy, enables the AI to learn without the necessity of a physical model. This also allows to learn from human-generated data. This means that the AI can be trained without operating directly at the STM, which saves time and operational costs.

After training from a rather small data set, the AI was further trained directly at the STM, where it manoeuvred the nanocar across a silver (111) surface. The AI is doing so by controlling the STM-tip position based on the position of the nanocar on the surface. The experiment showed that it is indeed possible to AI-control the nanocar. In a prime example, the AI showed an incredible success-rate of 89%, manoeuvring the nanocar at an average speed of 248 nm/h, which is more than double the speed compared to the race from Toulouse. Additionally, the experiment yields highly interesting insights that will help to create an efficient, and significantly improved AI that is more accurate and reliable, such that it can set itself apart from the manoeuvrability of humans.

Our results can easily be the basis for more sophisticated techniques of molecular manipulations where molecules are manoeuvred by AIs based on reinforcement learning and complemented by a deep neural network to analyse the current signal. The deep neural network can be used to find the correlations between the molecular manipulation and the induced current signal, which contains a unique rotation and translation pattern that is acting like a fingerprint for every molecule. This allows to identify and dislocate molecules at will, building the basis for future bottom-up constructions of nanotechnology.

Kurzfassung

Autonome Nanocars basierend auf bestärkendem Lernen

Bernhard Ramsauer
Institut für Festkörperphysik, Technische Universität Graz

Im April 2017 nahm ein österreichisch-texanisches Team der Universität Graz und der Rice University (Houston, TX) mit „Fahrer“ Grant Simpson (Graz) am weltweit ersten Molekül-Rennen teil. Bei diesem ersten Nanorennen der Welt, das am Center for Materials Development and Structure Studies (CEMES-CNRS) im französischen Toulouse stattfand, mussten die Fahrzeuge mithilfe eines Rastertunnelmikroskops (REMs) entlang eines vorgegebenen Parcours, eine Strecke von 100 Nanometern auf Gold bzw. 150 nm auf Silber inklusive zweier 45 °-Kurven, manövriert werden. Dabei durfte die Spitze des REM keinen direkten Kontakt mit dem Nanocar haben.

Nanocars mit unterschiedlichen Formen und Eigenschaften können auf einfache Weise hergestellt werden. Die Physik dahinter, welche für Bewegungen und Rotationen der einzelnen Moleküle verantwortlich ist, gestaltet sich allerdings als sehr komplex und beinhaltet auch die Wechselwirkung von Molekül zur Metallspitze sowie von Molekül zur Oberfläche. Ein Nanocar zu manövrieren und das Ergebnis einer Handlung vorherzusagen, ist deshalb für Menschen alles andere als einfach.

Das kann anhand des Rennens in Toulouse veranschaulicht werden. Obwohl das Rice-Graz-Team die Strecke von 150 Nanometern innerhalb von 1,33 Stunden zurücklegte und somit als Sieger des Rennens hervorging, ist in Bezug auf die Anzahl der tatsächlich erfolgreichen Manöver noch Luft nach oben. Im Laufe des Rennens waren in etwa 54 % der Zieh-Aktionen erfolgreich und demnach nur etwas höher als die Wahrscheinlichkeit, das Ergebnis eines Münzwurfs richtig zu erraten. Diese Beobachtung führte zur Idee, ein von künstlicher Intelligenz gesteuertes Nanocar zu entwerfen – was auch das Thema dieser Masterarbeit darstellt.

Durch die Implementierung einer künstlichen Intelligenz, welche auf bestärkendem Lernen basiert und Aktionen auch dann ausführen kann, wenn sich die Umgebung fortlaufend verändert, können einzelne Moleküle manipuliert werden. Die künstliche Intelligenz wird als off-policy-Algorithmus, auch bekannt als Q-Learning, implementiert. Durch den off-policy-Algorithmus kann die künstliche Intelligenz auch ohne das Vorhandensein eines physischen Modells lernen – demnach kann auch von Daten gelernt werden, die von Menschen generiert wurden. Da dazu nicht direkt am Rastertunnelmikroskop gearbeitet werden muss, werden Zeit und Kosten gespart.

Nachdem die künstliche Intelligenz zunächst von einigen wenigen Daten gelernt hatte, wurde sie direkt am Rastertunnelmikroskop trainiert. Die KI schafft dies, indem sie die Position der Metallspitze des REMs aufgrund der Positionierung des Nanocars auf der Oberfläche kontrolliert. Dieses Experiment zeigte, dass es durchaus möglich ist, ein Nanocar mittels einer KI zu steuern. Im erfolgreichsten Fall konnte die KI eine Erfolgsrate von 89 % erzielen, als das Nanocar mit durchschnittlich 248 nm/h und somit im Vergleich zum Rennen in Toulouse mehr als doppelt so schnell manövriert wurde. Durch das Experiment konnten außerdem wichtige Erkenntnisse für die Entwicklung einer effizienteren, genaueren und verlässlicheren KI gewonnen werden, die sich auch von der menschlichen Manövrierfähigkeit abhebt.

Unsere Ergebnisse können als Ausgangspunkt für komplexere Manipulationen an Molekülen dienen, bei der Moleküle mit Hilfe einer auf bestärkendem Lernen basierenden KI manövriert werden und das induzierte Stromsignal mit Hilfe eines Deep-Learning neuronalen Netzes (DLNN) analysiert wird. Dadurch können Moleküle identifiziert und willkürlich platziert werden, was die Grundlage für zukünftige Bottom-up-Konstruktionen in der Nanotechnologie darstellt.

The mind
drives the mass

PUBLIUS VERGILIUS MARO

Contents

1 Introduction

In the following chapters, I will introduce the world's first nanocar race - the structure of the world's fastest nanocar [8] - and an artificial intelligence designed to control it. Although the designed nanocar finished in first place, we will see that the manoeuvrability, even for an experienced human operator, is almost random - meaning an action leads to an unpredictable outcome. In order to enhance the controllability of the nanocar, a reinforcement-based artificial intelligence is used to control the nanocar on an beyond human-level of accuracy.

On the one hand, this thesis provides the complete design process for an artificial intelligence as well as the python code that is used to control the nanocar. On the other hand, it provides the physics and structure behind the nanocar and a glance on the theory of artificial intelligence by providing a detailed description of reinforcement learning and the applied learning algorithm, known as Q-Learning. The complete python code is fully annotated and for easier understanding described literally and figuratively in chapter 2. The code provides a program (agent) that can learn from human generated data and a program to control the scanning tunnelling microscope.

1.1 The nanocar

This section will provide a short introduction to the world's first nanocar race, the design choices for this particular nanocar - called Dipolar Racer, which closely follows [8], and shows the ability of humans to control nanocars.

1.1.1 The nanocar race

The world's first nanocar race took place on 28 and 29 April 2017 at the Centre for Materials Development and Structure Studies (CEMES-CNRS) in Toulouse, France. Six teams participated with their self-designed nanocars. The teams had to deposit their nanocar on a gold or silver (111) surface at $\sim$5 K and manoeuvre it over 100 nm or 150 nm respectively by using a scanning tunnelling microscope. The participants had to reach the goal within 36 hours. The nanocar could either be manoeuvred by using the tip-induced electric field gradient or the inelastic electron tunnelling current. Thus, no mechanical manipulation, such as pushing with the STM-tip, was allowed.

The deposition procedure is as follows. The nanocars were deposited on the metallic surface and then located by imaging the surface by using the STM. At the beginning, a large area is being imaged to find a racetrack that fulfils the rules of the race. These rules are for the racetrack to have at least two 45 ° turns and dependent on the surface, the racetrack has to be either 100 nm long for gold and 150 nm long for silver. Since the Dipolar Racer moved uncontrollably fast on a gold surface even during STM imaging, the team back then selected to race on silver, which solved this problem. The complete racetrack from the world's first race in Toulouse is shown in figure 1.1.

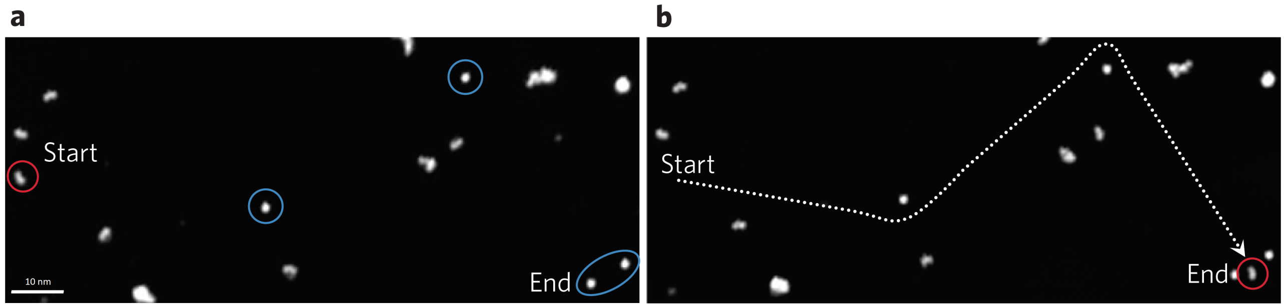


Figure 1.1: a: STM image (120 x 50 nm^2) of the Ag (111) surface at the start of the race showing a Dipolar Racer (red circle) on the left with two nearby nanocars, the two asperity pylons and the finish line between the juxtaposed pylons (blue circles). **b**: STM image of the same surface area where one Dipolar racer has crossed the finish line. The dotted line shows the 150 nm racetrack. The image is modified from reference [8]

1.1.2 The structure of the Dipolar Racer

In the following, the design features for optimal nanocar manipulation are explained. These are based on decades of STM manipulation and nanocar design expertise.

1. The **molecular weight** should be as low as possible, because it is difficult to deposit intact molecules under ultra-high vacuum conditions. A higher molecular weight provides more sites for surface adhesion. This in turn raises the diffusion barrier, and consequently slowing the Dipolar Racer.
2. The **wheels** should be aliphatic rather than alkenylic, aromatic or heteroatomic to minimize surface interactions. They should also be large enough to lift the chassis off the surface to minimize chassis-surface attraction. For the Dipolar Racer, the wheels are adamantane since they are aliphatic, while also being relatively spherical. The Dipolar Racer consists of two wheels, which are connected to opposite sites of the chassis. Since surface adhesion should be minimized, two wheels are a good choice for reducing surface interactions, while also lifting the chassis off the surface.
3. The **chassis** should be rigid and the **axles** as short as possible to prevent the overall structure from sagging towards the surface. This in turn decreases chassis-surface interactions. However, the axle also has to be long enough to minimize steric interactions between the wheels and the chassis and should be able to rotate freely around the axle to minimize rotational barriers.
4. The molecular structure should be **stable** enough to be deposited under ultra-high vacuum conditions, while also prevent bond breaking when a voltage pulse is applied at the STM-tip.

The structure of the nanocar, which in this specific case is called the Dipolar Racer, is shown in figure 1.2 and consists of two wheels, which are connected via axles to the chassis.

For translation on a surface, high forces are necessary to overcome the diffusion barrier. The easiest mechanism to overcome the diffusion barrier is 'pushing' the nanocar with the STM-tip by utilizing Pauli repulsion to translate the molecule. However, the rules of the race state that physical contact is forbidden, allowing only for tip-induced electric field gradient or inelastic electron tunnelling current to translate the nanocar.

Therefore, the Dipolar Racer was equipped with a strong net dipole in the chassis to improve the interaction with the electric field of the STM. The dipole is formed by two functional groups attached to a phenyl ring. The nitro group and the dimethylamine are connected to the phenyl ring and create a net dipole moment, shown in figure 1.2. To achieve a strong donor-acceptor interaction, the two functional groups have to be coplanar with the aromatic ring. This dipole supports the movement towards the STM-tip.

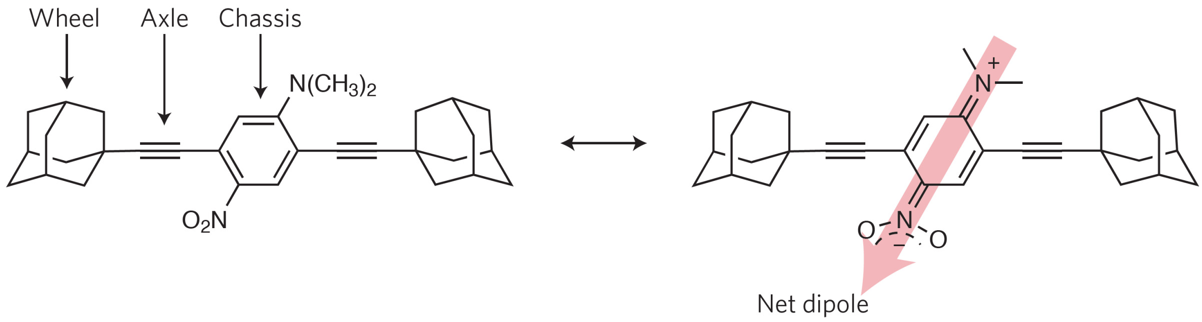


Figure 1.2: Molecular structure of the Dipolar Racer and its resonance form, which highlights the strong net dipole direction. The Dipolar Racer is ∼2.5 nm in length.[8]

1.1.3 The procedure of manipulating the nanocar

At first, the location of the nanocar is determined by imaging the surface with a low voltage. When the exact position is known, the lateral movement of the nanocar is induced by bringing the STM-tip towards the nanocar and applying a relatively high bias voltage. This creates a strong local electric field at the STM-tip with which the dipole moment interacts. If this field is sufficiently strong with respect to the diffusion barrier on the surface, a lateral displacement of the nanocar towards the STM-tip is induced. Afterwards, the nanocar is re-imaged with a low bias voltage to confirm its position. The schematic for a manipulation procedure is given in figure 1.3. A successful pulling action translates the molecule on average about 1 nm over the surface.

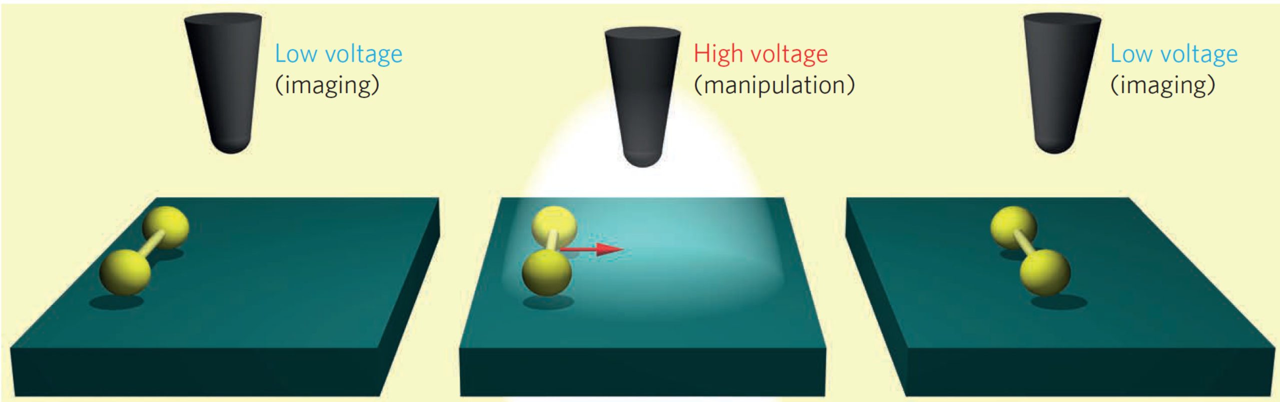


Figure 1.3: Schematic of the manipulation procedure. A low voltage (0.70 V) is used for imaging the molecule and a high voltage (1.8 V) is used to induce movement. [8]

However, since imaging the nanocar after every displacement step is the major bottleneck, as it limits the speed, it should be avoided if possible, as it is very time-consuming and takes between one and five minutes. Thus, instead of repeatedly imaging the surface, the tunnelling current during voltage pulses is measured and used as an indicator of how the nanocar moved towards the STM-tip. The tunnelling current signal has been shown to identify hopping distances and to distinguish between pulling, pushing and rolling modes during a lateral motion of the STM-tip over a molecule [1] and [5].

A tunnelling current profile, as shown in figure 1.4, is measured while a voltage pulse is applied.

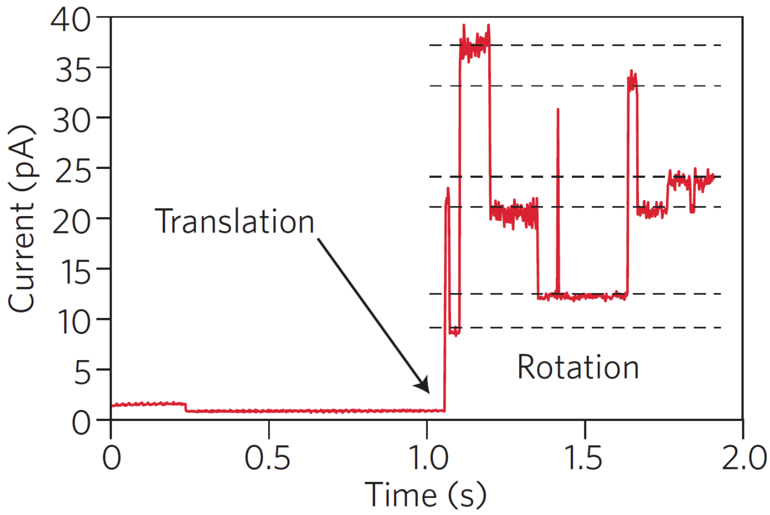


Figure 1.4: An order of magnitude jump in the current versus time plot indicates that the molecule has translated, after which the molecule rotates. [8]

Dependent on the translation behaviour of the nanocar, the profile may contain flat regions corresponding to no molecular motion and a region with abrupt and high current changes, which correspond to translation towards the tip and rotation under the tip. Thus, the current signal alone indicates if the translation of the Dipolar Racer was successful without imaging the surface after each step.

In the end, the Dipolar Racer completed the 150 nm silver-surface racetrack in a record time of 1 hour and 33 minutes, travelling at an average speed of almost 112 nm h^{-1}. Seeing these values, one might think that this works extremely well, and it does, but if we take a closer look at the data from Toulouse, there is a lot of time and potential unexploited.

1.1.4 The human's capability to control the nanocar

At first glance, controlling the nanocar over the surface is easy and straightforward, but for humans it is impossible to predict the outcome for a specific action. In figure 1.5 the successful and failed pulling attempts for the complete race from Toulouse are shown, exhibiting a successful pulling rate of about 54%, which is almost random and the predictability is slightly better than a coin flip.

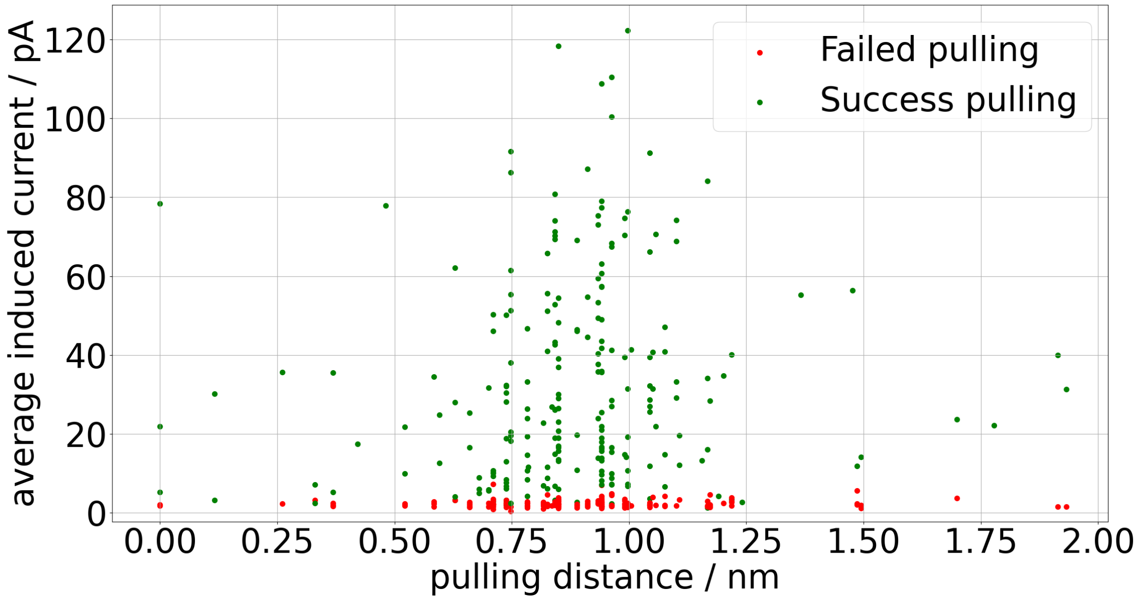


Figure 1.5: The race from Toulouse showed a pulling success rate of about 54%. A successful and failed pulling is indicated by either green or red dots respectively. A pulling action is considered to be successful, if the derivative of the current exceeds a certain threshold and failed otherwise. In general, the x-axis can be seen as the distance from the STM-tip to the nanocar or if the pulling action was successful - the travel distance of the nanocar.

Since the number of variables that have to be considered for its complex behaviour, it is extremely hard or impossible for humans to precisely control the nanocar. Thus, this would be a great opportunity to explore the performance of an artificial intelligence to manoeuvre the nanocar across the racetrack.

1.2 Artificial Intelligence

This chapter will provide you with the necessary concepts for this thesis and make you familiar with the kind of terminology that is used, when it comes to artificial intelligence or AI for short. However, since AI covers a very broad range of topics in the field of computer science, I will not go into much detail, as this would go beyond the scope of this master thesis. However, if you are highly interested in AI, there is a great book called *Artificial Intelligence: A Modern Approach* from *Stuart Russell, Peter Norvig*, on which parts of this chapter are based.

The understanding of *how we think* - meaning, how we perceive, predict, understand and process information has preoccupied humans for thousands of years.

The recent development regarding formulating algorithms that mimic thinking processes comprises a multitude of possibilities for solving highly complex problems, which are far beyond human's capability of solving. [9, p. 1]

The underlying potential to solve complex problems or finding meaning in seemingly random datasets, created a new field in computer science. This field is called artificial intelligence, which was invented in 1956 [9, p. 17] and is not just about understanding intelligence but also creating intelligent entities. During the 1990s, these created entities became known as "intelligent agents" [9, p. 26], which will be discussed in section 1.2.2.

1.2.1 Machine Learning

AI is a much broader field of study compared to machine learning (ML). In general, AI aims to make machines "intelligent" using multiple approaches and different learning algorithms, whereas ML focuses on making machines that can learn to perform tasks. Nevertheless, it is quite hard to define whether a machine or entity is intelligent, but it is clear that ML is a subfield of AI. [4, p. 3]

In the field of computer science, machine learning studies algorithms and techniques for automating solutions that are hard to program in computer language. A conventional program consists of two steps. During the first step, a detailed design for the program is created, in terms of *what* the program is supposed to do. During the second step, this detailed design has to be translated into a computer language. Despite a very clear and complete specification about the real environment, this second step is extremely challenging when it comes to real-world problems. This is where ML algorithms come into play. ML can solve many problems in a generic way, meaning that they do not require an explicit design or model of the real environment and are able to learn from data. [4, p. 2]

Machine learning can provide knowledge based on a large dataset by identifying patterns or regularities. This is done by algorithms that construct a statistical model based on the training data, but can also be applied to unknown datasets. [4, p. 4]

1.2.2 Intelligent Agents

The aim of the following section is to explain the terminology used in the field of AI. First and foremost, the concept and meaning of intelligent agents will be described by introducing the idea of an agent and the environment as well as the interaction between them. Moreover, the general terminology which is used in the field of AI research will be introduced.

How well an agent performs in a specific situation, strongly depends on the complexity of the task. However, an universal intelligence that is capable of solving each and every task does not exist.

1.2.2.1 Agent and Environment

The *Agent* is the computer program that is learning due to interactions with the *Environment*. The agent perceives the environment through sensors and operates upon it through actuators. [9, p. 34] This concept is illustrated in figure 1.6, where the agent is interacting and modifying the environment through the scanning tunnelling microscope (STM). Thus, it is immediately clear that for the precent case the STM is both sensor and actuator.

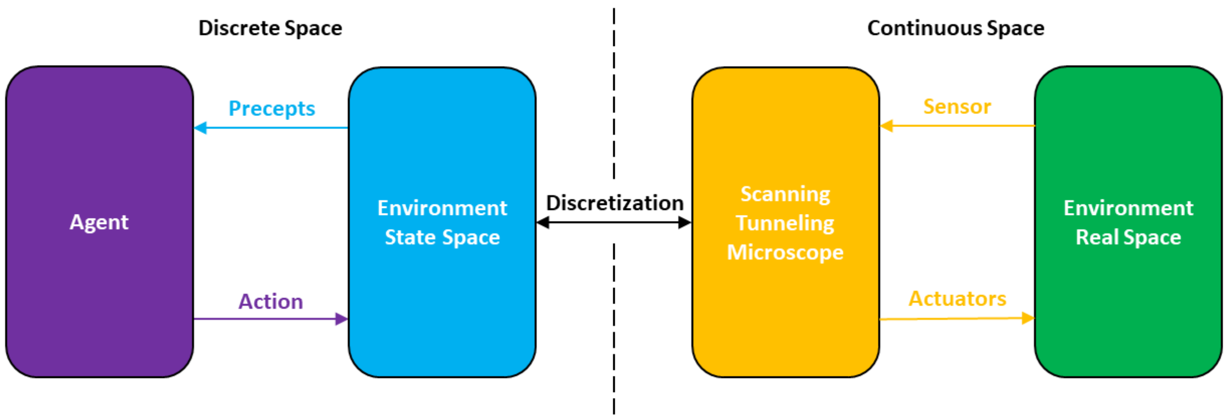


Figure 1.6: A schematic drawing of the agent interacting with the environment through the scanning tunnelling microscope, which functions as sensor and actuator.

The agent interacts sequentially with the environment, meaning there needs to be a notion of time to uniquely describe each time step. Thus, the system (agent + environment) starts at time 0 and is incremented by 1 before the next observation is received. [4, p. 196] As soon as the objective is achieved, the *episode* is finished.

The agent's choice of actions for a given situation, or *state*, can depend on the complete history of everything the agent has ever perceived. This perceived information for every time step is called *percept sequence.*

The agent's behaviour is described by the *agent function*, that maps any given state to an action. The agent is performing every action towards achieving the objective. Thus, the agent receives a *reward* for each action in order to determine its quality. Designing an excellent reward function is by no means trivial and highly influences the learning rate and performance of the agent.

1.2.2.2 Performance Measurement

An agent without any knowledge about the environment starts exploring the environment. At first, the agent performs random actions for which it gets feedback from the environment. This feedback is a numeric value, usually a real number and known as *reward.*

All agents are programmed with one objective: accumulating maximum reward from the environment due to the action that was taken. Thus, the agent has no direct knowledge about the environment, but it indirectly observes the environment via the reward function. This makes clear - how the *reward structure* is designed, depends on the task-specific objective. [9, p. 37]

1.2.2.3 The Nature of Environments

The state of the environment is a numerical description, which uniquely describes the environment at any given time. The *state* is described by a set of features called *state variables.* The state within the environment at a specific time is determined by the numeric values of these state variables.

The total number of possible environment states is given by the dot product of the number of values for each of the feature variables. E.g.: There are four feature variables and each contains 20 entries, then the state space of the environment or the total number of possible states is 160,000. This immediately implies the necessity of a discrete state space, as for real feature values the number of entries rapidly goes to infinity. This mapping from the *real space* to the *state space* is called *discretization.* This *discretization* is mostly caused by limited memory capacity. [4, p. 199]

Figure 1.7 shows the schematic drawing for how the agent is interacting with the discrete environment.

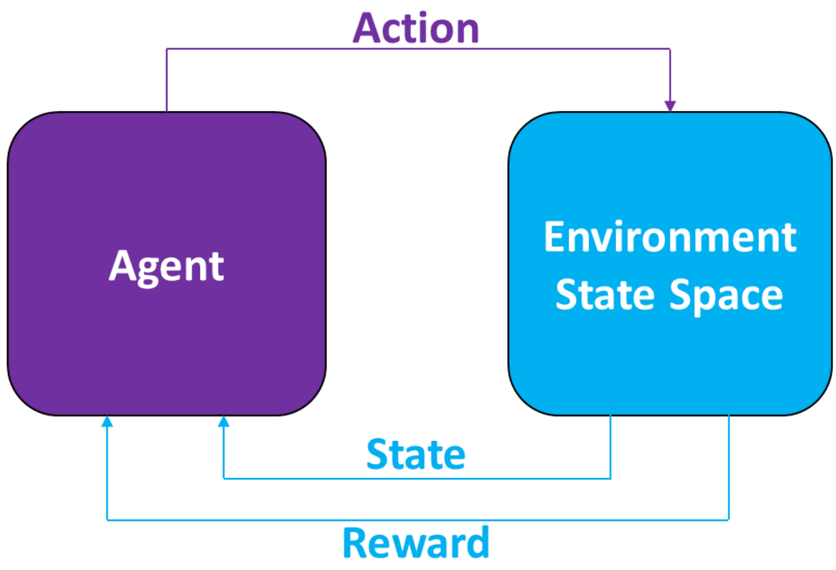


Figure 1.7: Interaction between agent and state space environments

At this point, it should be emphasised that the agent is not directly interacting with the real environment (*real space*), but the discrete environment *state space*. In other words, due to the discretization, the agent perceives the state space instead of the real space. A more detailed description about the environment is given in chapter 2.1.3.

The agent receives the state and the reward from the discrete environment and performs an action. After the action was performed, the time is incremented. Afterwards, the environment passes on the next state and reward to the agent. This creates a recurrent sequence of state s, reward r and action a $s_0, r_0, a_0, s_1, r_1, a_1, ...s_t, r_t, a_t, ...$, which is known as *trajectory*. A full trajectory from the initial state to the final state is known as *episode*. [4, p. 200]

1.2.2.3.1 Markov Decision Process

The agent-environment framing is described by a mathematical model known as Markov Decision Process (MDP). In order to formulate a finite MDP, the state space and action space has to be finite. The finite MDP is a model, where at any time t, from some state s_t and with some state transition probability, the system performs any action a_t that is available in this state s and for which a one-step reward *r(s,a)* is gained. [3, p. 3]

In a stochastic environment, the outcome is predictable for any given state and possible action within this state. If the chosen action at time t is independent of the history of all states or actions, up to t-1, then these states are known as *Markov states*. In reinforcement learning, we only consider environments that can be described in terms of Markov-states. Environments are described by Markov-state-environments because of their easy analysis and appliance to many real-world situations.

The mathematical description of MDPs is stated in chapter 1.3.1.

1.2.2.4 The Structure of Agents

The focus of AI is to design an agent program that maps from perceptions of the environments to actions. The architecture is made up of the computing device, sensors and actuators:

$$agent = architecture + program$$

There are numerous types of agents based on various methods for selecting actions to achieve certain objectives. The most interesting ones for reinforcement learning are called *learning agents*.

1.2.2.4.1 Learning Agents

In many areas of AI, learning agents are the state-of-the-art approach in creating intelligent agents. The huge advantage of learning agents is their ability to operate in initially unknown environments.

A learning agent, as shown in 1.8, can be divided into four conceptual components, which are known as *learning element*, *performance element*, *critic* and *problem generator*.

The performance element percepts an environment state and selects an action based on its knowledge.

The critic is rating the agents performance based on a performance standard.

The learning element receives feedback from the critic and determines how the actions should be modified to increase positive feedback in the future. The information gathered by the learning element is communicated with the knowledge of the performance element in order to update its knowledge data base.

The problem generator is suggesting new actions, that will lead to unknown responses from the environment and enable the agent to gather new experiences. This exploration of the environment will lead to suboptimal performance at first place; but it enables the agent to discover better actions for the future.

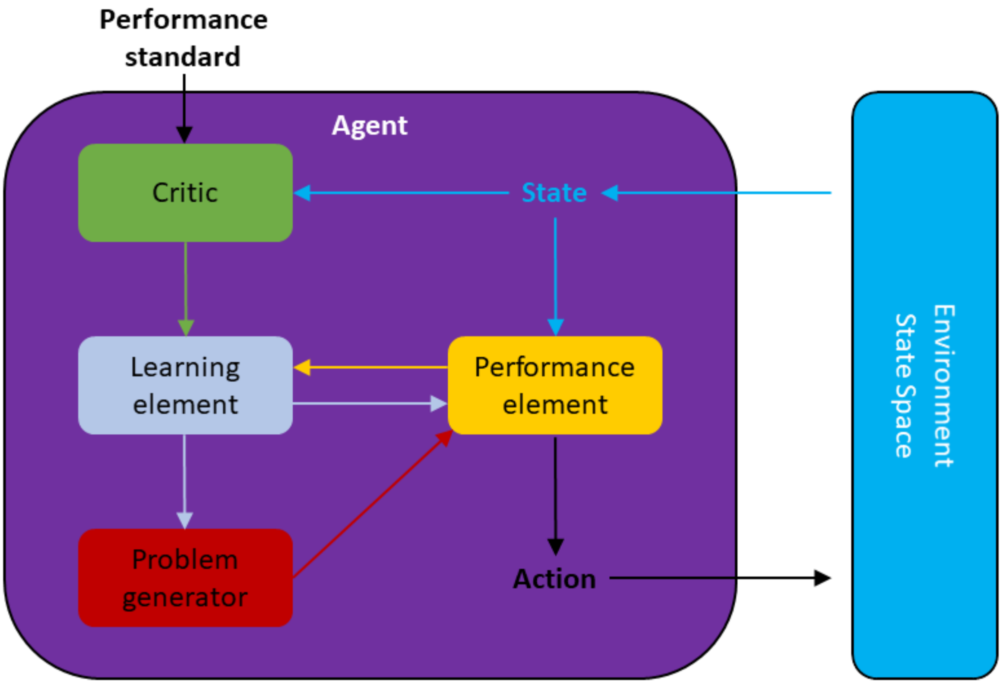


Figure 1.8: Conceptual components of a Learning Agent

Since we are now familiar with the terminology used when it comes to artificial intelligence, we can continue with reinforcement learning.

1.3 Reinforcement Learning

Reinforcement learning is learning by mapping states to actions, such that a numerical reward signal gets maximized. When the agent starts learning, it has no knowledge about the environment and does not know which actions are good or bad, so it discovers the environment by choosing random actions. Each action will then lead to a reward that judges the chosen action based on its performance. The goal of the agent is to accumulate the highest reward. These two characteristics, namely reward maximization and trial-and-error search, are the most important distinguishing features of reinforcement learning compared to other machine learning methods.

The problem of reinforcement learning is formalized using ideas from dynamical systems theory, known as the optimal control of incompletely-known Markov decision processes. The idea, as already mentioned in section 1.2.2, is to present the agent with an environment that captures the most significant aspects of the real problem, while interacting with this environment to achieve a goal. The agent must be able to observe the environment and take actions that affect its state within the environment, while also having a goal. Markov decision processes are intended to include these three features - perceive, action and goal. [11, pp. 1–2]

1.3.1 Finite Markov Decision Process

The following section will give a mathematical representation of finite Markov decision process (MDPs), which was already mentioned in section 1.2.2.3.1. This involves reward evaluation for choosing certain actions in specific situations. In MDPs, either the value function $V^*(s)$ of each state s is estimated by taking an optimal action a, or the state-value function $q^*(s,a)$ for each action in each state is estimated. This chapter closely follows [11, pp. 47–68].

1.3.1.1 Agent-Environment Interface

An MDP consists of a finite set of states, actions and rewards, noted as $(S,\ A,\ R)$ respectively. The agent interacts with the environment and at each time step t, the agent perceives some environment state $s_t \in S$ and selects an action based on this state $a_t \in A(s)$. After the action is performed, the time step is increased to $t+1$ and the agent receives a numerical reward $r \in R \subset \mathbb{R}$ and finds itself in a new state s_{t+1}. The state transition probability $p(s_{t+1}|s_t, a_t)$ is the probability that when performing action a_t in state s_t, the resulting state will be s_{t+1}, and is given by:

$$p(s_{t+1}|s_t, a_t) \doteq \sum_{r \in R} p(s_{t+1}, r|s_t, a_t) \tag{1.1}$$

The expected rewards for state-action pairs can be computed by:

$$r(s_t, a_t) \doteq \sum_{r \in R} r \sum_{s_{t+1} \in S} p(s_{t+1}, r|s_t, a_t) \tag{1.2}$$

1.3.1.2 Goals and Rewards

In reinforcement learning, the objective of the agent is formalized by a *reward* signal that is received from the environment. At each time step, the agent receives the reward as numerical value $r_t \in \mathbb{R}$. The goal is not to maximize the immediate reward, but the cumulative reward.

The use of a reward signal to formalize the idea of an objective is one of the most distinctive features of reinforcement learning. The formulation of a goal using only a reward signal might first appear to be limiting, but in practice it has proved to be flexible and applicable.

1.3.1.3 Returns and Episodes

The agent's goal is to maximize the cumulative reward received in the long run. Let us consider, the received rewards after time step t are denoted $r_{t+1}, r_{t+2}, r_{t+3}, \cdots$, then the maximum expected return, until the terminal state T is reached, is denoted G_t. In the simplest case, the return is the sum of the individual rewards:

$$G_t \doteq r_{t+1} + r_{t+2} + \cdots + r_T \tag{1.3}$$

After reaching time step T, the episode is finished. This type of return is useful for sequences with a terminal state. When this is not the case and the agent-environment interaction does not end and continues without any limitations, then the return formulated in this way is problematic, because for $T = \infty$ the return itself will be infinite.

Therefore, an additional factor has to be included in equation 1.3 to ensure that the expected *discounted return* is limited with increasing time steps.

$$G_t \doteq r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \cdots = \sum_{k=1}^{\infty} \gamma^k r_{t+k+1} \tag{1.4}$$

where γ is a parameter, $0 \leq \gamma \leq 1$, called the *discount factor*.

The discount rate can also show how relevant the immediate and the future reward is, as a discount rate will make future rewards worth only γ^{k-1} compared to immediate rewards.

1.3.1.4 Policies and Value Functions

Almost all reinforcement learning algorithms involve estimating *value functions* V(s), or *action-value functions* q(s,a) that estimate the expected future reward depending on what action is taken. The policy function π defines the action that the agent is going to perform for a certain state in the environment. Thus, a *policy* is a mapping from states in the environment to all possible actions that the agent can take, while every action has its probability to be chosen. The performance of the agent is represented by the expected reward $r(s_t, s_{t+1})$ under the policy function π. The function $V_\pi(s_t)$ is called *state-value function* for policy π:

$$V_\pi(s_t) = \mathbb{E}_\pi \Big[\sum_{k=1}^{\infty} \gamma^k r_{t+k+1} | s_t \Big] \tag{1.5}$$

The agent's objective is finding the best policy, which is the equivalent of accumulating maximum reward. This equation can be rephrased as the expected reward for taking action a_t in state s_t under policy π. The function $q_\pi(s_t, a_t)$ is called *action-value function* for policy π:

$$q_\pi(s_t, a_t) = \mathbb{E}_\pi\Big[\sum_{k=1}^{\infty} \gamma^k r_{t+k+1} | s_t, a_t\Big] \tag{1.6}$$

where $0 \leq \gamma \leq 1$ is the *discount factor* that determines the importance of rewards gained in the future.

1.3.1.5 Optimal Policies and Optimal Value Functions

The core of a reinforcement learning problem is finding a policy that achieves maximum reward in the long run. For finite MDPs, a policy π is defined to be better than or equal to another policy π' if its expected reward is greater than or equal to that of π', or in other words, if $v_\pi(s) \geq v_{\pi'}(s)$. There is always at least one policy that is better than or equal to all other policies. This is an optimal policy π_*.

Under an optimal policy, the state-value function is called the optimal *state-value function* v_*, and defined as:

$$v_*(s_t) \doteq \max_\pi v_\pi(s_t) \tag{1.7}$$

for $\forall s_t \in S$.

While their *optimal action-value function* q_* is defined as:

$$q_*(s_t, a_t) \doteq \max_\pi q_\pi(s_t, a_t) \tag{1.8}$$

for $\forall s_t \in S$ and $\forall a_t \in A(s)$.

Thus, we can write q_* in terms of v_* as follows:

$$q_*(s_t, a_t) = \mathbb{E}[r_{t+1} + \gamma v_*(s_{t+1}) | s_t, a_t]. \tag{1.9}$$

1.3.2 Temporal Difference Learning

Temporal Difference (TD) algorithms can learn directly from raw experience or datasets, either generated by other AIs or by humans without the necessity of modelling the environment dynamics. TD methods update their estimates based on already learned estimates to adjust and make more accurate predictions about the future, without waiting for the end of an episode, as it is the case in Monte Carlo methods. Updating the learned values immediately is known as *bootstrapping*.

The policy evaluation or *prediction* problem deals with the estimation of the value function v_π for a given policy π, while the *control* problem focuses on iteratively finding an optimal policy.

The value function gets updated for the next time step $t + 1$ by comparing the difference between the observed reward r_{t+1} and the estimate $V(s_{t+1})$:

$$V(s_t) \leftarrow V(s_t) + \alpha[\underbrace{r_{t+1} + \gamma V(s_{t+1})}_{TD\ target} - V(s_t)] \tag{1.10}$$

This method is called TD(0) or one-step TD, because it is updated immediately at the transition s_{t+1} by using the reward received in the next time step r_{t+1}.

This is a special case of the general TD(λ) method, where λ is a decay parameter with $0 \leq \lambda \leq 1$. For $\lambda = 1$ every value function $Q(\cdot, \cdot)$ that was visited during the episode gets updated at the end of the episode, we call this *Monte Carlo* (MC) methods. [10]

1.3.2.1 Q-Learning

The reinforcement learning algorithm that is most suitable for our purpose is called the Q-Learning algorithm and is based on Temporal Difference Learning. In temporal difference learning, an entry in the lookup table gets updated for every time step t by using the Q-learning algorithm, whose core is the Bellman equation 1.11.

The state at time t be s_t. The decision process begins at time 0 in the initial state s_0. At any time t, the possible action depends on the current state $a_t \in \Gamma(s_t)$, where the action a_t represents one or more control variables. After action a is taken, the state changes from s to a new state T(s,a) and the current pay-off from taking action a in state s is F(s,a). The discount factor $0 \leq \beta \leq 1$ is representing impatience.

$$V(s) = \max_{a \in \Gamma(s)} [F(s, a) + \beta V(T(s, a))] \tag{1.11}$$

Q-Learning is based on temporal difference learning and is a model-free approach of reinforcement learning. It enables an agent to act optimally in Markov Decision Processes by experiencing reward based on actions taken and without requiring a model for the environment.

Learning is considered to be off-policy, because the learned action-value function $Q(s_t, a_t)$ directly approximates the optimal action-value function q_* by taking the best action in the particular state s_t. This is known as a *greedy policy.* However, the policy still has an effect in determining which state-action pairs $Q(s_t, a_t)$ are visited and updated. An action-value function $Q(s_t, a_t)$ is updated by the following equation, which is based on the Bellman equation.

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha[\underbrace{r_{t+1} + \gamma Q(s_{t+1}, a)}_{Q-Learning\ target} - Q(s_t, a_t)] \tag{1.12}$$

The agent's next action a_{t+1} is chosen using the behaviour policy $a_{t+1} \sim \mu(\cdot|s_t)$, but the update of $Q(s_t, a_t)$ is performed using an alternative successor action a under policy π, $a \sim \pi(\cdot|s_t)$. Both, the behaviour policy μ and the target policy π, were updated. The target policy π is greedy with respect to $Q(s_t, a_t)$

$$\pi(s_{t+1}) = \arg\max_a Q(s_{t+1}, a) \tag{1.13}$$

and the behaviour policy μ is a greedy policy with respect to $Q(s_t, a_t)$. This is also the reason why Q-learning is off-policy. The action-values $Q(s_t, a_t)$ were updated using the next state action-values $Q(s_{t+1}, a)$ and the greedy action a. The *Q-Learning target* under an ϵ-greedy policy is given by:

$$\begin{aligned}
&\rightarrow \quad r_{t+1} + \gamma Q(s_{t+1}, a) && (1.14)\\
&= \quad r_{t+1} + \gamma Q(s_{t+1}, \arg\max_a Q(s_{t+1}, a)) && (1.15)\\
&= \quad r_{t+1} + \gamma \max_a Q(s_{t+1}, a) && (1.16)
\end{aligned}$$

In the end, substituting this expression for the *Q-Learning target* of equation 1.12, the Q-Learning Control Algorithm is given as:

$$Q^{new}(s_t, a_t) \leftarrow \underbrace{Q(s_t, a_t)}_{old\ value} + \underbrace{\alpha}_{learning\ rate} \overbrace{\underbrace{[\underbrace{r_{t+1}}_{reward} + \underbrace{\gamma}_{discount\ factor} \underbrace{\max_a Q(s_{t+1}, a)}_{estimate\ of\ optimal\ future\ value}}_{new\ value\ (temporal\ differenc\ target)} - \underbrace{Q(s_t, a_t)]}_{old\ value}}^{temporal\ difference} \tag{1.17}$$

By visiting all states and trying all actions repeatedly, it learns which actions are the best in each state. Thus, Q-Learning Control converges to the optimal action-value function $Q(s_t, a_t) \rightarrow q_*(s_t, a_t)$. [12, pp. 282–285]

This equation builds the core of the agent. Therefore, let us take a closer look and understand the meaning behind the equation, and how it can be tuned using the two hyperparameters α and γ. For the update process, we add the temporal difference times the learning rate α to the old Q-value $Q(s_t, a_t)$. The temporal difference includes the next step reward, which is received after action a_t was performed, plus the discount factor γ times the optimal future Q-value, which is the Q-value for the next state with the action that achieved the most reward. This is called the temporal difference target, which gets subtracted from the old Q-value.

The two hyperparameters α and γ get adjusted over time, as the agent's knowledge about the environment increases.

- α: essentially determines how important or high-weight future rewards are
- γ: determines how impactful already established action-value functions are compared to newly learned ones

The procedural form of the Q-learning algorithm is shown below:

Definition 1.3.2.1 Q-Learning (off-policy TD control) for estimating $\pi \sim \pi_*$

Algorithm parameters: step size $\alpha \in (0, 1]$, small rate of exploration $\epsilon > 0$
Initialize Q(s, a), for all s $\in \mathbb{S}^+, a \in \mathbb{A}(s)$, arbitrarily except that $Q(terminal, \cdot) = 0$
Loop for each episode:
    Initialize S
    Loop for each step of episode:
        Choose A from S using policy derived from Q (e.g. ϵ-greed policy)
        Take action A, observe R, S'
        $Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha[r_{t+1} + \gamma \max_a Q(s_{t+1}, a) - Q(s_t, a_t)]$
        $s_t \leftarrow s_{t+1}$
    until S is final

1.3.2.2 Q-Table

These action-value functions $Q(s_t, a_t)$, or Q-values, are stored in a Q-table. The Q-table is a multidimensional array, where the states can be seen as the pages of the array and the actions are the entries within a page.

In order to understand how Q-Learning updates its Q-values, the famous Taxi problem from the OpenAI Gym library is introduced [2]. The following figure 1.9 shows the Taxi environment, which consists of different fields either directly connected or separated by walls. At the start of an episode, the passenger and the taxi randomly spawn at the field, but the passenger can only spawn at one of the four possible pick-up or destination locations (R, G, Y, B), while the taxi can spawn anywhere except at the passenger location. The goal of the agent, alias the taxi-driver, is to pick up a passenger from one of the four locations and drop him off in another. The total number of states in this environment is given by the grid size 5x5, time another 5 for the possible locations of the passenger, namely the 4 pick-up locations and the location inside the taxi time another 4 for the four destination locations. The agent controls the taxi by using the six possible actions (down, up, right, left, pick-up, drop-off), which are chosen according to the entries in the Q-table. The taxi environment and the corresponding Q-table are shown in figure 1.9. The reward for a successful drop-off is +20, and -1 for every time step it takes. There is also a 10 point penalty for illegal pick-up and drop-off actions and also for driving into walls.

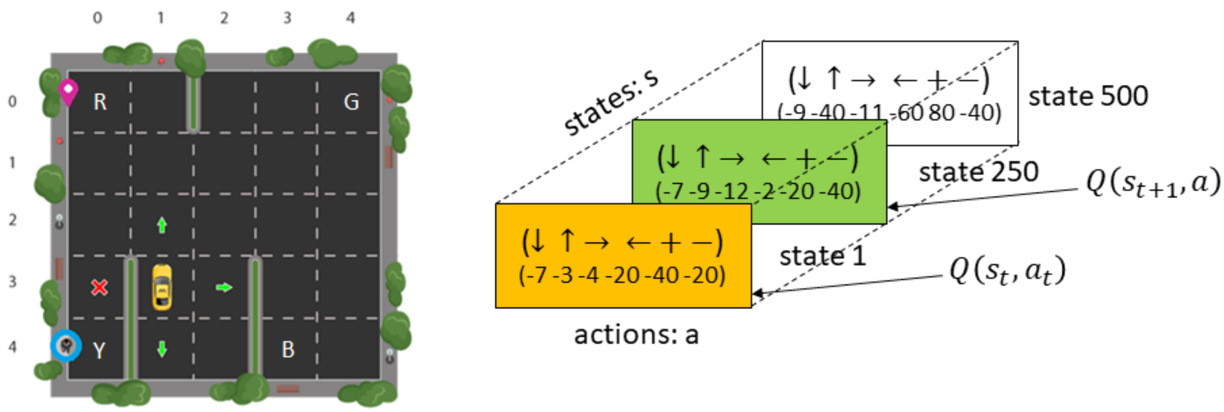


Figure 1.9: Left: OpenAI Gym Taxi environment [7] Right: The Q-table for the Taxi environment

The complete update process for a Q-table entry is given below. Assuming that the taxi is positioned as it is shown in figure 1.9 and moves up.

1. Being in state $s_t = 1$ (yellow page) and when performing action $a_t = \uparrow$, we end up in the next state: $s_{t+1} = 250$ (green page)
2. We receive a reward r_{t+1} from the environment that judges the quality of the action. The received reward is -1, because every time step is rewarded with -1. This encourages the agent to find the shortest way possible.
3. The Q-value at state 1, action $\uparrow$, given by $Q(1, \uparrow)$, gets updated by the Q-Learning algorithm
4. The update procedure adds the temporal difference target to the old Q-value. This includes the received reward $r_{t+1} = -1$ and the highest Q-value entry from the next state $\max_a Q(250, a) = -2$, which was taken from figure 1.9.

5. With the two hyperparameters being $\alpha = 0.95$ and $\gamma = 0.5$ the new Q-value is given as:

$$\begin{aligned}
Q^{new}(1,\uparrow) &\leftarrow Q^{old}(1,\uparrow) + 0.95\left[r_{t+1} + 0.50 \cdot \max_a Q(250,a) - Q^{old}(1,\uparrow)\right] \\
Q^{new}(1,\uparrow) &\leftarrow -3.00 \qquad + 0.95\,[-1.00 + 0.50 \cdot (-2.00) - (-3.00)\ ] \\
Q^{new}(1,\uparrow) &\leftarrow -2.05
\end{aligned}$$

This concludes the introduction chapter and provides all the information to understand the basics of reinforcement learning and the used Q-learning algorithm. This theoretical knowledge will be used to develop an AI written in the programming language Python.

2 Python Code Development

In the following chapters, I will explain how I designed an AI, which is capable of maneuvering a nanocar across a racetrack using a low-temperature scanning tunnelling microscope.

This chapter represents the Python code of the *Q-Learning* based AI. The Python code shows how the agent can learn either from human experience by using already existing datasets or by controlling the nanocar directly at the STM.

The first part of this chapter shows how the agent can **control the nanocar by using the STM**. The STM is connected to the agent via the OLE Control Interface. OLE (Object Linking & Embedding) is a protocol developed by Microsoft, that allows embedding and linking to objects. These objects can implement interfaces to export their functionality - like enabling a Python program to make use of these objects.

The second part of this chapter shows how the agent can **learn from human generated data**. This was implemented, because of the short time the STM was available and to train the agent beforehand from already existing data, like the nanocar race from Toulouse, and to use a pre-trained agent to drive at the STM.

In order to check how the agent would perform at the STM, a simulator was implemented, that provides a quick feedback on how the agent would select actions. Of course the simulation does not provide a physical feedback, which means it can not be used for training the agent, but it will represent the current learning state of the agent. The code of the simulator is given in the appendix 4.

The following chapters provide a fully annotated Python code as well as graphical representations to complement the code in order to give you a better imagination and allow for a much easier understanding.

2.1 Controlling the nanocar by the STM

This chapter explains the Python code I developed to manoeuvre a nanocar across a silver (111)-surface by giving commands to control the scanning tunnelling microscope. The code is explained by going through it one by one. First, the lowest level (hardware) is explained, followed by the GUI and the environment and, last but not least, the agent. In this way, every function that is used in a class is already explained beforehand and the code becomes more clear.

The overview of the code is illustrated by the flow diagram 2.1.

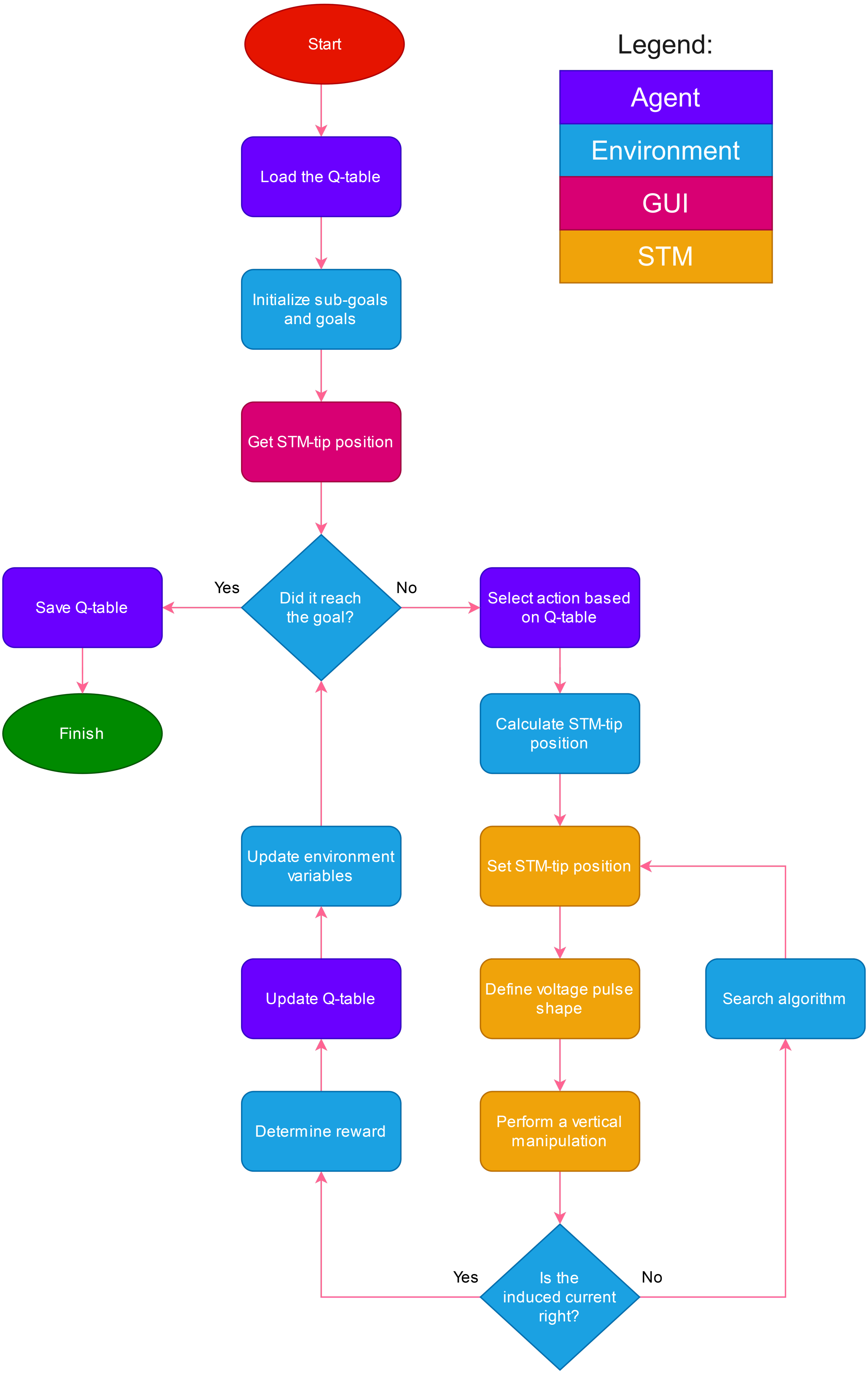


Figure 2.1: The flow diagram for manoeuvring the nanocar across a given race-track by controlling the STM. The Legend indicates to which class a processes belongs.

2.1.1 The Python to STM interface

The STM class utilizes the OLE control interface to connect with the STM and perform actions according to the agent's target. This should be seen as an interface class, where commands are rephrased to use the OLE control interface provided by Createc. The advantage here is that for an OLE enabled device the existing STM-class can be swapped out, while still being able to use the rest of the code. However, depending on your system, you will also have to adjust the threshold values in the environment class.

The STM is connected via *Ethernet* to the STM. For the OLE control interface to function, two packages are required: the *win32com.client* package, which contains a number of modules to provide access to automation objects, and the *pythoncom* package, which initializes COM-ports (hardware interface).

2.1.1.1 The code of the Python to STM interface

```
import numpy as np
import math
import glob
import os
import time
import pythoncom
import logging
import win32com.client
import matplotlib.pyplot as plt
from scipy import signal

class STM(object):
    """
    The class sends commands to the STM by using the OLE control protocol and interacts with the
    STMAFM software.

    Comment: If you want to see the available methods in python use dir(stm) and for properties use
    stm._prop_map_get_

    Methods
    -------
    connect()
        Initializes the connection to the STM/AFM program.

    update_parameters()
        Updates all parameters and synchronizes the parameters with the DSP (dual digital feedback
        controller).

    get_date()
        Reads the date from the STM.

    beep()
        Makes a beep sound and writes 'Beep' into the log-file.

    get_float_param(name)
        Reads the parameter specified by name and tries to convert it to float. The parameter is a
        string and has to be within the 'Basic Parameter'-Frame of the STM/AFM software.

    set_position()
        This sets the new position of the STM.

    get_relative_position()
        Returns the actual relative STM position.

    get_absolute_position()
        Returns the actual absolute STM position.

    define_voltage_pulse()
        Defines the voltage pulse.

    perform_vertical_manipulation()
        This performs a vertical manipulation and generates a current spectrum.

    perform_lateral_manipulation()
        This performs a lateral manipulation and creates a Z-topography. This is used for searching.
```

```

    get_current_spectrum()
        Returns the current spectrum.

    is_idle()
        Checks the status of the STM and returns true when idle.

    is_busy()
        Checks the status of the STM and returns true when busy.
    """
    def __init__(self):
        self.logger = logging.getLogger("STM")
        self.pos_STM = []
        self.voltage_STM = 0
        self.val_Current = 0
        self.val_Current_Duration = 0

    def connect(self):
        """
        Initializes the connection to the STM/AFM program.
        """
        self.logger.info("Connecting to STM")
        # Initializes the COM libraries for the calling thread
        pythoncom.CoInitialize()
        self.stm = win32com.client.Dispatch("pstmafm.stmafmrem")
        self.stm.serverneverclose()
        self.beep()
        self.update_parameters()

    def update_parameters(self):
        """
        Updates all parameters and synchronizes the parameters with the DSP (dual digital feedback
        controller).
        """
        self.logger.info("Synchronize all parameters with DSP")
        self.stm.updatedspfbparam()
        self.stm.updatedspparam()



    def get_date(self):
        """
        Reads the date from the STM.
        """
        date = self.stm.date
        self.logger.info("read current date: %s" % date)
        return date

    def beep(self):
        """
        Makes a beep sound and writes 'Beep' into the log-file.
        """
        self.logger.info("Beep!!!")
        self.stm.stmbeep()

    def get_float_param(self, name):
        """
        Reads the parameter specified by the argument and tries to convert it to float. The
        parameter is a string as it appears in the Basic Parameter form.

        Parameters
        ----------
        name : str
            String given by the Basic Parameter in the STM/AFM program and the menu
            bar under 'Forms' -> 'Basic Parameters'.

        Returns
        -------
        value : float
            The requested value from the STM as float variable - if possible.
        """
        value = self.stm.getparam(name)
        try:
            value = float(value)
        except:
            self.logger.error("$s cannot be read" % name)
        else:
```

```
            self.logger.info("read %s of %s" % (name, value))
        return value

    def set_position(self, pos_STM):
        """
        Moves STM-tip to new position. Coordinates are given in relative DAC units (relative: X,Y
        Offset and rotation are added afterwards) Control is returned after the move has been
        completely finished.

        Attributes
        ----------
        pos_STM : np.array(2)
            The position from the environment in DAC units.

        Functions
        ---------
        stm.move_tip_relofs(x_dac, y_dac, 2000.0, 0.0))
            1 | x_dac | single | X new position in relative DAC units
            2 | y_dac | single | Y new position in relative DAC units
            3 | Speed | single | Speed in DAC units/s
            4 | Units | integer| reserved
        """
        x_dac, y_dac = float(pos_STM[0]), float(pos_STM[1])
        self.stm.move_tip_relofs(x_dac,y_dac,2000,0)
        self.update_parameters()

    def get_relative_position(self):
        """
        Gets the relative position of the STM in DAC units.

        Functions
        ---------
        get_float_param('name')
            Returns the value of the standard parameter you passed over to the STM/AFM program.

        Return
        ------
        relative_stm_position : np.array(2)
            The relative position of the STM-tip.
        """
        relative_stm_position = np.array(np.zeros(2))
        relative_stm_position[0] = self.get_float_param('VertSpecPosX')
        relative_stm_position[1] = self.get_float_param('VertSpecPosY')
        return relative_stm_position

    def get_absolute_position(self):
        """
        Gets the absolute position of the STM in DAC units.

        Functions
        ---------
        get_float_param('name')
            Returns the value of the standard parameter you passed over to the STM/AFM program.

        Return
        ------
        absolute_stm_position : np.array(2)
            The absolute position of the STM-tip.
        """
        X_Offset = self.get_float_param('OffsetX')
        Y_Offset = self.get_float_param('OffsetY')
        X_Relativ = self.get_float_param('VertSpecPosX')
        Y_Relativ = self.get_float_param('VertSpecPosY')
        absolute_stm_position = np.array(np.zeros(2))
        absolute_stm_position[0] = X_Offset+X_Relativ
        absolute_stm_position[1] = Y_Offset+Y_Relativ
        return absolute_stm_position

    def define_voltage_pulse(self):
        """
        Sets the shape of the voltage pulse for controlling the nanocar. Voltage and time parameters
        are set individually to generate the voltage pulse.

        Parameter
        ---------

        Functions
```

```
        ---------
        stm.setparam('name', 'value')
            Sets the parameter called name to the desired value.
        """
        # Sets the duration of the voltage pulse:
        # The time per datapoint:
        # t_datapoint = DSP-Cycles (50kHz) x Vertmandelay = 0.02ms x Vertmandelay
        # The total time:
        # t = t_datapoint x number_of_datapoints = 0.02ms x 100 x 1000 = 2s

        # Zoffset: 54=0.5A, 65=0.6A, 76=0.7A, 87=0.8A, 98=0.9A, 109=1.0A, 271=2.5A
        self.stm.setparam('Zoffset','271')
        self.stm.setparam('Vertmandelay','100')
        self.stm.setparam('Vertmangain','9')

        self.stm.setparam('Vpoint0.t','0')
        self.stm.setparam('Vpoint1.t','5000')
        self.stm.setparam('Vpoint2.t','0')
        self.stm.setparam('Vpoint3.t','0')
        self.stm.setparam('Vpoint4.t','0')
        self.stm.setparam('Vpoint5.t','0')
        self.stm.setparam('Vpoint6.t','0')
        self.stm.setparam('Vpoint7.t','0')

        self.stm.setparam('Vpoint0.V','1800')
        self.stm.setparam('Vpoint1.V','1800')
        self.stm.setparam('Vpoint2.V','0')
        self.stm.setparam('Vpoint3.V','0')
        self.stm.setparam('Vpoint4.V','0')
        self.stm.setparam('Vpoint5.V','0')
        self.stm.setparam('Vpoint6.V','0')
        self.stm.setparam('Vpoint7.V','0')

        self.stm.setparam('Zpoint0.t','0')
        self.stm.setparam('Zpoint1.t','0')
        self.stm.setparam('Zpoint2.t','0')
        self.stm.setparam('Zpoint3.t','0')
        self.stm.setparam('Zpoint4.t','0')
        self.stm.setparam('Zpoint5.t','0')
        self.stm.setparam('Zpoint6.t','0')
        self.stm.setparam('Zpoint7.t','0')

        self.stm.setparam('Zpoint0.z','0')
        self.stm.setparam('Zpoint1.z','0')
        self.stm.setparam('Zpoint2.z','0')
        self.stm.setparam('Zpoint3.z','0')
        self.stm.setparam('Zpoint4.z','0')
        self.stm.setparam('Zpoint5.z','0')
        self.stm.setparam('Zpoint6.z','0')
        self.stm.setparam('Zpoint7.z','0')

        self.stm.updatedspmanipparam()

    def perform_vertical_manipulation(self):
        """
        Takes a vertical manipulation spectrum at the current image point X,Y. Control is returned
        after the spectrum has been completely finished. The tip remains at the current lateral
        position and the current signal is captured.

        Functions
        ---------
        stm.vertspectrum
            Takes a Vert.Spectrum at the current image point X,Y. Control is returned after the
            spectrum has been completely finished. The tip remains at the current lateral position.

        stm.vertsave
            Saves the current vertspecdata.
        """
        # Measures a vertspectrum at current position
        self.define_voltage_pulse()
        self.stm.vertspectrum()
        self.stm.vertsave()

    def perform_lateral_manipulation(self, start, end, steps):
        """
        Takes a lateral manipulation spectrum between start and end point where steps defines the
        number of measured points.
```

```

        Functions
        ---------
        latmanipxymove(Xstart, Ystart, Xend, Yend, steps, delay, preampgain,
                       biasvoltage, currentset)
        1 | Xstart      | integer | X start position in relative DAC units
        2 | Ystart      | integer | Y start position in relative DAC units
        3 | Xend        | integer | X end position in relative DAC units
        4 | Yend        | integer | Y end position in relative DAC units
        5 | steps       | integer | Number of steps
        6 | delay       | integer | Delay between steps in DSP Cycles
        7 | preampgain  | integer | Gain of Preamp during manipulation
        8 | biasvoltage | integer | Bias Voltage during manipulation
        9 | currentset  | integer | Current set point during manipulation in
                                    constant current mode

        Returns
        -------
        data : list([steps])
            Contains the Z-topography between start and end.
        """

        self.stm.setparam('Latmanmode','1')
        self.stm.setparam('Latchannelselectval', '1052673')
        self.stm.setparam('LatmanVolt','1000')
        self.stm.setparam('Latmangain','9')
        self.stm.setparam('Latmanlgi','12')
        self.stm.setparam('Latmanddx','12')
        self.update_parameters()
        self.stm.latmanipxymove(start[0], start[1], end[0], end[1], steps, 10, 9, 1000, 12)
        self.update_parameters()
        data = self.stm.latmandata(15,2)
        data = np.ravel(data)
        return data

    def get_current_spectrum(self):
        """
        Reads the current spectrum from the ADC channels of the STM/AFM program.

        stm.vertdata(channel,units)
            1 | channel | integer   | 0:Time in sec == 1:X == 2:Y == 3:Current_I
            2 | units   | integer   | 0:Default == 1:Volt == 2:DAC == 3:Ampere ==
                                      4:nm == 5:Hz

        Returns
        -------
        val_I : list([number of datapoints])
            Contains the current spectrum.
        """

        # Reads time signal in default units
        val_t = self.stm.vertdata(0,0)
        self.update_parameters()

        # Reads current signal from channel (ADC0) in DAC units
        val_I = self.stm.vertdata(3,2)
        return val_t, val_I

    def is_idle(self):
        """
        Checks the status of the STM and returns true when idle.
        """

        status = self.stm.scanstatus
        self.logger.info("STM status: %i" % status)
        if status == 0:
            self.logger.info("Checking STM status: STM is idle")
        else:
            self.logger.info("Checking STM status: STM is busy")
        return status == 0

    def is_busy(self):
        """
        Checks the status of the STM and returns true when busy.
        """
        return not self.is_idle()
```

2.1.2 The graphical user interface for environment initialization

The graphical user interface (GUI), shown in figure 2.2, allows to adjust the number of intermediate goals, also known as sub-goals, and shows a button that reads the current STM-tip position from the STM/AFM software (v.4.3) to initialize the environment. The initialization is done manually by measuring a vertical manipulation spectrum by using the "Single Spectrum" button within the software. A vertical manipulation spectrum measures the current signal at a specific x/y position on the surface and initializes the environment positions for the agent. The initialization spectra are saved as ".VERT-file" in the STM/AFM software. In the end, the necessary parameters for this initialization process can be loaded from the previously saved ".VERT-files" by right clicking the data and select "Load File with All Parameters" or by double clicking it. When all the goal positions are initialized, the GUI closes automatically and the agent takes control of the STM.

The agent needs the starting and goal position of the environment. The GUI is used to add additional sub-goals, because depending on the topography of the surface, it can be necessary to have sub-goals to manoeuvre around obstacles.

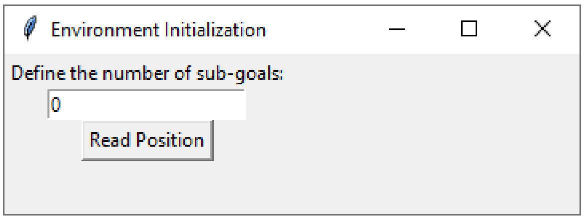


Figure 2.2: GUI to initialize the environment. The number of sub-goals is set in the textbox and a button click reads the relative position (VertX, VertY) of the currently loaded VERT-file.

2.1.2.1 The code of the GUI

```python
from stm import STM

import numpy as np
import collections
import logging
import random
import threading
import time
import tkinter as tk
import os

# Defines the settings for logging
logging.basicConfig(level=logging.INFO,
                    format='%(asctime)s - %(name)s - %(levelname)s - %(message)s',
                    filename='app.log')
console = logging.StreamHandler()
console.setLevel(logging.INFO)
formatter = logging.Formatter('%(name)-12s: %(levelname)-8s %(message)s')
console.setFormatter(formatter)
logging.getLogger('').addHandler(console)

class GUI(tk.Frame, threading.Thread):
    """
    The class visualizes the GUI. The button is used to read the position of the currently loaded
    VERT-file and the number in the textbox defines how many sub-goals the course has. This
    positional data is used to initialize the environment.

    Attributes
    ----------
    tk.Frame :  class
        A widget container from Tkinter.

    Methods
    -------
    create_widgets()
        Creates the button to initialize the environment.

    button_pressed()
        When pressed, the current STM-tip position is read and saved in an array.

    on_close()
        When the GUI is closed, the main window gets terminated and the AI takes
        control of the STM.
    """
    def __init__(self, stm, master=None):
        self.stm = stm
        super().__init__(master)
        threading.Thread.__init__(self)


        self.logger = logging.getLogger("GUI")
        self.master = master
        self.grid(column=0, row=0)

        self.master.protocol("WM_DELETE_WINDOW", self.on_close)

        # Defines the number of positions (>=2) to define the environment: start, goal
        self.number_of_points_in_environment = 2
        self.number_of_additional_points_in_environment = 0
        # Initializes the array to define the environment
        self.positions_to_define_environment = np.array(
                                               np.zeros([self.number_of_points_in_environment,2]))
        # Array index
        self.position_index = 0

        self.evt_get_position = threading.Event()
        self.evt_interrupted = threading.Event()
        self.evt_idle = threading.Event()

        self.start()

        self.create_widgets()

    def create_widgets(self):
        """
```

```
        Creates the initialization button in the GUI.
        """
        self.master.title("Environment Initialization")
        self.master.geometry('360x100')

        # Creates label and textbox
        tk.Label(self.master, text="Define the number of sub-goals:").grid(column=2, row=1)
        self.ent_number_additional_points = tk.Entry(self.master)
        self.ent_number_additional_points.grid(column=2, row=2)
        self.ent_number_additional_points.insert(0, '0')

        # Creates button
        self.btn = tk.Button(self.master)
        self.btn["text"] = "Read Position"
        self.btn["command"] = self.button_pressed
        self.btn.grid(column=2, row=4)

    def button_pressed(self):
        """
        When the button is pressed, the STM position is read from the latest loaded VERT-file.
        """
        if self.evt_idle.is_set():
            self.logger.info("button pressed")
            self.get_position()

    def on_close(self):
        """
        When the GUI is closed, the main window and all its widgets are terminated.
        """
        self.evt_interrupted.set()
        self.master.destroy()

    def run(self):
        """
        This method is representing the thread's activity. The GUI is terminated when the
        environment is completely initialized.
        """
        self.stm.connect()
        self.logger.info("start loop")
        self.evt_idle.set()

        # Loops until the environment is completely initialized
        while (not self.evt_interrupted.is_set() or self.position_index
        == self.number_of_points_in_environment+self.number_of_additional_points_in_environment-1):
            self.number_of_additional_points_in_environment = int(
                                                            self.ent_number_additional_points.get())

            if ((self.number_of_additional_points_in_environment+2)
                > len(self.positions_to_define_environment)):

                self.positions_to_define_environment = np.append(
                    self.positions_to_define_environment,
                    np.array(np.zeros([self.number_of_additional_points_in_environment,2])), axis=0)
                print(self.positions_to_define_environment)
            elif (self.number_of_additional_points_in_environment+2) < len(self.
    positions_to_define_environment):

                self.positions_to_define_environment = np.delete(
                    self.positions_to_define_environment,
                    self.number_of_additional_points_in_environment, axis=0)
                print(self.positions_to_define_environment)

            if self.evt_get_position.is_set():
                self.evt_idle.clear()

                if self.stm.is_busy():
                    self.evt_get_position.clear()
                    self.stm.beep()
                    continue

                self.positions_to_define_environment[self.position_index] = self.stm.
    get_relative_position()
                print(self.positions_to_define_environment)
                self.evt_get_position.clear()
                self.position_index+=1
                self.start_time = time.time()

```

```
            self.evt_idle.set()

            if self.position_index==self.number_of_points_in_environment+self.
    number_of_additional_points_in_environment-1:
                self.evt_interrupted.set()

        print(self.positions_to_define_environment)
        self.logger.info("exit loop")
        self.evt_interrupted.set()
        self.master.destroy()

    def get_position(self):
        """
        This method reads the tip-position from the currently loaded VERT-file.
        """
        if self.evt_idle.is_set():
            self.evt_get_position.set()
        else:
            self.logger.info("Can't get position: Device is not idle")

    def get_environment_positions(self):
        """
        Returns the initialized environment positions.
        """
        return self.positions_to_define_environment
```

2.1.3 The design of the environment

The environment contains all the information the agent needs to interact with the real world. The environment is a representation of the environment in the real world environment, but of course limited in the sense that only necessary information is tracked and synchronized between environments using the STM as sensor and actuator; like it is illustrated in figure 1.6.

The following chapter explains how the environment is designed. The Python code is described alongside with the schematic illustration 2.3 to allow for easier understanding. The schematic shows two situations, one where the manipulation step was successful and another where the manipulation step was unsuccessful, which means the nanocar translated undefined across the surface and has to be found again using a search algorithm.

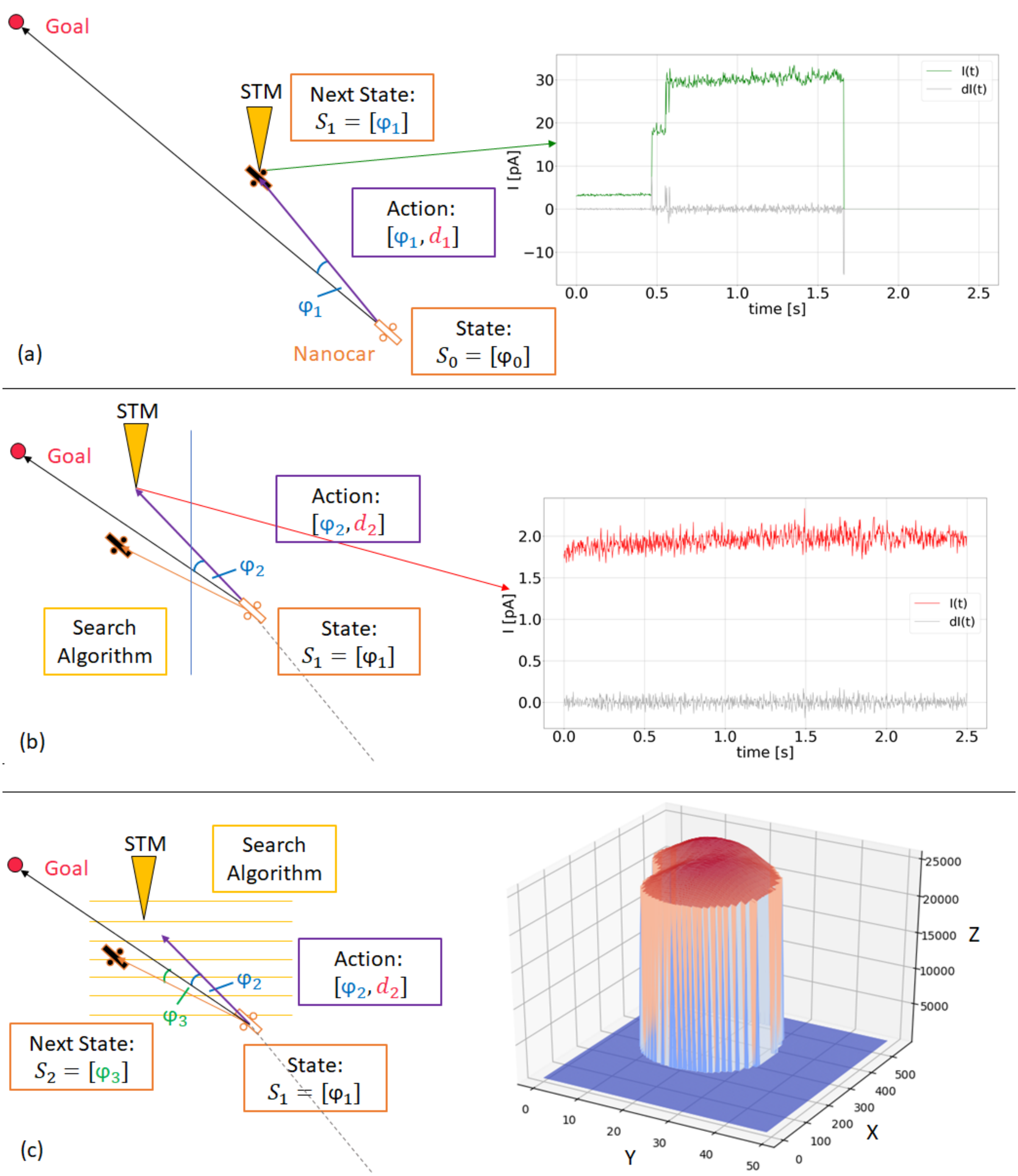


Figure 2.3: This schematic shows all the states and actions for (a) a successful manoeuvre step followed by an (b) unsuccessful manoeuvre step, for which the nanocar has to be (c) located by using the search algorithm. The first two graphs on the right represent the induced current spectra and its derivatives for a successful and a failed pulling action and the third graph represents the Z-topography of the nanocar after the search algorithm is completed.

In every time t, we know the position of the nanocar and the position of the goal. With this information, the current state of the nanocar can be determined and the agent chooses the *best action* in this particular state. Note: How the *best action* is evaluated, is part of the agent program and will be explained in the next chapter. The performed *action* from the agent's perspective is limited by the positioning of the STM-tip; being the most critical part anyway, and it has no control over the voltage pulse itself - which it could, but that would also increase the complexity.

To (a): We do not know how a state is defined yet, but let us assume the nanocar is currently determined by state φ_0. Then depending on this *state* φ_0, the agent chooses the *best action*, which determines where the STM-tip is positioned to pull the nanocar towards the tip. An *action* consists of an angle φ and a distance d. φ is defined as the angle between two vectors, namely the vector from **nanocar to goal** and from **nanocar to STM-tip**.

After the STM-tip is positioned at φ_1, d_1, a voltage pulse is applied for 2.5 s and an amplitude of 1.800 V. The high voltage at the sharp STM-tip creates a high electric field, which interacts with the dipole of the nanocar and attracts it towards the tip.

When the nanocar has moved below the tip while applying the voltage pulse, the tunnelling current drastically increases due to the decreasing tunnelling distance. Experiments and a lot of practice showed a successful step can be ensured, if the the derivative in the current shows a **significant step**. The performed action φ_1, d_1 indicates a successful pulling step, because of the relatively high derivative of the tunnelling current. The *next state* is simply given by the angle of the just performed action φ_1.

To (b): Now, the nanocar is in state φ_1 and the agent's *best action* is φ_2, d_2. After performing this action and applying the voltage pulse, the nanocar translates over the surface and no change in the tunnelling current is measured. Thus, the nanocar moved to some random position and got lost.

To (c): In order to find its position, a **search algorithm** kicks in. The algorithm performs multiple successive lateral manipulations, such that a square of 5 nm (twice the nanocar size) is scanned. This square is centred at half the trajectory of the previous known nanocar position and the latest tip position (where the nanocar should be when it would not be lost). The search algorithm creates a Z-topography of this area and calculates its centre of mass. The centre of mass for this area corresponds to the centre of the nanocar and hence its position is found. The parameters of the lateral manipulation are such that the position and orientation of the nanocar remains unchanged. In this case, the *next state* is not defined by the angle φ_2 of the just performed action, as it was before, but the angle φ_3, that was determined based on the position obtained by the search algorithm.

2.1.3.1 The reward function

Strictly speaking, the reward function is everything the agent perceives from the environment. There is no position the agent observes or current spectrum it measures. There is only the reward function it receives after every action and which determines how good or bad the performed action was.

As a consequence, the reward function determines the behaviour of the agent within the environment and is the most important choice to make in reinforcement learning. It is easy to define when the agent reached the goal, but it is much more difficult to design the reward function, such that it enables the agent to get there efficiently. Since the reward function determines the agent's behaviour, it is important to encode all the necessary information into the reward function to make sure it is representative of the behaviour you would like to see.

There are two behaviours the agent should learn in order to manoeuvre the nanocar successfully across any given racetrack. These behaviours are realized by using two separate reward functions.

The first reward function R_1 (2.1) encourages the agent to approach the goal. This means decreasing the distance at every time step leading to a positive reward. However, when this is not the case and the distance becomes greater than or equal to the previous time step, it gets penalized by receiving a negative reward, that is twice the highest positive reward it could receive. In this way, the agent wants to decrease the distance towards the goal for every time step. Penalizing equal distances also solves another undesired behaviour, namely the accumulation of maximum reward by just driving in circles around the goal.

The first reward function R_1 (2.1) in figure 2.4 shows the received reward plotted against Δx_t divided by d_{goal}, where Δx_t is the already covered distance and d_{goal} the initial distance between the nanocar and a sub-goal or the nanocar and the final goal.

$$R_1 = \begin{cases} 0.5\left(\frac{d_{goal}-x_t}{d_{goal}}\right) = 0.5\left(\frac{\Delta x_t}{d_{goal}}\right) & x_t < x_{t-1} \\ -1 & x_t \geq x_{t-1} \end{cases} \tag{2.1}$$

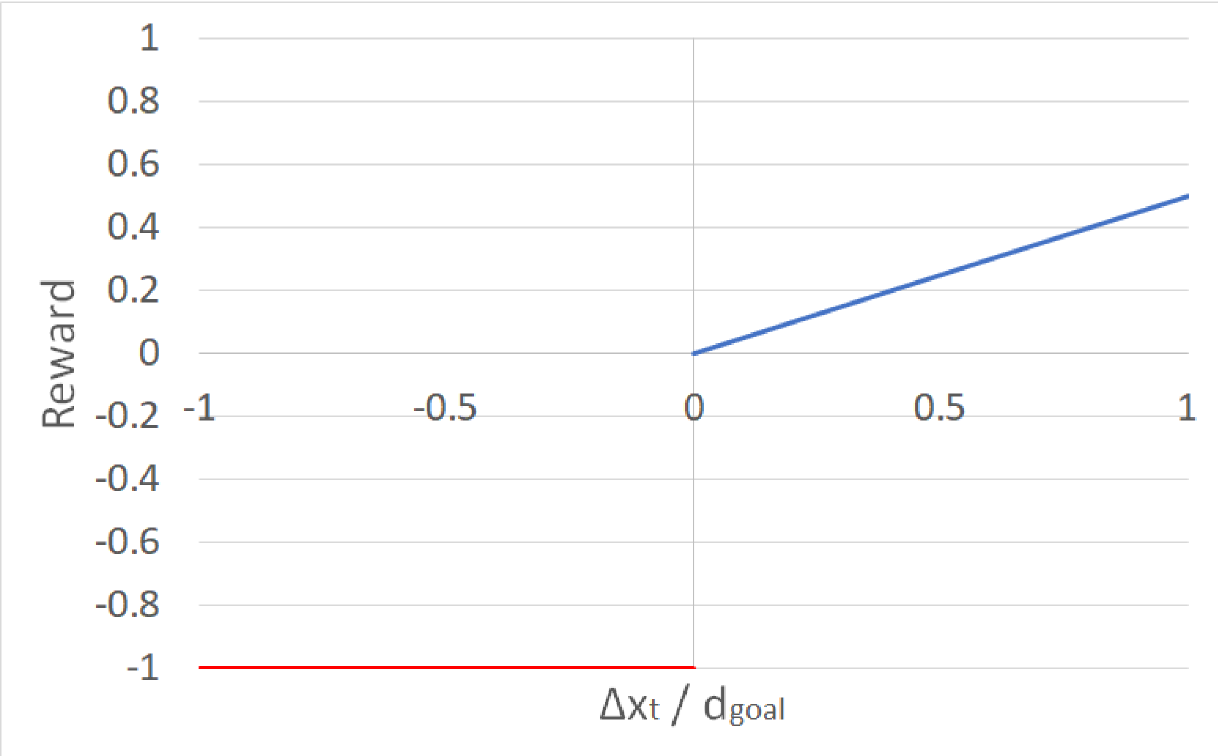


Figure 2.4: Reward function that encourages the agent to move towards the goal and decrease distance for every time step

The second reward function R_2 (2.2) encourages the agent to precisely pull the nanocar below the STM-tip. The reward function is raised by the power of 0.4, which gives it a steep curvature as the nanocar is close to the STM-tip and flattens the further away the nanocar has moved from the tip position. This form of the reward function encourages the agent to move the nanocar directly below the STM-tip.

The second reward function in figure 2.5 shows the reward versus x/d_{max}, where x is the distance from the nanocar to the STM-tip and d_{max} the largest distance where a pulling action can be successful, which is 2350 DAC units ($\hat{=}$ 13.19 $\mathring{A}$). This value comes from the experimental data of the race in Toulouse and will be discussed in chapter 2.2.

The exact position of the nanocar is obtained by determining the centre of mass of the nanocar. If the derivative of the current is **greater or equal** to a certain *threshold*, the position of the nanocar is assumed to be right below the STM-tip without further investigating its real spatial position. If the derivative of the current is **smaller** than a certain *threshold*, the search algorithm kicks in and determines the centre of mass of the nanocar. Thus, x the distance from the nanocar to the initial STM-tip position can be calculated. Thus, the reward function is in essence only calculated for unsuccessful pulling actions, as for a successful pulling actions the received reward is just 1.

$$R_2 = \begin{cases} 1 - \left(\frac{x}{d_{max}}\right)^{0.4} & x \leq d_{max} \\ 0 & x > d_{max} \end{cases} \tag{2.2}$$

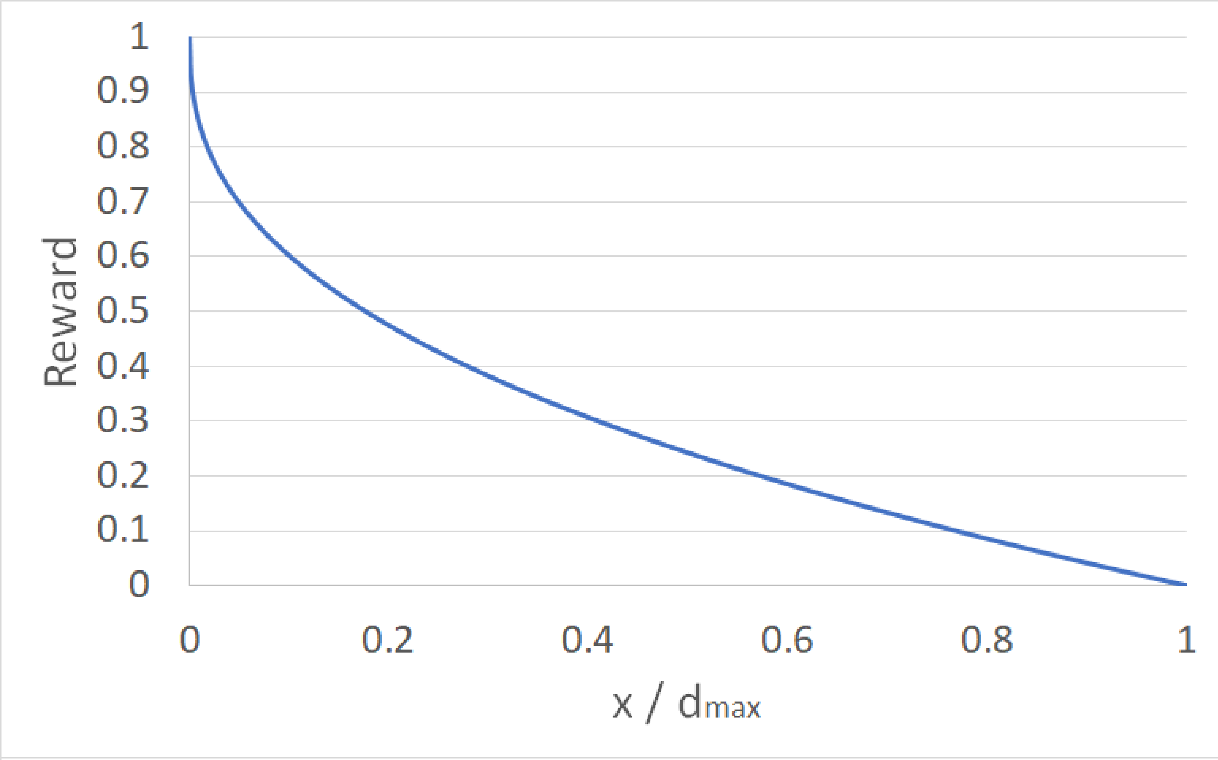


Figure 2.5: Reward function that encourages the agent to pull the nanocar as close to the STM-tip as possible

This concludes all the fundamentals necessary to easily understand the following Python code.

2.1.3.2 The code of the environment

```
import numpy as np
import math
import random
import itertools
import statistics

from scipy import ndimage
from scipy import signal
import matplotlib.pyplot as plt
from matplotlib import cm
from mpl_toolkits.mplot3d import Axes3D

import os
import glob
from datetime import datetime

import tkinter as tk
import socket # socket.gethostname()

from gui import GUI
from stm import STM

class EnvDriving(object):
    """
    This class represents the virtual environment which is essentailly a copy of the real
    environment but only with the parameters the agent needs.

    Methods
    -------
    init_env()
        Initializes the environment.

    init_reward_variables()
        Calculates the distance between consecutive sub-goals or sub-goal to goal.

```

```
    set_position()
        Sets the STM-tip position.

    get_relative_position()
        Overrides the relative STM-tip position of the environment by the position provided by the
        STMAFM program.

    get_current_spectrum()
        Returns the current spectrum of the latest vertical manipulation step.

    get_derivative_current()
        Calculates and returns the average current of the latest vertical manipulation step.

    define_voltage_pulse()
        Defines the voltage pulse that is used for pulling the nanocar towards the STM-tip.

    perform_vertical_manipulation()
        Performs a vertical manipulation by applying a defined voltage pulse and measures the
        induced current response.

    set_position_history()
        Saves either the position of the nanocar as long as its position is known or the position
        of the STM-tip while searching for it.

    update_environment_variables()
        Calculates the distance from the nanocar to the nearest goal; and from the nanocar to the
        final goal. Deletes the position of a goal when the goal is reached and also deletes the
        reward variable of the previous sub-goal distance.

    get_nanocar_position()
        Returns the latest known position of the nanocar.

    get_state_position_of_goals()
        Returns all the goal positions, like sub-goals and the final goal.

    get_total_distance()
        Returns the total distance from the nanocar to the final goal.

    unit_vector(vector)
        Returns the unit vector of the vector.

    distance_between_vectors(vector1, vector2)
        Returns the distance between two vectors.

    angle_between_vectors(v_base, v_car, v_goal)
        Return the angle in degrees between the two vectors, namely from 'v_base to v_car' and
        from 'v_base to v_goal'.

    calc_next_position(distance, alpha)
        Calculates the next position of the STM-tip by using the distance and angle chosen by the
        agent.

    check_current_pattern()
        Checks if the derivative of the current pattern measured after a pulling action and checks
        if a the treshhold is exceeded or not.

    search_car()
        Search for the nanocar in a circular pattern with increasing radius. A high current response
        will indicate, that the nanocar is below the STM-tip.

    reward_function()
        Calculates the reward to measure the performance of the agent's actions. The reward is
        calculated by using two functions.

    is_done()
        Checks if the episode is finished.
    """
    def __init__(self):
        self.directory_of_data = os.getcwd()+'/Data/1/'

        # Instantiation of the STM and connecting to the STMAFM program
        self.stm = STM()
        self.stm.connect()

        # Environment constants
        self.TRESHHOLD_CURRENT = 700        # Current treshhold for determining if the
                                            # nanocar is or is not below the tip.
```

```
        self.SEARCH_DISTANCE = 250
        self.HALF_SEARCH_LENGTH = 4000
        self.SEARCH_STEPSIZE = 250
        self.DISTANCE_REACH_GOAL = 2500     # Treshhold in DAC units between nanocar
                                            # and sub-goal/final goal

        # Environment variables
        self.position_nanocar = np.array(np.zeros(2))
        self.position_stm_tip = np.array(np.zeros(2))
        self.initial_stm_position = None
        self.position_of_environment = []
        self.number_of_manipulations = 0    # Number of manipulations
        self.min_height_values = 0
        self.max_height_values = 0
        self.current_spectrum = []
        self.derivative_current = []
        self.know_Car = True
        self.done = False

        # Initialize the environment using the GUI
        self.init_env()
        self.stm.connect()
        # State variables
        self.state_position_of_goals = np.array(self.position_of_environment[1:])
        self.state_position_of_nanocar_past_present = [self.position_nanocar, self.position_nanocar]

        # Reward variables and initialization
        self.reward  = 0
        self.DISTANCE_ERROR_MAX = 2250
        self.distance_to_nearest_goal = 0
        self.total_distance_to_goal = 0
        self.distance_subgoals = np.zeros(len(self.position_of_environment))
        self.init_reward_variables()

        try:
            files = glob.glob(self.directory_of_data + '*.CSV')

            if not files == []:
                latest_file = max(files, key=os.path.getmtime)
                with open(latest_file, newline='') as csv_file:
                    for line in csv_file.readlines(1):
                        self.number_of_episodes = int(line.split(',')[1])
            else:
                self.number_of_episodes = 0
                print("There are no previous episodes.")
        except OSError:
            self.number_of_episodes = 0
            print("The CSV file does not exist.")

        self.datetime_start = datetime.now()
        self.datetime_end = 0
        self.number_of_manipulations = 0
        self.number_of_successful_manipulations = 0
        self.number_of_failed_manipulations = 0
        self.total_reward_per_episode = 0
        self.number_of_searching = 0
        self.number_of_search_steps = 0
        self.average_steps_for_searching = 0
        self.x_history_nanocar = []
        self.y_history_nanocar = []
        self.x_history_searching_nanocar = []
        self.y_history_searching_nanocar = []
        # Total racetrack distance in [nm]
        self.total_distance = self.total_distance_to_goal*0.000561142

    def init_env(self):
        """
        Initializes the environment by using the GUI. Also the GUI is created here which is
        based on tkinter.

        The first stm-tip position selected with the GUI is equivalent to the nanocar position and
        the starting position, where the STM starts manouvering the nanocar.
        """
        # Creates the tkinter object
        root = tk.Tk()
        # Creates all widgets in the GUI
        gui = GUI(self.stm, master=root)
```

```
        # Calls the mainloop method which is inherited from Tk
        gui.mainloop()
        # Sets positions for the environment: Nanocar, Sub-goals and Goal
        self.position_of_environment = gui.get_environment_positions()
        # Sets first environment position equivalent to nanocar position and start
        # STM-tip position
        self.position_stm_tip = np.array(self.position_of_environment[0])
        self.position_nanocar = self.position_stm_tip

    def init_reward_variables(self):
        """
        Calculates the distance between all following sub-goals or sub-goal to goal that were set in
        the initialization step of the environment. These are necessary for the reward function.
        """
        # Distance between initial nanocar position to first sub-goal or already to the
        # final goal
        self.distance_subgoals[0] = np.linalg.norm(np.subtract(
            self.position_nanocar,
            self.position_of_environment[1]))

        # Distances between sucessive sub-goals and sub-goal to final goal.
        if len(self.position_of_environment) > 1:
            for i in range(1,len(self.position_of_environment)):
                self.distance_subgoals[i] = np.linalg.norm(np.subtract(
                    self.position_of_environment[i-1],
                    self.position_of_environment[i]))

    def set_position(self):
        """ Sets the STM-tip position either based on the agents choice or by the search-algorithm.

            Functions
            ---------
            stm.set_position(self.position_stm_tip)
                Sets the STM-tip position.

            set_position_history()
                Saves every STM-tip position.
        """
        self.stm.set_position(self.position_stm_tip)
        self.set_position_history()

    def get_relative_position(self):
        """
        Overrides the relative STM-tip position of the environment by the position provided by the
        STMAFM program.

        Functions
        ---------
        stm.get_relative_position()
            Overrides the position_stm_tip of the environment with the position given by the STMAFM
            program.
        """
        self.position_stm_tip = self.stm.get_relative_position()

    def get_current_spectrum(self):
        """ Returns the current spectrum of the latest vertical manipulation step.

            Function
            --------
            stm.get_current_spectrum()
                Reads the current spectrum from the ADC channels of the STMAFM program.

            Return
            ------
            self.current_spectrum : list([number of datapoints])
                Contains the current spectrum.
        """
        self.current_spectrum = self.stm.get_current_spectrum()
        return self.current_spectrum

    def get_average_current(self):
        """ Calculates and returns the average current of the latest vertical manipulation step.

            Functions
            ---------
            stm.get_current_spectrum()
                Reads the current spectrum from the ADC channels of the STMAFM program.
```

```python

            Return
            ------
            self.current_spectrum : list([number of datapoints])
                Contains the current spectrum.
        """
        current_spectrum = np.array(self.stm.get_current_spectrum())
        self.average_current = float(np.mean(current_spectrum[current_spectrum > 1000]))
        print(self.average_current)
        return self.average_current

    def get_derivative_of_current(self):
        """
        Calculates and returns the derivative of the current from the latest vertical
        manipulation step.

        Functions
        ---------
        stm.get_current_spectrum()
            Reads the current spectrum from the ADC channels of the STMAFM program.

        Return
        ------
        self.current_spectrum : list([number of datapoints])
            Contains the current spectrum.
        """
        time, current_spectrum = self.stm.get_current_spectrum()
        current_spectrum = np.ravel(current_spectrum)
        #current_spectrum_smoothed = signal.savgol_filter(current_spectrum,53,3)
        self.derivative_current = np.gradient(current_spectrum, axis=0)

        plt.plot(time, current_spectrum)
        #plt.plot(time, current_spectrum_smoothed)
        plt.plot(time, self.derivative_current)
        plt.show()

        return self.derivative_current

    def define_voltage_pulse(self):
        """
        Defines the voltage pulse that is used for pulling the nanocar towards the STM-tip.

        Functions
        ---------
        stm.define_voltage_pulse()
            Defines the voltage pulse by loading a .VZDATA-file.
        """
        self.stm.define_voltage_pulse()

    def perform_vertical_manipulation(self):
        """
        Performs a vertical manipulation by applying a defined voltage pulse and measures the
        induced current response.

        Functions
        ---------
        stm.perform_vertical_manipulation()
            Takes a vertical manipulation spectrum at the current image point (x,y).
        """
        self.stm.perform_vertical_manipulation()

    def set_position_history(self):
        """
        Saves either the position of the nanocar as long as its position is known or the position of
        the STM-tip while searching for it.
        """
        if self.know_Car == True:
            self.x_history_nanocar=np.append(self.x_history_nanocar,
                                             self.position_stm_tip[0])
            self.y_history_nanocar=np.append(self.y_history_nanocar,
                                             self.position_stm_tip[1])
        else:
            self.x_history_searching_nanocar=np.append(self.x_history_searching_nanocar,
                                                       self.position_stm_tip[0])
            self.y_history_searching_nanocar=np.append(self.y_history_searching_nanocar,
                                                       self.position_stm_tip[1])

```

```python
    def update_environment_variables(self):
        """
        Calculates the distance from the nanocar to the nearest goal; and from the nanocar to the
        final goal. Deletes the position of a goal when the goal is reached and also deletes the
        reward variable of the previous sub-goal distance.
        """
        # Calculates the distance to the nearest goal
        self.distance_to_nearest_goal = np.linalg.norm( np.subtract(
                                                        self.position_nanocar,
                                                        self.state_position_of_goals[0]))
        # Calculates the total distance to the goal
        self.total_distance_to_goal = self.distance_to_nearest_goal
        for i in range(1,len(self.state_position_of_goals)):
            self.total_distance_to_goal += np.linalg.norm(  np.subtract(
                                                            self.state_position_of_goals[i-1],
                                                            self.state_position_of_goals[i]))

        # When a sub-goal is reached, the sub-goal gets deleted. Also, the reward variable for the previous sub-goal distance gets deleted.
        if len(self.state_position_of_goals) > 0:
            if self.distance_to_nearest_goal < self.DISTANCE_REACH_GOAL:
                self.state_position_of_goals = np.delete(self.state_position_of_goals,0,0)
                self.distance_subgoals = np.delete(self.distance_subgoals,0,0)

    def get_nanocar_position(self):
        """
        Returns the latest known position of the nanocar.
        """
        return self.position_nanocar

    def get_state_position_of_goals(self):
        """
        Returns all the goal positions, like sub-goals and the final goal.

        Returns
        -------
        self.state_position_of_goals : lol
            The goal positions.
        """
        return self.state_position_of_goals

    def get_total_distance(self):
        """
        Returns the total distance from the nanocar to the final goal.

        Returns
        -------
        self.total_distance_to_goal : float
            The total distance from nanocar to goal.
        """
        return self.total_distance_to_goal

    def unit_vector(self, vector):
        """
        Returns the unit vector of the vector.

        Attributes
        ----------
        vector : list
            A vector.

        Return
        ------
        unit_vector : list
            The unit vector.
        """
        vector = np.array(vector)
        if vector.all() == 0:
            return [0,0]
        elif not vector.all() == 0:
            unit_vector = vector / np.linalg.norm(vector)
            return unit_vector

    def distance_between_vectors(self, vector1, vector2):
        """
        Returns the distance between two vectors.

```

```
        Attributes
        ----------
        vector1 : list
            Vector 1.
        vector2 : list
            Vector 2.

        Return
        ------
        vector_distance : float
            The distance between vector1 and vector2.
        """
        vector1 = np.array(vector1)
        vector2 = np.array(vector2)
        vector_distance = 0
        if not np.array_equal(vector1,vector2):
            vector_distance = np.linalg.norm(np.subtract(vector1,vector2))
        return vector_distance

    def angle_between_vectors(self, v_base, v_car, v_goal):
        """
        Returns the angle in degrees between the two vectors, namely from 'v_base to v_car' and from
        'v_base to v_goal'.

        Note: The function considers if the relative vector of the nanocar 'v_base to v_car' is
        positioned clockwise or counter-clockwise from the relative vector 'v_base to v_goal'.

        Attributes
        ----------
        v_base : list
            Vector to the basis.
        v_car : list
            Vector to the nanocar.
        v_goal : list
            Vector to the goal.

        Return
        ------
        angle : float
            The angle spanned by the two vectors: 'v_base to v_car' and from
            'v_base to v_goal'.
        """
        v_base = np.array(v_base)
        v_car = np.array(v_car)
        v_goal = np.array(v_goal)

        # Calculates the relative vectors of the nanocar and the goal
        v_car_rel = v_car-v_base
        v_goal_rel = v_goal-v_base

        # Calculates the unit vectors of the relative vectors nanocar and goal
        v_car_u = self.unit_vector(v_car_rel)
        v_goal_u = self.unit_vector(v_goal_rel)

        # Calculates the angle between the two relative vectors nanocar and goal
        angle = np.arccos(np.clip(np.dot(v_car_u, v_goal_u), -1.0, 1.0))*180/np.pi
        # Uses the property of the determinant that is, if the det < 0 the, relative
        # vector of the nanocar is clockwise to the relative vector of the goal.
        if np.linalg.det([v_goal_u,v_car_u])<0:
            angle = -angle
        return angle

    def calc_next_position(self, distance, alpha):
        """
        Calculates the next position of the STM-tip by using the distance and angle chosen by the
        agent.

        Attributes
        ----------
        distance : int
            The relative distance the STM-tip is position with respect to the
            position of the nanocar.
        alpha : int
            The relative angle at which the STM-tip is position relative to the
            vector reaching from the nanocar to the goal position.
        """
        # Converts alpha from degree to radiant
```

```
        alpha = alpha*np.pi/180
        theta = 0
        # Calculates the angle theta, which correlates the fixed STM coordination
        # system with the relative coordination system of the agent.
        dx = np.subtract(self.state_position_of_goals[0][0], self.position_nanocar[0])
        dy = np.subtract(self.state_position_of_goals[0][1], self.position_nanocar[1])

        if dx>0:
            theta = np.arctan(dy/dx)
        elif dx<0 and dy>=0:
            theta = np.arctan(dy/dx)+np.pi
        elif dx<0 and dy<0:
            theta = np.arctan(dy/dx)-np.pi
        elif dx==0 and dy>0:
            theta = np.pi/2
        elif dx==0 and dy<0:
            theta = -np.pi/2

        # Calculates STM-tip position in the fixed coordination system using the
        # relative angle alpha and the absolute angle theta
        pos_stm_x = int(round(self.position_nanocar[0]+distance*np.cos(alpha+theta),2))
        pos_stm_y = int(round(self.position_nanocar[1]+distance*np.sin(alpha+theta),2))

        # Sets the new STM-tip position
        self.position_stm_tip = np.array([pos_stm_x, pos_stm_y])
        self.set_position()
        # Increases the number of manipulation steps
        self.number_of_manipulations+=1

    def check_current_pattern(self):
        """
        Checks if the average current of the current pattern measured after a pulling action is
        higher than a certain treshhold.

        If this is:
        - TRUE: The position of the nanocar is below the STM-tip
        - FALSE: The position of the nanocar is not below the STM-tip and the
            search-algorithm is executed.

        Functions
        ---------
        get_derivative_current()
            Calculates the derivative to the STM-tip induced current after a pulling action.
        reward_function()
            Calculates the reward the agnet receives.
        search_car()
            Searching the nanocar if its lost.
        """
        self.get_derivative_of_current()
        if (abs(self.derivative_current) >= self.TRESHHOLD_CURRENT).any() and self.know_Car == True:  # I is RIGHT
            print("Current pattern is right!")
            self.number_of_successful_manipulations += 1
            self.position_nanocar = self.position_stm_tip.copy()
            print('Check I - Nanocar (X,Y): %s' % self.position_nanocar)
            self.state_position_of_nanocar_past_present = [
                self.state_position_of_nanocar_past_present[1],
                self.position_nanocar]
            self.initial_stm_position = None
            self.reward_function()

        elif (abs(self.derivative_current) < self.TRESHHOLD_CURRENT).any() and self.know_Car== True:  # I is WRONG
            print("Current pattern is wrong! == Car is lost ==")
            self.number_of_failed_manipulations += 1
            self.know_Car = False
            self.initial_stm_position = self.position_stm_tip.copy()
            print('Check I - STM-tip initial (X,Y): %s' % self.initial_stm_position)
            self.search_car()

    def search_car(self):
        """
        Search for the nanocar in a line-by-line pattern measuring the Z-topography centred around
        half the distance betweem the previous nanocar position and the current position of the
        STM-tip, where the nanocar should be. The centre of mass is calculated from the Z-topography
        and determines the nanocar's position.

```

```
        Functions
        ---------
        set_position()
            Sets the STM-tip position based on the search-algorithm.
        perform_lateral_manipulation(start, end, steps)
            Performs a vertical manipulation between the start and end point and returns the
            Z-Signal.
        """
        self.number_of_search_steps+=1
        # Determines the step size of the y-direction for the search algorithm
        step_size = 500

        # The center of the search-algorithm is the last pulling position of the STM-tip
        centre_of_search_float = np.subtract(self.initial_stm_position, self.position_nanocar)/2+
    self.position_nanocar
        centre_of_search_algorithm = [int(round(centre_of_search_float[0])),
                                      int(round(centre_of_search_float[1]))]
        print('Centre of search Algorithm (X,Y): %s' % centre_of_search_algorithm)

        # Necessary to convert DAC units into pixel
        deltaX = self.stm.get_float_param('Delta X [DAC]')
        deltaY = self.stm.get_float_param('Delta Y [DAC]')

        # Sets relative STM-tip to top left corner
        x_rel_start_for_search = int(round(centre_of_search_algorithm[0]-self.HALF_SEARCH_LENGTH))
        y_rel_start_for_search = int(round(centre_of_search_algorithm[1]-self.HALF_SEARCH_LENGTH))
        print(f'Relative Search param: x={x_rel_start_for_search} y={y_rel_start_for_search}')

        start_lateral_manipulation = [x_rel_start_for_search, y_rel_start_for_search]
        end_lateral_manipulation = np.add(self.position_stm_tip, [self.HALF_SEARCH_LENGTH*2,0])
        start_lateral_manipulation_pixel = [start_lateral_manipulation[0]/deltaX,
                                            start_lateral_manipulation[1]/deltaY]
        end_lateral_manipulation_pixel = [end_lateral_manipulation[0]/deltaX,
                                          end_lateral_manipulation[1]/deltaY]

        # Initialises lateral manipulation to know the number of datapoints the function will
    measure
        length_lat_manip_spectrum = len(self.stm.perform_lateral_manipulation(
            start_lateral_manipulation,
            end_lateral_manipulation,
            self.HALF_SEARCH_LENGTH*2))

        # Defines the number of points in the y-direction of the Z-topography
        number_of_steps = int(self.HALF_SEARCH_LENGTH*2/step_size)

        # Initialises the Z-topography
        z_topography = np.array(np.zeros([number_of_steps, length_lat_manip_spectrum]))

        # Performing lateral manipulations to record the Z-topography
        for y in range(0, self.HALF_SEARCH_LENGTH*2, step_size):
            self.position_stm_tip = [x_rel_start_for_search, y_rel_start_for_search+y]
            self.set_position()

            start_lateral_manipulation = [x_rel_start_for_search, y_rel_start_for_search+y]
            end_lateral_manipulation = np.add(self.position_stm_tip, [self.HALF_SEARCH_LENGTH*2,0])
            start_lateral_manipulation_pixel = [start_lateral_manipulation[0]/deltaX,
                                                start_lateral_manipulation[1]/deltaY]
            end_lateral_manipulation_pixel = [end_lateral_manipulation[0]/deltaX,
                                              end_lateral_manipulation[1]/deltaY]

            val_lateral_manipulation = self.stm.perform_lateral_manipulation(
                start_lateral_manipulation,
                end_lateral_manipulation,
                self.HALF_SEARCH_LENGTH*2)

            # Multiply by -1, because the Z-signal of the piezos is the inverse of the Z-topography.
            z_signal_to_z_topography = np.multiply(val_lateral_manipulation, -1)
            # Creates a Z-topography by filling the matrix row-by-row.
            z_topography[int(y/step_size)] = z_signal_to_z_topography

        # Setting all the Z-values below the mean Z-value to 1 to create an improved Z-topography
        centre_of_mass_threshold = np.mean(z_topography)

        super_threshold_indices = z_topography <= centre_of_mass_threshold
        z_topography_improved = z_topography.copy()
        z_topography_improved[super_threshold_indices] = 1

```

```python
        # Calculates the centre of mass from the improved Z-topography; given in indices
        centre_of_nanocar = ndimage.measurements.center_of_mass(z_topography_improved)
        print('Centre of Mass [indices]: %s' % centre_of_nanocar)

        # Rescales the centre of mass indices to DAC units
        centre_of_nanocar_DAC = [
            int(round(centre_of_nanocar[1]*self.HALF_SEARCH_LENGTH*2/length_lat_manip_spectrum)),
            int(round(centre_of_nanocar[0]*100))]

        print('Centre of Mass [DAC]: %s' % centre_of_nanocar_DAC)

        # Shows the Z-topography after searching is complete
        X, Y = np.mgrid[0:np.shape(z_topography)[0], 0:np.shape(z_topography)[1]]
        Z=z_topography[X,Y]
        fig = plt.figure()
        ax = Axes3D(fig)
        ax.plot_surface(X, Y, Z,
            rstride=1, cstride=1, cmap=cm.coolwarm, linewidth=1, antialiased=True)
        plt.show()

        # Calculates the absolute coordinates of the nanocar
        position_nanocar = [x_rel_start_for_search+centre_of_nanocar_DAC[0],
                            y_rel_start_for_search+centre_of_nanocar_DAC[1]]
        self.position_nanocar = [int(round(position_nanocar[0])),
                                 int(round(position_nanocar[1]))]
        print('Nanocar position (X,Y): %s' % self.position_nanocar)
        self.know_Car = True

    def reward_function(self):
        """
        Calculates the reward to measure the performance of the agents actions. The reward is
        calculated by using two functions.

        1. Reward function calculates how precisely the nanocar has moved below the STM-tip
        2. Reward function calculates how close the nanocar moved towards the goal.

        Functions
        ---------
        distance_between_vectors(vector1, vector2)
            Calclates the distance between two vectors.
        """
        self.reward  = 0

        if self.number_of_manipulations >= 1:
            position_of_nanocar_past = self.state_position_of_nanocar_past_present[0]
            position_of_nanocar_present = self.state_position_of_nanocar_past_present[1]
            position_of_nearest_goal = self.state_position_of_goals[0]

            # Calculates the distane to the goal before and after the pulling action
            distance_of_past_nanocar_to_goal = self.distance_between_vectors(
                                                                position_of_nanocar_past,
                                                                position_of_nearest_goal)
            distance_of_present_nanocar_to_goal = self.distance_between_vectors(
                                                                position_of_nanocar_present,
                                                                position_of_nearest_goal)
            difference_in_distance_from_goal_between_pulling_action = np.subtract(
                                                                distance_of_past_nanocar_to_goal,
                                                                distance_of_present_nanocar_to_goal)

            # Calculates by how much the nanocar translated to an unknown position
            if self.initial_stm_position is None:
                nanocar_deviates_from_initial_stm_position = 0
                self.initial_stm_position = position_of_nanocar_present
            else:
                nanocar_deviates_from_initial_stm_position = self.distance_between_vectors(
                                                                self.initial_stm_position,
                                                                position_of_nanocar_present)

            # Calculates the reward using two reward functions
            self.reward = 0
            # 1. Reward function
            if (difference_in_distance_from_goal_between_pulling_action > 0
            and self.total_distance_to_goal > 0):
                self.reward  += 0.5*(1-self.distance_to_nearest_goal/self.distance_subgoals[0])
            elif (difference_in_distance_from_goal_between_pulling_action <= 0
            and self.total_distance_to_goal >= 0):
                self.reward  -= 1
```

```python
            # 2. Reward function
            if nanocar_deviates_from_initial_stm_position <= self.DISTANCE_ERROR_MAX:
                self.reward += 1-math.pow(
                            nanocar_deviates_from_initial_stm_position/self.DISTANCE_ERROR_MAX,0.4)
        print(f'Reward: {self.reward}')

    def is_done(self):
        """ Checks if the episode is finished.

            Returns
            -------
            self.done : boolean
                Returns TRUE if the episode is finished.
        """
        if len(self.state_position_of_goals) <= 0:
            self.done = True
            self.datetime_end = datetime.now()
            self.number_of_episodes+=1
            print("The episode is finished!")
        return self.done
```

2.1.4 The creation of an agent

This is the final part of the program describing how the agent performs actions and learns by exploring and exploiting the environment.

The agent performs actions based on the learned Q-table. The Q-table relates states to actions, a so called state-action pair that is represented as a Q-value within the Q-table. The Q-table represents the knowledge database of the agent and is saved after an episode is finished.

2.1.4.1 The importance of the Q-table size

In table 2.1, you can see how fast the Q-table can drastically increase in size even when the environment is not that complicated. The number of Q-value entries is given by the stats (φ) times actions (φ, d).

The number of the Q-table entries is simply given by:

$$n = \varphi^2 * d \tag{2.3}$$

The agent uses angles φ ranges from -180 to +180°, where angles are ranging from -4 to +4° relative to the axis, which is defined by the line between the old nanocar position and the goal. These narrow angles are higher resolved by using a discretization of 1° and angles larger than ±4° with a discretization of 30°. The distance for a pulling action ranges from 1250 to 2350 DAC units ($\hat{=}$ 7.01 *to* 13.19 $\AA$). Within this range, experiments show that pulling actions are successful. The first row "Inflated states" shows the number of states with a step size of 1 unit for both φ and d. A visual representation of the angle discretization is shown in 2.6.

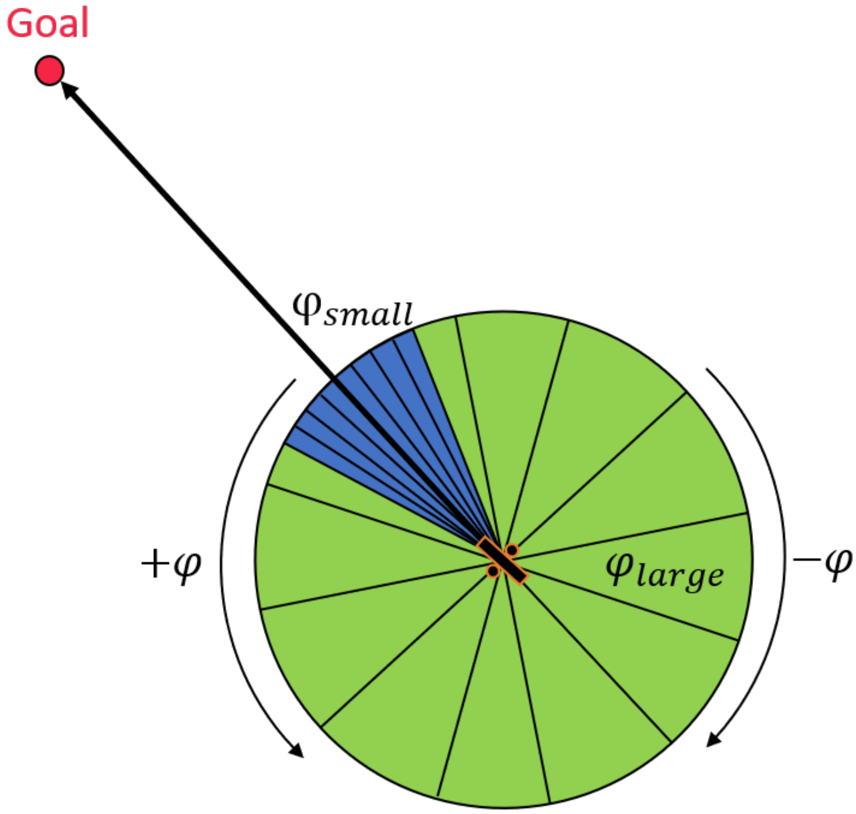


Figure 2.6: Angle discretization for states and actions

A single pulling action, while simultaneously measuring the current spectrum and saving it afterwards, takes about 2 s. This would take almost 2.3 years to at least visit every Q-table entry once, which is obviously an unfeasible approach. Therefore, the angle φ for states and actions were discretized by 20 and the distance d by 10, which by comparison will take about 5 h to fill up the whole Q-table once.

Note: These are theoretical numbers, as they assume every pulling action is successful and the nanocar will never be lost. Dependent on the resolution of the search algorithm, the nanocar position has recovered after a relatively large area of 5 by 5 nm, double the size of the nanocar, is scanned, which will take a minimum of 5 seconds.

Table 2.1: The size of the Q-table

φ ... off-axis angle; axis being the previous nanocar position to goal in °
d ... pulling distance in DAC units

n ... number of Q-table entries

	states	actions		
	φ / °	d / DAC	φ / °	n
Inflated states	360	1100	360	142,560,000
Discretization	30	10	30	9000
Discrete states	12	110	12	15,840
Addition φ_{small}	8		8	64
Discrete states	20	110	20	44,000

2.1.4.2 Discretization of the multidimensional Q-table

The Q-table in figure 2.7 is a multidimensional array of size 20 x 110 x 20. By extracting the states and actions for a specific Q-value from the indices of the array, the file size of the Q-table is decreased by a factor of three, which enhances performance due to faster writing it on the disc.

All possible states range from -180 to +180° and $\varphi_{small} = 4°$. To ensure indices are always positive values starting at 0 (page 0), an offset n_{offset} is applied. The offset n_{offset} is determined by the grade of discretization. In this case, two different discretization steps are used, namely $n_{discret\ large} = 30°$ for large and $n_{discret\ small} = 1°$ for small angles, the n_{offset} is given as follow:

$$n_{offset} = \begin{cases} n_{offset\ narrow} + n_{offset\ large} - 1 & , \varphi_{real} \leq \varphi_{small} \\ 2\ n_{offset\ narrow} + n_{offset\ large} - 1 & , \varphi_{real} > \varphi_{small} \end{cases} \tag{2.4}$$

, where φ_{real} is the perceived real angle which is non-discretized and continuous.

In this work, the offset n_{offset} is given by:

$$\begin{aligned} n_{offset} &= n_{offset\ narrow} + n_{offset\ large} - 1 \\ &= \frac{\varphi_{small}}{n_{discret\ small}} + \frac{\varphi_{large}}{n_{discret\ large}} - 1 \\ &= \frac{4}{1} + \frac{180}{30} - 1 \\ &= 4 + 6 - 1 = 9 \end{aligned}$$

The discrete angel $\varphi_{discrete}$ is simply given by the following equations:

$$n_{offset} = \begin{cases} \varphi_{discrete} = \frac{\varphi_{real}}{n_{discret}} & , \varphi_{real} < -\varphi_{small} \\ \varphi_{discrete} = \frac{\varphi_{real}}{n_{discret}} + n_{offset} & , else \end{cases} \tag{2.5}$$

Thus, if the nanocar is in state $\varphi = +3$, the performed action is chosen within page $3 + 9 = 12$.

When exploiting the environment, the agent chooses the highest Q-value entry within page 12. The position of the entry is uniquely defined by the index that can be decoded to determine the real action behind this index.

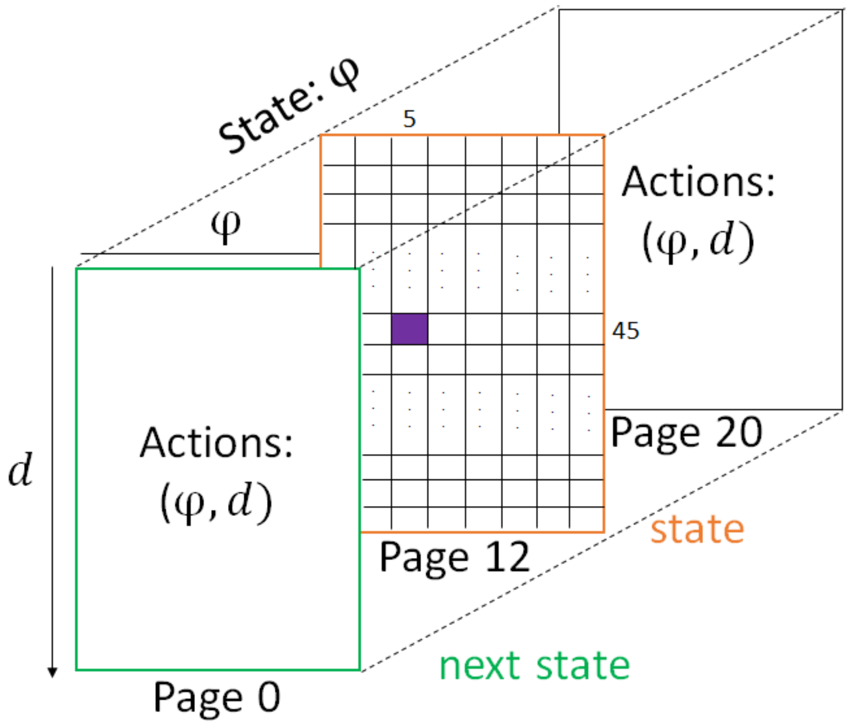


Figure 2.7: The multidimensional Q-table with two highlighted states. The current state of the nanocar is $\varphi_{real} = 5°$, which is page 12 in the Q-table. The highest Q-table entry is the action the agent performs, which is indicated by the purple square in column 45, row 5. This corresponds to action $\varphi_{real} = -10°$, $d_{real} = 1700$ *DAC units* ($\hat{=}$ 9.54 $\mathring{A}$). After this action is performed, the nanocar is in the next state $\varphi_{real} = -20°$, which is page 0.

2.1.4.3 Update process of the Q-table

The final part of this chapter explains how the Q-table gets updated and filled while manoeuvring along the racetrack and which improvements were implemented to learn more efficient when exploring the environment. The following figure 2.8 shows a one step Q-table update starting with (a) the previous figure 2.7. In addition to the previous example, the *Q-Learning equation* 1.17 is used to see how the Q-table entries were obtained and updated.

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha \left[r_{t+1} + \gamma \max_a Q(s_{t+1}, a) - Q(s_t, a_t) \right] \tag{2.6}$$

$$Q(s_t, a_t) \leftarrow 2.00 \qquad + 0.90 \, [0.81 + 0.77 \cdot 4.00 - 2.00] \tag{2.7}$$

$$Q(s_t, a_t) \leftarrow 3.70 \tag{2.8}$$

The agent gets the current state of the nanocar, $\varphi = -20°$, from the environment and determines the best action by looking up the highest Q-table entry in page 12. The action is performed and the nanocar translates over the surface to the next position. From this new position, the next state $\varphi = -5°$ and the reward $r_{t+1} = 0.82$ are determined by the environment and returned to the agent. In this next state, the highest Q-value $\max_a Q(s_{t+1}, a) = 4$ is used for the update process. The old Q-value gets updated using the Q-Learning algorithm and is replaced by $Q(s_t, a_t) = 3.7$.

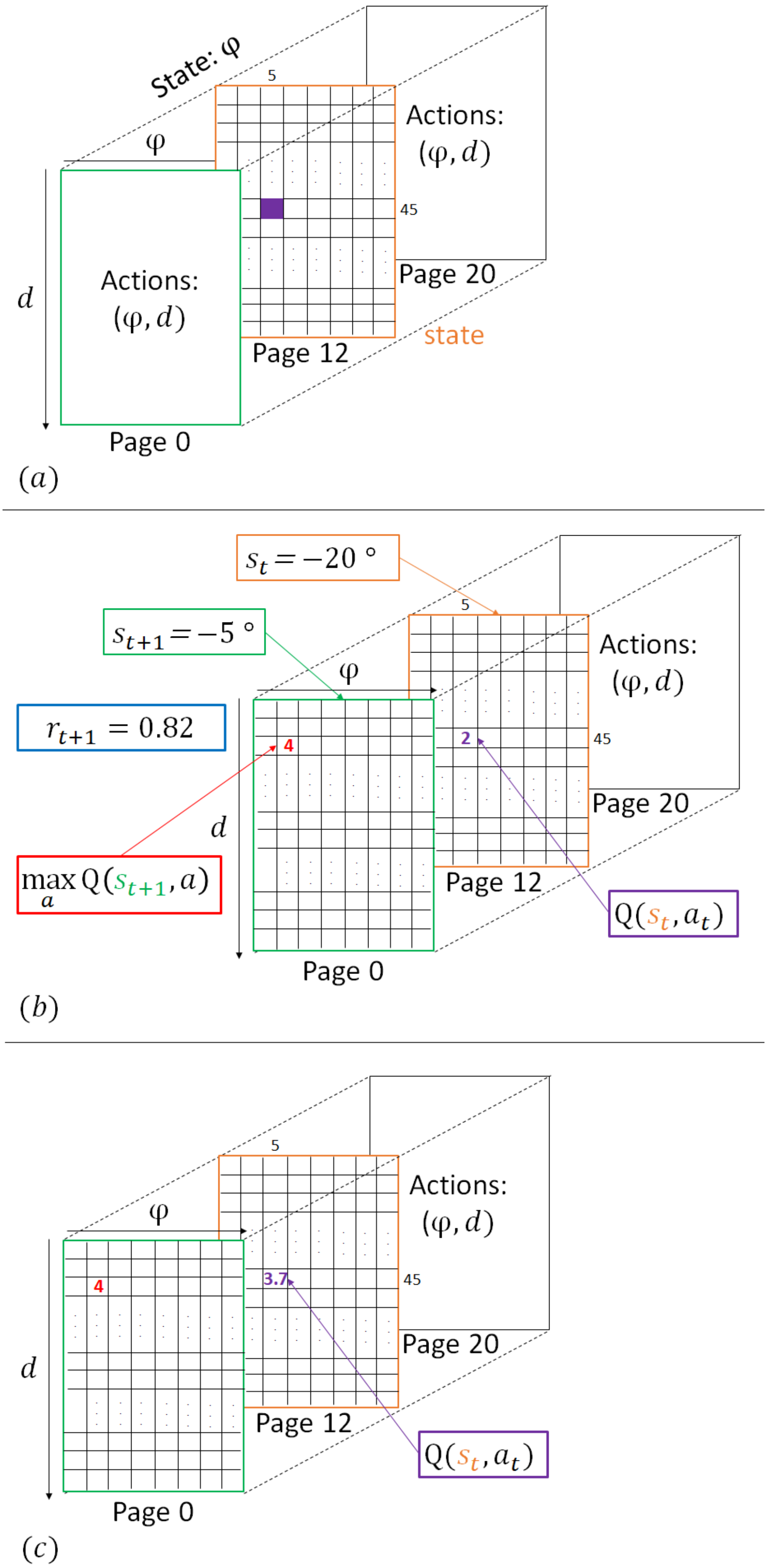


Figure 2.8: A Q-table update process for one time step. Starting at (a) the current state of the nanocar and the highest Q-value in this state is the performed action. After the action is performed, (b) the next state, the reward and the highest Q-value of the next state are determined and used to (c) update the old Q-value by applying the Q-Learning algorithm.

2.1.4.4 Enhanced exploration and exploitation

Although the used discretization of the Q-table reduces the number of Q-table entries from 142.56 million vs 44,000, it is still a very large number.

Therefore, the Q-table gets limited by narrowing the action space. This does not mean the Q-table itself is reduced, but the angles φ, from which the agent can choose, are limited. These limitations will be softened as the *limited Q-table* gets populated.

The reason for limiting the angles is based on the fact that the dipole of the nanocar enhances the manoeuvrability in three directions, namely at the position of the negative dipole at 0°, but also at a clockwise offset of about 45 and 225° to the negative dipole position. Considering the position at 225° is at the back of the nanocar, only the 0 and 45° positions are relevant for this discussion. (Grant Simpson, personal communication, March 19, 2020)

Here in the code, a preferred direction between 0 and 45° is assumed, in which the manoeuvrability is enhanced. At first, not the whole Q-table ranging from -180 to +180° is filled, but the action φ is limited between -4 and +4°, which reduces the Q-table entries to 8,800 compared to the former 44,000.

2.1.4.5 The code of the agent

```
from environment import EnvDriving

import numpy as np
import random
import math
import os
import glob
from datetime import datetime
import matplotlib.pyplot as plt
from pathlib import Path
import statistics

class QDriving(EnvDriving):
    """
    This class represents the agent program. The goal of the agent is to manouver a nanocar across a
    race-track and accumulate maximum reward. This is done by positioning the STM-tip based on the
    current state of the nanocar within the environment. The learning algorithm of the agent is
    based on an off-policy temporal difference algorithm, known as 'Q-Learning'.

    Methods
    -------
    convert_distance_to_index()
        Converts the distance into a sub-index for the Q-table.

    convert_angle_to_index()
        Converts the angle into a sub-index for the Q-table.

    evaluate_state()
        Evaluates the current state of the nanocar based on its position within the environment.

    select_action()
        The agent chooses the best action in a particular state based on the Q-table or
        by choosing a random action to explore the state.

    q_table_function()
        Calcuate the Q-Learning algorithm and updates the Q-table.

    save_q_table()
        Saves the Q-table as a binary file.
    """
    def __init__(self):
        # Directory to save the Q-table
        self.qtable_directory = os.path.dirname(os.getcwd())+'/Qtable/'

        # Q-learning hyperparameters
        self.ALPHA = 0.9
        self.GAMMA = 0.95

```

```python
        # Learning variables
        self.epsilon = 0.9   # Exploration rate [%]

        self.ANGLE_LOWER_LIMIT = -4
        self.ANGLE_UPPER_LIMIT = 4
        self.DISTANCE_LOWER_LIMIT = 1500
        self.DISTANCE_UPPER_LIMIT = 1900

        # Q-learning variables
        self.q_t = []
        self.q_tt = []
        self.q_tt_max = []

        # Discretization variables
        self.DISTANCE_MIN = 1250
        self.DISTANCE_MAX = 2350
        self.DISTANCE_DIV = 10
        self.DISTANCE_RANGE = self.DISTANCE_MAX-self.DISTANCE_MIN
        self.DISTANCE_STEP = int(self.DISTANCE_RANGE/self.DISTANCE_DIV)
        self.ANGLE_MIN = -30
        self.ANGLE_MAX = 30
        self.ANGLE_RANGE = self.ANGLE_MAX-self.ANGLE_MIN
        self.ANGLE_DIV = 2
        self.ANGLE_DIV_ROUGH = 30
        self.ANGLE_STEP = int(self.ANGLE_RANGE/self.ANGLE_DIV)

        self.ANGLE_RANGE_ROUGH = int((180-self.ANGLE_MAX)/self.ANGLE_DIV_ROUGH)
        self.POSITIVE_Q_TABLE_DISCRETIZATION = np.array(np.zeros(int(self.ANGLE_RANGE_ROUGH)))
        self.NEGATIVE_Q_TABLE_DISCRETIZATION = np.array(np.zeros(int(self.ANGLE_RANGE_ROUGH)))

        # Q-table initialization based on discretization variables
        for i in range(self.ANGLE_RANGE_ROUGH):
            # Additional 7 States: [ 30, 180]
            self.POSITIVE_Q_TABLE_DISCRETIZATION[i] = (self.ANGLE_MAX
                                                       + self.ANGLE_DIV_ROUGH*i
                                                       + self.ANGLE_DIV_ROUGH/2)
            # Additional 7 States: [-30,-180)
            self.NEGATIVE_Q_TABLE_DISCRETIZATION[i] = (self.ANGLE_MIN
                                                       - self.ANGLE_DIV_ROUGH*i
                                                       - self.ANGLE_DIV_ROUGH/2)

        # State variables
        self.state_angle = 0

        # Action variables
        self.action_distance = 0
        self.action_angle  = 0

        # Initialize environment
        self.env = EnvDriving()

        self.q_table = np.zeros([self.ANGLE_STEP+self.ANGLE_RANGE_ROUGH*2,
                                 self.DISTANCE_STEP+1,
                                 self.ANGLE_STEP+self.ANGLE_RANGE_ROUGH*2])

        # Load existing Q-table
        files =  glob.glob(f'{self.qtable_directory}*.npy')
        if not files == []:
            latest_file = max(files, key=os.path.getmtime)
            self.q_table = np.load(latest_file)
            print(latest_file)
            print(self.q_table[np.nonzero(self.q_table)])
            print('The Q-table is loaded!')
        else:
            print ("Q-table does not exist")

    def convert_distance_to_index(self, var):
        """
        Converts the distance into an index or sub-index. The distance is given by the distance
        between the STM-tip and the nanocar.

        Note: In general the index determines exactly where the entry is located in the Q-table.
        This subsequently means an entry of the multidimensional Q-table uniquely defines the state
        and the action.

        Return
        ------
```

```
            Returns the distance as index value.
        """
        var = np.round(var)
        index_of_var = 0
        if var < self.DISTANCE_MAX-self.DISTANCE_DIV and var > self.DISTANCE_MIN:
            index_of_var = round((var-self.DISTANCE_MIN)/self.DISTANCE_DIV)
        elif var >= self.DISTANCE_MAX-self.DISTANCE_DIV:
            index_of_var = round(
                (self.DISTANCE_MAX-self.DISTANCE_MIN-self.DISTANCE_DIV)/self.DISTANCE_DIV)
        return int(index_of_var)

    def convert_angle_to_index(self, var):
        """
        Converts the angle into a sub-index. The angle is given by the angle between the two vectors,
        namely the vector previous nanocar to goal position and previous nanocar to current nanocar
        position.

        Note: In general, the index determines exactly where the entry is located in the Q-table.
        This subsequently means an entry of the multidimensional Q-table uniquely defines the state and the action.

        Return
        ------
            Returns the angle as index value.
        """
        if var >= self.ANGLE_MIN and var <= self.ANGLE_MAX:
            index = int(np.around((var+self.ANGLE_MAX)/self.ANGLE_DIV,1)) + self.ANGLE_RANGE_ROUGH
        else:
            if var <= self.ANGLE_MIN:
                index = -(np.digitize(var,self.NEGATIVE_Q_TABLE_DISCRETIZATION)
                          + self.ANGLE_RANGE_ROUGH)
            elif var >= self.ANGLE_MAX:
                index =  (np.digitize(var,self.POSITIVE_Q_TABLE_DISCRETIZATION)
                          + self.ANGLE_RANGE_ROUGH
                          + self.ANGLE_STEP)
                if index == 40:
                    index = 0
        return index

    def evaluate_state(self):
        """
        Evaluates the current state of the nanocar based on its position within the environment.

        The state is given by the angle between the two vectors, namely the vector pointing from
        the previous nanocar to the goal and the previous nanocar to the current nanocar position.

        Functions
        ---------
        angle_between_vectors(v_base, v_car, v_goal)
            Return the angle in degrees between the two vectors, namely from
            'v_base to v_car' and from 'v_base to v_goal'.
        """
        # Calculates the state and sets the state to 0 before any manipulation was performed
        self.state_angle = 0
        if self.env.number_of_manipulations > 0:
            self.state_angle = self.angle_between_vectors(
                                                self.env.state_position_of_nanocar_past_present[0],
                                                self.env.state_position_of_nanocar_past_present[1],
                                                self.env.state_position_of_goals[0])

    def select_action(self):
        """
        The agent chooses the best action in a particular state based on the Q-table or by choosing a random action to explore the state.

        Exploitation: If two or more indices are equally good, meaning their Q-values are the same,
        the action is chosen randomly from these equally good actions.

        Exploration: EPSILON rate of exploration defines how often the agent takes a random action.
        At least in the beginning the agent's action space is limited, meaning that small angles and
        statistically better distances were chosen first.
        """
        self.evaluate_state()
        state_index = self.convert_angle_to_index(self.state_angle)
        action_index = np.zeros(2)

```

```python
        # Chooses the best action OR a random action that was never used before
        if random.uniform(0,1) < self.epsilon:
            # Calculate indices to corresponding limits
            lower_distance_index = self.convert_distance_to_index(self.DISTANCE_LOWER_LIMIT)
            upper_distance_index = self.convert_distance_to_index(self.DISTANCE_UPPER_LIMIT)+1
            lower_angle_index = self.convert_angle_to_index(self.ANGLE_LOWER_LIMIT)
            upper_angle_index = self.convert_angle_to_index(self.ANGLE_UPPER_LIMIT)+1

            # Determine all Q-table entries that were never used: Q-value == 0
            actions_never_used_index = np.where(self.q_table[state_index]==0)

            # Determine indices which are within the limit
            limited_actions_never_used_index = [
                                            (actions_never_used_index[0][:]<=upper_distance_index) &
                                            (actions_never_used_index[0][:]>=lower_distance_index) &
                                            (actions_never_used_index[1][:]<=upper_angle_index) &
                                            (actions_never_used_index[1][:]>=lower_angle_index)]

            # Select the actions that have never been used and are within the limits
            actions_never_used_index=[actions_never_used_index[0][limited_actions_never_used_index],
                                      actions_never_used_index[1][limited_actions_never_used_index]]

            # From all actions within the limit randomly chose one action
            action_random_never_used_index = np.random.randint(0,len(actions_never_used_index[0]))
            distance_never_used_index = actions_never_used_index[0][action_random_never_used_index]
            angle_never_used_index = actions_never_used_index[1][action_random_never_used_index]
            action_index = [distance_never_used_index, angle_never_used_index]
        else:
            # Select the best action
            action_best_index = np.where(self.q_table[state_index]
                                         == np.max(self.q_table[state_index]))

            # From equally good actions select one of them randomly
            action_random_best_index = np.random.randint(0,len(action_best_index[0]))
            distance_best_index = action_best_index[0][action_random_best_index]
            angle_best_index = action_best_index[1][action_random_best_index]
            action_index = [distance_best_index, angle_best_index]

        # Convert the index to real values in DAC units
        self.action_distance = self.DISTANCE_MIN + action_index[0]*self.DISTANCE_DIV
        if action_index[1] <= self.ANGLE_RANGE_ROUGH:
            self.action_angle = -180+action_index[1]*self.ANGLE_DIV_ROUGH
        elif action_index[1] >= self.ANGLE_RANGE_ROUGH + self.ANGLE_STEP:
            self.action_angle = (self.ANGLE_MAX + self.ANGLE_DIV_ROUGH*(action_index[1]
                                                                       - self.ANGLE_RANGE_ROUGH
                                                                       - self.ANGLE_STEP))
        else:
            self.action_angle = (self.ANGLE_MIN + self.ANGLE_DIV*(action_index[1]
                                                                 - self.ANGLE_RANGE_ROUGH))

        # Calculates the next STM-tip positon based on the agents choosen actions
        self.env.calc_next_position(self.action_distance, self.action_angle)

    def q_table_function(self):
        """
        Calcuate the Q-value based on the Q-Learning algorithm and updates the Q-table.

        Functions
        ---------
        convert_distance_to_index(var)
            Converts the distance into an index or sub-index. The distance is given by the distance
            between the STM-tip and the nanocar.
        convert_angle_to_index(var)
            Converts the angle into a sub-index. The angle is given by the angle between the two
            vectors, namely the vector previous nanocar to goal position and previous nanocar to
            current nanocar position.
        """
        if self.env.know_Car == True and self.env.number_of_manipulations > 1:
            q_t = 0
            q_tt_max = 0
            q_tt = 0

            # Action space: converts real actions to index values
            action_index = [self.convert_distance_to_index(self.action_distance),
                            self.convert_angle_to_index(self.action_angle)]

            # State space: converts real state to index value
```

```
            state_index = self.convert_angle_to_index(self.state_angle)
            next_state_index = action_index[1]

            # The Q-Learning algorithm
            q_t = self.q_table[state_index, action_index[0], action_index[1]]
            q_tt_max = np.max(self.q_table[next_state_index])
            q_tt = q_t + self.ALPHA*(self.env.reward + self.GAMMA*(q_tt_max) - q_t)
            self.q_table[state_index, action_index[0], action_index[1]] = q_tt
            self.save_q_table()

    def save_q_table(self):
        """
        Saves the Q-table as a binary file.
        """
        path = f'{self.qtable_directory}/qtable'
        now = datetime.now()
        current_time = now.strftime("%y-%m-%d_%H-%M-%S")
        path_with_timestamp = f'{self.qtable_directory}/{current_time}_qtable'

        try:
            print('The Q-table is saved!')
            np.save(path, self.q_table)
            np.save(path_with_timestamp, self.q_table)
            print(self.q_table[self.q_table>0])
        except:
            try:
                os.mkdir(self.qtable_directory)
                np.save(path, self.q_table)
                np.save(path_with_timestamp, self.q_table)
                print(self.q_table[np.nonzero(self.q_table)])
            except OSError:
                print("Creation of the directory %s failed" % path)
                print("Q-table could not be created.")
            else:
                print ("Successfully created the directory %s " % path)
```

2.1.5 The code of the main

```
#!/bin/env python3
from agent import QDriving
import numpy as np

import csv
from time import mktime

import logging
import tkinter as tk
import matplotlib.pyplot as plt


def analysis(agent):
    # Calculate Analysis Variables
    if agent.env.number_of_searching == 0:
        agent.env.average_steps_while_searching = 0
    else:
        agent.env.average_steps_while_searching = agent.env.number_of_search_steps/agent.env.
    number_of_searching

    timestamp_file = agent.env.datetime_end.strftime("%y-%m-%d_%H-%M-%S")
    path_with_timestamp = f'{agent.env.directory_of_data}/{timestamp_file}_episode_{agent.env.
    number_of_episodes}_epsilon_{agent.epsilon}.csv'
    time_difference_in_s = abs(mktime(agent.env.datetime_start.timetuple())-mktime(agent.env.
    datetime_end.timetuple()))
    speed = agent.env.total_distance/time_difference_in_s

    with open(path_with_timestamp, 'w', newline='') as csv_file:
        csv_write = csv.writer(csv_file)
        csv_write.writerow(['Episode', f'{agent.env.number_of_episodes}'])
        csv_write.writerow(['Epsilon', f'{agent.epsilon}'])
        csv_write.writerow(['Duration in s', f'{time_difference_in_s}'])
        csv_write.writerow(['Length', f'{agent.env.total_distance}'])
        csv_write.writerow(['Speed in nm / h', f'{speed}'])
        csv_write.writerow(['Manipulations', f'{agent.env.number_of_manipulations}'])
        csv_write.writerow(['Succeesful Manipulations',f'{agent.env.
    number_of_successful_manipulations}'])
        csv_write.writerow(['Failed Manipulations',f'{agent.env.number_of_failed_manipulations}'])
```

```
        csv_write.writerow(['Total reward per Episode',f'{np.around(agent.env.total_reward_per_episode,2)}'])
        csv_write.writerow(['Average Steps while Searching',f'{agent.env.average_steps_while_searching}'])
        csv_write.writerow(['== Positional Dataset =='])
        csv_write.writerows([['Goal'], np.swapaxes(agent.env.position_of_environment,0,1)[0], np.swapaxes(agent.env.position_of_environment,0,1)[1],
                             ['Nanocar'], agent.env.x_history_nanocar, agent.env.y_history_nanocar])
        csv_write.writerow(['Search-Algorithm'])
        for i in range(len(agent.env.x_history_searching_nanocar)):
            csv_write.writerow([agent.env.x_history_searching_nanocar[i], agent.env.y_history_searching_nanocar[i]])

def driving_routine(agent):
    agent.select_action()
    agent.env.perform_vertical_manipulation()
    agent.env.check_current_pattern()
    agent.q_table_function()
    agent.env.update_environment_variables()

def main():
    agent = QDriving()

    while not agent.env.is_done():
        driving_routine(agent)
    #agent.save_q_table()
    analysis(agent)
    plt.show()

if __name__ == "__main__":
    main()
```

2.2 Learning from human experience or existing data

The following section provides an example code for how an agent is able to learn from human generated data by using *VERT-files*, that are generated by the STM after an action is performed. This enables the agent to learn without the necessity of controlling the STM directly, which is saving time and operational costs. As in the previous section, the code starts with the lowest level, being the *filemanager*, followed by the environment and the agent program.

In the following flow diagram 2.9 the learning procedure is illustrated.

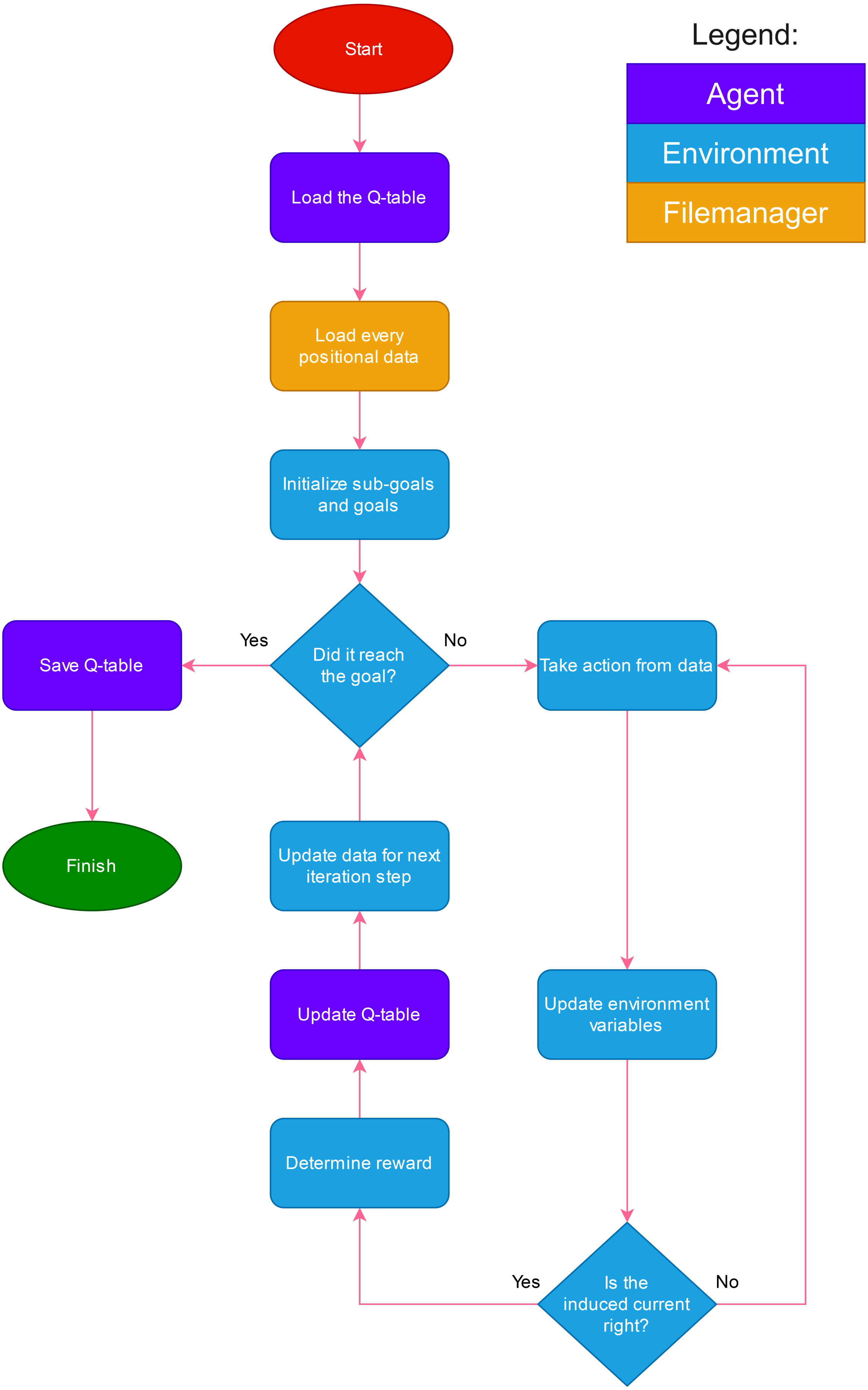


Figure 2.9: The flow diagram for training the agent from human generated data. The Legend indicates to which class a processes belongs.

2.2.1 The filemanager

The *filemanager* chronologically loads all VERT-files within a directory. A VERT-file contains the STM settings and most importantly the current response at each tip position. The complete directory is loaded, such that the agent has the complete trajectory from start to finish ahead of it and iterates through every time step by perceiving every state, the "performed" action, and its associated reward, as if it would control the STM directly.

2.2.1.1 The code of the filemanager

```
import time
from datetime import datetime
import os
import glob
import shutil
import math
import numpy as np

class FileManager(object):
    """ A class used to read and/or write the VERT-files for learning from human-generated data.

        Methods
        -------
        get_files : list
            Provides the complete path for every VERT-file within the 'directory' sorted by name.

        get_latest_file : str
            pProvides the complete path for the latest VERT-file in the 'directory'.

        get_num_files : int
            Provides the number of files within the given 'directory'.

        write_simulation_data(xy_data, know_Car=True)
            Writes artificial data with the STM-tip position and a high or low current dependent on
            weather the nanocar is below the tip or not (this is determined randomly).

        read_position : array(2)
            Read X/Y position from the VERT-file.
    """
    def __init__(self, directory_of_data):
        # A unique naming scheme for every written VERT-file
        self.last_timestamp = None
        # The number of files within the given 'directory'
        self.num_files = 0

    def get_files(self):
        """
        Returns the complete path for every VERT-file within the 'directory' and sorts it by name.

            Returns
            -------
            files : list
                A list of strings that contain the complete filepath of every VERT-
                file within the 'directory'
        """
        files = sorted(glob.glob('*.VERT'))
        self.num_files = len(files)
        return files

    def get_latest_file(self):
        files = sorted(os.listdir(os.getcwd()), key=os.path.getmtime)
        newest = files[-1]
        return newest

    def get_num_files(self):
        return self.num_files

    def write_simulation_data(self, xy_data, know_Car=True):
        dateTimeObj = datetime.now()
        timestampStr = f"{dateTimeObj.year}-{dateTimeObj.month}-{dateTimeObj.day}_{dateTimeObj.hour}-{dateTimeObj.minute}-{dateTimeObj.second}.{dateTimeObj.microsecond}"
        self.last_timestamp = timestampStr
        new_filename = f'{timestampStr}.VERT'
```

```python

        if know_Car == True:
            shutil.copyfile('Current_Right.VERT', new_filename)
        else:
            shutil.copyfile('Current_Wrong.VERT', new_filename)
        with open(new_filename, mode='r', encoding = "ISO-8859-1") as f:
            lines = f.readlines()
        with open(new_filename, mode='w', encoding = "ISO-8859-1") as f:
            lines[298] = '{:8d}{:8d}{:8d}{:10}'.format(1000, xy_data[0], xy_data[1], 1)+'\n'
            f.writelines(lines)

    def read_position(self, file=None):
        position = np.empty(2)

        if file is None:
            file = self.get_latest_file()

        with open(file, mode='r', encoding="ISO-8859-1") as f:
            f_data = f.read().split('\n')

        # X/Y-position from datafile
        xdac = float(f_data[298].split()[1])
        ydac = float(f_data[298].split()[2])

        # Offset correction
        offsetx = float(f_data[20].split('=')[1])
        offsety = float(f_data[21].split('=')[1])

        # Additional parameters
        dx = float(f_data[3].split('=')[1])
        dy = float(f_data[4].split('=')[1])
        nx = float(f_data[5].split('=')[1])
        ny = float(f_data[6].split('=')[1])

        rot = float(f_data[14].split('=')[1])

        driftxoff = 0
        driftyoff = 0

        # Rotation matrix:  cos -sin | xx xy
        #                   sin  cos | yx yy
        x_with_rotation= -(xdac*np.cos(rot*np.pi/180)-ydac*np.sin(rot*np.pi/180)+offsetx-driftxoff)
        y_with_rotation= -(xdac*np.sin(rot*np.pi/180)+ydac*np.cos(rot*np.pi/180)+offsety-driftyoff)

        position = np.array([xdac+offsetx,ydac+offsety])
        return position

    def read_current(self, file=None):
        if file is None:
            file = self.get_latest_file()

        with open(file, mode='r', encoding="ISO-8859-1") as f:
            f_data = f.read().split('\n')
        f_It = f_data[299:-1]      # Data for current and time

        t = []
        I = []
        for z in f_It:
            trunc = z.split()
            t.append(int(trunc[0]))
            I.append(float(trunc[3]))
        data_It = [t, I]
        return data_It

    def read_voltage(self, file=None):
        if file is None:
            file = self.get_latest_file()

        with open(file, mode='r', encoding="ISO-8859-1") as f:
            f_data = f.read().split('\n')
        f_Vt = f_data[299:-1]      # Data for current and time

        t = []
        V = []
        for z in f_Vt:
            trunc = z.split()
            t.append(int(trunc[0]))
```

```
            V.append(float(trunc[1]))
        data_Vt = [t, V]
        return data_Vt
```

2.2.2 The environment for learning

Although every VERT-file within a directory is loaded chronologically, the sub-goals that are evaluated by the environment are different than those the human headed for when maneuvering the nanocar towards a sub-goal. The reason for this is based on how the absolute position is defined, as (X, Y) are given relative to the latest image scanned. Figure 2.10 shows how the absolute position (X_{abs}, Y_{abs}) is determined by using the offset (X_{Offset}, Y_{Offset}) plus the relative position (X, Y) within the scanned image.

$$X_{abs} = X_{Offset} + X \tag{2.9}$$

$$Y_{abs} = Y_{Offset} + Y \tag{2.10}$$

However, the (X_{Offset}, Y_{Offset}) is not really consistent and shows a drift between images. Thus, when learning from data which is gathered from two recorded images, the data points do not process continuously, but show a random offset. This can be either due to thermal drift or due to the inaccurate coarse positioning system of the STM.

However, this problem is solved by calculating every distance of two successive points and if this distance is larger than 5000 DAC units, then the first point is defined as a sub-goal. The value of 5000 DAC units is a bit larger than double the distance (2350 DAC units), which is the largest distance where successful pulling actions can be achieved.

Note: Determining the absolute position is irrelevant for directly controlling the STM with the agent, because the agent only operates within the scanned image. Thus, all positions are determined relatively to the origin of the scanned image. If, for some reason the nanocar cannot be found by the search algorithm and a human has to take an image in order to locate the nanocar, the relative coordinates would change - meaning all goals would have to be re-initialized as the origin changes with the newly scanned image.

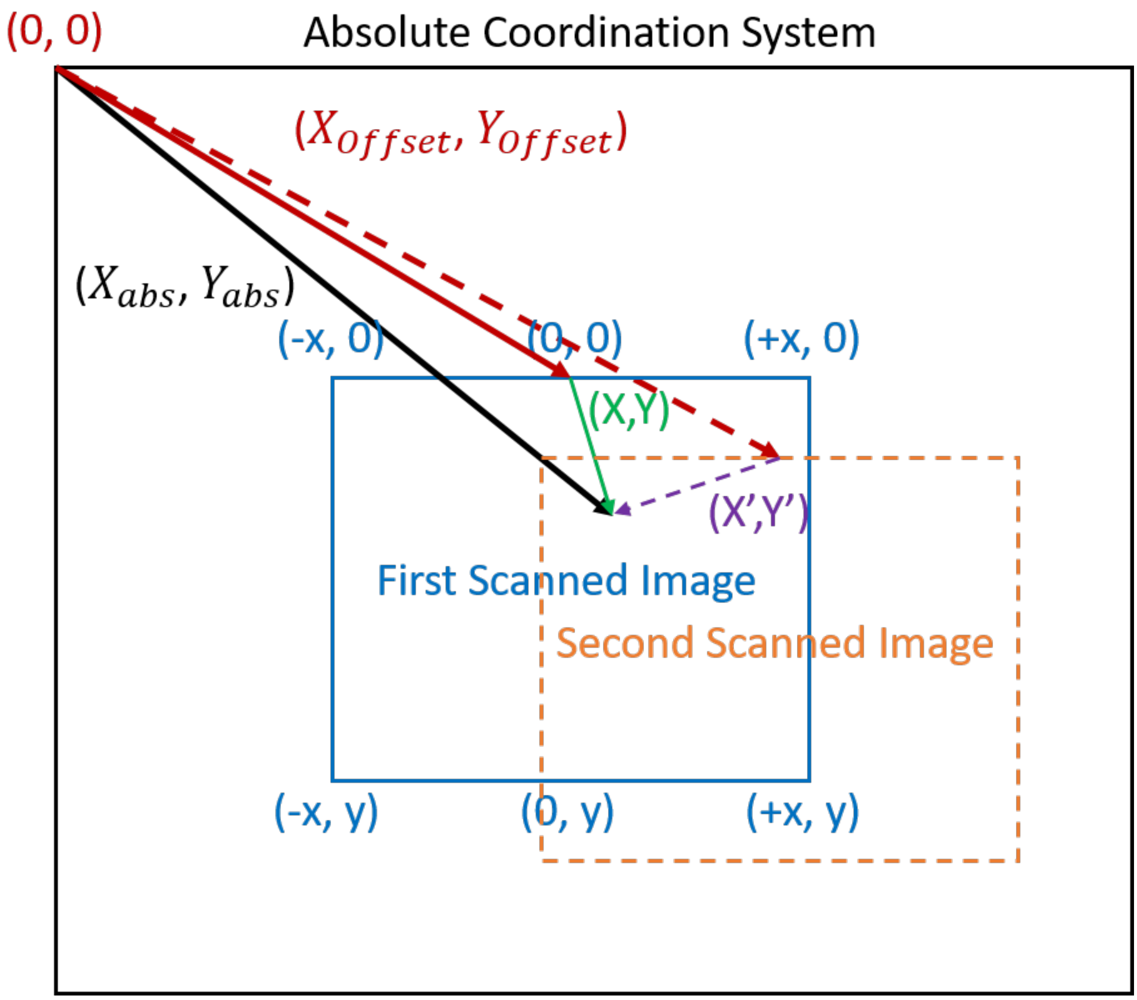


Figure 2.10: How the absolute and relative coordinates are defined by the STM. The origin of the scanned image is the center of the top boarder, while the offset and therefore the origin of the absolute coordination system is originated at the top left corner of the absolute coordination system.

2.2.2.1 The reward function

The *reward function* for learning is equivalent to the one defined in the environment section 2.1.3.1 of *Controlling the nanocar with the STM.*

2.2.2.2 The code of the environment

```
from filemanager import FileManager

import numpy as np
import math
import random
import os
import itertools
import statistics

import matplotlib.pyplot as plt
from matplotlib import cm
from mpl_toolkits.mplot3d import Axes3D

from scipy.signal import savgol_filter
import scipy.fftpack

class EnvLearning(FileManager):
    """
    This class represents the virtual environment generated from human data. This enables the agent
```

```
    to learn like itself is controlling the STM without the requirement of a real STM.

    Methods
    -------
    init_env()
        Initialize the environment.

    init_reward_variables()
        Calculates the distance between all following sub-goals or sub-goal to goal.

    load_position_data()
        Loads the absolute position from every VERT-files in the working directory.

    load_current_data()
        Loads the current spectra from every VERT-files in the working directory.

    load_goals()
        Evaluates the sub-goals and goal from the complete racetrack data.

    set_Position()
        Virtually sets the STM-tip to the next position.

    unit_vector(vector)
        Returns the unit vector of the vector.

    distance_between_vectors(vector1, vector2)
        Returns the distance between two vectors.

    angle_between_vectors(v_base, v_car, v_goal)
        Return the angle in degrees between the two vectors, namely from 'v_base to v_car' and from
        'v_base to v_goal'.

    calc_distance()
        Calculates the distance from the nanocar to the nearest goal; and from the nanocar to the
        final goal. Deletes the position of a goal when the goal is reached and also deletes the
        reward variable of the previous sub-goal distance.

    set_next_iteration()
        Updates all the data for the next iteration step.

    calc_average_current(current_spectrum)
        Calculates the average current from the current spectrum.

    check_current_pattern()
        Checks if the average current of the current pattern measured after a pulling action is
        higher than a certain treshhold.

    reward_function()
        Calculates the reward to measure the performance of the agent's actions. The reward is
        calculated by using two functions.

    is_done()
        Checks if the episode is finished.
    """
    def __init__(self):
        # Set the path of the data files as the working directory
        self.directory_of_data = os.getcwd()+'/Data/0/'
        os.chdir(self.directory_of_data)

        # Environment constants

        # Treshhold: know car position YES/NO?
        self.TRESHHOLD_CURRENT = 1000
        # Treshhold: distance above which a new sub-goal is defined
        self.TRESHHOLD_TO_EVALUATE_SUBGOAL = 5000

        # Environment variables
        self.number_of_iterations = 0
        self.initial_stm_position = None
        self.position_for_environment = []
        self.current_for_environment = []
        self.average_current_for_environment = []
        self.derivative_current_for_environment = []
        self.know_Car = True
        self.done = False
        # Inizializes the complete environment data from the 'directory'
        self.init_env()
```

```
        self.position_nanocar = np.array(self.position_for_environment[0])
        self.position_stm_tip = np.array(self.position_for_environment[0])

        # State variables
        self.state_position_of_goals = []
        self.load_goals()
        self.state_position_of_nanocar_past_present = [None, self.position_for_environment[0]]

        # Reward variables and initialization
        self.DISTANCE_ERROR_MAX = 2250
        self.distance_to_nearest_goal = 0
        self.total_distance_to_goal = 0
        self.distance_subgoals = np.zeros(len(self.state_position_of_goals))
        # Calculates distances between following environment positions
        self.init_reward_variables()
        # Calculates distances to the closest sub-goal and to the final goal
        self.calc_distance()

        # Statistic variables
        self.success = 0
        self.failure = 0

    def init_env(self):
        """
        Initialize the environment by loading a complete racetrack from VERT-files.

        A VERT-file is genereated after every vertical manipulation measurement and contains every
        setting of the STM.

        Functions
        ---------
        load_position_data()
            Loads the absolute position from the VERT-files for the given episode.
        load_current_data()
            Loads the measured spectrum from the VERT-files for the given episode.
        load_goals()
            Evaluates which data points are sub-goals or goals.
        """
        # Loads the positional data
        self.load_position_data()
        # Loads the current spectra
        self.load_current_data()
        # Evaluates sub-goals and the final goal
        self.load_goals()

    def init_reward_variables(self):
        """
        Calculates the distance between all following sub-goals or sub-goal to goal that were set
        in the initialization step of the environment. These are necessary for the reward function.
        """
         # Distance between initial nanocar position to first sub-goal or already to the final goal
        self.distance_subgoals[0] = np.linalg.norm(np.subtract(
                                                        self.position_nanocar,
                                                        self.state_position_of_goals[0]))

        # Distances between sucessive sub-goals and sub-goal to final goal.
        if len(self.state_position_of_goals) > 1:
            for i in range(1,len(self.state_position_of_goals)):
                self.distance_subgoals[i] = np.linalg.norm(np.subtract(
                                                        self.state_position_of_goals[i-1],
                                                        self.state_position_of_goals[i]))

    def load_position_data(self):
        """
        Loads the absolute position from every VERT-file in the working directory into a list.
        These positions represent the whole racetrack of an episode.
        """
        self.position_for_environment = []
        files = self.get_files()
        for file in files:
            self.position_for_environment.append(self.read_position(file))

    def load_current_data(self):
        """
        Loads the current spectra from every VERT-file in the working directory into a list.
        """
        self.current_for_environment = []
```

```
        files = self.get_files()
        for file in files:
            self.current_for_environment.append(self.read_current(file))
        for data in self.current_for_environment:
            self.average_current_for_environment.append(self.calc_average_current(data[1]))
            self.derivative_current_for_environment.append(np.gradient(data[1]))

    def load_goals(self):
        """
        Evaluates the sub-goals and goal from the complete racetrack data.

        A goal is evaluated by finding a position where its ensuing position is located futher away
        than a given treshhold. This has to be done in such a way, because the data gained by the
        STM is relative to the last taken image. This means, if for some reason the car could not
        be found, the surface has to be imaged. This changes the position of nanocar because its
        position is given by the relative position from the centre position of the image. Thus, the
        previous position does not correlate to the current position.
        """
        self.state_position_of_goals = []
        for i in range(1,len(self.position_for_environment)):
            # Defines a positon as a goal, if two points are further away than a given treshhold
            if np.linalg.norm(np.subtract(
                        self.position_for_environment[i-1],
                        self.position_for_environment[i])) >= self.TRESHHOLD_TO_EVALUATE_SUBGOAL:
                self.state_position_of_goals.append(self.position_for_environment[i])
        # The last position in a given racetrack is set to be the final goal
        self.state_position_of_goals.append(
                                self.position_for_environment[len(self.position_for_environment)-1])

    def set_position(self):
        """
        Virtually sets the STM-tip to the next position
        """
        self.position_stm_tip = self.position_for_environment[0].copy()

    def unit_vector(self, vector):
        """
        Returns the unit vector of the vector.

        Attributes
        ----------
        vector : np.array(len(vector))
            A vector.

        Return
        ------
        unit_vector : np.array(len(vector))
            The unit vector.
        """
        vector = np.array(vector)
        if vector.all() == 0:
            return [0,0]
        elif not vector.all() == 0:
            unit_vector = vector / np.linalg.norm(vector)
            return unit_vector

    def distance_between_vectors(self, vector1, vector2):
        """
        Returns the distance between two vectors.

        Attributes
        ----------
        vector1 : np.array(len(vector1))
            Vector 1.
        vector2 : np.array(len(vector2))
            Vector 2.

        Return
        ------
        vector_distance : float
            The distance between vector1 and vector2.
        """
        vector1 = np.array(vector1)
        vector2 = np.array(vector2)
        vector_distance = 0
        if not np.array_equal(vector1,vector2):
            vector_distance = np.linalg.norm(np.subtract(vector1,vector2))
```

```
        return vector_distance

    def angle_between_vectors(self, v_base, v_car, v_goal):
        """
        Return the angle in degrees between the two vectors, namely from 'v_base to v_car' and from
        'v_base to v_goal'.

        Note: The function considers if the relative vector of the nanocar 'v_base to v_car' is
        positioned clockwise or counter-clockwise from the relative vector 'v_base to v_goal'.

        Attributes
        ----------
        v_base : np.array(2)
            Vector to the basis.
        v_car : np.array(2)
            Vector to the nanocar.
        v_goal : np.array(2)
            Vector to the goal.

        Return
        ------
        angle : float
            The angle spanned by the two vectors: 'v_base to v_car' and from 'v_base to v_goal'.
        """
        v_base = np.array(v_base)
        v_car = np.array(v_car)
        v_goal = np.array(v_goal)

        # Calculates the relative vectors of the nanocar and the goal
        v_car_rel = v_car-v_base
        v_goal_rel = v_goal-v_base

        # Calculates the unit vectors of the relative vectors nanocar and goal
        v_car_u = self.unit_vector(v_car_rel)
        v_goal_u = self.unit_vector(v_goal_rel)

        # Calculates the angle between the two relative vectors nanocar and goal
        angle = np.arccos(np.clip(np.dot(v_car_u, v_goal_u), -1.0, 1.0))*180/np.pi
        # Use the property of the determinant that is, if the det < 0 the,
        # relative vector of the nanocar is clockwise to the relative vector of the goal.
        if np.linalg.det([v_goal_u,v_car_u])<0:
            angle = -angle
        return angle

    def calc_distance(self):
        """
        Calculates the distance from the nanocar to the nearest goal; and from the nanocar to the
        final goal. Deletes the position of a goal when the goal is reached and also deletes the
        reward variable of the previous sub-goal distance.
        """
        if len(self.position_for_environment) > 1:

            # Calculates the distance between the old and new stm-tip position
            self.moving_distance_stm_tip = np.linalg.norm(np.subtract(
                                                self.position_for_environment[0],
                                                self.position_for_environment[1]))

            # Calculates the distance to the nearest goal
            self.distance_to_nearest_goal = np.linalg.norm(np.subtract(
                                                self.position_nanocar,
                                                self.state_position_of_goals[0]))

            # Calculates the total distance to the goal
            self.total_distance_to_goal = self.distance_to_nearest_goal
            for i in range(1,len(self.state_position_of_goals)):
                self.total_distance_to_goal += np.linalg.norm(np.subtract(
                                                self.state_position_of_goals[i-1],
                                                self.state_position_of_goals[i]))

    def set_next_iteration(self):
        """
        Updates all the data for the next iteration step.

        This means the first entry in the list of positional data as well as reached sub-goals are
        deleted.
        """
        if len(self.position_for_environment) > 0:
```

```
            # Deletes the reached sub-goal
            if len(self.state_position_of_goals) > 0:
                if np.linalg.norm(np.subtract(self.position_for_environment[0],
                                              self.state_position_of_goals[0])) == 0:
                    self.state_position_of_goals = np.delete(self.state_position_of_goals, 0, 0)

            # Deletes the currentrly reached position in the list positional data
            self.position_for_environment = np.delete(self.position_for_environment, 0, 0)
            # Deletes the current spectrum that goes with the positional data
            self.derivative_current_for_environment = np.delete(
                                                    self.derivative_current_for_environment, 0, 0)
        self.number_of_iterations += 1

    def calc_average_current(self, current_spectrum):
        """ Calculates the average current from the current spectrum.

            Returns
            -------
            average_current : int
                The average current of the spectrum.
        """
        current_spectrum = np.array(current_spectrum)
        average_current = np.mean(current_spectrum[current_spectrum > 0])
        return average_current

    def check_current_pattern(self):
        """
        Checks if the derivative of the current pattern after a pulling action is higher than a
        certain treshhold.

        If this is:
        - TRUE: The position of the nanocar is below the STM-tip - hence it is known
        - FALSE: The position of the nanocar is not below the STM-tip - hence it is unknown and a
                 search-algorithm starts searching for the nanocar.

        Functions
        ---------
        reward_function()
            Calculates the reward the agnet receives.
        search_car()
            Searching the nanocar if the it got lost.
        """
        if ((abs(self.derivative_current_for_environment[0]) >= self.TRESHHOLD_CURRENT).any()
        and self.know_Car == True):                  # I is RIGHT
            print("Current pattern is right!")
            self.position_nanocar = self.position_stm_tip.copy()
            self.state_position_of_nanocar_past_present = [
                                                    self.state_position_of_nanocar_past_present[1],
                                                    self.position_nanocar]
            self.initial_stm_position = None
            self.reward_function()

        elif ((abs(self.derivative_current_for_environment[0]) < self.TRESHHOLD_CURRENT).any()
        and self.know_Car == True):                 # I is WRONG
            print("Current pattern is wrong! == Car is lost ==")
            self.know_Car = False
            self.initial_stm_position = self.position_stm_tip.copy()

        elif ((abs(self.derivative_current_for_environment[0]) >= self.TRESHHOLD_CURRENT).any()
        and self.know_Car == False):               # I is RIGHT
            print("Current pattern is right! == Car is found ==")
            self.know_Car = True
            self.position_nanocar = self.position_stm_tip.copy()
            self.state_position_of_nanocar_past_present = [
                                                    self.state_position_of_nanocar_past_present[1],
                                                    self.position_nanocar]
            self.reward_function()

    def reward_function(self):
        """
        Calculates the reward to measure the performance of the agent's actions. The reward is
        calculated by using two functions:

        1. Reward function calculates how precisely the nanocar has moved below the STM-tip
        2. Reward function calculates how close the nanocar moved towards the goal.

```

```
        Functions
        ---------
        distance_between_vectors(vector1, vector2)
            Calculates the distance between two vectors.
        """
        self.reward = 0

        if self.number_of_iterations >= 1:
            position_of_nanocar_past = self.state_position_of_nanocar_past_present[0]
            position_of_nanocar_present = self.state_position_of_nanocar_past_present[1]
            position_of_nearest_goal = self.state_position_of_goals[0]

            # Calculates the distance to the goal before and after the pulling action
            distance_of_past_nanocar_to_goal = self.distance_between_vectors(
                                                        position_of_nanocar_past,
                                                        position_of_nearest_goal)
            distance_of_present_nanocar_to_goal = self.distance_between_vectors(
                                                        position_of_nanocar_present,
                                                        position_of_nearest_goal)
            difference_in_distance_from_goal_between_pulling_action = np.subtract(
                                                        distance_of_past_nanocar_to_goal,
                                                        distance_of_present_nanocar_to_goal)

            # Calculates by how much the nanocar translated to an unknown position
            if self.initial_stm_position is None:
                nanocar_deviates_from_initial_stm_position = 0
                self.initial_stm_position = position_of_nanocar_present
            else:
                nanocar_deviates_from_initial_stm_position = self.distance_between_vectors(
                                                        self.initial_stm_position,
                                                        position_of_nanocar_present)

            # Calculates the reward using two reward functions
            self.reward = 0
            # 1. Reward function
            if (difference_in_distance_from_goal_between_pulling_action > 0
            and self.total_distance_to_goal > 0):
                self.reward += 0.5*(1-self.distance_to_nearest_goal/self.distance_subgoals[0])
            elif (difference_in_distance_from_goal_between_pulling_action <= 0
            and self.total_distance_to_goal >= 0):
                self.reward -= 1
            # 2. Reward function
            if nanocar_deviates_from_initial_stm_position <= self.DISTANCE_ERROR_MAX:
                self.reward += 1-math.pow(
                            nanocar_deviates_from_initial_stm_position/self.DISTANCE_ERROR_MAX,0.4)
        print(f'Reward: {self.reward}')

    def is_done(self):
        """ Checks if the episode is finished.

            Returns
            -------
            self.done : boolean
                Returns TRUE if the episode is finished.
        """
        if self.number_of_iterations >= self.get_num_files():
            self.done = True
            print("The training is finished!")
        return self.done
```

2.2.3 The learning agent

This code creates a Q-table by learning from human generated data. The chosen actions are already judged by the reward function of the environment. Thus, the performance of actions is pre-selected.

Important: The Q-table size has to be chosen, such that it corresponds with the final use case of the agent. Changing the discretization of states and actions afterwards is of course not possible, as it would break the correlation between state-action-pairs.

The Q-table size and discretization given in state space ranges from -40 to +40°, that is discretized by 2 leading to 21 states, centred around 0° with a discretization size of -1 to +1°. These settings are also used for the angle part of an action, while the distance is discretized by steps of 10 ranging from 1250 to 2350 DAC units → 110. A more detailed explanation is given in section 2.1.4.

2.2.3.1 The code of the agent

```
from environment import EnvLearning

import numpy as np
import math
import statistics
import os
from pathlib import Path
import matplotlib.pyplot as plt

class TDQLearning(object):
    """
    This class represents the agent program to learn from human data. The goal of the agent is to
    manouver a nanocar across a race-track and accumulate maximum reward. This is done by
    positioning the STM-tip based on the current state of the nanocar within the environment. The
    learning algorithm of the agent is based on an off-policy temporal difference algorithm, known
    as 'Q-Learning'.

    Methods
    -------
    convert_distance_to_index()
        Converts the distance into an sub-index for the Q-table.

    convert_angle_to_index()
        Converts the angle into an sub-index for the Q-table.

    evaluate_state()
        Evaluates the current state of the nanocar based on its position within the environment.

    select_move()
        The agent chooses the best action in a particular state based on the Q-table or
        by choosing a random action to explore the state.

    q_table_function()
        Calcuates the Q-Learning algorithm and updates the Q-table.

    save_q_table()
        Saves the Q-table as a binary file.
    """
    def __init__(self):
        # Directory to save the Q-table
        self.qtable_directory = os.path.dirname(os.getcwd())+'/Qtable/'

        # Q-learning hyperparameters
        self.ALPHA = 0.9
        self.GAMMA = 0.95

        # Q-learning variables
        self.q_t = []
        self.q_tt = []
        self.q_tt_max = []

        # Discretization variables
        self.DISTANCE_MIN = 1250
        self.DISTANCE_MAX = 2350
        self.DISTANCE_DIV = 10
        self.DISTANCE_RANGE = self.DISTANCE_MAX-self.DISTANCE_MIN
        self.DISTANCE_STEP = int(self.DISTANCE_RANGE/self.DISTANCE_DIV)
        self.ANGLE_MIN = -30
        self.ANGLE_MAX = 30
        self.ANGLE_RANGE = self.ANGLE_MAX-self.ANGLE_MIN
        self.ANGLE_DIV = 2
        self.ANGLE_DIV_ROUGH = 30
        self.ANGLE_STEP = int(self.ANGLE_RANGE/self.ANGLE_DIV)

        self.ANGLE_RANGE_ROUGH = int((180-self.ANGLE_MAX)/self.ANGLE_DIV_ROUGH)
        self.POSITIVE_Q_TABLE_DISCRETIZATION = np.array(np.zeros(int(self.ANGLE_RANGE_ROUGH)))
        self.NEGATIVE_Q_TABLE_DISCRETIZATION = np.array(np.zeros(int(self.ANGLE_RANGE_ROUGH)))

        # Q-table initialization based on discretization variables
        for i in range(self.ANGLE_RANGE_ROUGH):
            # Additional 7 States: [ 30, 180]
            self.POSITIVE_Q_TABLE_DISCRETIZATION[i] = (self.ANGLE_MAX
                                                       + self.ANGLE_DIV_ROUGH*i
                                                       + self.ANGLE_DIV_ROUGH/2)
            # Additional 7 States: [-30,-180)
```

```python
            self.NEGATIVE_Q_TABLE_DISCRETIZATION[i] = (self.ANGLE_MIN
                                                       - self.ANGLE_DIV_ROUGH*i
                                                       - self.ANGLE_DIV_ROUGH/2)

        # State variables
        self.state_angle = 0

        # Action variables
        self.action_distance = 0
        self.action_angle  = 0

        # Initialize environment
        self.env = EnvLearning()

        self.q_table = np.zeros([self.ANGLE_STEP+self.ANGLE_RANGE_ROUGH*2,
                                 self.DISTANCE_STEP+1,
                                 self.ANGLE_STEP+self.ANGLE_RANGE_ROUGH*2])

        # Load existing Q-table
        if Path(f"{self.qtable_directory}qtable.npy").is_file():
            self.q_table = np.load(f"{self.qtable_directory}qtable.npy")
            print(self.q_table[np.nonzero(self.q_table)])
        else:
            print ("Q-table does not exist")

    def convert_distance_to_index(self, var):
        """
        Converts the distance into a sub-index. The distance is given by the distance between the
        STM-tip and the nanocar.

        Note: In general the index determines exactly where the entry is located in the Q-table. This subsequently means an entry of
        the multidimensional Q-table uniquely defines the state and the action.

        Return
        ------
            Returns the distance as index value.
        """
        var = np.round(var)
        index_of_var = 0
        if var < self.DISTANCE_MAX-self.DISTANCE_DIV and var > self.DISTANCE_MIN:
            index_of_var = round((var-self.DISTANCE_MIN)/self.DISTANCE_DIV)
        elif var >= self.DISTANCE_MAX-self.DISTANCE_DIV:
            index_of_var = round((self.DISTANCE_MAX-self.DISTANCE_MIN-self.DISTANCE_DIV)/self.DISTANCE_DIV)
        return int(index_of_var)

    def convert_angle_to_index(self, var):
        """
        Converts the angle into an index or sub-index. The angle is given by the angle between the
        two vectors, namely the vector previous nanocar to goal position and previous nanocar to
        current nanocar position.

        Note: In general the index determines exactly where the entry is located in the Q-table.
        This subsequently means an entry of the multidimensional Q-table uniquely defines the state
        and the action.

        Return
        ------
            Returns the angle as index value.
        """
        if var >= self.ANGLE_MIN and var <= self.ANGLE_MAX:
            index = int(np.around((var+self.ANGLE_MAX)/self.ANGLE_DIV,1)) + self.ANGLE_RANGE_ROUGH
        else:
            if var <= self.ANGLE_MIN:
                index = -(np.digitize(var,self.NEGATIVE_Q_TABLE_DISCRETIZATION)
                          + self.ANGLE_RANGE_ROUGH)
            elif var >= self.ANGLE_MAX:
                index =  (np.digitize(var,self.POSITIVE_Q_TABLE_DISCRETIZATION)
                          + self.ANGLE_RANGE_ROUGH
                          + self.ANGLE_STEP)
                if index == 40:
                    index = 0
        return index

    def evaluate_action(self):
        """
```

```
        Evaluates the action state of the agent based on the positional data from the given
        environment.

        The action is given by the angle between the two vectors, namely the vector pointing from
        previous nanocar to goal and previous nanocar to current STM-tip position.

        Functions
        ---------
        angle_between_vectors(v_base, v_car, v_goal)
            Return the angle in degrees between the two vectors, namely from 'v_base to v_car' and
            from 'v_base to v_goal'.
        """
        self.action_distance = self.env.distance_between_vectors(
                                    self.env.state_position_of_nanocar_past_present[0],
                                    self.env.position_stm_tip)

        self.action_angle = self.env.angle_between_vectors(
                                    self.env.state_position_of_nanocar_past_present[0],
                                    self.env.position_stm_tip,
                                    self.env.state_position_of_goals[0])

    def evaluate_state(self):
        """
        Evaluates the current state of the nanocar based on its position within the environment.

        The state is given by the angle between the two vectors, namely the vector pointing from
        previous nanocar to goal and previous nanocar to current nanocar position.

        Functions
        ---------
        angle_between_vectors(v_base, v_car, v_goal)
            Return the angle in degrees between the two vectors, namely from 'v_base to v_car' and
            from 'v_base to v_goal'.
        """
        # Calculates the state and sets the state to 0 before any manipulation was performed
        self.state_angle = 0
        if self.env.number_of_iterations > 0:
            self.state_angle = self.env.angle_between_vectors(  self.env.state_position_of_nanocar_past_present[0],
                                                self.env.state_position_of_nanocar_past_present[1],
                                                self.env.state_position_of_goals[0])

    def q_table_function(self):
        """
        Calcuate the Q-value based on the Q-Learning algorithm and updates the Q-table.

        Functions
        ---------
        convert_distance_to_index(var)
            Converts the distance into an index or sub-index. The distance is given by the distance
            between the STM-tip and the nanocar.
        convert_angle_to_index(var)
            Converts the angle into a sub-index. The angle is given by the angle between the two
            vectors, namely the vector previous nanocar to goal position and previous nanocar to
            current nanocar position.
        """
        if self.env.know_Car == True and self.env.number_of_iterations > 1:
            q_t = 0
            q_tt_max = 0
            q_tt = 0

            self.evaluate_state()
            self.evaluate_action()

            # Action space: converts real actions to index values
            action_index = [self.convert_distance_to_index(self.action_distance),
                            self.convert_angle_to_index(self.action_angle)]

            # State space: converts real state to index value
            state_index = self.convert_angle_to_index(self.state_angle)
            next_state_index = action_index[1]

            # The Q-Learning algorithm
            q_t = self.q_table[state_index, action_index[0], action_index[1]]
            q_tt_max = np.max(self.q_table[next_state_index])
            q_tt = q_t + self.ALPHA*(self.env.reward + self.GAMMA*(q_tt_max) - q_t)
```

```
            self.q_table[state_index, action_index[0], action_index[1]] = q_tt

    def save_q_table(self):
        """ Saves the Q-table as a binary file.
        """
        np.save(f"{self.qtable_directory}/qtable", self.q_table)
        print(self.q_table[np.nonzero(self.q_table)])
```

3 Experiment and Proof of Concept

3.1 Experimental Setup

In this work, the nanocar manipulation is carried out on a *PAN Slider 4K LT-STM/AFM*, which is a low-temperature scanning tunnelling microscope (LT-STM) developed by Createc. The experiment was carried out at the setup shown in figure 3.1. The equipment for the experiment was kindly provided by the group of Leonhard Grill from the University of Graz.

The STM provides a fully open *OLE/COM control interface*, which allows the STM to be controlled by the agent program.

Both, preparation chamber and STM chamber, are cooled to 5 K. The synthesized nanocars are filled into a crucible and put inside the preparation chamber, where the nanocars get deposited on the surface by evaporating them at 150 °C for 30 min. After deposition, the sample was transferred into the STM chamber. The sample holder resides at room-temperature and therefore increases sample temperature while transferring it to the STM chamber. Since molecular movement is enhanced at elevated temperatures, the transfer time should be kept as short as possible.

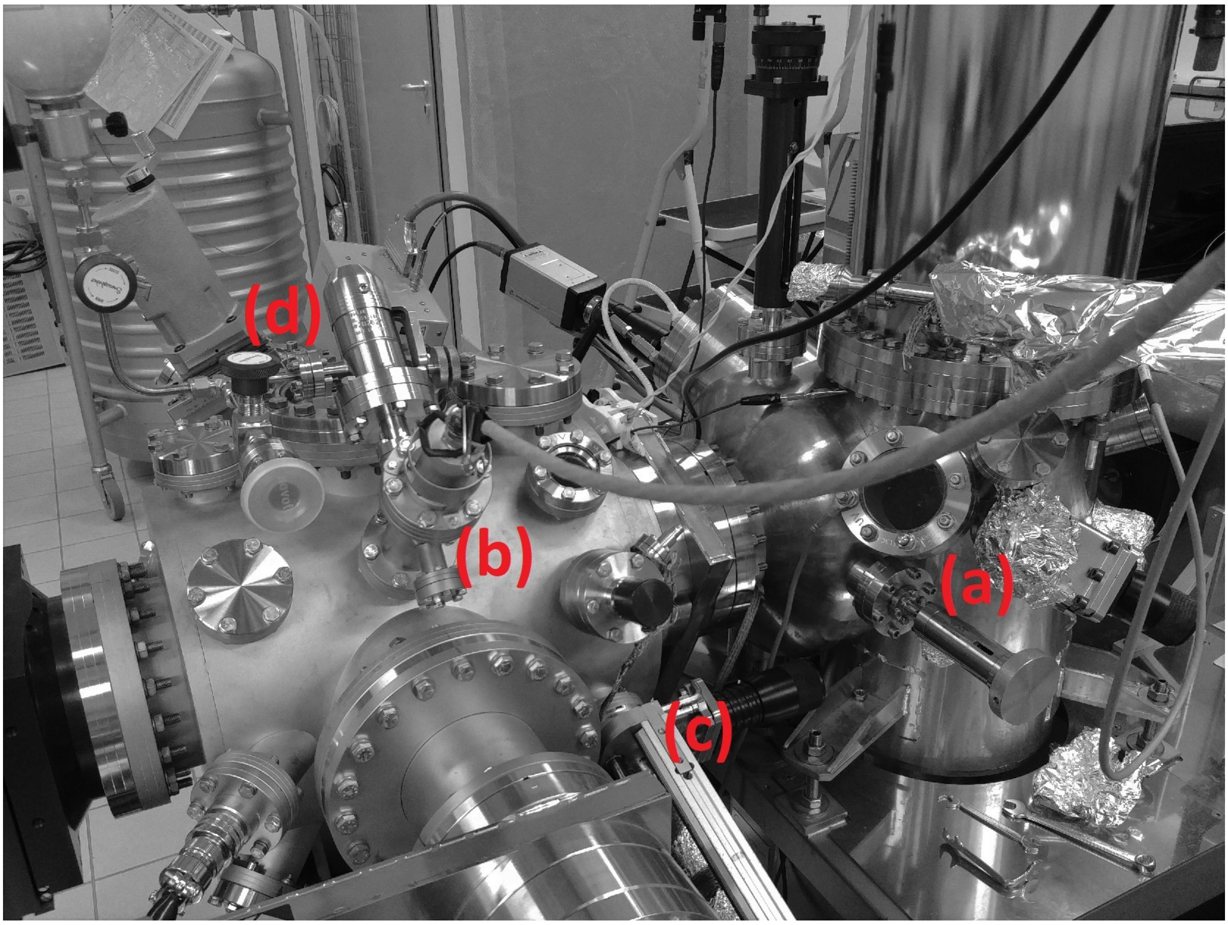


Figure 3.1: The PAN Slider 4K LT-STM/AFM is a low temperature STM. The nanocar is manoeuvred in (a) the STM chamber under UHV conditions. Inside the preparation chamber (b), the nanocar was deposited on the surface using an evaporator (c). Before depositing the nanocar, the surface was sputtering with an ion beam system (d) to provide an extremely flat and clean silver (111)-surface.

The nanocars are extracted from the island using a lateral manipulation. A lateral manipulation is a manoeuvre, where the STM-tip is moved within the xy-plane of the surface while maintaining a constant current. When extracting a molecule from an island, a small voltage in combination with a relatively high current is used and therefore the STM-tip approaches very close to the surface tearing out nanocars from the island.

While searching for the nanocar, a Z-topography is measured by using a higher voltage, but a much lower current. This moves the STM-tip further away from the nanocar and prevents additional translation or rotation. It should be emphasised that it is extremely important to not induce additional movement while searching, because the agent should learn the cause and effect for specific actions.

The manipulation of the nanocar was done using a vertical manipulation. The vertical manipulation is used for performing a voltage pulse at a given xy-position while maintaining a constant current. The electric field of the STM-tip interacts with the dipole of the nanocar and induces a movement towards the tip. The detailed settings for the different scenarios are given in table 3.1.

Table 3.1: Experimental condition and STM settings

T_s ... Temperature of the sample stage
p ... Pressure within the STM chamber
V_{bias} ... Bias voltage between tip and surface
I_t ... Tunnelling current between tip and surface
Z_{offset} ... Z approach towards the surface during the measurement

Parameter	Value
Conditions	
T_s	5 K
p	$5.00 \cdot 10^{-10}$ mbar
Nanocar extraction: lateral manipulation	
V_{bias}	0.010 V
I_t	0.300 nA
Z_{offset}	0.00 $\AA$
Nanocar manoeuvre: vertical manipulation	
V_{bias}	1.800 V
I_t	0.012 nA
Z_{offset}	2.50 $\AA$
Nanocar searching: lateral manipulation	
V_{bias}	1.000 V
I_t	0.012 nA
Z_{offset}	0.00 $\AA$

The following table 3.2 shows the conversion formulas for DAC to Angstroms, Ampere, Volt and Pixel units.

Table 3.2: Conversion formulas for DAC units to:

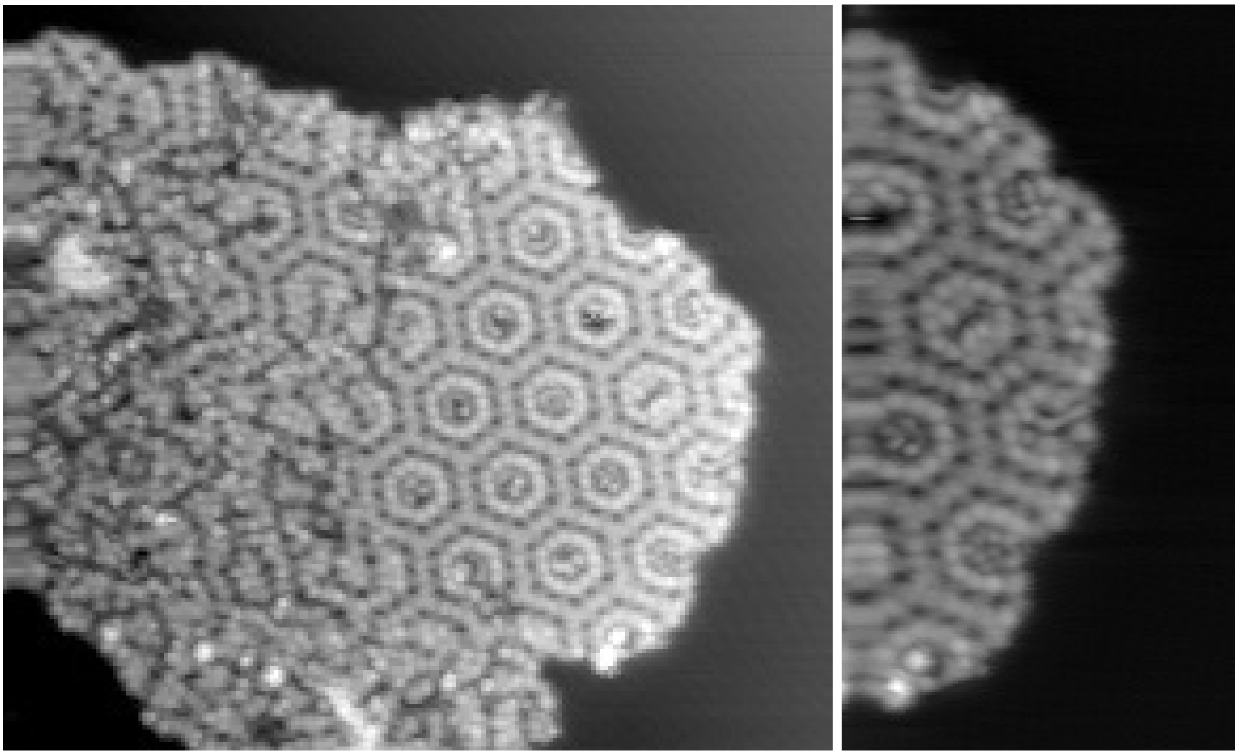
DAC ... DAC value
DAC_{Type} ... Digital to analogue converter (DAC) is 20 bit so its value is 20
Gain ... Gain for the piezocrystals in X and Y direction is given by 10
piezoconstant... Piezoconstant in X and Y direction is 29.42 for the STM used for learning and 43.50 for the STM used in the experiment
gainpreamp ... Tunnelling current amplification by a factor of 10

Unit	Formula
Angstroms	$DAC \cdot \frac{DAC_{Type}}{2^{DAC_{Type}}} \cdot GainX \cdot Xpiezoconst$
	$DAC \cdot \frac{DAC_{Type}}{2^{DAC_{Type}}} \cdot GainY \cdot Ypiezoconst$
Ampere	$DAC \cdot \frac{DAC_{Type}}{2^{DAC_{Type}} \cdot 10^{gainpreamp}} \cdot$
Volt	$DAC \cdot \frac{DAC_{Type}}{2^{DAC_{Type}}} \cdot GainX$
	$DAC \cdot \frac{DAC_{Type}}{2^{DAC_{Type}}} \cdot GainY$
Pixel	$\frac{DAC}{DeltaX}$
	$\frac{DAC}{DeltaY}$

3.2 Experiment

3.2.1 Nanocar extraction procedure

Before the agent can manoeuvre a nanocar, it has to be extracted from an island. Islands with well-ordered structures, shown in figure 3.2, are formed when only nanocars are present on the surface. If there are adsorbates within the island, the pattern gets disrupted or is not formed at all. The nanocars deposited on a silver (111)-surface will form large islands, which preferably start to grow at the step edges of a terrace.



(a) (b)

Figure 3.2: The STM image of an island on the right-hand side is mostly composed of nanocars (a) forming perfectly ordered structures. In the magnified image (b) of the island, the individual nanocars are resolved.

The complete extraction procedure is pictured in figure 3.3. A single nanocar can be extracted by performing lateral manipulations at the border of an island with the settings given in table 3.1. The extraction process can be considered successful, when a characteristic Z-signal is measured. An undamaged and fully functional nanocar is shaped like a peanut, shown in figure 3.3d.

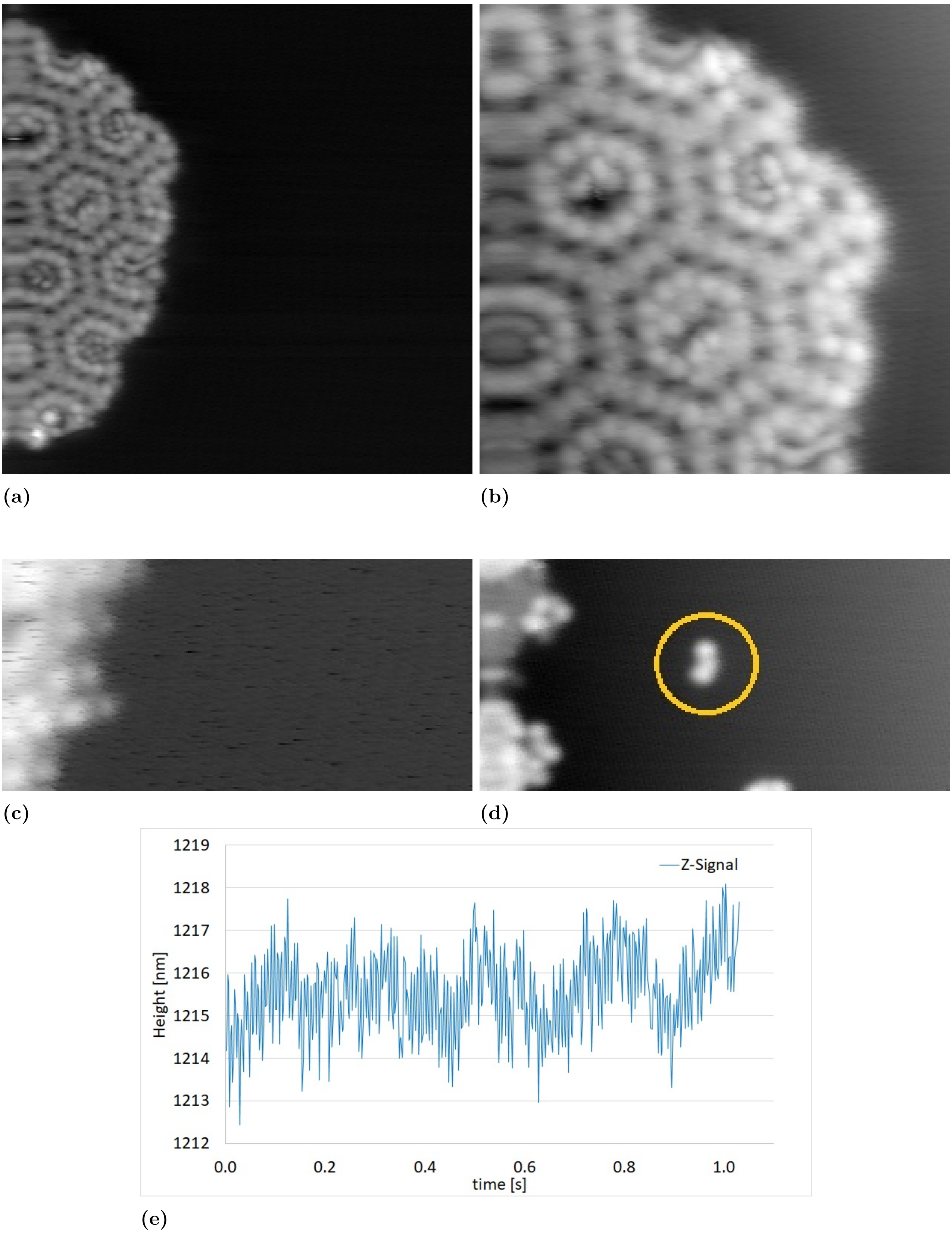


Figure 3.3: The extraction process of a single nanocar from an island. (a) Overview of an island composed of almost pure nanocars and a large empty space to manoeuvre afterwards. (b) Magnified image of pure nanocars at the border of the island and forming a well-ordered structure. (c) The boarder of the island, where a nanocar is extracted (d) Single nanocar extracted after several lateral manipulations. Also the island was torn apart in this procedure. (e) Characteristic feedback of the Z-signal while the nanocar is extracted from the island.

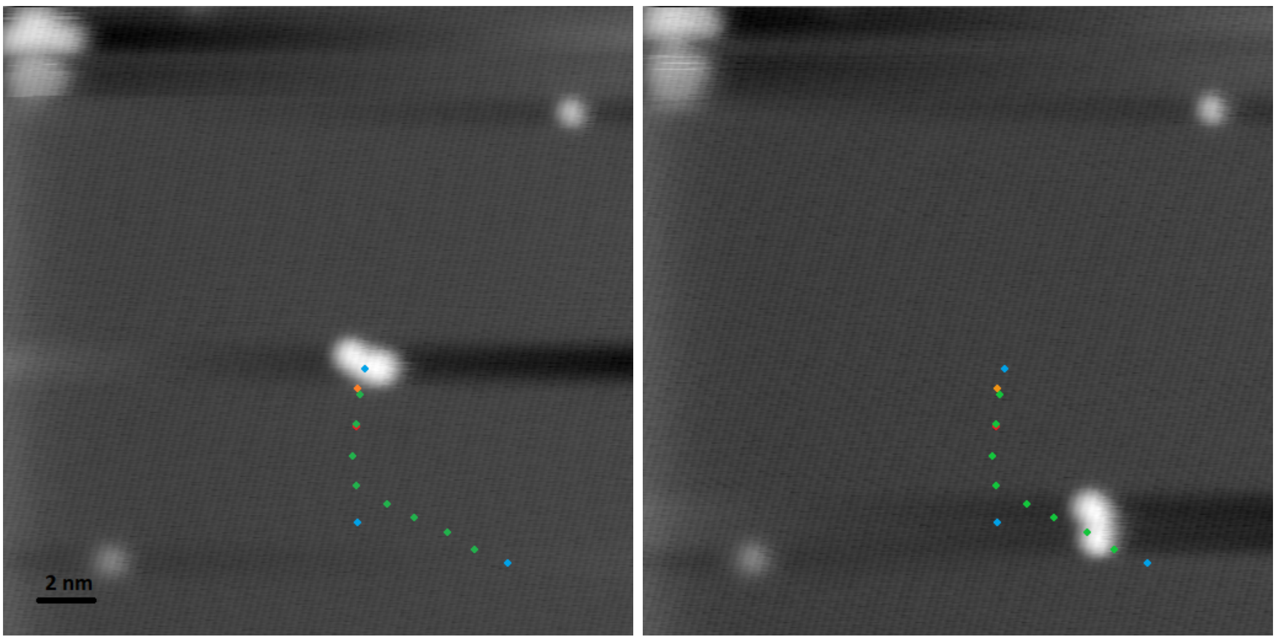
3.2.2 AI-controlled nanocar

After a single nanocar is extracted, it gets manoeuvred over a racetrack, as it is shown in figure 3.4. The environment for the agent is defined by the blue dots: the start, one sub-goal and the finish. The AI completed the racetrack with eight successful and one failed action and is showing a success-rate of 89%. The nanocar was manoeuvred over a distance of about 7.5 nm in 110 s, which means the nanocar was manoeuvred at a speed of 248 nm h^{-1} over the surface.

The analysis of the race gives interesting insights into the movement behaviour of nanocars, but also what crucial role its orientation plays relative to the positioning of the vertical manipulation.

The first vertical manipulation was successful and moved the border of the nanocar towards the STM-tip. Due to the suboptimally chosen starting position, the movement was just a small fraction. The second vertical manipulation did not promote a translation, but a rotation - leading to a failed action. The search algorithm was performed and determined the nanocars centre of mass, which was pretty close to the previous tip position, which is supporting the theory of rotation. This rotation moved the border of the nanocar closer to the next position of vertical manipulation, such that although the position is quite the same, this time the action succeeded.

After a pulling action, the border of the nanocar moved to the tip position, like it can be seen, when the nanocar reached the finish in figure 3.4b. This is clear, when considering the STM-tip is predominantly interacting with the dipole of the nanocar, which is pointing outwards and located at its boarder 1.2. Thus, not the centre of the nanocar, but the head and tail position of the dipole are the ones we are interested in.



(a) Start **(b)**

Figure 3.4: The AI manoeuvring the nanocar over a given racetrack defined by start and goals and solving the racetrack by manoeuvring the nanocar with eight successful actions and one failed action towards the goal. After a failed action, meaning the nanocar did not translate below the STM-tip, the search algorithm finds the position of the nanocar again.

The environment determines when the goal is reached by defining a distance around the goal. If the STM-tip is located within this range, the goal is supposed to be reached after a successful action. This can be seen at the sub-goal, where the AI changes the direction right before the sub-goal, and manoeuvres the nanocar straight towards the finish, where it again stops within a 1.4 nm radius around the goal.

4 Conclusion and outlook

In the proof of concept 3.2.2, the AI impressively demonstrated its performance. In the prime example shown here, the nanocar was manoeuvred with eight successful steps towards the goal showing an success-rate of 89%; compared to 54% for humans. Hence impressive stats were accomplished, as the nanocar solved a 7.5 nm racetrack in 110 s moving at an average speed of 248 nm h^{-1}. In the first nanocar race in Toulouse, a 150 nm racetrack was solved in about 1.33 h, which corresponds to an average speed of 112 nm h^{-1}.

Our experiment showed the alluring prospect of reinforcement learning based AI in controlling single molecules across a surface. However, not every racetrack could be solved with the current version of the agent and the issues that persist could not be solved, as this would go beyond the scope of this thesis. There are minor and major solutions required - like defining states in terms of the dipole orientation and using a deep-neural network to analyse the current - that will tackle this issue and improve reliability as well as universality of the AI.

In this thesis, the state is based on the fact that the AI will figure out which action is the best in a particular state, and this state is given by the angle between the vectors, starting at the old nanocar position once to the to goal and the other to the current nanocar position. This is not the most elegant way of defining the state of the nanocar, because the nanocar on a FCC (111)-surface has a six-fold symmetry.

A more sophisticated definition for the states would be to use the angle between dipole direction of the nanocar relative to the direction of the goal. In that way, the orientation of the nanocar would be completely defined. This would be done by extending the search algorithm and determining the dipole direction via the central axis of the nanocar, because the axis and the dipole orientation are simply shifted by an angle of 90 °.

This approach would be ideal for learning the perfect actions to the corresponding states and vice versa - knowing the effect (next state) for any taken action. In order to learn perfect correlations while still being competitive, there would have to be a training mode and a performance mode. In the training mode, where speed is irrelevant, a topography profile of the nanocar is recorded after every action in order to determine its exact state. While in performance mode, the topography is only recorded when the nanocar gets lost and the agent assumes to know every state of the nanocar due to the correlation of the performed action.

The universality of the agent could be realised by complementing the existing AI, which is responsible for manoeuvring the nanocar with a deep neural network. The neural network analyses the current signal, which contains a unique rotation and translation pattern that is acting like a fingerprint for every molecule. This allows molecules to be identified and provide insight into how they move during a manipulation.

This can easily be the foundation for more sophisticated techniques of molecular manipulations, where the AI is not limited to specific molecules, but every molecule can be placed at will - forming the basis for autonomous assembly and future bottom-up constructions of nanotechnology.

List of Figures

Bibliography

[1] L. Bartels, G. Meyer, and K.-H. Rieder. “Basic Steps of Lateral Manipulation of Single Atoms and Diatomic Clusters with a Scanning Tunneling Microscope Tip”. In: *Phys. Rev. Lett.* 79 (4 July 1997), pp. 697–700. DOI: 10.1103/PhysRevLett.79.697. URL: https://link.aps.org/doi/10.1103/PhysRevLett.79.697.

[2] Greg Brockman et al. *OpenAI Gym*. 2016. eprint: arXiv:1606.01540.

[3] Adam Shwartz (eds.) Eugene A. Feinberg. *Handbook of Markov Decision Processes: Methods and Applications*. reprint. Springer, 2002. ISBN: 9780792374596.

[4] Rebala Gopinath, Ravi Ajay, and Churiwala Sanjay. *An Introduction to Machine Learning*. Springer International Publishing, 2019.

[5] L. Grill et al. “Rolling a single molecular wheel at the atomic scale”. In: *Nature Nanotechnology* (2 Feb. 2007), pp. 95–98. DOI: https://doi.org/10.1038/nnano.2006.210.

[6] Rapenne Gwénaël and Joachim Christian. “The first nanocar race”. In: *Nature Reviews Materials* 2.6 (June 6, 2017), p. 17040. DOI: 10.1038/natrevmats.2017.40. URL: https://doi.org/10.1038/natrevmats.2017.40.

[7] Kansal Satwik and Martin Brendan. *Reinforcement Q-Learning from Scratch in Python with OpenAI Gym*. URL: https://www.learndatasci.com/tutorials/reinforcement-q-learning-scratch-python-openai-gym/.

[8] Grant J. Simpson et al. “How to build and race a fast nanocar”. In: *Nature Nanotechnology* 12 (2017), pp. 604–606.

[9] Peter Norvig Stuart Russell. *Artificial Intelligence: A Modern Approach*. 3rd. Prentice Hall Series in Artificial Intelligence. Prentice Hall, 2010. ISBN: 9780136042594.

[10] Richard S. Sutton. “Learning to predict by the methods of temporal differences”. In: *Machine Learning* 3 (1988), pp. 9–44. URL: https://doi.org/10.1007/BF00115009.

[11] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. Second. The MIT Press, 2018. URL: http://incompleteideas.net/book/the-book-2nd.html.

[12] Christopher J. C. H. Watkins. “Q-learning”. In: *Machine Learning* 8 (1992), pp. 279–292. URL: https://doi.org/10.1007/BF00992698.

Appendix

The code of the Simulator

The filemanager

```python
import time
from datetime import datetime
import os
import glob
import shutil
import math
import numpy as np

class FileManager(object):
    """ A class used to read and/or write the VERT-files for either learning form human data or doing simulations.

        Methods
        -------
        get_files : list
            provides the complete path for every VERT-file within the 'directory' sorted by name
        get_latest_file : str
            provides the complete path for the latest VERT-file in the 'directory'
        get_num_files : int
            provides the number of files within the given 'directory'
        write_simulation_data(xy_data, know_Car=True)
            writes artificial data with the STM-tip position and a high or low current dependent on weather the nanocar is below the tip or not (this is determined randomly)
        read_position : array(2)
            read X/Y positin form the VERT-file
    """
    def __init__(self, directory_of_data):
        self.directory_of_data = directory_of_data
        # A unique naming scheme for every written VERT-file
        self.last_timestamp = None
        # The number of files within the given 'directory'
        self.num_files = 0

    def get_files(self):
        """ Returns the complete path for every VERT-file within the 'directory' and sorts it by name

            Returns
            -------
            files : list
                A list of strings that contain the complete filepath of every VERT-file withing the 'directory'
        """
        files = sorted(glob.glob('*.VERT'))
        self.num_files = len(files)
        return files

    """o os.chdir messes up the path as the next time it is called it    """
    def get_latest_file(self):
        files = sorted(os.listdir(self.directory_of_data), key=os.path.getmtime)
        newest = files[-1]
        return newest

    def get_num_files(self):
        return self.num_files

    def write_simulation_data(self, xy_data, know_Car=True):
        dateTimeObj = datetime.now()
        timestampStr = f"{dateTimeObj.year}-{dateTimeObj.month}-{dateTimeObj.day}_{dateTimeObj.hour}-{dateTimeObj.minute}-{dateTimeObj.second}.{dateTimeObj.microsecond}"
```

```
        self.last_timestamp = timestampStr
        new_filename = f'{timestampStr}.VERT'

        if know_Car == True:
            shutil.copyfile('Current_Right.VERT', new_filename)
        else:
            shutil.copyfile('Current_Wrong.VERT', new_filename)
        with open(new_filename, mode='r', encoding = "ISO-8859-1") as f:
            lines = f.readlines()
        with open(new_filename, mode='w', encoding = "ISO-8859-1") as f:
            lines[298] = '{:8d}{:8d}{:8d}{:10}'.format(1000, xy_data[0], xy_data[1], 1)+'\n'
            f.writelines(lines)

    def read_position(self, file=None):
        position = np.empty(2)

        if file is None:
            file = self.get_latest_file()

        with open(file, mode='r', encoding="ISO-8859-1") as f:
            f_data = f.read().split('\n')

        # X/Y-position from datafile
        xdac = float(f_data[298].split()[1])
        ydac = float(f_data[298].split()[2])

        # Offset correction
        offsetx = float(f_data[20].split('=')[1])
        offsety = float(f_data[21].split('=')[1])

        # Additional parameters
        dx = float(f_data[3].split('=')[1])
        dy = float(f_data[4].split('=')[1])
        nx = float(f_data[5].split('=')[1])
        ny = float(f_data[6].split('=')[1])

        rot = float(f_data[14].split('=')[1])

        driftxoff = 0
        driftyoff = 0

        # Rotation matrix:  cos -sin | xx xy
        #                   sin  cos | yx yy
        x_with_rotation = -(xdac*np.cos(rot*np.pi/180) - ydac*np.sin(rot*np.pi/180) + offsetx-
    driftxoff)
        y_with_rotation = -(xdac*np.sin(rot*np.pi/180) + ydac*np.cos(rot*np.pi/180) + offsety-
    driftyoff)

        position = [xdac, ydac]
        return position

    def read_current(self, file=None):
        if file is None:
            file = self.get_latest_file()

        with open(file, mode='r', encoding="ISO-8859-1") as f:
            f_data = f.read().split('\n')
        f_It = f_data[299:-1]      # Data for current and time

        t = []
        I = []
        for z in f_It:
            trunc = z.split()
            t.append(int(trunc[0]))
            I.append(float(trunc[3]))
        data_It = [t, I]
        return data_It

    def read_voltage(self, file=None):
        if file is None:
            file = self.get_latest_file()

        with open(file, mode='r', encoding="ISO-8859-1") as f:
            f_data = f.read().split('\n')
        f_Vt = f_data[299:-1]      # Data for current and time

        t = []
```

```
        V = []
        for z in f_Vt:
            trunc = z.split()
            t.append(int(trunc[0]))
            V.append(float(trunc[1]))
        data_Vt = [t, V]
        return data_Vt
```

The environment

```
from filemanager import FileManager

import numpy as np
import math
import random
import os
import glob
from datetime import datetime
import csv
import itertools
import statistics
from scipy.signal import savgol_filter
import scipy.fftpack

class EnvSimulation(FileManager):
    def __init__(self, pos_Env):
        self.directory_of_data = os.getcwd()+'/Data/1/'

        # Environment constants
        self.TRESHHOLD_CURRENT = 4000       # Current treshhold for determining if the nanocar is or is not below the tip.
        self.SEARCH_DISTANCE = 250
        self.SEARCH_STEPSIZE = 50
        self.DISTANCE_REACH_GOAL = 2500     # Treshhold in DAC units between nanocar and sub-goal/final goal

        # Environment variables
        self.position_of_environment = pos_Env
        self.position_nanocar = np.array(self.position_of_environment[0])
        self.position_stm_tip = np.array(np.zeros(2))
        self.initial_stm_position = None
        self.current_spectrum = []
        self.average_current = 0
        self.know_Car = True
        self.done = False
        self.position_nanocar_random = [None, None]
        self.set_current_spectrum_right()

        # State variables
        self.state_position_of_goals = np.array(self.position_of_environment[1:])
        self.state_position_of_nanocar_past_present = [self.position_nanocar, self.position_nanocar]

        # Reward variables and initialization
        self.reward  = 0
        self.DISTANCE_ERROR_MAX = 2350
        self.distance_to_nearest_goal = 0
        self.total_distance_to_goal = 0
        self.distance_subgoals = np.zeros(len(self.position_of_environment))
        self.init_reward_variables()        # Calculates distances between following environment positions
        self.calc_distance()                # Calculates distances to the closest sub-goal and to the final goal


        # Analysis Variables FIXME: try catch if episodes.csv does not exist
        try:
            files = glob.glob(self.directory_of_data + '*.CSV')

            if not files == []:
                latest_file = max(files, key = os.path.getctime)
                print(latest_file)
                with open(latest_file, newline='') as csv_file:
                    for line in csv_file.readlines(1):
                        self.number_of_episodes = int(line.split(',')[1])
            else:
                self.number_of_episodes = 0
```

```
                print("There are no previous episodes.")
        except OSError:
            self.number_of_episodes = 0
            print("The CSV file does not exist 2")

        self.datetime_start = datetime.now()
        self.datetime_end = 0
        self.number_of_manipulations = 0
        self.number_of_successful_manipulations = 0
        self.number_of_failed_manipulations = 0
        self.total_reward_per_episode = 0
        self.number_of_searching = 0
        self.number_of_search_steps = 0
        self.average_steps_for_searching = 0
        self.x_history_nanocar = []
        self.y_history_nanocar = []
        self.x_history_searching_nanocar = []
        self.y_history_searching_nanocar = []
        self.total_distance = self.total_distance_to_goal*0.000561142

    def init_reward_variables(self):
        """ Calculates the distance between all following sub-goals or sub-goal to goal that were set in the initialization step of the environment.
            These are necessary for the reward function.
        """
        # Distance between initial nanocar position to first sub-goal or already to the final goal
        self.distance_subgoals[0] = np.linalg.norm(np.subtract(self.position_nanocar,self.position_of_environment[1]))

        # Distances between sucessive sub-goals and sub-goal to final goal.
        if len(self.position_of_environment) > 1:
            for i in range(1,len(self.position_of_environment)):
                self.distance_subgoals[i] = np.linalg.norm(np.subtract(self.position_of_environment[i-1],self.position_of_environment[i]))

    def set_position(self):
        """ Writes simulation data
        """
        #self.write_simulation_data(self.position_stm_tip, self.know_Car)  # know_Car is necessary for "test data" writing

    def random_Car_Data(self):
        if np.random.randint(0,100) < 60:
            print('Random Nanocar')
            #self.write_simulation_data(self.position_stm_tip, False)
            #range_rnd_pos = self.DISTANCE_ERROR_MAX/np.sqrt(2)/2 # Enable!
            range_rnd_pos = self.DISTANCE_ERROR_MAX/np.sqrt(2)/5

            pos_rnd_x = np.random.randint(-range_rnd_pos, range_rnd_pos)
            pos_rnd_y = np.random.randint(-range_rnd_pos, range_rnd_pos)
            self.position_nanocar_random[0] = int(np.round(self.position_stm_tip[0] + pos_rnd_x))
            self.position_nanocar_random[1] = int(np.round(self.position_stm_tip[1] + pos_rnd_y))
            self.set_current_spectrum_wrong()
        #else:
            #self.set_position()

    def set_position_history(self):
        """ Saves either the position of the nanocar as long as its position is known or the position of the STM-tip while searching for it.
        """
        if self.know_Car == True:
            self.x_history_nanocar=np.append(self.x_history_nanocar, self.position_stm_tip[0])
            self.y_history_nanocar=np.append(self.y_history_nanocar, self.position_stm_tip[1])
        else:
            self.x_history_searching_nanocar=np.append(self.x_history_searching_nanocar, self.position_stm_tip[0])
            self.y_history_searching_nanocar=np.append(self.y_history_searching_nanocar, self.position_stm_tip[1])

    def calc_distance(self):
        """ Calculates the distance from the nanocar to the nearest goal; and from the nanocar to the final goal.
            Deletes the position of a goal when the goal is reached and also deletes the reward variable of the previous sub-goal distance.
        """
        # Calculates the distance to the nearest goal
```

```
        self.distance_to_nearest_goal = np.linalg.norm(np.subtract(self.position_nanocar, self.state_position_of_goals[0]))
        # Calculates the total distance to the goal
        self.total_distance_to_goal = self.distance_to_nearest_goal
        for i in range(1,len(self.state_position_of_goals)):
            self.total_distance_to_goal += np.linalg.norm(np.subtract(self.state_position_of_goals[i-1],self.state_position_of_goals[i]))

        # When a sub-goal is reached, the sub-goal gets deleted. Also, the reward variable for the previous sub-goal distance gets deleted.
        if len(self.state_position_of_goals) > 0:
            if self.distance_to_nearest_goal < self.DISTANCE_REACH_GOAL:
                # When a goal is reached, the relative angle changes dramatically, this has to be compensated by adding the absolute angle
                self.state_position_of_goals = np.delete(self.state_position_of_goals,0,0)
                self.distance_subgoals = np.delete(self.distance_subgoals,0,0)

    def get_nanocar_position(self):
        """ Returns the latest known position of the nanocar.
        """
        return self.position_nanocar

    def get_state_position_of_goals(self):
        """ Returns all the goal positions, like sub-goals and the final goal.

            Returns
            -------
            self.state_position_of_goals : np.array(len(self.position_of_environment[1:]), 2)
                The goal positions.
        """
        return self.state_position_of_goals

    def get_total_distance(self):
        """ Returns the total distance from the nanocar to the final goal.

            Returns
            -------
            self.total_distance_to_goal : float
                The total distance from nanocar to goal.
        """
        return self.total_distance_to_goal

    def unit_vector(self, vector):
        """ Returns the unit vector of the vector.  """
        vector = np.array(vector)
        if vector.all() == 0:
            return [0,0]
        elif not vector.all() == 0:
            return vector / np.linalg.norm(vector)

    def distance_between_vectors(self, vector1, vector2):
        """ Returns the distance between two vectors.

            Attributes
            ----------
            vector1 : np.array(len(vector1))
                Vector 1.
            vector2 : np.array(len(vector2))
                Vector 2.

            Return
            ------
            vector_distance : float
                The distance between vector1 and vector2.
        """
        vector1 = np.array(vector1)
        vector2 = np.array(vector2)
        vector_distance = 0
        if not np.array_equal(vector1,vector2):
            vector_distance = np.linalg.norm(np.subtract(vector1,vector2))
        return vector_distance

    def angle_between_vectors(self, v_base, v_car, v_goal):
        """ Return the angle in degrees between the two vectors, namely from 'v_base to v_car' and from 'v_base to v_goal'.

```

```
            Note: The function considers if the relative vector of the nanocar 'v_base to v_car' is positioned
            clockwise or counter-clockwise from the relative vector 'v_base to v_goal'.

            Attributes
            ----------
            v_base : np.array(2)
                Vector to the basis.
            v_car : np.array(2)
                Vector to the nanocar.
            v_goal : np.array(2)
                Vector to the goal.

            Return
            ------
            angle : float
                The angle spanned by the two vectors: 'v_base to v_car' and from 'v_base to v_goal'.
        """
        v_base = np.array(v_base)
        v_car = np.array(v_car)
        v_goal = np.array(v_goal)

        # Calculates the relative vectors of the nanocar and the goal
        v_car_rel = v_car-v_base
        v_goal_rel = v_goal-v_base

        # Calculates the unit vectors of the relative vectors nanocar and goal
        v_car_u = self.unit_vector(v_car_rel)
        v_goal_u = self.unit_vector(v_goal_rel)

        # Calculates the angle between the two relative vectors nanocar and goal
        angle = np.arccos(np.clip(np.dot(v_car_u, v_goal_u), -1.0, 1.0))*180/np.pi
        # Use the property of the determinant that is, if the det < 0 the,
        # relative vector of the nanocar is clockwise to the relative vector of the goal.
        if np.linalg.det([v_goal_u,v_car_u])<0:
            angle = -angle
        return angle

    def set_current_spectrum_right(self):
        self.current_spectrum = np.array(self.read_current(self.directory_of_data+'/Current_Right.VERT'))

    def set_current_spectrum_wrong(self):
        self.current_spectrum = np.array(self.read_current(self.directory_of_data+'/Current_Wrong.VERT'))

    def get_average_current(self):
        """ Calculates and returns the average current of the latest vertical manipulation step.

            Functions
            ---------
            stm.get_current_spectrum()
                Reads the current spectrum from the ADC channels of the STMAFM program.

            Return
            ------
            self.current_spectrum : list([number of datapoints])
                Contains the current spectrum.
        """
        self.average_current = int(np.mean(self.current_spectrum[self.current_spectrum > 0]))
        return self.average_current


    """ Calculates the position of the STM-tip due to a given moving_length ================================ """
    def calc_next_position(self, length, angle):
        angle = angle*np.pi/180
        theta = 0
        """ Defines the direction the nanocar has to drive. This is the relative
        direction the tip will be positioned next, while the distance will be
        solved by the neural network. """
        dx = np.subtract(self.state_position_of_goals[0][0], self.position_nanocar[0])
        dy = np.subtract(self.state_position_of_goals[0][1], self.position_nanocar[1])

        if dx>0:
            theta = np.arctan(dy/dx)
        elif dx<0 and dy>=0:
```

```
            theta = np.arctan(dy/dx)+np.pi
        elif dx<0 and dy<0:
            theta = np.arctan(dy/dx)-np.pi
        elif dx==0 and dy>0:
            theta = np.pi/2
        elif dx==0 and dy<0:
            theta = -np.pi/2

        #print(f'Angle: {angle*180/np.pi}')
        print(f'Theta: {theta*180/np.pi}')
        print('Nanocar position: %s' % self.position_nanocar)
        print('Goal position: %s' % self.state_position_of_goals[0])
        pos_STM_x = int(np.round(self.position_nanocar[0] + length*np.cos(angle+theta),2))
        pos_STM_y = int(np.round(self.position_nanocar[1] + length*np.sin(angle+theta),2))

        self.position_stm_tip = [pos_STM_x, pos_STM_y]

        # Sets the STM-position or puts the nanocar with a certain percentage to a random position
        self.random_Car_Data()
        self.number_of_manipulations += 1

    def check_current_pattern(self):
        """ Checks if the average current of the current pattern measured after a pulling action is higher than a certain treshhold.
            If this is:
            - TRUE: The position of the nanocar is below the STM-tip - hence it is known
            - FALSE: The position of the nanocar is not below the STM-tip - hence it is unknown and a search-algorithm starts searching for the nanocar.

            Functions
            ---------
            get_average_current()
                Calculates the average current induces to the STM-tip after a pulling action.
            reward_function()
                Calculates the reward the agnet receives.
            search_car()
                Searching the nanocar if the it got lost.
        """
        self.get_average_current()

        if self.average_current >= self.TRESHHOLD_CURRENT and self.know_Car == True:    # I is RIGHT
            print("Current pattern is right!")
            self.number_of_successful_manipulations += 1
            self.position_nanocar = self.position_stm_tip.copy()
            self.state_position_of_nanocar_past_present = [self.state_position_of_nanocar_past_present[1], self.position_nanocar]
            self.initial_stm_position = None
            self.reward_function()

        elif self.average_current < self.TRESHHOLD_CURRENT and self.know_Car == True:    # I is WRONG
            print("Current pattern is wrong! == Car is lost ==")
            self.number_of_failed_manipulations += 1
            self.know_Car = False
            self.initial_stm_position = self.position_stm_tip.copy()
            self.search_car()

        elif self.average_current >= self.TRESHHOLD_CURRENT and self.know_Car == False:    # I is RIGHT
            print("Current pattern is right! == Car is found ==")
            self.know_Car = True
            self.position_nanocar = self.position_stm_tip.copy()
            self.state_position_of_nanocar_past_present = [self.state_position_of_nanocar_past_present[1], self.position_nanocar]
            self.reward_function()

    def search_car(self):
        ' XXX XXX XXX Adjust search parameters such that it is in the dimension of the nanocar XXX XXX XXX '
        """ Search for the nanocar in a circular pattern with increasing radius. A high current response will indicate, that the nanocar is below the STM-tip.

            Functions
            ---------
            define_voltage_pulse_searching()
```

```
                Defines the voltage pulse to search for the nanocar such that it does not translate when the voltage is applied.
            set_position()
             Sets the STM-tip position based on the search-algorithm.
        """
        # The center of the search-algorithm is the last pulling position of the STM-tip
        centre_of_search_algorithm = self.position_stm_tip.copy()
        self.number_of_searching+=1
        search_steps=0
        # Positions the STM-tip in a circular pattern and search pattern with increasing radius
        for radii, phi in itertools.product(range(self.SEARCH_STEPSIZE, 10000, self.SEARCH_STEPSIZE), range(0, 370, 5)): #FIXME: radii and angle step size
            dx = radii*np.cos(phi*np.pi/180)
            dy = radii*np.sin(phi*np.pi/180)
            self.position_stm_tip = [int(round(centre_of_search_algorithm[0] + dx)), int(round(centre_of_search_algorithm[1] + dy))]
            search_steps+=1
            self.check_distance_to_random_nanocar()
            self.check_current_pattern()
            self.set_position_history()
            self.number_of_search_steps+=1

            # If the current pattern is right, searching is finished
            if self.know_Car == True:
                break

    def check_distance_to_random_nanocar(self):
        distance = self.distance_between_vectors(self.position_stm_tip, self.position_nanocar_random)
        if distance <= self.SEARCH_DISTANCE:
            self.set_current_spectrum_right()

    def reward_function(self):
        """ Calculates the reward to measure the performance of the agents actions. The reward is calculated by using two functions.
            1. Reward function calculates how precisely the nanocar has moved below the STM-tip
            2. Reward function calculates how close the nanocar moved towards the goal.

            Functions
            ---------
            distance_between_vectors(vector1, vector2)
                Calclates the distance between two vectors.
        """
        self.reward  = 0

        if self.number_of_manipulations >= 1:
            position_of_nanocar_past = self.state_position_of_nanocar_past_present[0]
            position_of_nanocar_present = self.state_position_of_nanocar_past_present[1]
            position_of_nearest_goal = self.state_position_of_goals[0]

            # Calculates the distane to the goal before and after the pulling action
            distance_of_past_nanocar_to_goal = self.distance_between_vectors(position_of_nanocar_past, position_of_nearest_goal)
            distance_of_present_nanocar_to_goal = self.distance_between_vectors(position_of_nanocar_present, position_of_nearest_goal)
            difference_in_distance_from_goal_between_pulling_action = np.subtract(distance_of_past_nanocar_to_goal, distance_of_present_nanocar_to_goal)

            # Calculates by how much the nanocar translated to an unknown position
            if self.initial_stm_position is None:
                nanocar_deviates_from_initial_stm_position = 0
                self.initial_stm_position = position_of_nanocar_present
            else:
                nanocar_deviates_from_initial_stm_position = self.distance_between_vectors(self.initial_stm_position, position_of_nanocar_present)

            # Calculates the reward using two reward functions
            self.reward = 0
            # 1. Reward function
            if difference_in_distance_from_goal_between_pulling_action > 0 and self.total_distance_to_goal > 0:
                self.reward  += 0.5*(1-self.distance_to_nearest_goal/self.distance_subgoals[0])
            elif difference_in_distance_from_goal_between_pulling_action <= 0 and self.total_distance_to_goal >= 0:
                self.reward  -= 1
            # 2. Reward function
            if nanocar_deviates_from_initial_stm_position <= self.DISTANCE_ERROR_MAX:
```

```
                    self.reward  += 1-math.pow(nanocar_deviates_from_initial_stm_position/self.
    DISTANCE_ERROR_MAX,0.4)
            self.total_reward_per_episode += self.reward
            print(f'Reward: {self.reward}')

    def is_done(self):
        if len(self.state_position_of_goals) <= 0:
            self.done = True
            self.datetime_end = datetime.now()
            self.number_of_episodes+=1
            print("The course was solved!")
        return self.done
```

The agent

```
import numpy as np
import random
import math
import os
import glob
import matplotlib.pyplot as plt
from pathlib import Path
import statistics
from environment import EnvSimulation
from datetime import datetime

class TDQSimulation(object):
    """ This class represents the agent program.
        The goal of the agent is to manouvers a nanocar across a race-track and accumulate maximum
    reward.
        This is done by positioning the STM-tip based on the current state of the nanocar within the
     environment.
        The learning algorithm of the agent is based on an off-policy temporal difference algorithm,
     known as 'Q-Learning'.

        Methods
        -------
        convert_distance_to_index()
            Converts the distance into an sub-index for the Q-table.
        convert_angle_to_index()
            Converts the angle into an sub-index for the Q-table.
        evaluate_state()
            Evaluates the current state of the nanocar based on its position within the environment.
        select_move()
            The agent chooses the best action in a particular state based on the Q-table or
            by choosing a random action to explore the state.
        q_table_function()
            Calcuate the Q-Learning algorithm and updates the Q-table.
        save_q_table()
            Saves the Q-table as a binary file.
    """
    def __init__(self, pos_Env):
        # Directory to save the Q-table
        self.qtable_directory = os.path.dirname(os.getcwd())+'/Qtable/'

        # Q-learning hyperparameters
        self.ALPHA = 0.9
        self.GAMMA = 0.95

        # Learning variables
        self.epsilon = 0.7   # Exploration rate [%]

        self.ANGLE_LOWER_LIMIT = -4
        self.ANGLE_UPPER_LIMIT = 4
        self.DISTANCE_LOWER_LIMIT = 1500
        self.DISTANCE_UPPER_LIMIT = 1900

        # Q-learning variables
        self.q_t = []
        self.q_tt = []
        self.q_tt_max = []

        # Discretization variables
        self.DISTANCE_MIN = 1250
        self.DISTANCE_MAX = 2350
        self.DISTANCE_DIV = 10
```

```
        self.DISTANCE_RANGE = self.DISTANCE_MAX-self.DISTANCE_MIN
        self.DISTANCE_STEP = int(self.DISTANCE_RANGE/self.DISTANCE_DIV)
        self.ANGLE_MIN = -30
        self.ANGLE_MAX = 30
        self.ANGLE_RANGE = self.ANGLE_MAX-self.ANGLE_MIN
        self.ANGLE_DIV = 2
        self.ANGLE_DIV_ROUGH = 30
        self.ANGLE_STEP = int(self.ANGLE_RANGE/self.ANGLE_DIV)

        self.ANGLE_RANGE_ROUGH = int((180-self.ANGLE_MAX)/self.ANGLE_DIV_ROUGH)
        self.POSITIVE_Q_TABLE_DISCRETIZATION = np.array(np.zeros(int(self.ANGLE_RANGE_ROUGH)))
        self.NEGATIVE_Q_TABLE_DISCRETIZATION = np.array(np.zeros(int(self.ANGLE_RANGE_ROUGH)))

        # Q-table initialization based on discretization variables
        for i in range(self.ANGLE_RANGE_ROUGH):
            # Additional 7 States: [ 30, 180]
            self.POSITIVE_Q_TABLE_DISCRETIZATION[i] = self.ANGLE_MAX+self.ANGLE_DIV_ROUGH*i+self.
    ANGLE_DIV_ROUGH/2
            # Additional 7 States: [-30,-180)
            self.NEGATIVE_Q_TABLE_DISCRETIZATION[i] = self.ANGLE_MIN-self.ANGLE_DIV_ROUGH*i-self.
    ANGLE_DIV_ROUGH/2

        # State variables
        self.state_angle = 0

        # Action variables
        self.action_distance = 0
        self.action_angle  = 0

        # Initialize environment
        self.env = EnvSimulation(pos_Env)

        self.q_table = np.zeros([self.ANGLE_STEP+self.ANGLE_RANGE_ROUGH*2, self.DISTANCE_STEP+1,
    self.ANGLE_STEP+self.ANGLE_RANGE_ROUGH*2])

        # Load existing Q-table
        files =  glob.glob(f'{self.qtable_directory}*.npy')
        if not files == []:
            latest_file = max(files, key=os.path.getmtime)
            self.q_table = np.load(latest_file)
            print(latest_file)
            print(self.q_table[np.nonzero(self.q_table)])
            print('The Q-table is loaded!')
        else:
            print ("Q-table does not exist")



    def convert_distance_to_index(self, var):
        """ Converts the distance into an index or sub-index. The distance is given by the distance
    between the STM-tip and the nanocar.

            Note: In general the index determines exactly where the entry is located in the Q-table.
     This subsequently means an entry of
            the multidimensional Q-table uniquely defines the state and the action.

            Return
            ------
                Returns the distance as index value.
        """
        var = np.round(var)
        index_of_var = 0
        if var <= self.DISTANCE_MAX and var >= self.DISTANCE_MIN:
            index_of_var = np.round((var-self.DISTANCE_MIN)/self.DISTANCE_DIV,1)
        elif var > self.DISTANCE_MAX:
            index_of_var = np.round((self.DISTANCE_MAX-self.DISTANCE_MIN)/self.DISTANCE_DIV,1)
        return int(index_of_var)

    def convert_angle_to_index(self, var):
        """ Converts the angle into an sub-index. The angle is given by the angle between the two
    vectors, namely the vector
            previous nanocar to goal position and previous nanocar to current nanocar position.

            Note: In general the index determines exactly where the entry is located in the Q-table.
     This subsequently means an entry of
            the multidimensional Q-table uniquely defines the state and the action.

```

```
            Return
            ------
                Returns the angle as index value.
        """
        if var >= self.ANGLE_MIN and var <= self.ANGLE_MAX:
            return int(np.around((var+self.ANGLE_MAX)/self.ANGLE_DIV,1)) + self.ANGLE_RANGE_ROUGH
        else:
            #var = np.around(var,-1)
            if var <= self.ANGLE_MIN:
                return -np.digitize(var,self.NEGATIVE_Q_TABLE_DISCRETIZATION) + self.ANGLE_RANGE_ROUGH
            elif var >= self.ANGLE_MAX:
                index = np.digitize(var,self.POSITIVE_Q_TABLE_DISCRETIZATION) + self.ANGLE_RANGE_ROUGH + self.ANGLE_STEP
                if index == 40:
                    index =0
                return index

    def evaluate_state(self):
        """ Evaluates the current state of the nanocar based on its position within the environment.

            The state is given by the angle between the two vectors, namely the vector pointing from
            previous nanocar to goal and previous nanocar to current nanocar position.

            Functions
            ---------
            angle_between_vectors(v_base, v_car, v_goal)
                Return the angle in degrees between the two vectors, namely from 'v_base to v_car' and from 'v_base to v_goal'.
        """
        # Calculates the state and sets the state to 0 before any manipulation was performed
        self.state_angle = 0
        if self.env.number_of_manipulations > 0:
            self.state_angle = int(self.env.angle_between_vectors(  self.env.state_position_of_nanocar_past_present[0],
                                                                  self.env.state_position_of_nanocar_past_present[1],
                                                                  self.env.state_position_of_goals[0]))

    def select_move(self):
        """ The agent chooses the best action in a particular state based on the Q-table or
            by choosing a random action to explore the state.

            XXX Choose small angles first to fill the Q-table at smaller angles first XXX
            XXX Explore function to explore state using random actions XXX
        """
        self.evaluate_state()
        print(self.state_angle)
        state_angle_index = self.convert_angle_to_index(self.state_angle)
        action_index = np.zeros(2)

        if random.uniform(0,1) < self.epsilon:
                # Calculate indices to corresponding limits
                lower_distance_index = self.convert_distance_to_index(self.DISTANCE_LOWER_LIMIT)
                upper_distance_index = self.convert_distance_to_index(self.DISTANCE_UPPER_LIMIT)+1
                lower_angle_index = self.convert_angle_to_index(self.ANGLE_LOWER_LIMIT)
                upper_angle_index = self.convert_angle_to_index(self.ANGLE_UPPER_LIMIT)+1

                # Determine all Q-table entries that were never used: Q-value == 0
                actions_never_used_index = np.where(self.q_table[state_angle_index]==0)

                # Determine indices which are within the limit
                limited_actions_never_used_index = [(actions_never_used_index[0][:]<= upper_distance_index) &
                                                    (actions_never_used_index[0][:]>= lower_distance_index) &
                                                    (actions_never_used_index[1][:]<= upper_angle_index) &
                                                    (actions_never_used_index[1][:]>= lower_angle_index)]

                # Select the actions that are never used and are within the limits
                actions_never_used_index = [actions_never_used_index[0][limited_actions_never_used_index],
                                            actions_never_used_index[1][limited_actions_never_used_index]]

```

```python
                # From all actions within the limit randomly chose one action
                action_random_never_used_index = np.random.randint(0,len(actions_never_used_index[0]))
                distance_never_used_index = actions_never_used_index[0][action_random_never_used_index]
                angle_never_used_index = actions_never_used_index[1][action_random_never_used_index]
                action_index = [distance_never_used_index, angle_never_used_index]
        else:
            # Select the best action
            action_best_index = np.where(self.q_table[state_angle_index]==np.max(self.q_table[state_angle_index]))

            # From equally good actions select one of them randomly
            action_random_best_index = np.random.randint(0,len(action_best_index[0]))
            distance_best_index = action_best_index[0][action_random_best_index]
            angle_best_index = action_best_index[1][action_random_best_index]
            action_index = [distance_best_index, angle_best_index]

        self.action_distance = self.DISTANCE_MIN + action_index[0]*self.DISTANCE_DIV
        if action_index[1] <= self.ANGLE_RANGE_ROUGH:
            self.action_angle = -180+action_index[1]*self.ANGLE_DIV_ROUGH
        elif action_index[1] >= self.ANGLE_RANGE_ROUGH + self.ANGLE_STEP:
            self.action_angle = self.ANGLE_MAX+(action_index[1]-self.ANGLE_RANGE_ROUGH-self.ANGLE_STEP)*self.ANGLE_DIV_ROUGH
        else:
            self.action_angle = self.ANGLE_MIN + (action_index[1]-self.ANGLE_RANGE_ROUGH)*self.ANGLE_DIV

        # Calculates the next STM-tip positon based on the agents choosen actions
        self.env.calc_next_position(self.action_distance, self.action_angle)

        print(f'State in deg: {self.state_angle}')
        print(f'Action in DAC: {self.action_distance}')
        print(f'Action in deg: {self.action_angle}')

    def q_table_function(self):
        """ Calcuate the Q-value based on the Q-Learning algorithm and updates the Q-table.

            Functions
            ---------
            convert_distance_to_index(var)
                Converts the distance into an index or sub-index. The distance is given by the distance between the STM-tip and the nanocar.
            convert_angle_to_index(var)
                Converts the angle into an sub-index. The angle is given by the angle between the two vectors, namely the vector
                previous nanocar to goal position and previous nanocar to current nanocar position.
        """
        if self.env.know_Car == True and self.env.number_of_manipulations > 1:
            q_t = 0
            q_tt_max = 0
            q_tt = 0

            # Action space: converts real actions to index values
            action_index = [self.convert_distance_to_index(self.action_distance),
                            self.convert_angle_to_index(self.action_angle)]

            # State space: converts real state to index value
            state_index = self.convert_angle_to_index(self.state_angle)
            next_state_index = action_index[1]

            # The Q-Learning algorithm
            q_t = self.q_table[state_index, action_index[0], action_index[1]]
            q_tt_max = np.max(self.q_table[next_state_index])
            q_tt = q_t + self.ALPHA*(self.env.reward + self.GAMMA*(q_tt_max) - q_t)
            self.q_table[state_index, action_index[0], action_index[1]] = q_tt

    def save_q_table(self):
        """ Saves the Q-table as a binary file.
        """
        path = f'{self.qtable_directory}/qtable_simulation'
        now = datetime.now()
        timestamp_file = now.strftime("%y-%m-%d_%H-%M-%S")
        path_with_timestamp = f'{self.qtable_directory}/{timestamp_file}_qtable_simulation'

        try:
            print('The Q-table is saved!')
```

```
            np.save(path_with_timestamp, self.q_table)
            print(self.q_table[np.nonzero(self.q_table)])
        except:
            try:
                os.mkdir(self.qtable_directory)
                np.save(path, self.q_table)
                np.save(path_with_timestamp, self.q_table)
                print(self.q_table[self.q_table>0])
            except OSError:
                print("Creation of the directory %s failed" % path)
                print("Q-table could not be created.")
            else:
                print ("Successfully created the directory %s " % path)
```

The main

```
from agent import TDQSimulation

import matplotlib.pyplot as plt
import statistics
import numpy as np
import math
import csv
from itertools import zip_longest
from time import mktime

def draw_position_driving(agent):
    plt.figure(2)
    ax = plt.axes()
    scatter1 = plt.scatter(None, None, color='red', marker=".", s=100)
    scatter2 = plt.scatter(None, None, color='green', marker=".", s=100)
    scatter3 = plt.scatter(None, None, color='grey', marker=".", s=200)

    scatter2 = plt.scatter(agent.env.x_history_nanocar, agent.env.y_history_nanocar, color='green',
    marker=".", s=100)
    scatter1 = plt.scatter(agent.env.x_history_searching_nanocar, agent.env.
    y_history_searching_nanocar, color='red', marker=".", s=100)
    x_data_Goal = []
    y_data_Goal = []
    print(len(pos_Env))
    print(pos_Env[0][0])
    for i in range(len(pos_Env)):
        scatter3 = plt.scatter(pos_Env[i][0], pos_Env[i][1], color='grey', marker=".", s=200)

    plt.title('Part of the race-track from the nanocar race in Toulouse', fontsize=24)
    plt.xlabel('X / a.u.', fontsize=24)
    plt.ylabel('Y / a.u.', fontsize=24)
    plt.legend((scatter1, scatter2), ('Failed pulling', 'Successful pulling'), scatterpoints=1, loc=
    'best', prop={'size': 20})
    plt.xlim(0, 200000)
    plt.ylim(0, 200000)
    plt.draw()


def simulation_routine(agent):
    agent.select_move()             # Includes set_position() and set_Current/Voltage | For Testing
    : write Artificial Data
    agent.env.check_current_pattern()
    agent.env.calc_distance()
    agent.q_table_function()
    agent.env.set_position_history()
    draw_position_driving(agent)

def epoch_is_done(episode):
    final_episode = 100
    return episode==final_episode

def analysis(agent):
    # Calculate Analysis Variables
    if agent.env.number_of_searching == 0:
        agent.env.average_steps_while_searching = 0
    else:
        agent.env.average_steps_while_searching = agent.env.number_of_search_steps/agent.env.
    number_of_searching

    timestamp_file = agent.env.datetime_end.strftime("%y-%m-%d_%H-%M-%S")
```

```
    path_with_timestamp = f'{agent.env.directory_of_data}/{timestamp_file}_episode_{agent.env.number_of_episodes}_epsilon_{agent.epsilon}.csv'
    time_difference_in_s = abs(mktime(agent.env.datetime_start.timetuple())-mktime(agent.env.datetime_end.timetuple()))
    speed = agent.env.total_distance_to_goal/time_difference_in_s

    with open(path_with_timestamp, 'w', newline='') as csv_file:
        csv_write = csv.writer(csv_file)
        csv_write.writerow(['Episode', f'{agent.env.number_of_episodes}'])
        csv_write.writerow(['Epsilon', f'{agent.epsilon}'])
        csv_write.writerow(['Duration in s', f'{time_difference_in_s}'])
        csv_write.writerow(['Length', f'{agent.env.total_distance}'])
        csv_write.writerow(['Speed in nm / h', f'{speed}'])
        csv_write.writerow(['Manipulations', f'{agent.env.number_of_manipulations}'])
        csv_write.writerow(['Succeesful Manipulations',f'{agent.env.number_of_successful_manipulations}'])
        csv_write.writerow(['Failed Manipulations',f'{agent.env.number_of_failed_manipulations}'])
        csv_write.writerow(['Total reward per Episode',f'{np.around(agent.env.total_reward_per_episode,2)}'])
        csv_write.writerow(['Average Steps while Searching',f'{agent.env.average_steps_while_searching}'])
        csv_write.writerow(['== Positional Dataset =='])
        csv_write.writerows([['Goal'], np.swapaxes(agent.env.position_of_environment,0,1)[0], np.swapaxes(agent.env.position_of_environment,0,1)[1],
                             ['Nanocar'], agent.env.x_history_nanocar, agent.env.y_history_nanocar])
        csv_write.writerow(['Search-Algorithm'])
        for i in range(len(agent.env.x_history_searching_nanocar)):
            csv_write.writerow([agent.env.x_history_searching_nanocar[i], agent.env.y_history_searching_nanocar[i]])


pos_Env = np.array([[37000, 10000], [16000,35000]])# [10000,70000], [60000,180000], [150000,75000]])
x_data_Goal=[]
y_data_Goal=[]

def main():
    agent = TDQSimulation(pos_Env)

    while not agent.env.is_done():
        simulation_routine(agent)
    analysis(agent)
    agent.save_q_table()
    plt.show()



if __name__ == "__main__":
    main()
```