Reinforcement-learning-based navigation for autonomous mobile robots in unknown environments

M.M.A.M. (Mohammed) Alhawary

MSc Report

Committee:
Prof.dr.ir. P.C. Breedveld
Dr.ir. J.B.C. Engelen
N. Botteghi, MSc
Dr. M. Poel

August 2018

032RAM2018
Robotics and Mechatronics
EE-Math-CS
University of Twente
P.O. Box 217
7500 AE Enschede
The Netherlands

Abstract

Mobile robot navigation in an unknown environment is an important issue in autonomous robotics. Current approaches to solve the navigation problem, such as roadmap, cell decomposition and potential field, assume complete knowledge about the navigation environment. However, complete knowledge about the environment can be hardly obtained in practical applications where obstacles locations and surface friction properties are unknown. On the other hand, navigation in an unknown environment can be phrased as a reinforcement learning (RL) problem, because it is only possible to discover the optimal navigation plan through trial-and-error interaction with the environment. The goal of this project is to control a skid-steering mobile robot to navigate in an unknown environment with obstacles and a slippery floor using reinforcement learning techniques. The main task studied is the navigation to a goal location in the shortest time while avoiding obstacles and overcoming the skidding effects.

The standard (model-free) Q-learning algorithm is widely used to discover optimal trajectories in unknown navigation environments. However, it converges to these trajectories with undesirably-slow rates. The Dyan-Q approach extends the Q-learning with online-constructed models about the environment properties (obstacles, slippage, etc.) resulting in a model-based Q-learning platform. In this thesis, we examine using a multinomial probabilistic model to describe the state transition probabilities of the system dynamics. Moreover, two ideas to improve the learning performance of the Dyna-Q approach are suggested. The first is to prioritize the simulated learning experiences to be around the shortest discovered navigation trajectories between the initial and the target states. The second is to learn a parametric kinematic model for the robot motion that can be used to simulate the motion characteristics of the robot in unvisited locations.

It was shown that utilizing the models to simulate learning experiences makes the robot more robust to stochastic effects caused by skidding. The experimental results proved that the model-based RL algorithms converge faster to sub-optimal policies with higher success rates than the model-free Q-learning. Therefore, model-based algorithms have a better long-term performance in terms of less divergences from the discovered sub-optimal trajectories.

Contents

1 Introduction

1.1 Context

Navigation of autonomous mobile robots includes reaching a target location within a specified time while avoiding obstacles. When a complete knowledge of the environment is assumed, different global path planning algorithms [35] (roadmap, cell decomposition and potential field) can be applied. However, complete knowledge about the environment can be hardly achieved in practical applications where obstacles locations and surface friction properties are unknown. Adding to this, sliding is an inherent property of some mobile robots such as skid-steering-mobile-robots (SSMRs). All of which make global path planning algorithms fail in such stochastic unstructured navigation environments. On the other hand, navigation in such conditions can be naturally phrased as a reinforcement learning (RL) problem. Since it is only possible to discover (sub)optimal navigation trajectories through a trial-and-error interaction with the environment [12]. In a RL context, navigation consists of two interacting phases: *exploration phase* and *exploitation phase*. During *exploration phase*, the robot will navigate randomly to learn models about environment properties (obstacles, slippage, etc.), its own dynamics, and effects of its decisions on the environment. Then, during *exploitation phase*, the robot will make use of these models to plan (sub)optimal navigation trajectories.

1.2 Reinforcement learning

Reinforcement learning (RL) is a machine learning paradigm that enables a robot to autonomously find out an optimal behaviour through trial-and-error interaction with the environment. The traditional decision making pipeline for robots control divides the control problem into a set of sequential abstract sub-problems as shown in figure 1.1 and solves each subproblem separately. Adopting this traditional methodology leads to loss of information while moving from one abstract sub-problem to the next resulting in a sub-optimal performance. Instead of explicitly detailing the solution to a problem, in reinforcement learning the designer provides only a scalar value that measures the one-step performance of the robot [16].

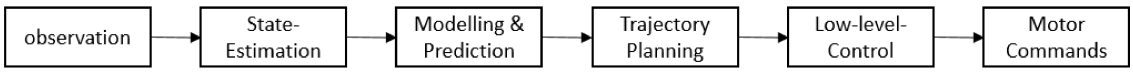


Figure 1.1: Traditional decision making pipeline for robots control [16].

Reinforcement learning (RL) assumes that the robot motion can be described by a set of states, S, and the RL agent (the robot) can take one of a fixed number of actions, A. At each time step, the agent observes the current state ,s_t, and choose an action, a_t to take. After taking the action, the agent is granted a reward, r_{t+1}, that reflects how good the action was, and observes the new state of the robot s_{t+1}. The goal of RL is to take this experience information $(s_t,a_t,r_{t+1},s_{t+1})$, and learn a mapping from each state (or each state-action pair) to a measure of the long-term value of being in that state (or that state-action pair), known as the optimal value function. According to the Bellman principle of optimality, being in a state taking an action with the maximum optimal value (the greedy action) will lead to the optimal policy that maximizing the return R (the summation of the rewards in the long run). A RL-agent can reach the state optimal values through a sequence of numerical updates to these values based on the interaction $(s_t,a_t,r_{t+1},s_{t+1})$ (value-function update). Accordingly, the agent updates the adopted control policy taking actions that transfer the robot to states with maximum optimal value (policy update). The decision making diagram of RL control agent is shown in figure 1.2.

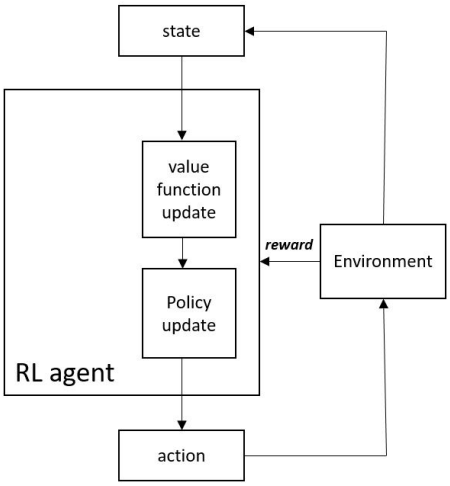


Figure 1.2: Reinforcement learning decision making diagram for robots control.

The most common-used reinforcement learning algorithm in the literature to solve autonomous navigation is Q-learning [11]. The Q-learning optimal value function is defined as

$$Q^*(s,a) = E[R(s,a) + \gamma max_{a'} Q^*(s',a')] \tag{1.1}$$

This represents the expected value of the reward for taking action a from state s, ending in state s', and acting optimally from then on. The parameter γ is known as the discount factor, and measures how much attention is paid to future rewards. Once we have the optimal Q-function for each state-action pair $Q^*(s,a)$, it is easy to deduce the optimal policy, $\pi^*(s)$, by simply selecting the action with the largest state-action pair value,

$$\pi^*(s) = argmax_a Q(s,a) \tag{1.2}$$

If the system is fully described by a set of finite states, the Q-function is typically stored in a table, indexed by the state-action pair. It is initialized by arbitrary values and iteratively updated to approximate the optimal Q-function based on the observation of the world. Every time that the robot takes an action, an experience tuble, $(s_t, a_t, r_{t+1}, s_{t+1})$, is generated. The Q-function table for state s and action a is then updated as follows

$$Q(s_t,a_t) \leftarrow (1-\alpha)Q(s_t,a_t) + \alpha(r_{t+1} + \gamma max_{a'} Q^*(s',a')) \tag{1.3}$$

Under some reasonable conditions [26], this is guaranteed to converge to the optimal Q-function, $Q^*(s,a)$. Q-learning is known as $off-policy$ RL algorithm. This means that the distribution from which the training samples are generated has no effect on the policy learned.

1.3 Problem statement

This research project aims to develop a RL-based navigation system to enhance the navigation capabilities of skid-steering-mobile robots (SSMRs). These capabilities are quantified by the robot ability to discover navigation trajectories to move to a target location in minimum time avoiding obstacles and slippery areas. This will be achieved by overcoming the main challenges that are usually faced when formalizing a robotic navigation problem as a reinforcement-learning problem, namely [12]:

- **Curse of dimensionality:** states and actions for most robots are inherently continuous. However, RL algorithms are designed for discrete-time events which necessitates discretizing states and actions spaces. Generally, discretizing n-dimensional state-space

with m levels will result in m^n different unique states. This exponential growth in the number of states leads to very slow convergence rates of the RL algorithm. It is possible to discretize the 2D space of the motion to obtain a grid world representation of the map. However, in case of SSMRs with non-holonomic constraints, it is not possible to simply move in the lateral directions. Therefore, the action space is usually discretized with four actions (move-forward, move-backward, turn-left, and turn-right). These actions are orientation-dependant necessitating including the robot orientation as a state [3]. Including the orientation as an internal state increases the dimension of the system so that the expected convergence time increases significantly.

- **Curse of real-world samples:** learning from interaction with the real environment is expensive in terms of time, human supervision and finance as explained in [12]. It is essential to create efficient RL algorithms that can learn from a small number of trials with the physical environment instead of limiting the memory consumption or the computational complexity.
- **Curse of under-modelling and model uncertainty:** One way to decrease the need to real-world interaction is to depend on accurate simulation models. However, small model errors due to under-modelling may cause the behaviour of the simulated robot to diverge from the real-world system. In SSMRs, sliding is an inherent motion property resulting in high levels of stochasticity in the robot model. Such property results in difficulties in accurately modelling the navigation environment due to the nonlinear stochastic slippage properties.
- **Curse of goal specification:** The goal of RL algorithms is to maximize the accumulated long-term reward. Although specifying the reward is simpler than detailing the behaviour itself, in practice, it is challenging to define a good reward function for the robot task. Since traditional binary rewards used in classical RL algorithms barely succeeds in practical robotic applications, a reward can be fully specified in terms of features of the space in which the RL algorithm operates, a problem which is known as *reward shaping*.

1.4 Literature review

This section presents how the reinforcement learning paradigm has been combined in navigation systems in the previous research. Based on the analysis of the previous research, the aims and the objectives of this research projects will be motivated.

1.4.1 State space representations

The representation of system states determines the dimensional properties of the system which have huge impacts on the speed of convergence of the RL algorithm. In [3], the pose states of the mobile robot $[x, y, \theta]$ were discretized with 15 quantization levels each. This results in a total system with 13500 state-action pairs. The experiment carried out took 15000 iterations for the algorithm to converge to the optimal policy given that the work-space dimensions are $1x1.2m$. This relatively slow convergence rate has motivated representing continuous state spaces. This is achieved by extending the traditional Q-learning algorithm by a state-value approximation function. In the traditional Q-learning, the value of each state-action pair is tabularly represented. However, value function approximators aim to learn a parametric function that can map each state-action pair to its optimal value. There are several possible representations for these parametric functions. In [17],[34],[30] where intensive sensor data are processed, the following neural networks structures were utilized as value function approximators:

- **Continuous state-space clustering:** this approach has been presented in [17] to solve the structure credit assignment problem in the representation of the navigation environment. This problem states that if the state space is highly discretized, the same credit

(function value) for a certain sequence of actions may be distributed among many internal regions of the state space leading to very slow convergence. To solve this problem, the states of the grid are clustered into a set of discrete states. Each cluster of states will share the same state-action Q-value to represent a cluster-action pair. The learning is based on clustering the state space to a group of similar states and different different ones and is based on learning the optimal cluster-action pair value for each cluster. The continuous representation of the state space is replaced by a set of N discrete states (clusters) using a Kohonen neural network structure with a winner-take-all learning rule 1.3.

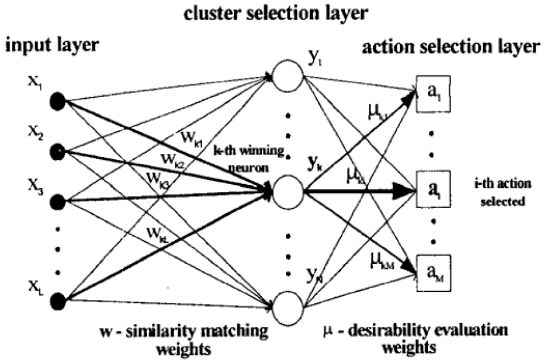


Figure 1.3: The neural network structure of the learning system [17] .

The left part of the neural network structure clusters the input state vector $x = [x_1\ x_2\ ..x_L]$ into N clusters using a set of weight vectors $\{\vec{w}_1,\ \vec{w}_2\ ..,\vec{w}_j,..,\vec{w}_N\}$, where $\vec{w}_j$ connects the input vector $\vec{x}$ to the j cluster. The input state vector is assigned to the the k cluster which has the winning neuron k and is selected by the following similarity matching rule:

$$k : \|\vec{x} - \vec{w}_k\| \leq \|\vec{x} - \vec{w}_j\|\ \forall j \tag{1.4}$$

The winning vector $\vec{w}_k$ is updated using the following rule:

$$w_k(\vec{t}+1) = \vec{w_k}(t) + \eta(\vec{x(t)} - \vec{w_k}(t)) \tag{1.5}$$

For the input state $\vec{x}$, k^{th} neuron is the winning one and the action-values for each action a in a set of M actions are $\{Q(\vec{x},a_1) = \mu_{k1}\ Q(\vec{x},a_2) = \mu_{k2}\ ...Q(\vec{x},a_M) = \mu_{kM}\}$. The chosen action at each time t is the one which has the largest action-value Q leading to a greedy policy over the action-values. The update rule for the state-action estimates μ_{ji} is:

$$\mu_{ji}(t) = \mu_{ji}(t-1) + \alpha r^*(t) \tag{1.6}$$

where $r^*(t)$ is the expected return based on the received reward $r(t)$ and is calculated by the following equation:

$$r^*(t) = r(t) + \gamma max|_a Q(x_t,a_t) - Q(x_{t-1},a_{t-1}) \tag{1.7}$$

The initial estimates $\mu(0)$ are set to zero.

- **Neural Q-learning:** this approach is presented in [34]. Assuming that the action space is discretized to N possible actions ($A = a_i, i = 1,2,..,N$), N multi-layer feedforward neural networks with an input layer, a hidden layer and an output layer are used to approximate N action value functions $Q(s,a_i)$ as shown in figure 1.4. The input to the neural network is the sensor readings expected which are denoted by $(s_1,s_2,...s_m)$. where m is the number of the input layer. The hidden layer of the ith neural network has n neurons with

corresponding weights $q^i_{kl}(k = 1,2,..m, l = 1,2,..n)$. The Sigmoid function is used as an activation function for the hidden layer and y^i_l denotes the lth hidden layer neuron of the ith neural network as explained by the following equation:

$$y^i_l = \frac{1}{1 + exp(-\zeta x^i_l)} \tag{1.8}$$

$$x^i_l = \sum_{j=1}^{m} q^i_{jl} s_j \tag{1.9}$$

where ζ is a real constant number. On the other hand, a linear weighted combination function is used as an activation function for the output layer of the ith neural network.

$$Q(s, a_i) = \sum_{l=1}^{n} w^i_l y^i_l \tag{1.10}$$

The parametrization of the network parameters q^i_{kl} and w^i_l is done through an unsupervised temporal difference updates as explained in [34]. The mobile robot uses the neural network outputs $Q(s, a_i)$ to select actions with the maximum Q-value.

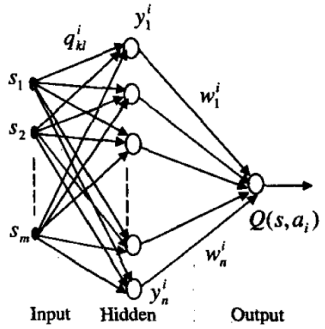


Figure 1.4: Neuron Q-learning [34].

The idea contructs n neural networks given n actions to predict the Q-value of each action independently. On the other hand, other neural Q-learning algorithms construct multi-output neural networks to predict the action-value of all the action simultaneously as explained in the following algorithm.

- **Experience-replay neural Q-learning:** this approach has been presented in [30]. This approach uses a back-propagation neural network to approximate the Q-value for each action given the mobile robot states as shown in figure 1.5. It uses a different principle to parameterize the network. This principle is based on storing all the navigation experiences (s_t, a_t, R_t, s_{t+1}) in a replay memory D. These experiences will be used back as a training examples in a random order. This breaks the similarity of subsequent training examples that might lead the network into a local minimum. This technique is called experience replay. It makes the training task similar to usual supervised learning. For updating the weights θ of the neural network, random minibatch of size N is taked from D. For every tuple (s_j, a_j, R_j, s_{j+1}) in the batch, feedforward propagation is made for the current state s_j based on the current parameters θ to predict the $Q(s_j, a_j, \theta)$ value. If s_{j+1} is terminal an evaluation equal to the terminal reward $y_j = R_{terminal}$ is assigned. Otherwise, the value of s_{j+1} is predicted to be $Q(s'_j, a'_j, \theta')$ and the evaluation is assigned to be $y_j = r + max_{a'_j} Q(s'_j, a'_j, \theta')$. Finally, the loss function $L(\theta)$ is

$$L(\theta) = \frac{1}{N} \sum_{i=1}^{n} (y_j - Q(s_j, a_j, \theta)) \tag{1.11}$$

Using $L(\theta)$, the weights θ will be updated using back-propagation and a gradient descent algorithm.

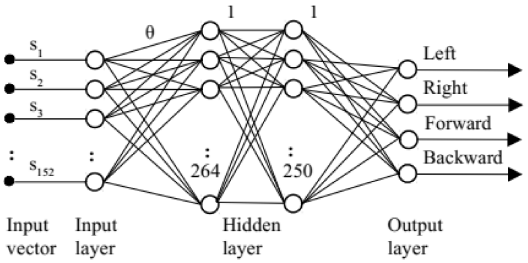


Figure 1.5: The inputs are the states of the robot, the outputs are the q-value corresponding to each possible action of the mobile robot [30].

All the three approaches have been validated through simulations and it has been proven to have faster convergence rates than the traditional Q-learning.

1.4.2 Reinforcement learning algorithms

[25] and [26] uses the Q-learning for navigation and obstacle avoidance. They have used a sparse reward strategy such that the agent receive value of 1 while reaching the target, -1 on hitting an obstacle and 0 otherwise. The robot has been trained for several times and the learned policy has been evaluated by exposing the robot to the same environment with a static goal and a single static obstacle. The robot was able to successfully reach the target showing some potentials of using RL for mobile robots navigation. However, the time taken to reach the target was relatively long. This has motivated developing some modifications to the standard Q-learning algorithm to accelerate the learning in high-dimension navigation environments. Examples of these modifications are learning by demonstration, hybrid RL-fuzzy logic and neural Q-learning.

Learning by demonstration

In [26], they supply example trajectories to the learning agent and split the learning process into two phases. Firstly, the robot is being controlled by a supplied control policy by either a coded navigation program or manually. During this phase the robot stores the states, actions and rewards generated by the supplied policy. The learning agent uses these data to bootstrap information into its action-value approximators. The key feature of this phase is that the RL system does not control the robot. In this phase, the learning system is not assumed complete enough to adequately control the robot. The first phase aims to expose the robot to reward-giving states instead of consuming too much time exploring uninteresting parts of the state space that give zero reward. The learning agent is not storing the trajectories generated by the supplied policy but it uses the experience to train the action-value approximator. Secondly, when the action-value approximator is sufficiently-trained, the learned policy from the first phase is used to control the robot using the standard Q-learning algorithm. By dividing the learning into two phases, the agent gains the ability to bootstrap information into the action-value approximation function before using the learned approximation function to control the robot. This guarantees that in the second phase, the robot will be able to find the reward-giving states and will not stall. The idea of learning by demonstration has become popular recently [26] and has been proved to accelerate the learning.

Hybrid reinforcement-learning and fuzzy logic

These algorithms try to combine reinforcement learning and fuzzy logic learning techniques. In [5], a hybrid approach using Fuzzy Logic and Reinforcement Learning is proposed. They

have noted the method by Environment Exploration Method (EEM). The navigation system has two basic modules: avoidance behaviour and goal-seeking behaviour. The avoidance behaviour and the goal-seeking behaviour are fuzzy engines that map the sensor readings to a fuzzy output which is the optimal action. The fuzzy rules are built using reinforcement learning rather than supervised learning which requires less evaluation data.
In [35], they propose a navigator which consists of three components: obstacle avoidance, move-to-goal and a fuzzy behaviour supervisor. The obstacle avoidance components receives sensory data that are fuzzified then the rules are learned through RL. After constructing fuzzy rules, the decisions are made to generate the output actions. The same principle applies for the move-to-goal component. The fuzzy behaviour supervisor module receives the output of the other two components and generates an appropriate action. The result of the algorithm showed that this method produced higher performance and did not suffer from local minima. However, both methods are applicable only to static environments and cannot be utilized for dynamic ones.
An other hybrid approach is presented in [10]. This approach introduces an enhanced dynamic self-generated fuzzy Q-learning approach (EDSGFQL) for automatically-generated fuzzy inference systems (FISs). The structure identification and the parameter estimations for the FISs are obtained using reinforcement learning. The Q-learning is used to cluster the input space to generate the FISs. Also, RL is used to adjust and omit fuzzy rules dynamically. EDSGFQL was used for a navigation task for a Khepera mobile robot. The simulations showed that the robot has succeeded in wall following and obstacles avoidance tasks for static environments.
Another hybrid learning method that uses a fuzzy inference system structured based on the generalized dynamic fuzzy neural network algorithm was proposed by [9]. Supervised learning is applied for the neuro-fuzzy controller and a reinforcement-learning actor-critic method is used so that the system can re-adapt to a new environment without human interventions. The reinforcement learning is used to tune the parameters of the fuzzy rules. The simulations have proven the system ability to solve the obstacles avoidance problem in a static environment.

Neural Q-leaning

As presented in the previous subsection, representing a continuous state space requires modifying the standard Q-learning algorithm to include a value function approximator. In the context of autonomous navigation, neural networks have been used as efficient function approximators. There are three neural network structures that have been used with Q-learning to achieve autonomous navigation which are presented in 1.4.1. The first method (state-space clustering) presented in [17] has a better convergence rate than the regular neural Q-learning method. In this approach, discrete Q-learning is used in stead of continuous Q-learning. The learning is based on clustering input data into similar and dissimilar groups. The method uses a discrete representation for the action space. The clustering algorithm performs faster than other neural Q-learning methods but it is only applicable to static environments [11].
The probabilistic roadmap method (PRM) has been combined with the Q-learning to implement a hybrid approach for the robot navigation in [21]. The PRM method is based on connecting the start location of the robot to the target location by drawing random nodes in an assumed obstacles-free space. The Q-learning is used when an obstacle blocks a pre-planned trajectory to determine the best action to be taken to avoid the obstacle. The performance of the method is degraded for dynamic navigation environments.

1.4.3 Reward shaping

The are several ways to shape the reward function to solve the navigation problem. These ways are listed below:

- **Binary reward:** This is a very simple strategy. It gives 1 reward on reaching the goal and zero otherwise. This strategy is commonly-used in classical reinforcement learning problems that do not include a dynamical behaviour. Adopting this method makes the convergence rate very slow specially in highly-dimensional state spaces where the probability of finding the goal is significantly small [26].

- **Sparse reward:** It is similar to the binary reward. In addition, it adds negative reward on hitting the obstacles and zero otherwise. This approach to shape the reward function has been used intensively in the literature ([25], [26] and [3]).

- **Potential-based reward:** this idea has been introduced in [20]. It aims to find a transformation to the sparse reward function that incorporates knowledge about the navigation environment in the design of the reward function. The approach assigns a potential value $\phi(s)$ to each cell in the navigation grid. This potential is determined based on the distance between the cell and the goal and the distance between the cell and known obstacles. The transformation applied to the sparse reward will be $R' = R + F$ where

$$F(s, a, s') = \gamma\phi(s') - \phi(s) \tag{1.12}$$

1.4.4 Conclusion

From the previous review it can be seen that there is a trend among the researchers to adopt the continuous neural Q-learning to solve the navigation problem. On the other hand, they did not address the potentials of including environment models together with the reinforcement learning to accelerate the learning process. This approach is known as *model-based reinforcement learning* or *indirect reinforcement learning*. The idea has been introduced in [4], [27] and [23] where several model-based Q-learning algorithms like dyna-Q and prioritized-sweeping were elaborated. Although they provide great potentials in speeding up the convergence rate of the Q-learing, they have not been used intensively in the RL-based autonomous navigation. Similarly, models for robot motions are not commonly utilized to plan motion trajectories together with the reinforcement learning. Moreover, the problem of slippery surfaces and sliding motion of SSMRs are not addressed in the literature. These reasons motivate directing the research of this project to the area of integrating models with standard Q-learning. We suggest learning several models online (an environment model, a robot model and a slippage model). These models are used to extend the standard Q-learning algorithm to achieve (sub)optimal motion trajectories.

1.5 Project aims and objectives

The common trend in the current research is to combine standard Q-learning algorithm with neural networks to accelerate the learning process (Neural Q-leaning). On the other hand, the model-based reinforcement learning is proven to accelerate the convergence rate more than the standard Q-learning [4]. The aim of this project is to explore the potentials of the model-based reinforcement leaning for autonomous navigation tasks in unknown stochastic environments. This is obtained by modifying the standard Q-learning algorithm by extending it with online learned models of the robot motion and the navigation environment (obstacles locations and slippage properties). The aim of the project can be accomplished through exploring the following research objectives:

- How to model the transition of the robot states in the environment stochastically?

- How to model the motion of the SSMR robot to integrate it with the Q-learning algorithm?

- What is the effect of discretization level of the action and the state spaces on the learning speed and the optimality of the learned trajectories?

- What is the effect of including models with the standard Q-learning on the performance of the reinforcement-learning system?

Motivated solutions are validation by designing a RL-based navigation system for a small SSMR so that it can explore optimal trajectories to given targets in an unknown environments.

This report shows the research work aimed to achieve these research objectives and is organized as follows:

Chapter 2 summarizes the theories involved in the design of the RL-based navigation systems. It elaborates more on the standard Q-learning algorithm and how to modify it to implement a model-based reinforcement learning through the Dyna-Q algorithm. Then, it introduces multinomial probabilistic models which will be used to model the transition of discrete system states based on data gathered from the environment. Finally, it introduces the kinematic model of SSMR and Bayesian ridge regression algorithm used to estimate the SSMR kinematic model using the data gathered from the robot and the environment.

Chapter 3 shows how the navigation problem can be formulated as a reinforcement learning problem to be solved using a model-based Q-learning algorithm. Firstly, it explains requirements on the system representation necessary to use the Q-learning algorithm. Then, it discusses selecting the parameters of the Q-learning algorithm in the context of the autonomous navigation. After that, it analyzes the nature of motion of Skid-Steering-Mobile-Robots elaborating the requirements needed to model their motion. Finally, it shows the set-up built to run the real experiments and the limitations it introduces to the system performance.

Chapter 4 proposes a design structure of a reinforcement-learning-based navigation system. The learning system is designed based on the Dyna-Q platform modified by creating a multinomial probabilistic model to describe the dynamics. The algorithm is enhanced by creating an online-structured Bayesian regression model to learn the kinematic model of the mobile robot. Finally, the architecture of the software used to run the set-up in real-time is presented.

Chapter 5 provides a detailed analysis of the obtained results of the simulation and the real-time experiments.

2 Background

2.1 Q-learning algorithm

2.1.1 Finite Markov decision process

The solution of reinforcement learning problem is framed by modelling the control process as a finite Markov decision process (MDP) [28]. In a MDP, the controller is called the *agent* and everything outside the agent is called the *environment*. The agent at state S_t can interact with the environment (figure 2.1) by applying an action A_t and the consequence of this interaction is receiving a numerical reward R_{t+1} (a scalar value needed to be maximized by the agent over time) and the agent moves to a new state S_{t+1}.

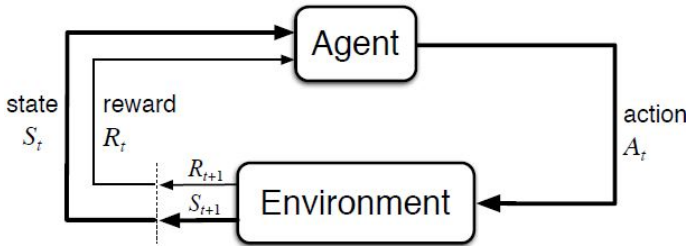


Figure 2.1: The agent-environment interaction in a Markov decision process [28].

In a finite MPD, there are two random variables R and S with a well-defined discrete probability distribution that depends only on the current state s and the applied action a. So for all next state $s' \in S$, $r \in R$ and $a \in A(s)$ it applies:

$$p(s', r|s, a) = Pr[S_t = s', R_t = r|S_{t-1} = s, A_{t-1} = a] \tag{2.1}$$

$$\sum_{s' \in S} \sum_{r \in R} p(s', r|s, a) = 1 \tag{2.2}$$

The probabilities p completely characterize the environment's dynamics. If it is sufficient to know the current state s_t and the current action a_t to determine the probability of the next state s_{t+1} and the next reward r_{t+1} then the system is said to have the Markov property.
The expected rewards for state-action-next state pairs as a three-argument function $r : S \times A \times S \rightarrow \Re$,

$$r(s, a, s') = E[R_t|S_{t-1} = s, A_{t-1} = a, S_t = s'] \tag{2.3}$$

The agent's goal is to maximize the total amount of this expected reward received from the environment. This means maximizing not the immediate reward, but the cumulative reward in the long run. If the sequence of rewards received after time step t denoted $R_{t+1}, R_{t+2}, R_{t+3}, ...$, then maximizing the value of this sequence is denoted by maximizing the *exptected return*, where the return G_t:

$$G_t = R_{t+1} + R_{t+2} + R_{t+3} + ... + R_T \tag{2.4}$$

where T is a final time step. If the task representing the agent interaction with the environment can be naturally broken into sequences that end with a specific terminal state S^+, then this task is called an *episodic task*. In episodic tasks, the agent's state is reset to a starting state after reaching the terminal state. The time of termination T is a random variable that normally varies between episodes. It is also possible to discount future rewards to be maximized by introducing the discount rate parameter γ where $0 \leq \gamma \leq 1$. Thus, the goal is to maximize the discounted return:

$$G_t = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + ... = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \tag{2.5}$$

2.1.2 Policies and value functions

Reinforcement learning algorithms involve estimating value functions. They are functions of states (or of state-action pairs) that estimate how good for the agent to be in that state in terms of maximizing the future reward. Since the rewards to be received in the future depends on the actions taken, value functions for system states depend on the strategy adopted to select actions which are called *policies*. A policy π defines a probability distribution of selecting an action a at each state s ($\pi(a|s)$) for each $a \in A(s)$ and $s \in S$. The *value function* of a state s under a policy π, is the expected return starting from s following π:

$$v_\pi(s) = E_\pi[G_t|S_t = s] = E_\pi[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1}|S_t = s] \tag{2.6}$$

Similarly, the action-value function of action a in state s under policy π is defined:

$$q_\pi(s,a) = E_\pi[G_t|S_t = s, A_t = a] = E_\pi[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1}|S_t = s, A_t = a] \tag{2.7}$$

The reinforcement learning problem is solved by finding the policy that maximizes the reward over the long run. A policy π' is said to be better a policy π ($\pi' > \pi$) if and only if $v_{\pi'}(s) > v_\pi(s)$ for all $s \in S$. There is always at least one policy that is better than or equal other policies which is the *optimal policy*. Optimal policies share the same optimal action-value function, denoted by q_*:

$$q_*(s,a) = max|_\pi q_\pi(s,a) \tag{2.8}$$

This optimal action-value function gives the expected return of taking an action a in state s following the optimal policy:

$$q_*(s,a) = E[R_{t+1} + \gamma v_*(S_{t+1})|S_t = s, A_t = a] \tag{2.9}$$

2.1.3 Off-policy Q-learning

The Bellman optimality equation states that the value of a state under an optimal policy must equal the expected return of the best action at that state:

$$v_*(s) = max|_a \sum_{s',r} p(s', r|s, a)[r + \gamma v_*(s')] \tag{2.10}$$

This last equation represents the Bellman optimality equation for v_*. The Bellman optimality equation for q_* is

$$q_*(s,a) = max|_a \sum_{s',r} p(s', r|s, a)[r + \gamma max|_{a'} q_*(s', a')] \tag{2.11}$$

For finite MDPs, the Bellman optimality equation for v_π has a unique solution independent of the policy. The Bellman optimality equations for an n states system is a system of n equations in n unknowns. If the dynamics p of the environment are known, then it is possible to solve for v_* by solving systems of non-linear equations and consequently solving a related set of equations for q_*. If the dynamics of the environment are unknown, it is still possible to estimate the optimal action value functions q_* through the experience by interacting with the environment and updating a numerical back-up equation for each state-action pair value $Q(s,a)$.
Q-learning algorithm (figure 2.2) is a powerful tool for estimating the optimal action value functions q_* of a stochastic Markovian system with unknown dynamics. It uses the following back-up updating rule:

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha[R_{t+1} + \gamma max|_a Q(S_{t+1}, a) - Q(S_t, A_t)] \tag{2.12}$$

Where $0 < \alpha \leq 1$ parameter is called the *learning rate*. Under the usual stochastic approximation conditions, Q-learning algorithm converges with probability 1 to q_* if all the state-action pairs are visited and updated. Thus, the policy to be followed during the exploration phase of the environment is not the learned optimal policy (*greedy*) policy. In stead, actions are selected randomly with a probability ϵ at each state to enhance exploring new state-action pairs. This exploration policy is called the $\epsilon - greedy$ policy. Since the learned policy through the back-up updating rule π_* is different from the policy followed during the learning process, Q-learned is called an off-policy reinforcement learning algorithm. In this case, the learned action value function Q, directly approximate q_* independent of the policy being followed.

Q-learning (off-policy TD control) for estimating $\pi \approx \pi_*$

Algorithm parameters: step size $\alpha \in (0, 1]$, small $\varepsilon > 0$
Initialize $Q(s, a)$, for all $s \in \mathcal{S}^+, a \in \mathcal{A}(s)$, 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-greedy)
    Take action A, observe R, S'
    $Q(S, A) \leftarrow Q(S, A) + \alpha\big[R + \gamma \max_a Q(S', a) - Q(S, A)\big]$
    $S \leftarrow S'$
  until S is terminal

Figure 2.2: Q-learning algorithm for approximating q_* [28].

2.2 Dyna-Q algorithm

Q-learning algorithm is a *model-free* reinforcement learning algorithm since it depends only on learning to update action value functions for each state-action pair. On the other hand, *model-based* reinforcement learning algorithms depend on planning using a model of the environment structured online during learning [28]. A model of the environment allows the agent to make predictions about the next states and the next rewards. If the system is stochastic, environment models produce a description of all possibilities with their probabilities leading to *probabilistic distribution models*.
Models can be used to simulate experiences. Given a starting state and action, a distribution model generates all probabilities of possible transitions. Similar to model-free methods, simulated experiences generated by the distribution model are used for the estimation of the value functions using the same back-up update equations [28]. If the standard Q-learning algorithm is used with data generated by a distribution model, the model-based algorithm converges under the same conditions that the one-step Q-learning converges for the real environment.

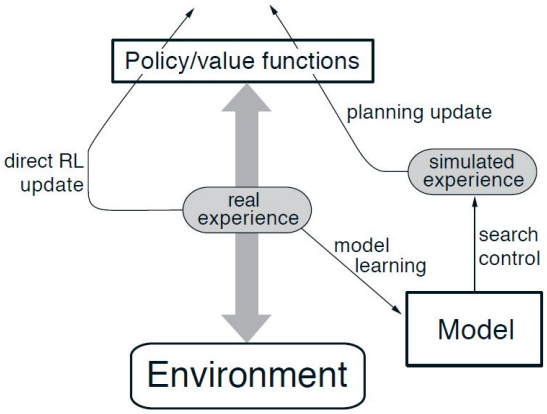


Figure 2.3: The general Dyna-Q architecture [28].

If planning is done online while interacting with the environment, collected data representing state transitions are used to modify the model and consequently affect the planning. Dyna-Q architecture, shown in figure 2.3, is a platform proposed in [27] that integrates model-learning, planning and value functions updating. After each interaction with the environment, the experience $(S_t, A_t \rightarrow R_{t+1}, S_{t+1})$ is used for the planning update (figure 2.3). Moreover, it is stored in a look-up table that maps each (S_t, A_t) to equivalent (R_{t+1}, S_{t+1}). This model learning based assumes that the model is deterministic. During planning, the Q-planning algorithm randomly samples only from state-action pairs that have been experienced before and stored in the table. The data stored in the table-based model are used to simulate hypothetical experiences that can directly update the Q-values of state-action pairs just as if they really happened. The agent performs N hypothetical experiences after each real-world interaction with the environment. The same back-up update rule is used both for learning from the real experience and for planning from simulated experiences.

Tabular Dyna-Q

Initialize $Q(s,a)$ and $Model(s,a)$ for all $s \in \mathcal{S}$ and $a \in \mathcal{A}(s)$
Loop forever:
- (a) $S \leftarrow$ current (nonterminal) state
- (b) $A \leftarrow \varepsilon$-greedy(S,Q)
- (c) Take action A; observe resultant reward, R, and state, S'
- (d) $Q(S,A) \leftarrow Q(S,A) + \alpha\big[R + \gamma \max_a Q(S',a) - Q(S,A)\big]$
- (e) $Model(S,A) \leftarrow R, S'$ (assuming deterministic environment)
- (f) Loop repeat n times:
  - $S \leftarrow$ random previously observed state
  - $A \leftarrow$ random action previously taken in S
  - $R, S' \leftarrow Model(S,A)$
  - $Q(S,A) \leftarrow Q(S,A) + \alpha\big[R + \gamma \max_a Q(S',a) - Q(S,A)\big]$

Figure 2.4: The tabular Dyna-Q algorithm [28].

Figure 2.4 shows a pseudo-code of the general Dyna-Q algorithm that assumes a deterministic environment. Integrating planning with learning accelerates the convergence to the optimal q_* significantly. Because any change in the q-value of a state-action pair is going to propagate to values of other pairs through hypothetical experiences. On the other hand, when a new information is gained, the model is updated and the planning will decide different ways of behaving based on the updated model.

2.3 Prioritized Sweeping

The Dyna-Q algorithm presented in the previous chapter selects state-action pairs to simulate hypothetical experiences uniformly from the stored experience. The hypothetical experiences can be more efficient if the simulated updates are focused on particular state-action pairs. The prioritized sweeping algorithms focuses on state-action pairs whose values have changed recently. If the values of these pairs have changed, then the values of the predecessor states may change accordingly. Then, actions leading into them needed to be updated, and then their predecessor states may have changed. Thus, the algorithm works backward from arbitrary states that have changed their values, either performing updates or ending the propagation. This idea is called *backward focusing* of planning computations.

Prioritized sweeping for a deterministic environment

Initialize $Q(s,a)$, $Model(s,a)$, for all s,a, and $PQueue$ to empty
Loop forever:
(a) $S \leftarrow$ current (nonterminal) state
(b) $A \leftarrow policy(S,Q)$
(c) Take action A; observe resultant reward, R, and state, S'
(d) $Model(S,A) \leftarrow R, S'$
(e) $P \leftarrow |R + \gamma \max_a Q(S',a) - Q(S,A)|$.
(f) if $P > \theta$, then insert S,A into $PQueue$ with priority P
(g) Loop repeat n times, while $PQueue$ is not empty:
$S,A \leftarrow first(PQueue)$
$R,S' \leftarrow Model(S,A)$
$Q(S,A) \leftarrow Q(S,A) + \alpha\left[R + \gamma \max_a Q(S',a) - Q(S,A)\right]$
Loop for all $\bar{S},\bar{A}$ predicted to lead to S:
$\bar{R} \leftarrow$ predicted reward for $\bar{S},\bar{A},S$
$P \leftarrow |\bar{R} + \gamma \max_a Q(S,a) - Q(\bar{S},\bar{A})|$.
if $P > \theta$ then insert $\bar{S},\bar{A}$ into $PQueue$ with priority P

Figure 2.5: The tabular prioritized sweeping algorithm [28].

It is convenient to prioritize the updates based on the change of the state-action pairs values and perform them in order of priority. The planning algorithm performs this idea is called *prioritized sweeping*. The state-action pairs are stored in a priority queue being updated based on the change of each state-action pair value prioritized by the amount of the change. When the top of the queue is updated, the effect on each predecessor pair is computed. If the effect generates a pair value greater than a threshold, the pair is inserted in the queue with a new priority. In this way, the effects of changes are efficiently propagated until stillness. The full algorithm is explained in figure 2.5.

2.4 Multinomial probabilistic models

The general Dyna-Q algorithm presented in the previous section assumes a deterministic environment. However, in the context of autonomous navigation for SSMRs, sliding is an inherent system property that leads to stochastic dynamics. Thus, the learned model should be probabilistic. The basic idea to learn a probabilistic model is that not predicting a deterministic next state and a deterministic reward, but a probability distribution over next states and next rewards [27]. For discrete finite MDPs, it is possible to represent the state transition probabilities $p(s',r|s,a)$ using multinomial distributions [14].
If the result of a stochastic process can be described by a discrete variable that take one of possible K exclusive states, it is convenient to represent this variable with a 1-of-K scheme [7]. The variable is represented by a K-dimensional vector x in which one of the elements x_k is 1 and

the remaining elements are 0. Such vectors satisfy $\sum_{k=1}^{K} x_k = 1$. If the probability of $x_k = 1$ is denoted by the parameter μ_k, then the distribution of x:

$$p(x|\mu) = \prod_{k=1}^{K} \mu_k^{x_k} \tag{2.13}$$

where $\mu = (\mu_1, .., \mu_K)^T$ such that $\mu_k \geq 0$ and $\sum_k \mu_k = 1$. Now consider a data set D of N independent observations $x_1, .., x_N$. The corresponding likelihood function takes the form:

$$p(D|\mu) = \prod_{n=1}^{N} \prod_{k=1}^{K} \mu_k^{x_{nk}} = \prod_{k=1}^{K} \mu_k^{(\sum_n x_{nk})} = \prod_{k=1}^{K} \mu_k^{m_k} \tag{2.14}$$

$m_k = \sum_n x_{nk}$ represents the number of observations of $x_k = 1$. Given the data set D it is required to solve for μ. This solution is achieved by finding μ that maximizes the likelihood function. Maximizing the likelihood is achieved by maximizing its logarithmic function $ln[p(D|\mu)]$ accounting for the constraint $\sum_k \mu_k = 1$. This can be achieved by using a Lagrangian multiplier λ:

$$\sum_{k=1}^{K} m_k ln(\mu_k) + \lambda \sum_{k=1}^{K} \mu_k - 1 \tag{2.15}$$

By setting the derivative of the previous equation with respect to μ_k to zero, the solution for μ_k is:

$$\mu_k = -m_k/\lambda \tag{2.16}$$

To solve for the Lagrangian multiplier λ,(2.16) is substituted into the constraint $\sum_k \mu_k = 1$. Thus, the maximization of the likelihood is given in the form:

$$\mu_k^{ML} = \frac{m_k}{N} \tag{2.17}$$

which is the fraction of the N observations whose $x_k = 1$. The *multinomial distribution* is a joint distribution of the quantities $m_1, ..., m_k$ given the parameter μ on the total number of N observations. It takes the form:

$$Mult(m_1, .., m_K|\mu, N) = \frac{N!}{m_1!m_2!..m_K!} \prod_{k=1}^{K} \mu_k^{m_k} \tag{2.18}$$

The normalization coefficient is the number of ways to partition N objects into K groups of size $m_1, .., m_K$. These variables m_k are subject to the constraint:

$$\sum_{k=1}^{K} m_k = N \tag{2.19}$$

2.5 Skid-Steering-Mobile-Robots kinematic model

The previous section shows how it is possible to build probabilistic models for the next states and rewards for each state action pair (s, a) given the experience data set $(D = (s'_1, r_1)..(s'_N, r_N))$. This means that it is only possible to simulate hypothetical experiences for already-visited state-action pairs (s, a). If it is required to predict probability distributions for the next states and rewards for action-action pairs that have not been visited before, an accurate model for the robot motion must be developed. This model is also necessary for simulating the designed reinforcement learning algorithms before validating them in real-time.
In [13], a kinematic model for SSMRs is presented. This kinematic model can be developed by analyzing the robot free body diagram shown in figure 5.2 .

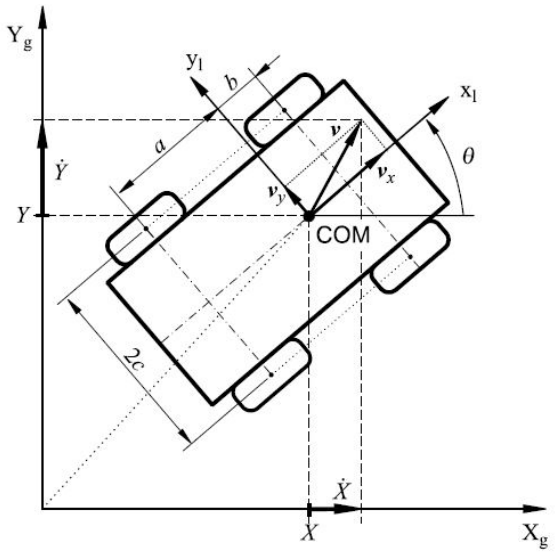


Figure 2.6: SSMR free body diagram [13].

Suppose that the robot linear velocity vector expressed in the robot reference frame (l) is $v = [v_x \ v_y \ 0]^T$ and the angular velocity vector is $\Omega = [0 \ 0 \ \omega]^T$. If $q = [XY\theta]^T$ represents the robot COM position X and Y and orientation θ expressed with respect to the inertial frame (g), then $\dot{q} = [\dot{X}\dot{Y}\dot{\theta}]^T$ represents the vector of the generalized velocities. From figure 2.6, the $\dot{X}$ and $\dot{Y}$ are related to v_x and v_y by:

$$\begin{bmatrix} \dot{X} \\ \dot{Y} \end{bmatrix} = \begin{bmatrix} cos(\theta) & -sin(\theta) \\ sin(\theta) & cos(\theta) \end{bmatrix} \begin{bmatrix} v_x \\ v_y \end{bmatrix} \tag{2.20}$$

Because of the planner motion, the relation $\omega = \dot{\theta}$ is valid.

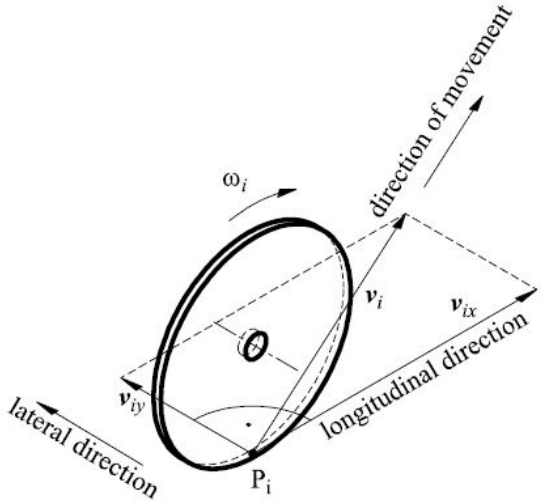


Figure 2.7: Velocities of one wheel [13].

Suppose that the i-th wheel rotates with an angular velocity $\omega_i(t)$, where $i = 1,2,3,4$ which are the control inputs. Assume that the point of contact between the wheel and the surface is P_i (figure 2.7), unlike most common wheeled robots, the lateral slip velocity v_{iy} is non-zero. This is because the lateral skidding is necessary when the SSMR changes orientation. Therefore, this

component v_{iy} is only zero if the robot moves in a straight line. On the other hand, for simplification, the longitudinal slip can be neglected so that the following relation can be developed.

$$v_{ix} = r_i \omega_i \tag{2.21}$$

where v_{ix} is the longitudinal component of the total wheel velocity v_i of the i-th wheel expressed in the robot reference frame. r_i denotes the effective rolling radius of the i-th wheel.

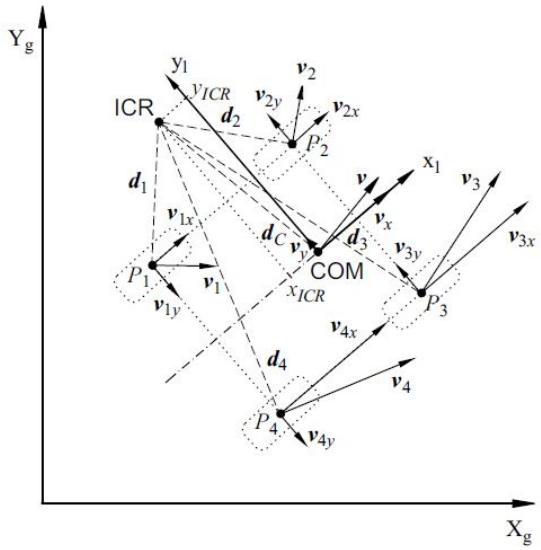


Figure 2.8: Wheel velocities [13].

To develop a kinematic model, the robot body rotation with respect to an instantaneous centre of rotation point ICR is defined through the radius vector $d_i = [d_{ix} d_{iy}]^T$ and $d_C = [d_{Cx} d_{Cy}]^T$. Consequently, based on the geometry of figure 2.8 , it holds that:

$$\frac{||v_i||}{||d_i||} = \frac{||v||}{||d_C||} = |\omega| \tag{2.22}$$

Or,

$$\frac{v_{ix}}{-d_{iy}} = \frac{v_x}{-d_{Cy}} = \frac{v_{iy}}{d_{ix}} = \frac{v_y}{d_{Cx}} == \omega \tag{2.23}$$

By defining the ICR coordinates in the robot reference frame as

$$ICR = (x_{ICR}, y_{ICR}) = (-d_{Cx}, -d_{Cy}) \tag{2.24}$$

Then,

$$\frac{v_x}{y_{ICR}} = \frac{v_y}{x_{ICR}} = \omega \tag{2.25}$$

From figure 2.8, the coordinated of vectors d_i satisfy the following relationships:

$$d_{1y} = d_{2y} = d_{Cy} + c \tag{2.26}$$

$$d_{3y} = d_{4y} = d_{Cy} - c \tag{2.27}$$

$$d_{1x} = d_{4x} = d_{Cx} - a \tag{2.28}$$

$$d_{2x} = d_{3x} = d_{Cx} + b \tag{2.29}$$

where a, b and c are positive kinematic parameters of the robot (figure 2.6). After combining equation 2.23 with equations (2.26, 2.27, 2.28 and 2.29), the following relationships can be obtained:

$$v_L = v_{1x} = v_{2x} \tag{2.30}$$

$$v_R = v_{3x} = v_{4x} \tag{2.31}$$

$$v_F = v_{2y} = v_{3y} \tag{2.32}$$

$$v_B = v_{1y} = v_{4y} \tag{2.33}$$

where v_L and v_R represent the longitudinal coordinates of the left and right wheel velocities, v_F and v_B are the lateral coordinates of the front and rear wheels velocities. Accordingly, the wheel velocities are related to the robot velocities by the following equation:

$$\begin{bmatrix} v_L \\ v_R \\ v_F \\ v_B \end{bmatrix} = \begin{bmatrix} 1 & -c \\ 1 & c \\ 0 & -x_{ICR}+b \\ 0 & -x_{ICR}-a \end{bmatrix} \begin{bmatrix} v_x \\ \omega \end{bmatrix} \tag{2.34}$$

Assuming that the effective rolling radius is $r_i = r$ for each wheel, then, the following approximated relations between the angular wheel velocities and the robot velocities can be developed:

$$\begin{bmatrix} v_x \\ \omega \end{bmatrix} = r \begin{bmatrix} \frac{\omega_L+\omega_R}{2} \\ \frac{-\omega_L+\omega_R}{2c} \end{bmatrix} \tag{2.35}$$

The accuracy of the previous equation depends on the longitudinal slip and can be valid if this longitudinal slip is not dominant. Finally, to complete the kinematic model, the lateral velocity v_y is constrained by the rotation of the robot and can be expressed by the following equation:

$$v_y + x_{ICR}\dot{\theta} = 0 \tag{2.36}$$

This last equation is not integrable. It describes a nonholonomic constraint that can be written in the form:

$$[-sin(\theta)cos(\theta)x_{IRC}][\dot{X}\dot{Y}\dot{\theta}]^T = A(q)\dot{q} = 0 \tag{2.37}$$

The position x_{ICR} has a critical influence on the stability of the mobile robot [2]. If the origin of the robot reference frame is placed in the middle of the distance between the front and the rear axles, the value of x_{ICR} is bounded by the following relationship [2]:

$$x_{ICR} \in [-a, b] \tag{2.38}$$

2.6 Multi-output Bayesian regression

It is very difficult to predict the exact motion of SSMRs depending on the kinematic model described in the previous section. The reasons for that are this model assumes a pure slip-free rolling in the longitudinal direction which limits the capabilities of the model to predict v_x and ω of the robot practically. It is possible to increase the model accuracy by an online learning of the kinematic parameters r and c of equation 3.7 [13]. Moreover, the only possible way to predict the v_y component constrained by the non-holonomic constraint (equation 3.8) is to experimentally learn a parametric representation of the parameter x_{ICR} as a function of the robot speed and location [33]. This introduces a multi-output regression problem that aims

to learn the robot kinematic parameters (r, c and x_{IRC}) of the SSMR by observing and fitting data representing wheel speeds $[\omega_L \omega_R]^T$ and robot speed $[v_x \omega v_y]^T$. For such multi-output regression problems the training data set of N instances takes the following form [6]:

$$D = (x^{(1)}, y^{(1)}), ..., (x^{(N)}, y^{(N)}) \tag{2.39}$$

Where the input feature vector is $x^{(l)} = (x_1^{(l)}..x_m^{(l)})$ and the output vector is $y^{(l)} = (y_1^{(l)}..y_d^{(l)})$. The task is to learn a multi-output regression model from D to a function h that assigns to each instance l targed values Ω:

$$h : \Omega_{X_1} \times ... \times \Omega_{X_M} \rightarrow \Omega_{Y_1} \times ... \times \Omega_{Y_d}$$
$$x = (x_1, ..., x_m) \longmapsto y = (y_1, ..., y_d)$$

If the output vector contains elements that represents independent variables, it is possible to adopt the single-target (ST) method, where a multi-output model with a an output vector of length d is decomposed into d single-target models. Each model is trained based on a transformed data set: $D_i = (x^{(1)}, y_i^{(1)}), ..., (x^{(N)}, y_i^{(N)}), i \in 1, .., d$ to predict the value of a single-target variable Y_i.

Since the multi-output problem has been transformed into several single-target problems, any of-the-shelf single-target regression algorithm can be used [6]. The Bayesian regression method is introduced [7]. The Bayesian method treats the fitting problem from a probabilistic perspective. Given a data set of N input values $x = (x_1, ..., x_N)^T$ and their corresponding target values $t = (t_1, ..., t_N)^T$, it is possible to express uncertainty over the value of the target variable using a probability distribution. The mean value of this probability distribution is the parametric polynomial function of the input vector $y(x, w)$:

$$y(x, w) = w_0 + w_1 x + w_2 x^2 + .. + w_M x^M = \sum_{j=0}^{M} w_j x^j \tag{2.40}$$

Therefore, it is possible to represent the uncertainty over this mean by a precision parameter β which is the inverse variance of the distribution. This distribution is assumed to be Gaussian and can be represented by:

$$p(t|x, w, \beta) = \mathcal{N}(t|y(x, w), \beta^{-1}) \tag{2.41}$$

The aim is to determine the values of the unknown parameters w and β given the training data x,t by maximizing the likelihood. If the data are drawn independently from the distribution, then the likelihood function is given by

$$p(t|x, w, \beta) = \prod_{n=1}^{N} \mathcal{N}(t|y(x, w), \beta^{-1}) \tag{2.42}$$

To maximize a likelihood function, it is convenient to maximize its logarithm which can be expressed by

$$lnp(t|x, w, \beta) = -\frac{\beta}{2} \sum_{n=1}^{N} y(x_n, w) - {t_n}^2 + \frac{N}{2} ln\beta - \frac{N}{2} ln(2\pi) \tag{2.43}$$

To solve for the parameter vector w that maximizes the likelihood (w_{ML}), the last two terms are omitted since they don't depend on w. Thus, to maximize the first term is equivalent to minimizing its negative which is minimizing the *sum-of-squares error function.*

$$E(w) = \frac{1}{2} \sum_{n=1}^{N} y(x_n, w - t_n)^2 \tag{2.44}$$

Similarly maximizing with respect to β gives:

$$\frac{1}{\beta_{ML}} = \frac{1}{N}\sum_{n=1}^{N} y(x_n, w_{ML} - t_n) \tag{2.45}$$

Having determined the parameters w_{ML} and β_{ML}, it is possible to make a probabilistic prediction for a new input x that gives a distribution over t not just a point estimate:

$$p(t|x, w_{ML}, \beta_{ML}) = \mathcal{N}(t|y(x, w_{ML}), \beta_{ML}^{-1}) \tag{2.46}$$

To create a fully Bayesian-based regression model, it is necessary to introduce prior distribution over the parameters w that will be influenced by the likelihood of the training data x, t to give an adjusted posterior distribution over w. First of all, it is possible to assume a Gaussian distribution as a prior distribution for the parameters w:

$$p(w|\alpha) = \mathcal{N}(w|0, \alpha^{-1}I) = \frac{\alpha}{2\pi}^{(M+1)/2} exp{-\frac{\alpha}{2}w^T w} \tag{2.47}$$

where α is the precision of the distribution, and $M+1$ is the total number of elements of vector w. Accoring to the Bayes' theorom, the posterior distribution for w is proportional to the product of the prior distribution and the likelihood function:

$$p(w|x, t, \alpha, \beta) \propto p(t, x, w, \beta)p(w|\alpha) \tag{2.48}$$

To determine w, it is required to maximize the posterior distribution. By taking its negative of its logarithm, it is possible to maximize the posterior distribution by minimizing the following function:

$$\frac{\beta}{2}\sum_{n=1}^{N} y(x_n, w - t_n)^2 + \frac{\alpha}{2}w^T w \tag{2.49}$$

This means that including a prior distribution allows minimizing the values of the parameter vector w to prevent over-fitting which can be controlled by the precision parameter α.

3 Problem Analysis

The Q-learning algorithm is widely used in the literature to search for optimal autonomous navigation paths. The power of such an algorithm lies in its ability to estimate a unique solution for the optimal action-value function q_* even if the environment dynamics are unknown and stochastic. This chapter shows how the navigation problem can be formulated as a reinforcement learning problem that can be solved using the model-based Q-learning algorithm. Firstly, it explains requirements of system representation which are necessary to use the Q-learning algorithm. Then, it analyzes the parameters of the Q-learning algorithm and the Dyna-Q algorithm showing how they influence the optimality and the speed of convergence of the navigation problem. Finally, it analyzes the nature of motions of Skid-Steering-Mobile-Robots showing the stochasticity it introduces to the system dynamics and elaborates on the requirements needed to model their motion.

3.1 System representation

In the context of the reinforcement learning problem, the system should be fully described by three concepts: the state, the action and the reward. These concepts have been fully defined in section 2.1. The navigation system representation is guided by the requirements imposed by the Q-learning algorithm on its states, actions and rewards representations to guarantee the convergence to (sub)optimal policies.

3.1.1 States representation requirements

The standard Q-learning algorithm is guaranteed to converge to an optimal policy for a finite episodic Markovian decision process. To satisfy this convergence condition the following considerations are taken into account:

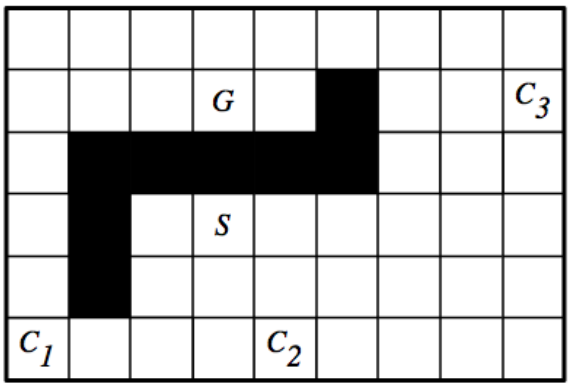


Figure 3.1: A grid-world map representation [24].

- **Finite representation:** this means that the set of states to describe the system must be represented in a discrete form. The navigation environment is assumed to contain static objects. Consequently, actions applied by the robot on the environment do not change the environment's properties but change only the pose of the robot in the navigation environment. Therefore, it is sufficient to consider the position of the robot inside the environment as the system states. Usually, for an autonomous navigation problem, the map is discretized into a set of squares along the x and y axis leading to a formulation of a grid-world problem [12] as shown in figure 3.1. However, for Skid-Steering-Mobile-Robots (SSMRs), the evolution of the system states is orientation-dependant (section 2.5). This means that it is necessary to include the orientation of the robot θ with respect to an in-

ertial frame to achieve a minimal representation for the system. This orientation state must also be discretized.

- **Episodic representation:** to make the navigation process episodic, at least one of the system states has to be defined as a terminal state s_T [28]. It is convenient to assign the target location where the robot should navigate to as a system's terminal state. If the robot successfully reaches this goal location, it should receive a maximum reward r_T to increase the system desirability to move to this terminal state. In order to start another training episode, the robot should be reset to a starting state which is different from the terminal state (section 2.1).

- **Markovian representation:** a Markovian system only requires knowledge about its current state $s \in S$ and its dynamics (state transition probabilities) to predict all possible future states $p(s'|s,a) \forall s, s' \in S$ given the action $a \in A(s)$. This means that the past trajectories of system states don't influence the future states probabilities. One of the possible reasons for the system to be non-Markovian is low discretization levels.

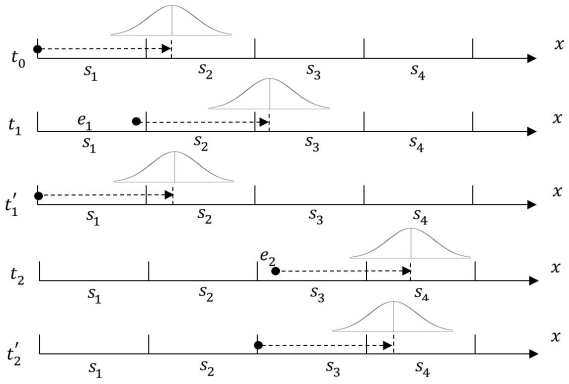


Figure 3.2: Actual and predicted locations for one directional motion in low discrete levels.

Figure 3.2 shows how low discretization levels lead to violating the Markov property. Assume that the robot can move only a long one direction with the shown discretization levels. The transition probability after applying a move-forward action is given by a Gaussian distribution shown in the figure 3.2. After applying the action, the robot is predicted to move from s_1 to s_2 with probability $P_{s_1} \approx 1$ or it can remain in state s_1 with probability $P_{s_2} = 1 - P_{s_1} \approx 0$. Assume that the latter case happened (Axis t_1). Thus, after the motion the observed state is s_1 and from the observer point of view the robot has not executed any movement. Thus, the prediction for the following state at time t_1' will remain the same as time t_0 with a probability P_{s_1} to move to s_2 and a probability P_{s_2} to remain in s_1. This prediction diverges from the reality shown in Axis t_1 where the real probabilities are $P_{s_3} > 0.5$ to move to $s3$ and $P_{s_2} = 1 - P_{s_3} < 0.5$ to remain in s_2. Now after executing the forward action the robot moves to state s_3 (Axis t_2). This transition is observed and the robot is at s_3 from the observer prospective. Now, the prediction of the following state given at axis t_2' is more accurate and similar to the real distribution given by axis t_2. The reason for that is the ability of the model to predict a distribution over the following state depends on the quantization error e. In the case when the error is large (e_1 in axis t_1) the model prediction is less accurate than when the error is small (e_2 in axis t_2). Therefore, since low discretization levels increase the possibility to have high quantization errors, it lowers the system's Markovian properties. This is because it is not sufficient to know the system dynamics (state transition probabilities) to predict the future states, in addition, it is needed to know the quantization errors. For the navigation problem, this can be avoided by having sufficiently high discretization levels. This helps to avoid problems

associated with the non-Markovian behaviour such as mis-classifying free and obstacle states as elaborated in figure 3.3.

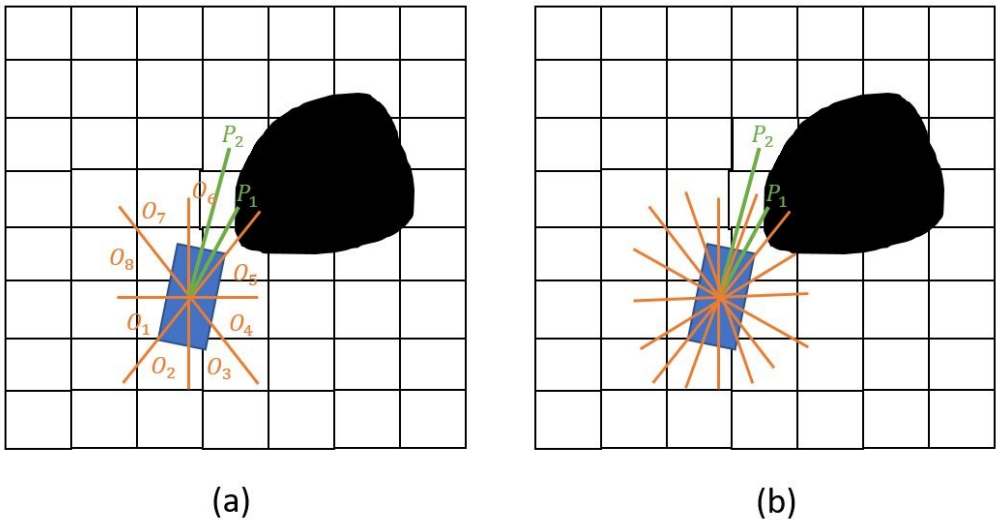


Figure 3.3: (a) Low discretization levels lead to assigning ortientation level O_6 as a free state if path P_2 is executed instead of path P_1 even if it is an obstacle state. (b) This misclassification does not occur if the orientation level is highly discretized

3.1.2 Actions representation requirements

Another necessary condition for the Q-learning algorithm convergence is that every state s in the state space S has to be visited [28]. In practice, sub-optimal solutions can be discovered even if the whole state space is not fully spanned. However, it is necessary to guarantee that the actions defined over each state will enable exploring paths between the initial and the target states. This can only be guaranteed if each state ($s \in S$) is reachable from at least one different state ($s' \in S$) by at least one action a in the action space defined over s' ($a \in A(s')$). This condition can be satisfied by analyzing how the velocity states of the robot change due to the wheel velocities (section 2.5). This relation is given by:

$$\begin{bmatrix} v_x \\ \omega \end{bmatrix} = r \begin{bmatrix} \frac{\omega_L+\omega_R}{2} \\ \frac{-\omega_L+\omega_R}{2c} \end{bmatrix} \tag{3.1}$$

and:

$$v_y + x_{ICR}\dot{\theta} = 0 \tag{3.2}$$

Where $[v_x\ \omega\ v_y]^T$ is the COM robot velocity vector expressed in the robot local frame (figure 2.6). This means that in order for all the states $[v_x\ \omega\ v_y]^T$ to evolve, at least one of the applied actions should have wheels velocities (ω_L and ω_R) with different values. This is sufficiently obtained if the action space is chosen to be (forward, backward, left and right) defined over every discrete state of the state space. The conditions on the wheels velocities that should be applied for each action can be described as follows:

- **Forward:** $\omega_L = \omega_R > 0$
- **Backward:** $\omega_L = \omega_R < 0$
- **Left:** $\omega_L < 0$ and $\omega_R > 0$ and $|\omega_L| = |\omega_R|$

- **Right:** ω_L >0 and ω_R <0 and $|\omega_L| = |\omega_R|$

3.1.3 Reward functions representations:

The Q-learning algorithm converges to an optimal policy that maximizes rewards given by the environment on the long-term. Reward functions should be designed to achieve the required goal of the control problem. In classical reinforcement learning algorithms, the agent receives a reward of value 1 if the task is fully accomplished correctly and 0 otherwise. This reward shaping idea is called *binary reward*. However, it is not efficient to depend on such an idea for navigation problems because if the state space is significantly large, the chance of finding the reward is very small [26]. Therefore, there is a need to depend on dynamic reward functions where the reward given changes with the state transitions even if the task is not fully completed. For the navigation problem, a *dense reward* function can be defined based on the distance between the mobile robot, the obstacles and the target locations . The main disadvantage of such method is that dense reward functions can cause local minima problems [26]. On the other hand, it is sufficient to evaluate the mobile robot behaviour even if the task is not completed by giving -1 for hitting an obstacle or moving on a sliding area, 1 for reaching the goal and 0 otherwise. This shaping idea is known as *sparse reward* [26]. The sparse reward shaping has been proven to be sufficient to guarantee convergence of the Q-learning algorithm when used for several autonomous navigation tasks [11]. Finally, in [4], the idea of the reward function transformation has been suggested to accelerate the convergence of the Q-learning algorithm. This idea has been first introduced in [20]. The idea aims to find a transformation to the sparse reward function that incorporates knowledge about the navigation environment in the design of the reward function. The approach assigns a potential value $\phi(s)$ to each cell in the navigation grid. This potential is determined based on the Manhattan distance between the cell and the goal and the Manhattan distance between the cell and known obstacles. The transformation applied to the sparse reward will be $R' = R + F$ where R is the sparse reward value and F is calculated by the potential difference:

$$F(s, a, s') = \gamma\phi(s') - \phi(s) \tag{3.3}$$

This reward shaping idea is called *potential-based reward* and it is very interesting option because it incorporates knowledge about the navigation problem inside the reward function design. The only disadvantage is that the potential function for each state must be recalculated for each state every time the navigation map is updated increasing the computational time of the solution.

Q-learning (off-policy TD control) for estimating $\pi \approx \pi_*$

Algorithm parameters: step size $\alpha \in (0, 1]$, small $\varepsilon > 0$
Initialize $Q(s, a)$, for all $s \in \mathcal{S}^+, a \in \mathcal{A}(s)$, 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-greedy)
    Take action A, observe R, S'
    $Q(S, A) \leftarrow Q(S, A) + \alpha\big[R + \gamma \max_a Q(S', a) - Q(S, A)\big]$
    $S \leftarrow S'$
  until S is terminal

Figure 3.4: Q-learning algorithm for approximating q_* [28].

3.2 Q-learning parameters selection

The Q-learning algorithm aims to converge to the optimal action-value functions q_* for each state-action pair. It convergences numerically by updating the action-value using the back-up updating loop given by algorithm shown in figure 3.4.

The speed of convergence of the algorithm is affected by the choice of the following parameters: initial values Q_0, the learning rate α, the discount rate γ and the exploration factor ϵ. The optimal choice for these parameters is problem-dependant. Any previous knowledge about the stochastic process can be implicitly included in the choice of these parameters to accelerate the convergence of the algorithm. This section shows how these parameters can be selected optimally for the purpose of the autonomous navigation.

3.2.1 Initial values Q_0 selection

The Q-learning algorithm can be initialized by selecting arbitrary values for each action-value function q_0. However, they can be selected to supply some prior knowledge about expected rewards [28] as discussed in the following subsections.

Optimistic initial values

Selecting initial values can be used as a simple strategy to enhance exploration [28]. This is obtained by choosing high "optimistic" initial values for the action-value function. During the learning process, whichever action is selected, the reward is less than the initial estimate for q; the learner will explore other actions being dissatisfied with the rewards it is receiving. In the beginning, the optimistic method performs worse since it explores more, but with time it performs better since the exploration decreases with time. Figure 3.5 shows the effect of choosing optimistic initial values on the convergence speed of the grid world problem.

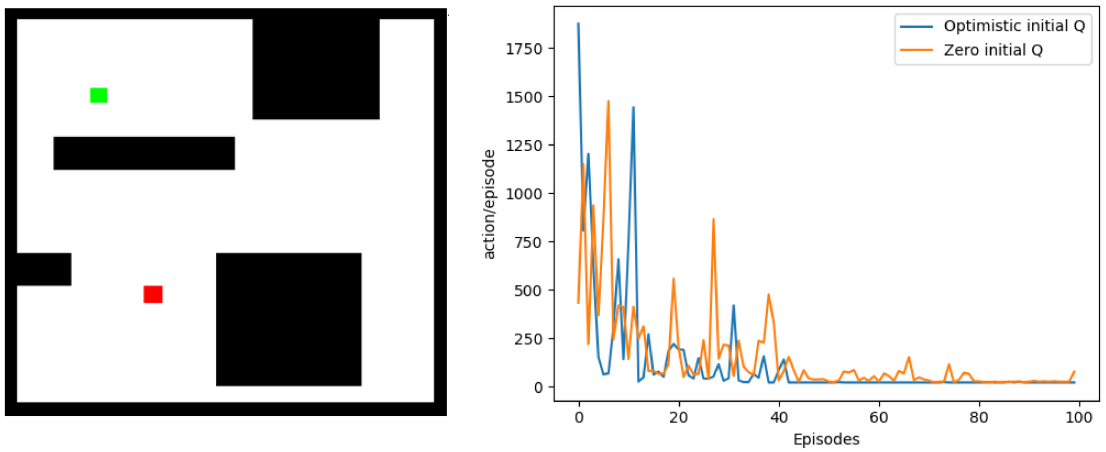


Figure 3.5: (a) A grid world problem; the task is to learn an optimal path from the green to the red cell. (b) Assigning optimistic initial values for Q has worse behaviour at the beginning since it explores more but reaches the optimal path quicker than zero initial values (after 40 episdoes).

Biased initial values

The Q-learning algorithm converges to the optimal policy π_* regardless of the policy applied to explore the environment. It is possible to make the exploration policy biased to paths that lead to the target assuming obstacle-free environment. This strategy aims to follow paths that lead to the target location faster to accelerate finding the positive reward from the environment. The idea is to increase the initial action-value q_0 for actions that will bring the robot to follow

these paths. If one of these paths intersects with an obstacle, the negative reward from the environment will cause the action-value function of these actions to decrease, and consequently, diverging from these paths. In order to determine which actions should be given higher initial values at each state, figure 3.6 is analyzed.

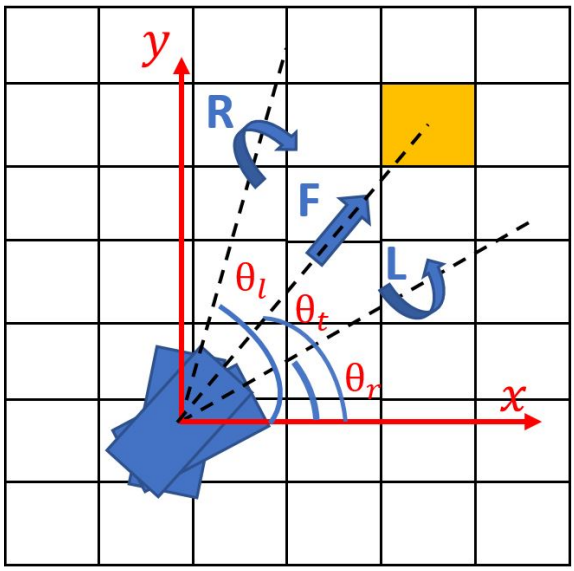


Figure 3.6: The preference is given to a specific action depending on the orientation of the robot

The target orientation θ_t is defined by the angle between the horizontal x and the line drawn between the robot COM and the target. If the robot orientation is the same as θ_t, the optimal action to be taken is the move-forward action. On the other hand, if the robot orientation is on the left of this line ($\theta_l > \theta_t$), taking a turn-right action will adjust the robot orientation to align with the line. Similarly, if the robot orientation is on the right of this line ($\theta_r < \theta_t$) the optimal action to be taken is the turn-left action. The assignment of biased initial values can be done by the following pseudo-code:

Result: Biased initial q_0
Initialize x_{target} ;
Initialize y_{target} ;
for *every (x,y,θ) state* **do**
  $\theta_{target} = tan^{-1}(y_{target} - y / x_{target} - x)$;
  $e = \theta_{target} - \theta$;
  if $e == 0$ **then**
    $q_0[(x, y, \theta), F] >> 0$;
  end
  if $e < 0$ **then**
    $q_0[(x, y, \theta), L] >> 0$;
  end
  if $e > 0$ **then**
    $q_0[(x, y, \theta), R] >> 0$;
  end
end

Algorithm 1: Assigning biased q_0 for actions at each state (x, y, θ).

Adopting this approach to initialize q_0 values has been proven to be powerful in finding paths that lead to the target location in the exploration phase.

3.2.2 Learning rate α selection

The learning rate α is used to give a desired weight to the recent reward R over the past rewards represented in the previous action-value $Q(S,A)$ while updating it, as shown in figure 3.4. This learning rate takes a values between 0 and 1 ($\alpha \in [0,1]$) such that the larger the value, the more the weight given to the recent expected return ($R + \gamma max|_a[Q(s',a')]$) over the previous $Q(S,A)$ and vice versa. In the case where the stochastic process is non-stationary, the environment supplies a variable reward $r(t)$ to the learning agent if the same action a is applied at the same state s. This means that this reward changes with time and the learner has to track its change. To achieve that, the most recent received reward must have higher weight over previous rewards, so the learning rate must take a high value (≈ 1). In the context of the navigation problem where the environment is static, the stochastic process can be considered stationary. However, in the beginning of the learning process, since the algorithm starts from arbitrary initial values q_0, the received rewards must have high weights over q_0. Therefore, in the beginning of the learning process, the learning rate α is set to a huge value. Then, if the learning agent starts reaching an approximate optimal action-values q_*, the learning rate α can be adjusted to a lower value to filter the effect of faulty rewards. In the autonomous navigation problem, faulty rewards can occur due to mis-classifying free and obstacle states or due to skidding effects. Thus, the idea of decreasing the learning rate α with time can limit these effects on the optimal action-values q_*. Algorithm 2 shows how to adjust the value of α when the action-values $Q(S,A)$ getting close their optimal values. This is detected if the change in their values after the numerical update δ is less than a threshold θ for each (S,A) pair.

Result: Adjusted learning rate α
Initialize α ;
Initialize θ ;
Initialize δ ;
Initialize $discount\_factor$;
while *True* **do**
  $\delta = \min$ ($Q(S,A) - Q(S,A)_{prev}$,δ);
  if $\delta < \theta$ **then**
    $\alpha* = discount\_factor$;
  end
end

Algorithm 2: Assigning biased q_0 for actions at each state (x, y, θ).

3.2.3 Discount rate γ selection

The discount rate γ gives a desired weight between the effect of the current reward R and the effect of future rewards encoded in the value of the next state $V(S') = max|_a Q(S',a)$ on the update of the current action-value $Q(S,A)$ as shown in figure 3.4. This parameter can be set between 0 and 1 ($\gamma \in [0,1]$). Since the target of the reinforcement leaning problem is to maximize long-term rewards, this parameter is usually set to a high value to encourage moving to states with higher optimal values $V^*(s')$. For the purpose of accelerating the learning in navigation problems, this parameter must be set to a huge value (≈ 1), specially in the case of sparse reward shaping where all free states give a 0 reward ($R = 0$). In this case, the learner can depend more on the value of the next state $V(S') = max|_a Q(S',a)$ to update the value of $Q(S,A)$ because a zero reward will not cause any update to its value.

3.2.4 Exploration factor ϵ selection

The Q-learning algorithm can only converge if every state-action pair is visited at least once during the learning process. Therefore, the exploration is an inherent requirement for any reinforcement learning algorithm. At each state, selecting the most optimal action according to

the current q-values is denoted by *exploitation* and this exploitation policy is called the *greedy policy* $\pi_*(s)$. On the other hand, it is required to visit new state-action pairs in the state-action spaces, by selecting actions randomly with probability ϵ resulting in adopting the *ϵ-greedy policy*. For navigation tasks in an unknown environment, the exploration factor ϵ must have high value in the beginning of the learning process for the following two reasons:

- It helps to discover various parts of the state space till finding states that supply the (positive) reward.
- It helps to build a more accurate model for the state transition probabilities of the system needed to implement a model based Q-learning (section 2.2).

Since the navigation environment is assumed to be static, it is advisable to decease the exploration rate with time after the action-value functions are getting close to their optimal values $q*(s,a)$. This can be done by multiplying the exploration factor ϵ by a discount factor. This reduction happens after each episode termination or after discovering a shorter navigation path to the target location as shown in algorithm 3 in section 4.1.2. This approach is denoted by *discounted ϵ-greedy*. Finally, in the literature several approaches are described to optimize or to direct the exploration some of which are described in the subsections below.

Softmax action selection

One drawback of the ϵ-greedy method is that it explores equally among all actions[28]. The softmax method varies the action probabilities as a graded function of estimated value. The greedy-action will be given highest probability, but all others are ranked and weighted according to their Q-value. One of these softmax action selection rules uses a Gibbs, or Boltzmann, distribution. It assigns to each action a a probability according to the following rule

$$P(\alpha) = \frac{e^{Q(\alpha)/\tau}}{\sum e^{Q(b)/\tau}} \tag{3.4}$$

where τ is a positive parameter called the temperature and b denotes each non-greedy action. High temperatures cause the actions to be nearly equi-probable. Low temperatures cause a great difference in selection probability of actions according to their Q-values.

Directed exploration method

This method aims to direct the exploration towards more interesting areas in the state-action space instead of treating all non-greedy actions equally [4]. An exploration bonus is added to the Q-values to reflect the added value of selecting this action for the sake of exploration. Thus the Q-value for each state-action pair is modified to be

$$Q^+(s,a) = Q(s,a) + \eta\sqrt{m(s,a)}/n(s,a) \tag{3.5}$$

where η is a constant, $m(s,a)$ is the number of time steps since action a was last tried in s. $n(s,a)$ is the total number of trying a in s. Then, this modified Q-value is substituted in Boltzmann distribution (3.4) to assign probabilities of selecting each action. Such a method is proven to be beneficial while depending on model-based Q-learning [4]. Otherwise, model-based Q-learning algorithms converge to sub-optimal policies. For this reason, it seems to be a promising exploration strategy for the navigation problem.

3.3 Dyna-Q-based navigation

Q-learning algorithm is a *model-free* reinforcement learning algorithm since it depends only on the learning to update action value functions for each state-action pair. On the other hand, *model-based* reinforcement learning algorithms depend on planning using a model of the environment structured online during learning [28]. A model of the environment allows the agent

to make predictions about the next states and the next rewards. If the system is stochastic, environment models produce a description of all possibilities with their probabilities leading to *distribution models*. Models can be used to simulate interaction experiences. Given a starting state and an action, a distribution model generates probabilities of all the possible transitions. Similar to model-free methods, simulated experiences generated by the distribution model are used to estimate the value functions using back-up update equations.

Tabular Dyna-Q

Initialize $Q(s,a)$ and $Model(s,a)$ for all $s \in \mathcal{S}$ and $a \in \mathcal{A}(s)$
Loop forever:
- (a) $S \leftarrow$ current (nonterminal) state
- (b) $A \leftarrow \varepsilon$-greedy(S,Q)
- (c) Take action A; observe resultant reward, R, and state, S'
- (d) $Q(S,A) \leftarrow Q(S,A) + \alpha\big[R + \gamma \max_a Q(S',a) - Q(S,A)\big]$
- (e) $Model(S,A) \leftarrow R, S'$ (assuming deterministic environment)
- (f) Loop repeat n times:
  - $S \leftarrow$ random previously observed state
  - $A \leftarrow$ random action previously taken in S
  - $R, S' \leftarrow Model(S,A)$
  - $Q(S,A) \leftarrow Q(S,A) + \alpha\big[R + \gamma \max_a Q(S',a) - Q(S,A)\big]$

Figure 3.7: The tabular Dyna-Q algorithm [28].

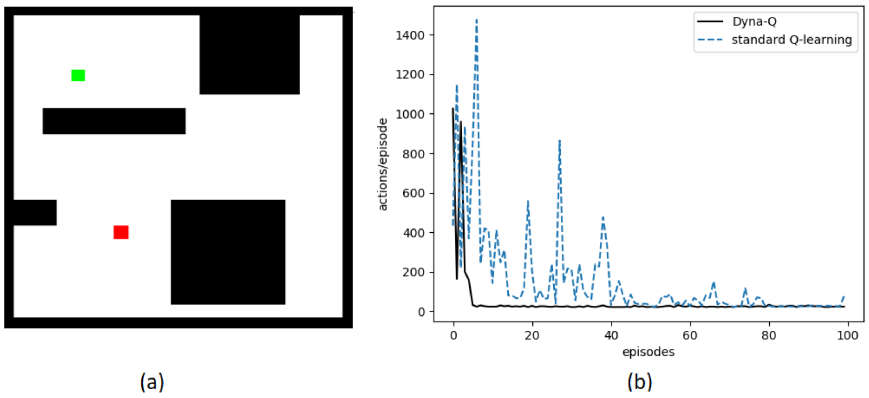


Figure 3.8: (a) A grid world problem; the task is to learn an optimal path from the green to the red cell. (b) Using the Dyna-Q algorithm is really powerful in accelerating the Q-learning convergence (after 7 episodes).

The Dyna-Q algorithm (figure 3.7) is a platform proposed in [27] that integrates model-learning, planning and value functions updating. After each interaction with the environment, the experience $(S_t, A_t \rightarrow R_{t+1}, S_{t+1})$ is used for the back-up update. Moreover, it is stored in a look-up table that maps each (S_t, A_t) to equivalent (R_{t+1}, S_{t+1}). During planning, the Q-planning algorithm randomly samples only from state-action pairs that have been experienced before and stored in the table. The data stored in the table-based model are used to simulate hypothetical experiences that can directly update the Q-values of state-action pairs just as if they really happened. The agent performs N hypothetical experiences after each real-world interaction with the environment. Integrating planning with learning accelerates the convergence to the optimal q_* significantly. The is due to the fact that any change in the q-value of a state-action pair is going to propagate to values of other pairs through hypothetical experiences. This is assumed to be very powerful specially in the case where the state space is

significantly large as in autonomous navigation problems. In order to show the potentials of the Dyna-Q algorithm, it has been simulated for the grid-world problem as shown in figure 3.8. Unlike the standard Q-learning that takes about 80 episodes to converge, it converges really fast in 7 episodes. The results of applying it to real navigation tasks are discussed in chapter 5.

3.3.1 Necessary adjustments to the navigation problem

The general Dyna-Q algorithm presented in [27] assumes a deterministic dynamical environment. However, in the context of autonomous navigation for SSMRs, sliding is an inherent system property that leads to stochastic dynamics. Thus, it is required for the learned model to be probabilistic. The basic idea is to learn a model that does not predict a deterministic next state and a deterministic reward, but a probability distribution over next states and next rewards [27]. For discrete finite MDPs, it is possible to represent the state transition probabilities $p(s', r|s, a)$ using multinomial distributions [14]. The design of the algorithm to learn these multinomial distributions is presented in section 4.1.2. However, it should be noted that the online-constructed models can never represent the navigation environment identically especially if a limited number of samples have been observed. When the model is inaccurate, the planning process will compute sub-optimal policies [28]. Using these multinomial models, it is only possible to simulate hypothetical experiences for already-visited state-action pairs (s, a). If it is required to predict probability distributions for the next states and rewards for state-action pairs that have not been visited before, an accurate model for the robot motion must be developed. Possibilities to construct this motion model for SSMRs depend on their motion characteristics that will be analyzed in the following section.

3.4 Motion analysis of Skid-Steering-Mobile-Robots

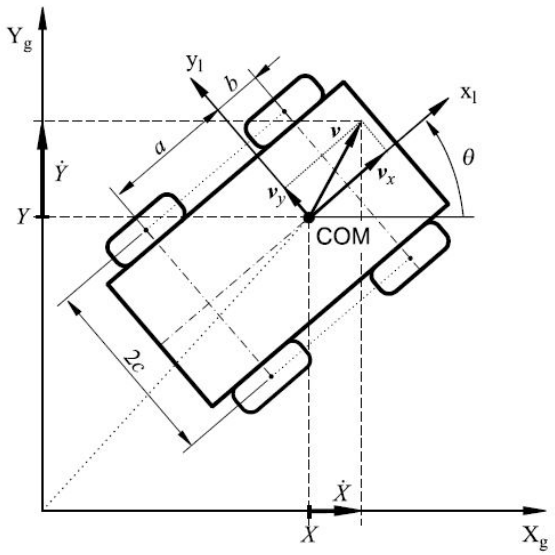


Figure 3.9: SSMR free body diagram [13].

The kinematic model for SSMRs has been explained in section 2.5. The generalized velocity (figure 3.9) equation can be described by:

$$\begin{bmatrix} \dot{X} \\ \dot{Y} \\ \dot{\theta} \end{bmatrix} = \begin{bmatrix} cos(\theta) & -sin(\theta) & 0 \\ sin(\theta) & cos(\theta) & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} v_x \\ v_y \\ \omega \end{bmatrix} \tag{3.6}$$

The vector $[v_x v_y \omega]$ represents the car velocity in its local reference frame. This velocity vector is related to the wheels angular velocity by the following equation:

$$\begin{bmatrix} v_x \\ \omega \end{bmatrix} = r \begin{bmatrix} \frac{\omega_L + \omega_R}{2} \\ \frac{-\omega_L + \omega_R}{2c} \end{bmatrix} \tag{3.7}$$

and v_y is constrained by the rotation of the robot and can be expressed by the following equation:

$$v_y + x_{ICR}\dot{\theta} = 0 \tag{3.8}$$

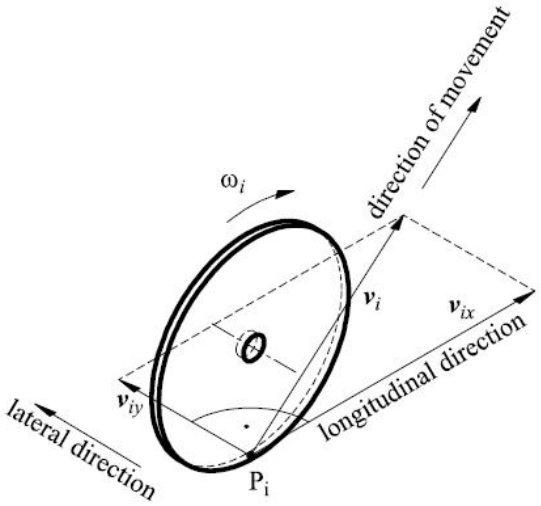


Figure 3.10: Velocities of one wheel [13].

The model represented by equation 3.7 assumes that the longitudinal slip is not dominant in the robot motion. This assumption is no longer valid if the robot is navigating above slippery surfaces. To understand the effect of the slippage on the wheels motion, figure 3.10 is analyzed. The velocity of the wheel in the longitudinal direction v_{ix} depends only on the rotation of the wheel ω_i in case of a pure rolling. However, if the wheel slides, the term *longitudinal wheel slip* λ_i is defined to fully describe the wheels motion in the longitudinal direction [18]:

$$\lambda_i = \frac{v_{ix} - r\omega_i}{v_{ix}} \tag{3.9}$$

This longitudinal wheel slip depends on the normal force acting on the wheel, the road surface conditions and tire characteristics [19]. In [1], the relation between the longitudinal wheel slip and the friction coefficient is analyzed.

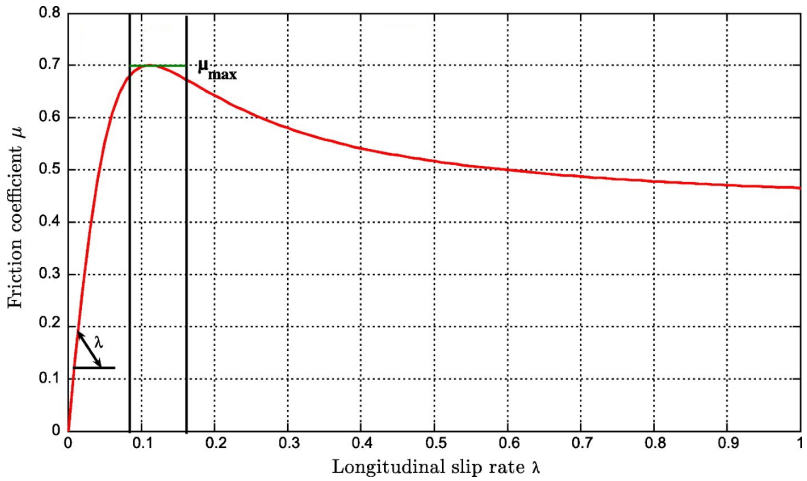


Figure 3.11: Longitudinal slip as a function of friction coefficient [1].

Figure 3.11 shows how the longitudinal slip varies with the friction coefficient.

For navigating over slippery planar surfaces, this longitudinal slip depends on the discretized (x, y) state of the robot location. In order to build an accurate model for the robot motion the longitudinal slip should be learned online and it should be location-dependant.
On the other hand, it is possible to increase the model accuracy and to account for the slippage effects by online learning of the kinematic parameters r and c of equation 3.7 [13]. Moreover, the only possible way to predict the v_y component constrained by the non-holonomic constraint (equation 3.8) is to experimentally learn a parametric representation of the parameter x_{ICR} as a function of the robot speed and location [33]. This introduces a multi-output regression problem that aims to learn the robot kinematic parameters (r, c and x_{IRC}) of the SSMR by observing and fitting data representing the wheel speeds $[\omega_L \omega_R]^T$ and the robot speeds $[v_x\ \omega\ v_y]^T$. To simplify the problem, r and c parameters are learned independently from x_{ICR}. For the learned r and c parameters the input feature vector is $x^{(l)} = (\omega_L^{(l)}\ \omega_R^{(l)})$ and the output vector is $y^{(l)} = (v_x^{(l)}\ \omega^{(l)})$. This definition matches the kinematic equation 3.7 and defines a multi-output regression problem. On the other hand, for the learned parameter x_{ICR} the input feature is $x^{(l)} = \omega^{(l)}$ and the output is $y^{(l)} = v_y^{(l)}$. This fits with the non-holonomic constraint equation 3.8 and defines a single target regression problem. According to [6], since the two elements of the output vector of the first regression problem ($y^{(l)} = (v_x^{(l)}\ \omega^{(l)})$) represent two independent variables, it is possible to decompose the problem into two separate regression problems each solves for 1 element of the output vector. Therefore, it is possible to learn r and c independently by defining two independent output vectors $y_1^{(l)} = v_x^{(l)}$ and $y_2^{(l)} = \omega^{(l)}$ for the input feature vector $x^{(l)} = (\omega_L^{(l)}\ \omega_R^{(l)})$. Since the multi-output problem has been transformed into three single-target problems, one of-the-shelf single-target regression algorithm can be used [6]. To account for the noise in the collected features and outputs data, a Bayesian ridge regression algorithm is used to learn the kinematic model of the robot experimentally [7].

4 Learning Algorithms Design

This chapter proposes a design structure of a reinforcement-learning-based navigation system guided by the analysis introduced in chapter 3. The algorithm is designed based on the Dyna-Q platform which is modified by creating a multinomial probabilistic model to describe the dynamics of the system in the form of learned state transition probabilities. The algorithm is enhanced by creating an online-structured Bayesian regression model to learn the kinematic model of the mobile robot and use it to estimate the robot motion for newly-visited states. These models are used to simulate hypothetical experiences which is assumed to accelerate the convergence of the Q-learning algorithm. Finally, an idea to prioritize the simulated experience around the shortest path experienced by the robot is discussed.

4.1 Dyna-Q platform design

4.1.1 The boundary between the agent and the environment

It is essential to clarify the boundaries between the agent and the environment in order to fully determine the system observable states. The agent's internal states represent the pose $[x\ y\ \theta]$ of the COM of the mobile robot. These states are fully observable to the agent by a ceiling camera as explained in section 5.2.3. The state transitions of the robot is modeled as an MDP as was explained in section 3.1.1. The environment represents all the interacting components that cause the state of the COM of the robot to change. This includes the wheels dynamics, the surface friction properties and the distributed obstacles in the navigation environment. The properties of the environment are assumed static because the friction properties and the obstacles locations are assumed unchanging during the training period. Therefore, applied action changes only the pose of the robot which is only considered as the system state. The state observer algorithm (section 5.2.3) stores the current and the previous pose state of the mobile robot and calculates the reward accordingly. Thus, the rewards are not given from the environment directly but they are computed inside the algorithm and they are supplied to the agent consequently. More elaborations on how the rewards are computed are explained in section 4.1.3.

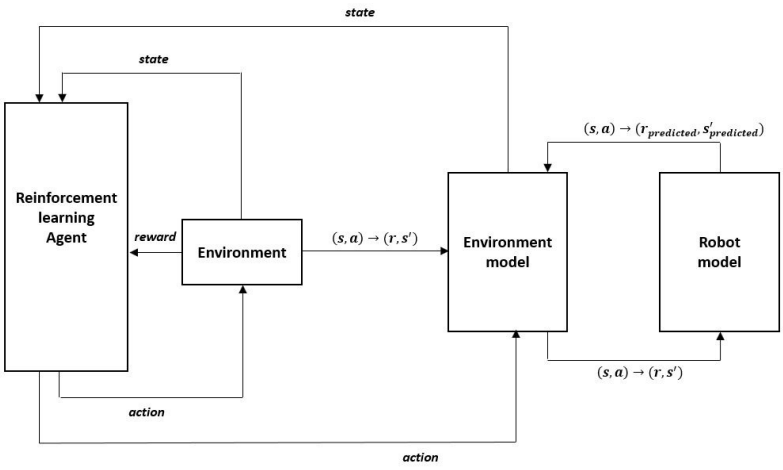


Figure 4.1: Interacting components of the reinforcement learning navigation system.

The structure of the Dyna-Q platform designed in Python-based IDE is shown in figure 4.1. Figure 4.1 shows an abstract representation of the interacting components of the reinforcement learning algorithm based on a Dyna-Q platform.

- **The reinforcement-learning agent:** this component stores the data related to the action-value functions for each state-action pair $Q(S, A)$. These action-values are updated based on the standard Q-learning back-up update rule explained in section 2.1.3 while training the agent in simulations or in real-time experiments. The agent needs to be initialized with the parameters needed to implement the Q-learning algorithm guided by the selection rules presented in section 3.2. By analyzing the action-values q stored, an exploration or an exploitation policy is followed to apply an action to the environment.

- **The environment:** this component represents the physical properties of the navigation environment that interacts with the reinforcement-learning agent. In simulation, as will be explained in this section, it represents a simulated behaviour of a SSMR navigating in a slippery environment with obstacles. If the learning agent applies an action on the environment, the action will cause the change of the SSMR states. The agent receives as feedback the new states and a scalar reward value. This reward value characterizes the desirability of the applied actions in terms of reaching the goal, hitting an obstacle, or moving on a slippery area. The selection of the reward values are guided by rules presented in section 3.1.3. In real-time, this component represents data obtained from the real-robot to represent its state and data sent to it to execute the actions. This will be explained in details in section 5.2.3

- **The environment model:** this component is created by storing all the navigation experiences $(s, a) \rightarrow (s', r)$ to build a distribution model for the system dynamics $p(s', r)|(s, a)$. Since the state-action spaces are discrete, this distribution model is constructed to be a multinomial probabilistic model (section 2.4) and will be used to simulate hypothetical motion experiences that can directly update the action-values q for every state-action pair.

- **The robot model:** this component represents an area-dependant learned kinematic model of the SSMR. This model is constructed online by gathering data representing the wheel velocities $[\omega_L \ \omega_R]$ and the robot velocities $[v_x \ v_y \ \omega]$ and using a Bayesian regression model to fit these data (section 4.2). This model is used while creating hypothetical experiences to predict future states and rewards for state-action pairs that have not been experienced before and are not stored in the environment model. The model assumes an obstacles-free environment while making predictions.

4.1.2 Platform realization

The interacting components are realized through python classes. In this section the structure of the three main classes necessary for the Dyna-Q algorithm (QLearn class, env class and world_model class) are presented. The last component (robot_model) which is an extension to the main algorithm is introduced in section 4.2. In this section, an abstracted representation for these classes is shown to explain how their data are structured and their methods functionality.

The reinforcement-learning agent

The agent contains the parameters needed to implement the Q-learning algorithm (α, γ and ϵ) and it has a dictionary q that maps every state-action pair to its action value. The main two functions of the agent are:

- **Actions selection policy:** It is possible to select an action to be applied on the environment based on a desired policy π. This policy is chosen to be an $\epsilon - greedy$ policy to ensure sufficient exploration of the state space.

  Result: $\epsilon - greedy$ action
  Initialize $rand \in [0,1]$;
  if $rand > \epsilon$ **then**
  | action = $argmax|_a\{q(s,a)\}\ \forall a$;
  else
  | greedy_action = $argmax|_a\{q(s,a)\}\ \forall a$;
  | action = $argmax|_a\{q(s,a)\}\ \forall a \neq greedy\_action$;
  end

  Algorithm 3: Selecting an action based on an $\epsilon - greedy$ policy.

  The policy is shown by algorithm 3. A random variable is selected between 0 and 1 ($rand \in [0,1]$). The action with the highest value is selected if ($rand > \epsilon$). Otherwise, the action with the second highest value is selected. This exploration strategy is called *softmax action selection* (section 3.2). The QLearn class applies this function by the method **chooseAction()**.

- **Action-values backup:** the aim of the algorithm is to converge to the optimal action-values for every state-action pair $q_*(S,A)$. This is achieved by doing a numerical backup update for the value $Q(S_t,A_t)$ of each (S_t,A_t) after receiving a reward R_{t+1} and moving to a new state S_{t+1} according to the following rule:

$$Q(S_t,A_t) \leftarrow Q(S_t,A_t) + \alpha[R_{t+1} + \gamma max|_a Q(S_{t+1},a) - Q(S_t,A_t)] \tag{4.1}$$

  It is also possible to store the rewards received for n interaction steps and use weighted sum of them to do the backup of the state-action pair in the beginning of these n steps (S_{t-n+1}, A_{t-n+1}). This weighted sum is called the *discounter return* and is defined by [28]:

$$G_{t:t+n} = R_{t+1} + \gamma R_{t+2} + ... + \gamma^{n-1} R_{t+n} + \gamma^n Q_{t+n-1}(S_{t+n}, A_{t+n}), n \geq 1, 0 \leq t \tag{4.2}$$

  Following this definition the updating rule is:

$$Q(S_{t-n+1}, A_{t-n+1}) \leftarrow Q(S_{t-n+1}, A_{t-n+1}) + \alpha[G_{t:t+n} - Q(S_{t-n+1}, A_{t-n+1})] \tag{4.3}$$

  This idea is called *n-step bootstrapping* and it is desirable for systems where it is needed to wait for a significant length of steps to receive a recognizable change in the given reward. This is useful for navigation problems if the orientation level is highly-discretized. The QLearn class applies this backup function by the method **learn(state_prev,action,state,reward)**.

The environment

The environment can be fully characterized by the physical properties of the SSMR, slippage coefficients at each state and the obstacles configurations. The env class is developed to represent the environment interacting with the agent. The structure of this class for the real-time experiments is presented in section 5.2.3. In simulation, this class is formed by two classes. The SSMR class (section 5.4) is used to represent the motion properties of the robot and slippage properties at every positional state (x,y). In order to simulate different obstacles configurations, the library *gym-pathfinding* developed by [31] is used. The library enables creating discrete navigation environments with obstacles at different configurations 4.2.

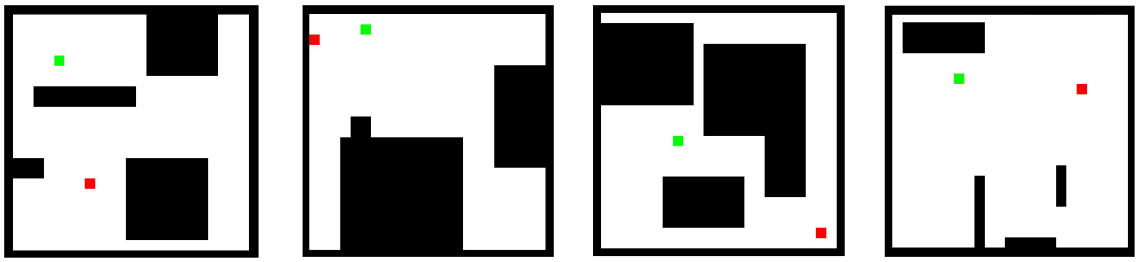


Figure 4.2: Discrete navigation environments generated by *gym-pathfinding* with different obstacles configurations.

Result: Reward R_{t+1}, Next state S_{t+1}
Initialize S_t ;
for *every step(action A) call* **do**
  Execute action A by the SSMR ;
  Observe S_{t+1} of the SSMR ;
  if $St = S_{t+1}$ **then**
    $R = -1$;
  else
    if $S_{t+1} = target$ **then**
      $R = 1$;
      Set Terminal to True ;
    else
      $R = f(S_t, S_{t+1})$;
    end
  end
  $S_t = S_{t+1}$;
  Return R_{t+1}, S_{t+1} ;
end

Algorithm 4: The environment-agent interaction loop.

Result: Initial state Si
Set Terminal to False ;
Set S_{t+1} to S_i ;

Algorithm 5: The environment reset algorithm.

The agent interacts with the environment through the sequence diagram explained in algorithm 4. The situation at which the agent applies an action on the environment is called a step and is implemented by calling a step() function with the desired action. When the agent applies this *step*, the motion of the robot is simulated based on the selected desired action A. Then, the new state of the robot S_{t+1} is observed. Based on the change of this state, the environment returns a scalar reward to the agent. If this state is not changing after applying the action, this is an indication for hitting an obstacle, so a reward -1 is returned. On the other hand, if the new state is the target location, it is an indication that the task has been completed successfully, so a reward 1 is sent to the agent. Otherwise, the reward can be a function of the previous and current states. For example, if the Euclidean distance between the new and current state is significantly small which indicates slipping, a negative reward can be sent to the agent. The calculated reward R_{t+1} and the new state S_{t+1} are returned to the agent to calculate an updated value for the state-action pair (S, A). When the target is reached a terminal flag (algorithm 4) is set to true. Then, the agent starts a new training episode, so the robot should be reset to its initial state S_i by algorithms 5.

Multinomial probabilistic world model

Creating an online model of the environment dynamics is the core-stone of the model-based reinforcement learning. This model can be used to simulate hypothetical training experiences and update the action-values q as if they have occurred in reality. Therefore, any change in the q-value of a state-action pair is going to propagate to values of other pairs through hypothetical experiences. This helps accelerating the convergence of the algorithm significantly. The basic idea to create this model is to build a look-up table that maps each state-action pair (s,a) to a next state and a reward (s',r). This idea $((s,a) \rightarrow (s',r))$ is valid while dealing with deterministic processes. If the process is stochastic, the model should represent a probability distribution over the next states and rewards $p(s',r)|(s,a)$, which is called a *distribution model* [28]. In the case where the output of the stochastic process is discrete, the probabilistic models that can describe the process are *multinomial distribution models* [7]. In the context of the navigation problem, applying a distinct action A_d at a particular state S_i can be treated as an independent stochastic event that can yield a sequence of discrete K observations $\{({S'_j}^1, R^1), ({S'_j}^2, R^2), ..., ({S'_j}^K, R^K)\}$. Note that the elements of each yield (S_j, R) are dependent. Therefore the mapping is modified to:

$$(S_i, A_d) \rightarrow \{({S'_j}^1, R^1), ({S'_j}^2, R^2), ..., ({S'_j}^K, R^K)\}, d \in \{1,2,3,4\}, i \neq j \in \{1, ..., n \times m \times o\} \tag{4.4}$$

where d is the number of possible actions at each state i and $n \times m \times o$ represents the total number of discrete states when the discretization levels of the x, y and θ states are n, m and o respectively. The aim of such mapping is to build a probability for observing a particular yield (S'_j, R) given (S_i, A_d). Assume that this yield has been observed m times out of K observation. Therefore,

$$p(S'_j, R)|(S_i, A_d) = \mu_j = \frac{m}{K} \tag{4.5}$$

Result: world_model
Initialize world_model table with (S_t, At) key ;
for *every* (S_t, At) **do**
    Initialize S_{t+1} $[(S_t, At)]$array ;
    Initialize $R_{t+1}[(S_t, At)]$ array ;
    Initialize prob table with (S_{t+1}, R_{t+1}) as a key ;
end
for *every* $(S_t, At) \rightarrow (S_j, R_j)$ **do**
    Add S_j to $S_{t+1}[(S_t, At)]$;
    Add R_j to $R_{t+1}[(S_t, At)]$;
    Initialize m = length ([$S = S_j$ for S in $S_{t+1}[(S_t, At)]$]) ;
    Update prob$[(S_j, R_j)]$ = m/length$(S_{t+1}[(S_t, At)])$;
end

Algorithm 6: World model updating algorithm.

The bias of this model depends only on the noise in the observations so it can be assumed to be unbiased if sufficiently large K observations are captured $(K \rightarrow \infty)$. This implies that the influence of the model on updating action-values q improves with time while training. In python, this world model class is created with a python dictionary to map every state and action to all the observed next states and next rewards. The class has a method **update_world_model()** (algorithm 6) that is called after each real interaction to store the experience. It also contains a method **hypothetical_experience(N)** (algorithm 7) that implements N hypothetical q updates

once being called. For these hypothetical experiences, the nexts states and next rewards are chosen based on their probabilities of occurrence calculated by equation 4.5.

Result: N simulated q updates
for *N times* **do**
  Randomly select (S_t, At) from world_model ;
  Select random i in range [0, length $S_{t+1}[(S_t, At)]$] ;
  Calculate $S' = S_{t+1}[(S_t, At)][i]$;
  Calculate $R' = R_{t+1}[(S_t, At)][i]$;
  Update $Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha[R' + \gamma max|_a Q(S', a) - Q(S_t, A_t)$;
end

Algorithm 7: Hypothetical experience algorithm.

4.1.3 Dyna-Q algorithm

Based on the components and the algorithms explained in the previous section, the Dyna-Q training algorithm is presented. In the beginning, some considerations regarding state space discretization options and the reward shaping are discussed.

State-space discretization

The fineness of the discretization levels has a great impact on the quality of the Dyna-Q learning algorithm. From one side, having very fine discrete levels makes the convergence rate of the algorithm very slow. This is for two reasons; the first one is the probability of finding states that gives a positive reward significantly decreases. Secondly, it requires a huge number of state transitions to span the whole state space which is needed to guarantee convergence. On the other hand, if the state space is coarsely-discretized, the process tends to be non-Markovian based on the analysis provided in 3.1.1. This is undesirable because the Q-learning algorithm is reliable only for (partial) Markovian stochastic processes. Therefore, finding the most suitable state-space discretization levels is a challenge that depends on properties of the navigation problems such as the map area, the robot size and obstacles sizes and shapes. For the purpose of carrying out reinforcement-learning training experiments, the discretization levels are considered essential design parameters.

Reward shaping

Rewards are given to the agent to evaluate its behaviour based on algorithm 4. The previous state of the agent is compared with its new state and the environment returns a scalar reward based on the change of its state. If the state does not change after applying the action, this is an indication for hitting an obstacle, so a reward -1 is returned. On the other hand, if the new state is the target location, it is an indication that the task has been completed successfully, so a reward 1 is sent to the agent. Otherwise, the reward can be a function of the previous and current states. For example, if the Euclidean distance between the new and current state is significantly small which indicates slipping, a negative reward can be sent to the agent. Besides, the idea of assigning a potential to its discrete cell and using the change in potential to give rewards to the agent has been discussed in section 3.1.3. The approach assigns a potential value $\phi(s)$ to each cell in the navigation grid. The potential is determined based on the distance between the cell and the goal and the distance between the cell and known obstacles. The transformation applied to the sparse reward will be $R' = R + F$ where R is the sparse reward value and F is calculated by the potential difference:

$$F(s, a, s') = \gamma\phi(s') - \phi(s) \tag{4.6}$$

This reward shaping idea is called *potential-based reward* and it is very attracting option because it incorporates knowledge about the navigation problem inside the reward function design. The only disadvantage is that the potential function for each state must be recalculated

for each state every time the navigation map is updated increasing the computational time of the solution. Therefore, the idea will be compared to the traditional sparse reward shaping and if there are no significant improvements, the sparse reward shaping will be used to avoid the computational time cost of the potential-based reward shaping.

4.1.4 Orientation-based rewards

The potential-based reward shaping function introduced in section 4.1.3 assigns rewards to the robot by analyzing only its discrete position (x, y). As explained in figure 4.3, the reward is -1 around obstacles, 1 at the target. Otherwise, the reward will be calculated using the Manhattan distance between the discrete cell and the target defined as follows:

$$d_{MAN} = |x_{target} - x| + |y_{target} - y| \tag{4.7}$$

Thus, the reward becomes $r = \frac{1}{d_{MAN}^2}$ which is defined to be position-based reward. It is possible to include the orientation in the calculation of the reward shaping function. The criterion will be that orientations which point towards free areas are better than orientations that point towards obstacles. Therefore, it is needed to predict the effect of applying a front action on the current orientation to determine whether it will lead to an obstacle or not (figure 4.3). Accordingly, the orientation-based reward can be calculated by the following equation:

$$r_o = \sum_{o-i}^{o+i} r_p(s'|(s, a = F)) \tag{4.8}$$

The reward r_o at a specific orientation o depends on the sum of position-based rewards r_p of the next state if a forward action is applied at this orientation o and for preceding and following i orientations. As shown in figure 4.3, orientations that point towards obstacles have more negative position rewards sum than orientations that point to free areas. Since the idea depends on the next states if a forward action is applied, it will consequently depend on the model created to predict the next states. Therefore, this idea will be applied only to model-based RL algorithms (the smart planning prioritized Dyna-Q and the extended Dyna-Q algorithms). The standard Q-learning algorithm uses only the position-based reward function.

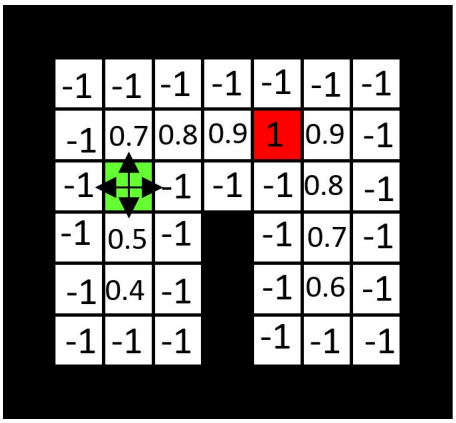


Figure 4.3: The position-based reward is given for each discrete cell. The robot location is the green cell whose position reward is 0.6. However, different robot orientations have different orientation-based rewards which depend on executing a forward action. For example, the up orientation has an orientation reward ($r_{up} = -1-1-1+0.8+0.7 = -1.5$). On the other hand, the down orientation has an orientation reward ($r_{down} = -1-1-1-1+0.5 = -3.5$). Therefore, the robot state whose orientation points up has a larger value.

Result: Approximately optimal q_*
Initialize α, θ_α, γ, ϵ , θ_ϵ and Q_0 (section 3.2) ;
for *N episodes* **do**
  Reset the environment and get $S_t = S_i$ (algorithm 5) ;
  if $\epsilon > \theta_\epsilon$ **then**
    ϵ *= discount factor ;
  end
  if $\alpha > \theta_\alpha$ **then**
    α *= discount factor ;
  end
  for *M trials* **do**
    Select A for S_t based on an $\epsilon - greedy$ policy π (algorithm 3) ;
    Apply a step on the environment using A and Observe S_{t+1} , R_{t+1} (algorithm 4) ;
    Update world model (algorithm 6) ;
    Update $Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha[R_{t+1} + \gamma max|_a Q(S_{t+1}, a) - Q(S_t, A_t)]$;
    Execute N hypothetical experiences (algorithm 7) ;
    if *Not Terminal* **then**
      $S_t = S_{t+1}$;
    else
      break ;
    end
  end
end

Algorithm 8: The Dyna-Q learning algorithm.

Dyna-Q training loop

The reinforcement learning training algorithm is structured based on the previous algorithms as shown in algorithm 8. The algorithm implements N training episodes where N should be adjusted based on the dimension of the state space to guarantee having sufficient training that allows the convergence to q_*. For each episode, the agent applies M actions on the environment based on the $\epsilon - greedy$ policy and updates the action-values $Q(S_t, A_t)$ for each state-action pair. The episode terminates if the target location is reached or after M trails to optimize the computational time of the algorithm. When the episode terminates, the environment is reset and the agent starts a new episode from S_i.

4.2 Learning SSMRs kinematic model

The probabilistic model used in the previously introduced Dyna-Q platforms provides probability distributions over next states and rewards for already visited states ($p(S_{t+1}, R_{t+1})|(S_t, A_t)$). By obtaining a generic model that can predict the robot behaviour at any admissible state, it is possible to execute hypothetical experiences at any location of the map to update the action-values q much faster. For SSMRs, it is possible to learn the kinematic model of the robot model online to improve its predictability [13][33]. This section introduces the algorithm used to learn the kinematic model of the SSMR slip-dependant kinematic model discussed in section 5.1.1. This model maps the wheel velocities of the mobile robot $[\omega_L \ \omega_R]^T$ and the evolution of the robot velocities expressed in a local frame $[v_x \ v_y \ \omega]^T$ (figure 1.4.1) by the following equation:

$$\begin{bmatrix} v_x \\ v_y \\ \omega \end{bmatrix} = \begin{bmatrix} \frac{r(1-\lambda_L)}{2} & \frac{r(1-\lambda_R)}{2} \\ \frac{X_{ICR} r(1-\lambda_L)}{2c} & \frac{-X_{ICR} r(1-\lambda_R)}{2c} \\ \frac{-r(1-\lambda_L)}{2c} & \frac{r(1-\lambda_R)}{2c} \end{bmatrix} \begin{bmatrix} \omega_L \\ \omega_R \end{bmatrix} \tag{4.9}$$

The kinematic parameters that need to be learned are the longitudinal slip coefficients at each wheel λ_l , λ_R, the effective wheel radius r and the instantaneous centre of rotation X_{ICR}. The longitudinal slip coefficients (λ_l , λ_R) and the effective wheel radius r relates the output vector $[v_x\ \omega]^T$ to the input vector $[\omega_L\ \omega_R]^T$ at each state. Similarly, the instantaneous centre of rotation X_{ICR} relates the output v_y to the input ω. The analysis that has been introduced in section 3.4 recommends dividing this multi-output regression problem between $[v_x\ v_y\ \omega]^T$ and $[\omega_L\ \omega_R]^T$ to the following independent single-target regression problems :

- $h_1 : \Omega\omega_L \times \Omega\omega_R \rightarrow \Omega v_x, \mathbf{x} = (\omega_L, \omega_R) \longmapsto \mathbf{y} = v_x$
- $h_2 : \Omega\omega_L \times \Omega\omega_R \rightarrow \Omega\omega, \mathbf{x} = (\omega_L, \omega_R) \longmapsto \mathbf{y} = \omega$
- $h_3 : \Omega\dot{\theta} \rightarrow \Omega v_y, \mathbf{x} = \dot{\theta} \longmapsto \mathbf{y} = v_y$

To solve these three problems, data representing the inputs and the outputs are collected from the environment at each discrete $x-y$ state. Thus, the collected data sets have the following form:

$$\mathbf{D}_{A_i} = \{(x^{(1)}, y^{(1)}), .., (x^{(N)}, y^{(N)})\} \tag{4.10}$$

Where A_i represents the area where the data $\mathbf{D}$ have been collected (figure 4.4), $x^{(l)} = (\omega_L^{(l)}, \omega_R^{(l)}, \dot{\theta})$ represents the input feature instance l and $y^{(l)} = (v_x, \omega, v_y)$ represents the output feature instance l. Accordingly, three regression models $[h_1\ h_2\ h_3]_{A_i}$ should be learned for each A_i discrete grid. The models are created to fit the data using a suitable regression method. The Bayesian ridge regression is capable of build strong regression models that are robust to noisy data sets. Characteristics of the Bayesian regression method have been presented in section 2.6. It is possible to create Bayesian ridge regression model in a Python-based environment using sci-kit library [22].
A look-up table with an entry key of the discrete grid coordinate A_i is created and the data captured at each grid are stored accordingly. After each step, the data captured are divided into two subsets. One set is used to train the model and the other is used to validate the model through calculating the coefficient of determination R^2 of the prediction. The coefficient of determination is defined by:

$$R^2 = 1 - \frac{S_{res}}{S_{tot}} \tag{4.11}$$

Where S_{res} is the residual sum of squares of each prediction error e_i):

$$S_{res} = \sum_i (y_i - t_i)^2 = \sum_i e_i^2 \tag{4.12}$$

And S_{tot} is called the total sum of squares and it is proportional to variance in the data and is defined by:

$$S_{tot} = \sum_i (y_i - \bar{y}), \bar{y} = \frac{1}{n}\sum_{i=1}^{n} y_i \tag{4.13}$$

The coefficient of determination R^2 can be used to assess the model predictability [22]. The best possible score is 1 and a model that always predicts the expected value of y regardless of the input features would get an R^2 worst score of 0.

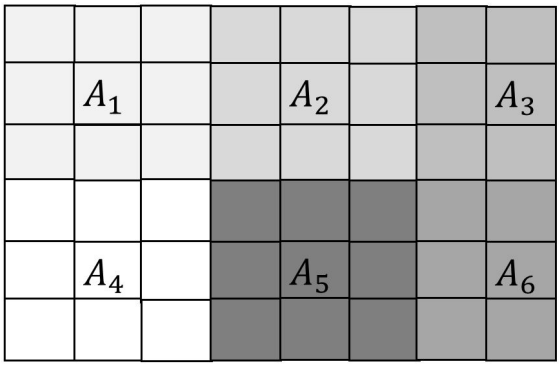


Figure 4.4: Dividing the map into a set of areas A_i different from the discrete levels of the state space.

The power of these models is that they can be created with respect to grids A_i with coarser discretization levels than the discrete levels of the state-space. For example, as shown in (figure 4.4), the navigation map can be divided into a number of areas A_i and models created at each area can be used to predict the robot motion in states enclosed by it. Therefore, a look-up table is created to map each state S_t to the area that includes it A_i.

Result: N simulated q updates
Initialize δ_{R^2} ;
for *N times* **do**
  Randomly select (S_t, At) ;
  if (S_t, At) *in world_model* **then**
    Select random i in range [0, length $S_{t+1}[(S_t, At)]$] ;
    Calculate $S' = S_{t+1}[(S_t, At)][i]$;
    Calculate $R' = R_{t+1}[(S_t, At)][i]$;
    Update $Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha[R' + \gamma max|_a Q(S', a) - Q(S_t, A_t)$;
  else
    Look-Up A_i that includes S_t ;
    Calculate $R^2_{A_i}$;
    if $R^2_{A_i} > \delta_{R^2}$ **then**
      Predict S_{t+1} using $[h_1 h_2 h_3]_{A_i}$;
      Predict R_{t+1} using (S_t, S_{t+1}) (algorithm 4) ;
      Update $Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha[R_{t+1} + \gamma max|_a Q(S_{t+1}, a) - Q(S_t, A_t)$;
    end
  end
end

Algorithm 9: Hypothetical experience algorithm.

The created robot motion model can be integrated into the architecture of the learning system as shown in figure 4.1 extending the world probabilistic model. Therefore, the algorithm 7 is modified to make predictions using the robot model for state-action pairs that have not been visited before (algorithm 9). If the state-action pair (S_t, A_t) has not been visited and is not stored in the world model loop-up table, it is possible to use the robot motion model to predict (S_{t+1}, R_{t+1}) for this pair. The algorithm finds which area A_i includes S_t and makes the (S_{t+1}, R_{t+1}) prediction only if the coefficient of determination of the model is larger than a defined threshold δ_{R^2}.

4.3 Smart planning

While simulating hypothetical experiences, state-action pair are selected randomly. It is possible to prioritize this selection for state-action pairs whose action-values change more rapidly

in the real experience updating. This idea is called prioritized sweeping (PS) (section 2.5) and has been argued by [4] to optimize the usage of computational resources for more interesting state-action pairs. Another similar idea that has been proved to accelerate the convergence to a (sub-optimal) path is to keep track of the shortest trajectory taken to reach the goal and prioritize the hypothetical experiences around it. This makes the reward propagates from the target state to all the states around the trajectory which makes the action-values of state-action pairs over this trajectory converge much faster.

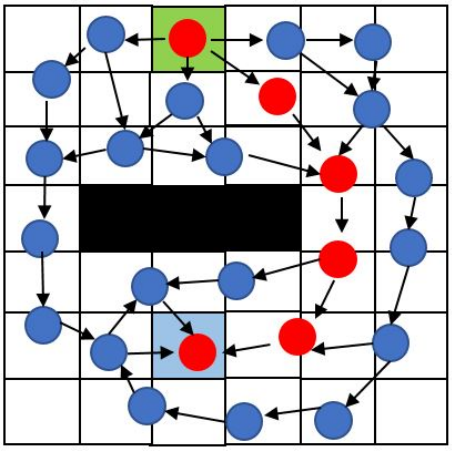


Figure 4.5: The red trajectory represents the shortest discovered path between the green and the blue cells. By simulating the transitions along the red trajectory after the reward is received, the effect of the reward propagates along the trajectory and to other state-action pairs that lead to it.

The idea is illustrated by figure 4.5. The algorithm stores all the paths taken to reach the goal during the exploration phase. Once a shorter path is experienced, a hypothetical experience is simulated along all the state-action pairs among it starting from the last state that received the positive reward. This makes the effect of the positive reward propagate all over the state-action pairs on the trajectory. In the following episodes, the agent is more likely to take optimal greedy actions to keep itself around that trajectory whose state-action pairs have the highest values. After that, when the hypothetical updates are done randomly, other state-action pairs values that lead to this trajectory are also updated. Therefore, the agent will have a tendency to take greedy actions to move towards the shortest discovered path. This idea is significantly powerful in improving the learning performance as will be discussed in chapter 6.

5 Experimental Design

This chapter shows the simulation and the real-time platforms designed to validate the proposed algorithms presented in chapter 4. It is required to determine whether adopting model-based reinforcement-learning approaches improves the learning performance of Skid-Steering-Mobile-Robots in unknown environments. In the literature, reinforcement learning algorithms are compared based on the following performance measures:

- **The cumulative reward:** It is defined by the sum of scalar rewards harvested over a training episode. It provides a quantitative measure to the quality of the robot trajectory all over the episode. If the movement trajectory avoids obstacles and reaches the goal in minimum steps, the effect of the positive reward received when reaching the target dominates the rewards sum. In contrast, if the major portions of the trajectory surround obstacles or slippery areas, the negative rewards received will dominate the sum. The ideal behaviour of the cumulative rewards during the training is explained by figure 5.1.a.

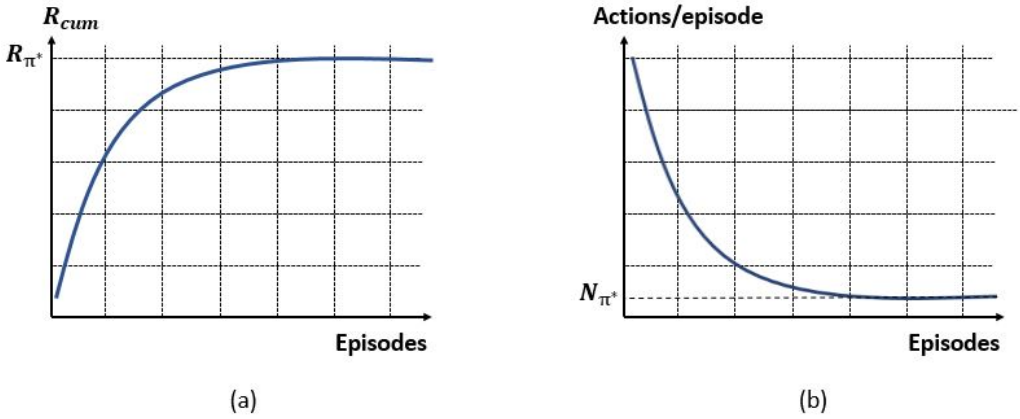


Figure 5.1: (a) The ideal cumulative rewards behaviour. They increase during the training and saturate when the optimal policy is reached. (b) The ideal number of actions/episode behaviour. They decrease during the training and saturate when the optimal policy is reached.

- **The number of actions per episode:** The length of a trajectory is characterized by the number of actions executed between the initial and the target state. In initial training episodes, the exploration phase dominates. Therefore, the number of actions taken to reach the goal is large. Once the action-values start to converge, the exploitation phase follows the learned optimal policy and the number of steps taken to reach the goal decreases. The ideal behaviour of the episode length during the training is explained by figure 5.1.b.

- **The convergence speed:** It is characterized by the number of episodes required to converge to an approximate optimal policy. Thus, it is usually desirable to minimize such a number to achieve high efficiency. The convergence is detected if the minimum change for each updated action-value q is less than a threshold θ_c.

- **The success rate:** The training episode terminates if the goal is reached or if the number of actions exceeds a predefined trials number ($trials_{max}$). In the long-term behaviour, each episode always terminates when the goal is reached since the number of actions to reach the goal by the optimal policy is less than $trials_{max}$. However, it is possible to evaluate algorithms behaviours in the exploitation phase by comparing the number of

successful episodes that have terminated by reaching the goal (N_s) to the total number of training episodes after the convergence (N_{total}). Accordingly, the term *success rate* **SR** is defined for every algorithm:

$$SR_{N_{total}} = \frac{N_s}{N_{total}} \tag{5.1}$$

The designed experiments are aimed to provide an appropriate platform to compare between designed RL algorithms according to the previously mentioned performance measures. The first section of this chapter shows how a simulation platform has been structured to validate the designed algorithms in a Python-based environment. Then, it shows the properties of the experiments implemented using simulations whose results will be explained in chapter 6. The second section shows how a real-time experimental set-up has been designed to carry out real-time RL experiments. It explains the requirements needed for the set-up motivating its hardware and software structures. Then, it explained the properties of the RL experiments to be implemented in real-time whose results are explained in chapter 6.

5.1 Experiments in simulation

It is required to provide a simulation model for the motion of SSMRs that can be used to validate the designed reinforcement-learning algorithms before implementing the real-time experiments. The reason for that is that the time needed to validate reinforcement-learning algorithms in reality is undesirably-long [12]. Therefore, by having a sufficiently-abstract model that can simulate the robot motion, it is possible to design and test reinforcement learning algorithms much faster.

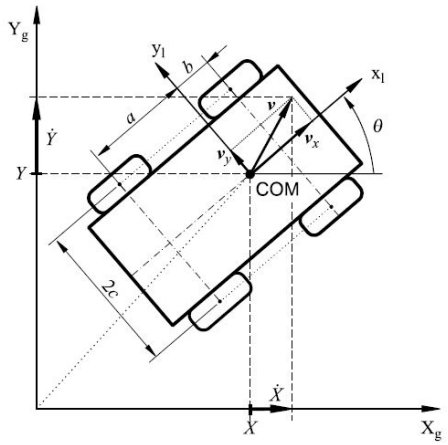


Figure 5.2: SSMR free body diagram [13].

5.1.1 Slip-dependant kinematic model

In the literature, the kinematic and the dynamic models for four-wheels SSMRs are provided for the purpose of trajectory tracking control. To use such models, the following assumptions are considered [18]:

1. The robot centre of mass coincides with its body geometric center.
2. The contact between the ground and the wheels is reduced to a point contact.
3. The rolling resistance force of the wheels are negligible.
4. Each side's two wheels has the same rotating speed.

5. The normal forces applied to the robot from the ground are equally distributed among the four wheels.

6. The robot motion is on a flat surfaces and the four wheels are always in contact with the ground surface.

Under these assumptions, the relation between the wheels velocity vector $[\omega_L \omega_R]^T$ and the generalized velocity vector $[\dot{X}\dot{Y}\dot{\theta}]$ (figure 5.2) can be derived as shown in section 2.5 to be [13]:

$$\begin{bmatrix} \dot{X} \\ \dot{Y} \\ \dot{\theta} \end{bmatrix} = \begin{bmatrix} cos(\theta) & -sin(\theta) & 0 \\ sin(\theta) & cos(\theta) & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \frac{r}{2} & \frac{r}{2} \\ \frac{X_{ICR}r}{2c} & \frac{-X_{ICR}r}{2c} \\ \frac{-r}{2c} & \frac{r}{2c} \end{bmatrix} \begin{bmatrix} \omega_L \\ \omega_R \end{bmatrix} \tag{5.2}$$

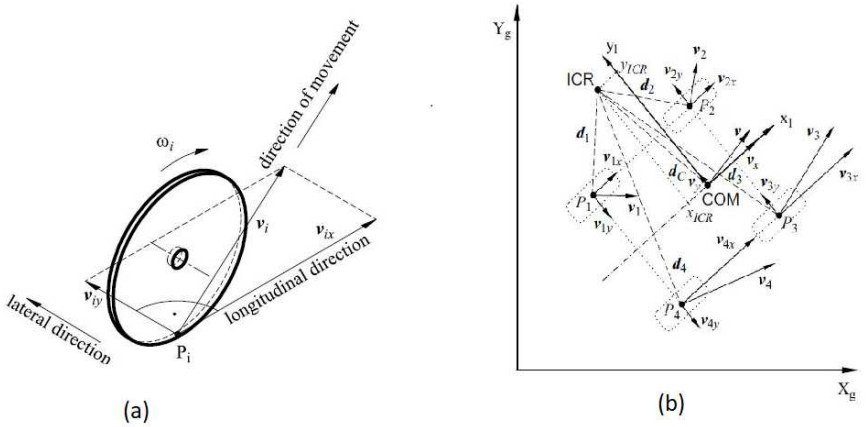


Figure 5.3: (a) Velocities components at one wheel. (b) Resolved velocities components of the robot body [13].

It is difficult to depend only on this model to simulate an accurate motion for SSMRs. The drawback of this model lies in the pure rolling assumption in the longitudinal x-direction (figure 5.3.a). In practice, sliding in longitudinal and lateral directions always occurs for several reasons. Longitudinal sliding can occur while moving on slippery surfaces with low friction coefficients [2]. In addition, sliding in the lateral direction is an inherent property of the SSMRs motion. It occurs when the robot changes orientation where reactive friction forces are usually more dominant than inertial forces [18] and thus they influence the robot motion significantly. This implies that the dynamic properties of SSMRs play an important role in determining the their motions. These sliding properties have a non-linear behaviour that depends on various factors such as shapes and materials of the wheels tires and the ground [2]. Therefore, developing a dynamic model for SSMRs has been discussed intensively in the literature. However, these models need a lot of computational effort to calculate complex dynamics since it requires a number of integral operations [2][33]. Thus, some trials have been discussed in the literature to include slip coefficients in the kinematic model of SSMRs [33][18]. In the approach presented in [18], the longitudinal wheel slip λ_i at each wheel is defined by the ratio between the difference of the wheel velocity and its center velocity, and the wheel velocity:

$$\lambda_i = \frac{r\omega_i - v_{ix}}{r\omega_i} = -\frac{\Delta v_{ix}}{r\omega_i}, i = 1, ..., 4 \tag{5.3}$$

where r is the effective wheel radius, ω_i is the angular velocity of wheel i and $\Delta v_{ix} = v_{ix} - r\omega_i$ is defined to be the longitudinal slip velocity of the i^{th} wheel/ground contact point P_i (figure 5.3). Under assumption 4, it is valid to assume $\lambda_1 = \lambda_2 = \lambda_L$ and $\lambda_3 = \lambda_4 = \lambda_R$ as $\omega_1 = \omega_2 = \omega_L$

and $\omega_3 = \omega_4 = \omega_R$. According to this definition, the longitudinal slip can be included into the equations of the left and the right longitudinal velocities to be $v_{Lx} = r(1-\lambda_L)\omega_L$ and $v_{Rx} = r(1-\lambda_R)\omega_R$. By substituting these quantities into equation 5.2, the generalized velocity vector $[\dot{X}\dot{Y}\dot{\theta}]$ can be expressed in the matrix form to be as follows:

$$\begin{bmatrix} \dot{X} \\ \dot{Y} \\ \dot{\theta} \end{bmatrix} = \begin{bmatrix} cos(\theta) & -sin(\theta) & 0 \\ sin(\theta) & cos(\theta) & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \frac{r(1-\lambda_L)}{2} & \frac{r(1-\lambda_R)}{2} \\ \frac{X_{ICR}r(1-\lambda_L)}{2c} & \frac{-X_{ICR}r(1-\lambda_R)}{2c} \\ \frac{-r(1-\lambda_L)}{2c} & \frac{r(1-\lambda_R)}{2c} \end{bmatrix} \begin{bmatrix} \omega_L \\ \omega_R \end{bmatrix} \tag{5.4}$$

This slip-dependant kinematic model represented by equation 5.4 will be used to simulate the motion of the SSMR. It is assumed to provide a sufficient abstraction to the real slippage behaviour because of the following two reasons:

- The action space (move-front, move-back ,turn-left and turn-right) is discretized and the mobile robot will be forced to stop after executing each action to limit the growth of dynamical effects that occur during changing the orientation along a continuous movement trajectory.
- The slippage effects will be simulated by assigning λ_L, λ_R and X_{ICR} independently for each discrete x and y state according to the following values boundaries: $\lambda_L \in [0,1], \lambda_R \in [0,1]$ [18] and $x_{ICR} \in [-a,b]$ (figure 5.2) [2]. By choosing high discretization levels for the x and y states, the kinematic model will approximate the sliding motion of the SSMR in the real environment.

5.1.2 Python implementation

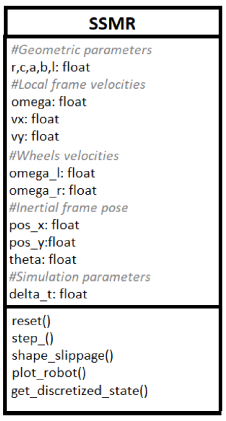


Figure 5.4: Python class diagram for the SSMR.

The SSMR class shown in figure 5.4 has been built to simulate the robot behaviour given by equation 5.4. Figure 5.4 shows the following methods that can be used by the user to simulate the robot behavior:

- **reset():** this method can be used to reset the robot location to an initial position in the navigation map at the beginning of each training episode.
- **step_(action):** this method can be used to apply an action on the environment. The action variable can take the following integer values: 0 for move-front, 1 for move-back, 2 for turn-left and 3 for turn-right.
- **shape_slippage():** this method can be used to determine slip coefficients at each $x-y$ robot discrete location.

- **plot_robot():** this method can be used to plot the simulated robot location after executing actions.
- **get_discretized_state():** this method returns the discrete x, y and θ pose of the robot.

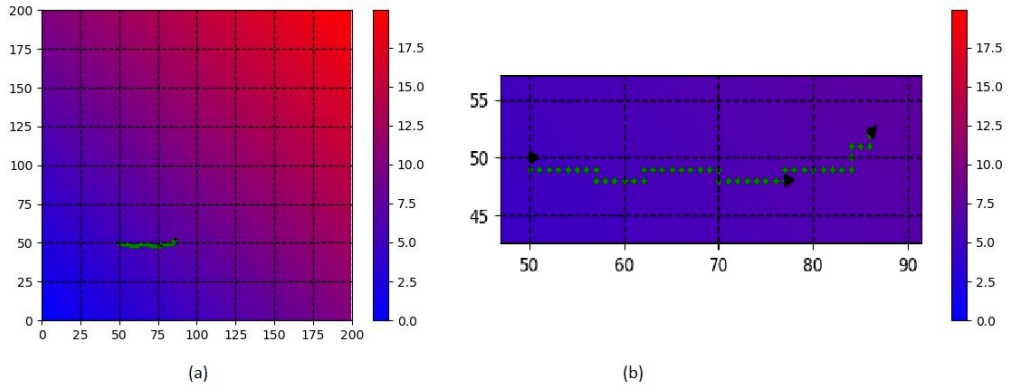


Figure 5.5: (a) The slip coefficients distribution over a 200x200 grid map. (b) The robot moves faster and almost in a straight line in the first 100 actions because the slip is small. After that, when the slip increases the robot diverges from the straight line and moves slower.

It is possible to use this class to simulate the behaviour of the robot given a map with a desired slip distribution. In figure 5.5.a, the slip is designed to increase with the coordinates of the x-y discrete locations. The robot applies 200 move-forward actions. As shown in figure 5.5.b, in the first 100 moves where the slip is small, the robot moves faster and almost in a straight line. After that, when the slip increases the robot diverges from the straight line and moves slower.

5.1.3 Experiments properties

To validate the designed reinforcement learning algorithms in simulation, the map shown in figure 5.6 is used. The map is created using gym-pathfinding library developed by [31]. Other map configurations (figure 4.2) will be used to validate whether the designed algorithms are map-invariant. The motion of the robot in the map simulates the kinematic behaviour of a SSMR as explained in section 5.1.1.

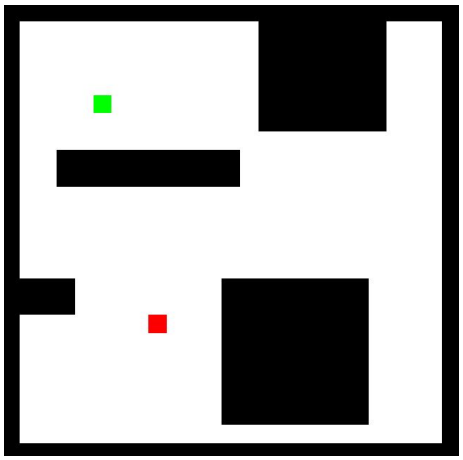


Figure 5.6: The task is to learn an optimal trajectory to navigate from the green to the red cell.

The simulated experiments share the parameters properties defined by table 5.1. The selection of the values of these parameters is guided by the selection criteria detailed in section 3.2. The

Parameter	Value	Behaviour
Exploration factor (ϵ)	70% → 1%	Decreases with time
Discount factor (γ)	0.9	constant
Learning rate (α)	1 → 0.9	Decreases with time
Convergence threshold (θ_c)	0.001	constant
Maximum trials per episode (M_{max})	500 → len(shortest path)	Re-assignment if shorter path found
Number of episodes (N)	2000	constant
Discretization level along x-Axis (n)	25	constant
Discretization level along y-Axis (m)	25	constant
Discretization level around θ (o)	20	constant

Table 5.1: Experiments parameters.

exploration factor ϵ is initialized with a high value of 70% to encourage applying exploratory actions, then it is discounted after each episode. In order to overcome possible local minima the exploration is not fully terminated, but the exploration factor is reduced to 1% Accordingly, the agent starts to adopt a greedy policy over the action-values. The discount factor γ is set to a huge value of 0.9 so that the learn depends on the optimal value of the next state $V(s') = max|_a Q(s', a')$ to update the action-value $Q(s, a)$. The learning rate α is set to a huge of 1 to eliminate the bias to the initial values Q_0 by providing higher weight to the received rewards over the pre-assumed initial values Q_0. Because the environment is assumed to be static, it is not expected to receive a time-variant reward signal $r(t)$ from the same next state. Therefore, any change in the given reward from the same next state is considered to be faulty due to skidding and its effect on the action-value needed to be reduced. Thus, after convergence, the learning rate α is reduced to a value of 0.9 to reduce the effect of these noisy rewards. As detailed in section 4.1.3, the discretization levels are designed by trial-and-error based on the learning experiment. The selected levels $25 \times 25 \times 20$ have proven to provide a satisfactory performance.

5.2 Experiments in real-time

5.2.1 Setup requirements

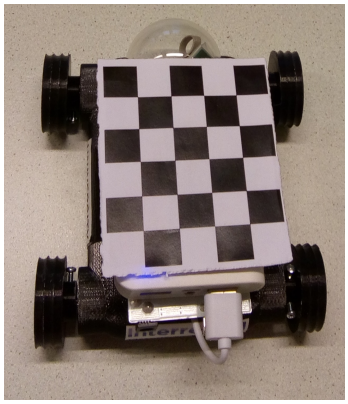


Figure 5.7: Four-wheel mobile robot used for the experiments.

A four-wheel Skid-Steering-Mobile robot (figure 5.7) has been built to validate the designed algorithms. It has been designed with the following properties to enable implementing training experiments for sufficiently-long time to guarantee convergence:

- It has a Bluetooth module to connect to a master PC that runs the reinforcement learning algorithm so that the robot can receive actions to be executed and send data representing the wheels angles.
- It uses a ceiling camera (figure 5.8) as a sensor that can detect a visual pattern on the top of the robot (figure 5.7) to fully observe the robot states $[x\ y\ \theta]^T$.
- It has a high capacity power supply to allow carrying out experiments for long time without any human supervising.
- The two left motors are connected in parallel so as the two right motors to ensure identical motions assumed by the model.
- Optical encoders are connected to each motor to feedback rotating wheels angles.

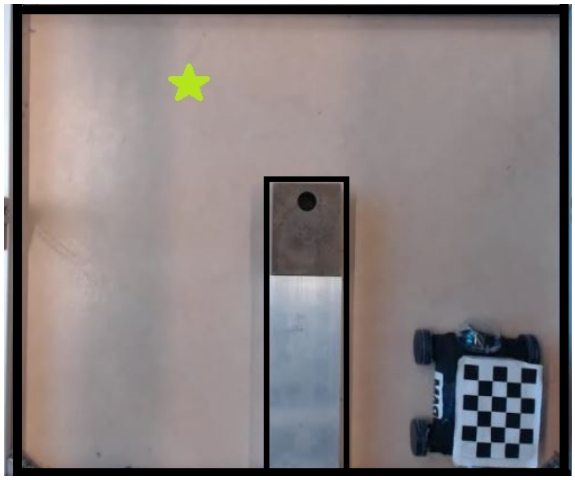


Figure 5.8: A photo of the environment taken by a ceiling camera.

5.2.2 Hardware realization

The hardware has been realized using the components shown in table 5.2. The schematic design of the interacting hardware is shown in figure 5.9.

Component function	Used component	number
Microcontroller	Arduino DUE	1
DC motor with integrated optical encoders	EN-2619-SR-IE2-16	4
Bluetooth module	HC-05	1
H-Bridge driver	L298N	1
Camera	Logitech 1080Hd	1
Power bank	VOLTCRAFT 7800 mAh	1

Table 5.2: Components used to build the setup.

The reinforcement learning algorithm should be run by a master PC that sends encoded data to the Bluetooth module HC-05 which represents the actions to be executed by the robot. The micro-controller (Arduino DUE) decodes these data to determine which action (move-front, move-back, turn-left or turn-right) should be applied. Accordingly, the micro-controller sends PWM data representing desired left and right motor speeds to the H-bridge driver (L298N). The motor driver applies the correct voltage values (V_R and V_L) to the motors to execute the required action.

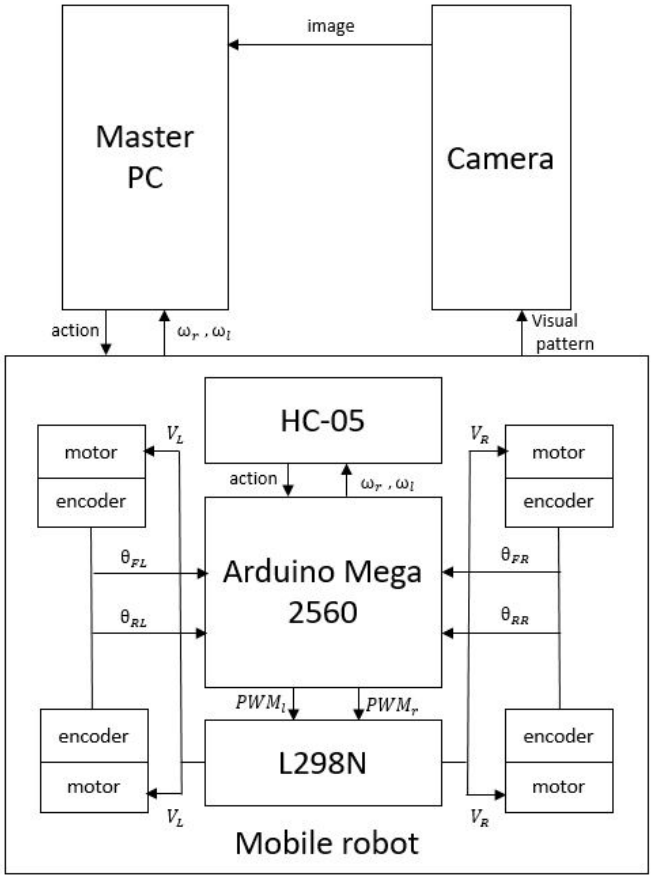


Figure 5.9: A schematic diagram of interacting hardware components.

After executing an action, the encoders send the new rotating angle of each wheel $(\theta_{FR},\theta_{FL},\theta_{RR},\theta_{RL})$ to the micro-controller. The micro-controller calculates accordingly the wheel speeds (ω_l,ω_r) which are sent to the PC through the Bluetooth module (HC-05). In addition, the camera tracks the visual pattern installed on the top of the robot and sends an image about the changing environments. These images are analyzed to extract the new mobile robot state (x,y,θ). The design of the software that controls this hardware set-up is presented together with relevant algorithms in section 5.2.3.

5.2.3 Software realization

This section shows the design of the software used to run and communicate between the hardware components of the setup (figure 5.9). The reinforcement learning algorithm is run in a python-based development environment. In order to communicate with the hardware of the camera and the car, a class real_env is created. This class is composed of two main components:

- **The communication server:** it is built using Python *serial* library. It encodes the selected actions to be executed into letters: F,B,L,R for move-forward, move-backward, turn-left and turn right respectively. These letters are sent to the car through the Bluetooth module that is connected to the micro-controller. It encodes the received data to the following functions: **moveForward(), moveBackward, moveLeft() and moveRight()** to be executed by the car. In order to synchronize the motion of the robot with the sent command data, the algorithm waits for a signal which indicates that the action is fully executed before sending a new command to the robot (algorithm 10). On the other hand,

the micro-controller sends the encoders reading to be processed by the reinforcement-learning algorithm.

- **The states observer:** Once the action is fully executed, the algorithm starts processing the image captured by the camera that contains the new pose of the pattern that is attached to the robot. The algorithm uses the *open CV* library to track the position and the orientation of a frame attached to the visual pattern. Accordingly, the planar $[x\ y]$ location of the mobile robot and its orientation around the z-Axis (the red frame) are extracted and discretized with the desired discretization levels.

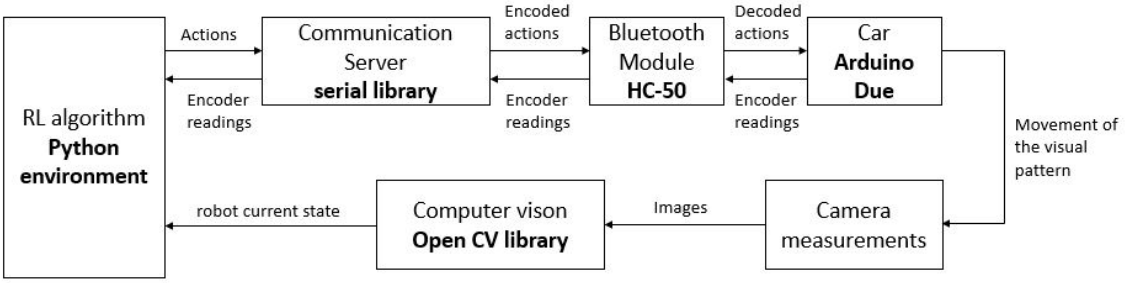


Figure 5.10: The software architecture of the setup interacting components.

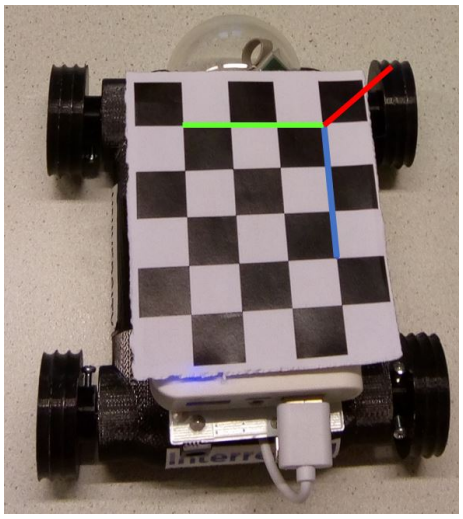


Figure 5.11: The tracked frame in the visual pattern image.

Before starting the real experiment the map is calibrated to determine the limits of the measurements along the x and the y axes ($x_{min}, x_{max}, y_{min}$ and y_{max}). Accordingly, the discrete

level of each grid can be calculated using algorithm 11. These calculated states are fed back to the learning algorithm to close the algorithm loop.

Result: Synchronized communication between Python and Arduino Due
for *every action command* **do**
    Encode the action to the equivalent letter. ;
    Send data to the Bluetooth module. ;
    while *not isReady()* **do**
        Pass ;
    end
end
Function isReady() {
Decode inputData received from the Bluetooth module. ;
if *inputData* == *'ready'* **then**
    return True ;
else
    return Fasle ;
end
}

Algorithm 10: Algorithm used to synchronize commands sent from the RL learner to the mobile robot.

Result: $x\_discrete$, $y\_discrete$, $\theta\_discrete$
Initialize $x_{min}, x_{max}, y_{min}$ and y_{max} ;
Initialize n, m and o discretization levels ;
while *True* **do**
    Capture the image ;
    Detect chessboard pattern ;
    if *pattern found* **then**
        Solve for exact x, y and θ ;
        $x\_discrete = floor(\frac{x \times n}{x_{max} - x_{min}})$;
        $y\_discrete = floor(\frac{y \times m}{y_{max} - y_{min}})$;
        $\theta\_discrete = floor(\frac{tan^{-}1(sin(\theta)/cos(\theta)) \times o}{2\pi})$;
        Return $x\_discrete$, $y\_discrete$, $\theta\_discrete$;
    end
end

Algorithm 11: Algorithm used to calculate discrete $[x, y, \theta]$ states.

5.2.4 Experiments properties

The designed algorithms will be validated in real-time through the navigation task shown in figure 5.12. This section shows the results of adopting the prioritized Dyna-Q smart planning algorithm. The results of using the standard Q-learning algorithm and extending the Dyna-Q with the robot kinematic model are also discussed.

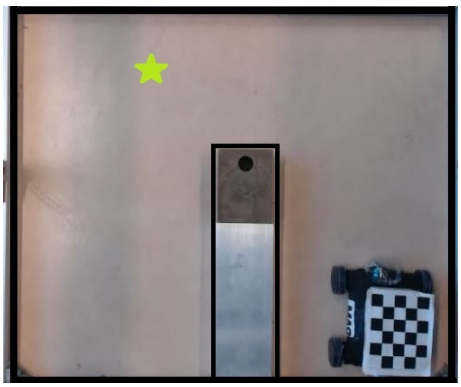


Figure 5.12: The map used to make the real-time experiments. The star marks the target location.

Reinforcement-learning algorithm in real-time takes a significant long time. Therefore, it is not feasible to run the experiment for too many episodes, even though, a map with small dimensions (62cm x 50.5cm) is used (figure 5.12). The number of episodes to run the experiments has been determined by trying the fastest algorithm and checking the time needed for its convergence pattern to stabilize. It has been proved that 190 episodes are sufficient to converge to a sub-optimal behaviour for the prioritized smart-planning algorithm. Therefore, other algorithms are experimented for 190 episodes to compare them. Making real-time experiments has the problem of non-stationary rewards. Non-stationary rewards are received by the robot due to the skidding. When the robot applies the same action in the same state, it is not possible to move to a unique next-state, but due to the skidding properties, it is probable that the robot will visit a set of next-states. Thus, the reward given by the environment is non-stationary and depends on which next-state the robot will move to. In order to avoid the effect of these noisy rewards, the minimum value of the learning rate α factor is kept smaller than simulations at 0.8. The minimum exploration factor ϵ is also kept at a large value of 20%. This is particularly useful in case the robot is trapped in local minima. Finally, the action space has been reduced to only "move-forward", "turn-right" and "turn-left" movements. The reason for this is that including the move-backward action increases the probability of getting stuck in local minima during the training. The parameters used for real-time experiments are used in table 5.3.

Parameter	**Value**	**Behaviour**
Exploration factor (ϵ)	40% → 20%	decreases with time
Discount factor (γ)	0.9	constant
Learning rate (α)	1 → 0.8	decreases with time
Convergence threshold (θ_c)	0.001	constant
Max trials per episode (M_{max})	300 → avg(episodes lengths)	decreases with time
Number of episodes (N)	190	constant
Discretization level along x-Axis (n)	20	constant
Discretization level along y-Axis (m)	20	constant
Discretization level around θ (o)	20	constant

Table 5.3: Real-time experiments parameters.

6 Results

In this chapter, the reinforcement learning algorithms presented in chapter 4 are validated using a Python-based environment. The algorithms are tested using the simulation model explained in section 5.1.1. Validating the designed algorithms using simulations is desirable before testing them on the real set-up since the real-time experiments are significantly long. Firstly, the standard Q-learning algorithm is parameterized and simulated. Then, a Dyna-Q algorithm which builds a probabilistic model of the system is also simulated. After that, the algorithm is extended with the smart planning technique which was presented in section 4.3 where the shortest obtained trajectory is used to steer the simulated hypothetical experiences. Similar algorithms are tried on the real robot in real-time training experiments. Eventually, the kinematic model of the robot is learned online to extend the probabilistic model of the dyna-Q algorithm and its influence on the learning behaviour is analyzed. The designed algorithms are compared based on the performance measures explained in the beginning of chapter 5.

6.1 Simulation results

6.1.1 Standard Q-learning algorithm

The Q-learning algorithm has been simulated for 4000 episodes since 2000 episodes were not sufficient to reach the convergence. Figure 6.1c shows the explored map with the discovered optimal trajectory. Each cell of the map is colored with a color representing its optimal value according to the color map shown in the figure 6.1c. The value of each discrete cell is determined based on the maximum value of each state-action pair it contains as described by the following equation:

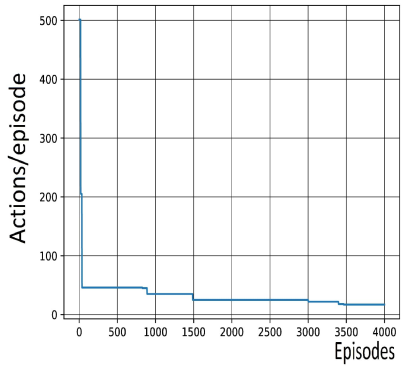
$$V(s) = argmax|_a[Q(s,a) \forall a] \tag{6.1}$$

Accordingly, the optimal discovered trajectory navigates along the discrete cells whose optimal values increase (increasing in the color gradient) from the initial state to the target state. Figure 6.1a shows that the training reaches convergence after 3465 episodes.

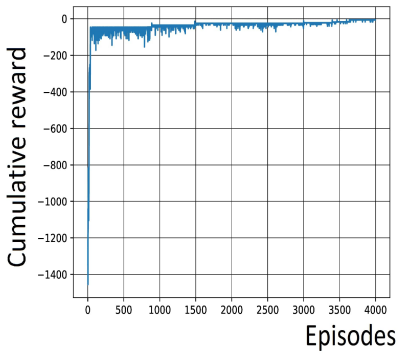
Measure	Value
The cumulative long-term reward	-7
Number of action at optimal policy	17 actions
Convergence speed	After 3465 episodes
Success rate	0.23275

Table 6.1: Training results for applying the standard Q-learning algorithm.

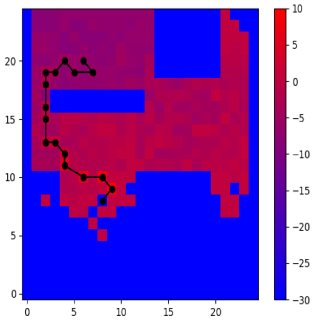
Once the optimal path is discovered, the agent adopts a greedy policy towards it. Therefore, after converging, the successful episodes increase significantly and most of the episodes have the same number of actions, 17, which is the length of the optimal trajectory. Figure 6.1b shows the cumulative rewards all over the training. After the convergence, there are some oscillations in the cumulative reward graphs since the agent deviates from the optimal path in some episodes due to the stochasticity of the environment.
The results of the standard Q-learning algorithm shows that the convergence is significantly slow. The reason for that is the value for each state-action pair q which is only updated if the pair is visited in a real interaction. The state-space has 12500 unique states that give 50000 state-action pairs. The average length of the training episode for the current training experiment is 25 trials. Therefore, at least 2000 episode is required to guarantee that every pair has been visited assuming that the exploration is completely unbiased ($\epsilon = 100\%$). However, in the case of the $\epsilon - greedy$ policy adopted by the Q-learning algorithm, it takes more episodes than 2000 to sufficiently visit state-action pairs and achieve the convergence.



(a) The number of executed actions for each training episode for the standard Q-learning algorithm.



(b) The cumulative rewards for each training episode for the standard Q-learning algorithm.

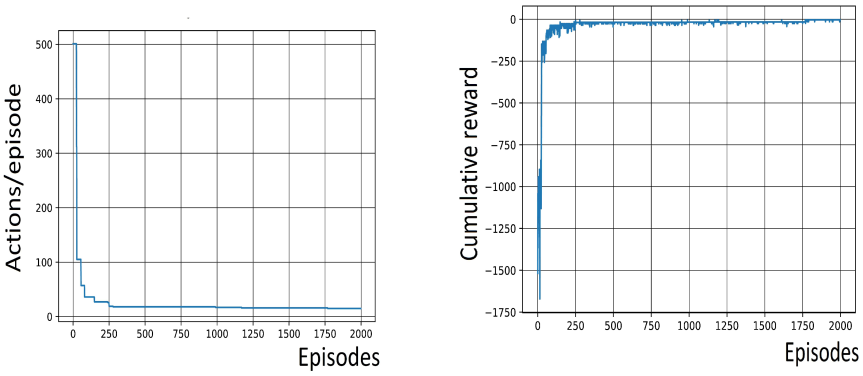


(c) The discovered map and the sub-optimal discovered trajectory. Each cell in the map has a color representing its value according the right-hand-side color map.

Figure 6.1: Training results for applying the standard Q-learning algorithm.

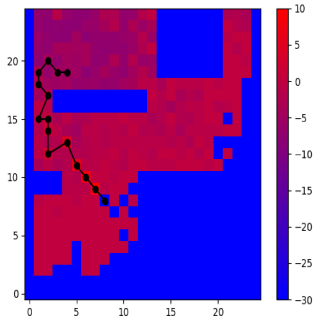
6.1.2 Dyna-Q algorithm

The Dyna-Q algorithm is simulated according to algorithm 3.7 by executing 100 hypothetical experiences after each real interaction experience. As expected, The Dyna-Q algorithms converges faster than the standard Q-learning algorithm in 1768 episodes. Figure 6.11c shows the explored map with the discovered optimal trajectory. The discovered trajectory is similar to the one discovered by the standard Q-learning algorithm (15 actions). Equivalently, the optimal path is the trajectory resulted from following a greedy-policy over the action-values q. Therefore, it follows discrete cells with increasing optimal values. After the convergence, the dyna-Q algorithm has significantly higher number of successful episodes than the standard Q-learning algorithm. Since the Dyna-Q algorithm selects states of the hypothetical experiences randomly, simulated training treats all the state-action pairs equally. Therefore, some computational resources are wasted since the algorithm does not keep track of the newly-updated state-action pairs and does not propagate their new values to their connecting pairs. The following section shows how our suggested algorithm, the prioritized smart planning, can improve the performance of the traditional Dyna-Q.



(a) The number of executed actions for each training episode for the Dyna Q-learning algorithm.

(b) The cumulative rewards for each training episode for the Dyna Q-learning algorithm.



(c) The discovered map and the sub-optimal discovered trajectory. Each cell in the map has a color representing its value according the right-hand-side color map.

Figure 6.2: Training results for applying the Dyna Q-learning algorithm.

Measure	Value
The cumulative long-term reward	-7
Number of action at optimal policy	15 actions
Convergence speed	After 1768 episodes
Success rate	0.936

Table 6.2: Training results for applying the Dyna-Q learning algorithm.

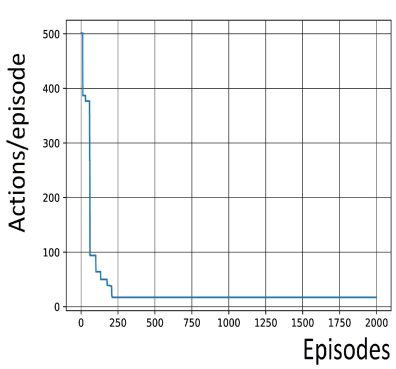
6.1.3 Smart planning algorithm

Smart planning prioritizes the simulated hypothetical experiences of the Dyna-Q by giving higher priorities to state-action pairs lying on the shortest discovered trajectory. Once a shorter path is experienced, a hypothetical experience is simulated along all the state-action pairs among this path starting from the last state that received the goal reward. The simulation transfers the effect of the positive reward given by the target state to each state-action pair on the trajectory. This has been proven to be powerful in accelerating the convergence of the Dyna-Q algorithm. The algorithm converges significantly fast after 211 episodes (figure 6.3a). The optimal discovered trajectory, as shown in figure 6.3c, is obtained by following a greedy policy defined over the optimal states values which is achieved by following states with higher color

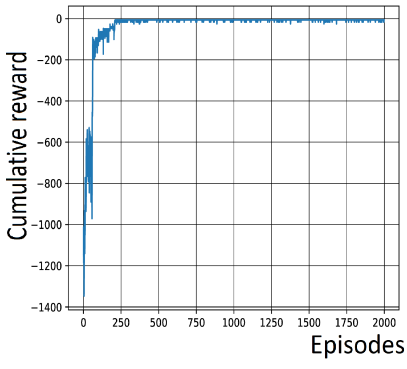
gradient. The success rate after the convergence is significantly high and consequently, it is clear that the cumulative reward plot (figure 6.3b) has less deviations from the maximum after the convergence than the standard Q-learning algorithm. The reason for this is that the model-based Q-learning is less sensitive to noisy rewards caused by environment's stochastic effects. Model-based algorithms store all the next-states experienced by robot at a given state-action pair. It updates the q-value of the pair based on the next-state that has the highest probability. Therefore, if the robot experiences a noisy reward through a less probable next-state, the model will correct its effect by simulating experiences using a more probable next-state.

Measure	Value
The cumulative long-term reward	-7
Number of action at optimal policy	17 actions
Convergence speed	After 211 episodes
Success rate	0.932

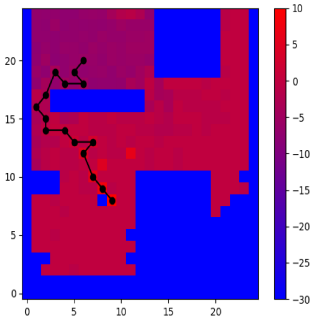
Table 6.3: Training results for applying the smart planning prioritized Dyna-Q learning algorithm.



(a) The number of executed actions for each training episode for the smart planning Dyna-Q learning algorithm.



(b) The cumulative rewards for each training episode for the smart planning Dyna-Q learning algorithm.



(c) The discovered map and the sub-optimal discovered trajectory. Each cell in the map has a color representing its value according the right-hand-side color map.

Figure 6.3: Training results for applying the smart planning Dyna Q-learning algorithm.

6.1.4 Extending the Dyna-Q with a learned kinematic model

As mentioned in section 4.2, a learned kinematic model for the robot motion can be used to extend the probabilistic multinomial model used by the Dyna-Q platform. Guided by the model structure elaborated in 4.2, the map is divided into a set of 5×5 areas (figure 6.4) and for each area, three Bayesian regression models are created to predict the three robot velocity components $[v_x \; v_y \; \omega]^T$. The simulation has demonstrated that depending on the learned kinematic models has significantly decreased the time needed for convergence. Without adopting any prioritized planning, the algorithm converges to the optimal policy in 317 episodes as shown in figure 6.5a. The reason for that can be explained by analyzing figure 6.6 that shows how frequent the algorithm depends on the learned kinematic model to make predictions. The figure shows how many predictions are made per each training episode. The algorithms learns the kinematic models of some discrete cells in the first 9 episodes and starts making predictions accordingly. The algorithms uses the learned kinematic model between the 9^{th} to the 150^{th} episodes such that the number of predictions per episode decreases with time. After the 150^{th} episode, the algorithm stops depending on the learned kinematic model since most of state-action pairs have been already visited and stored in the multinomial probabilistic model. The predictions made using the learned kinematic model have a remarkable influence on accelerating the convergence of the algorithm in 317 (figure 6.5a) episodes instead of 1768 episodes in case of the traditional Dyna-Q (figure 6.11a).

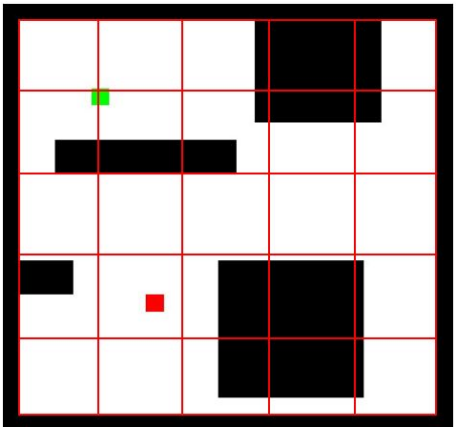
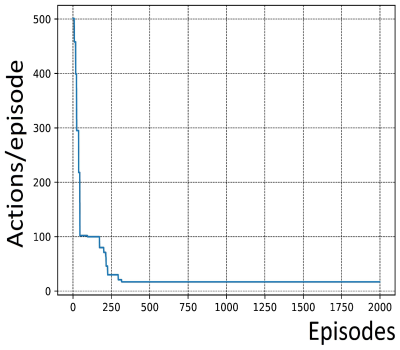


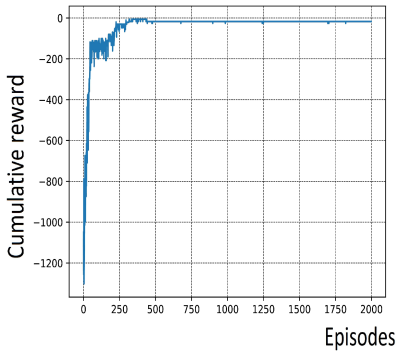
Figure 6.4: Dividing the map into 5×5 different areas.

Measure	Value
The cumulative long-term reward	-6
Number of action at optimal policy	17 actions
Convergence speed	After 317 episodes
Success rate	0.55

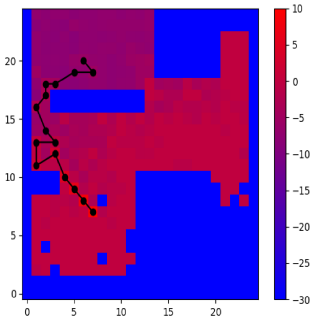
Table 6.4: Training results for applying the extended Dyna-Q learning algorithm.



(a) The number of executed actions for each training episode for the Dyna-Q algorithm extended by the learned kinematic model.



(b) The cumulative rewards for each training episode for Dyna-Q algorithm extended by the learned kinematic model.



(c) The discovered map and the sub-optimal discovered trajectory. Each cell in the map has a color representing its value according the right-hand-side color map.

Figure 6.5: Training results for applying the kinematic model with the Dyna Q-learning algorithm.

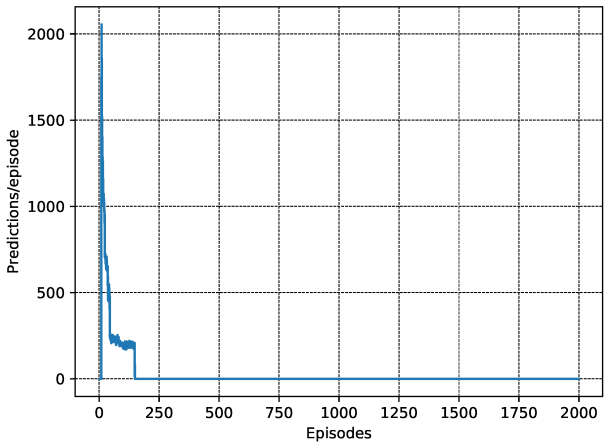
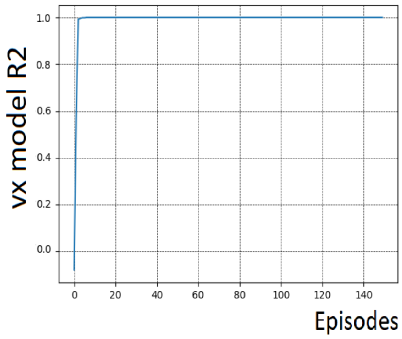
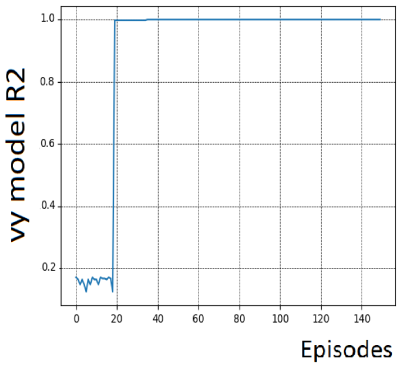


Figure 6.6: The number of executed predictions for each training episode.

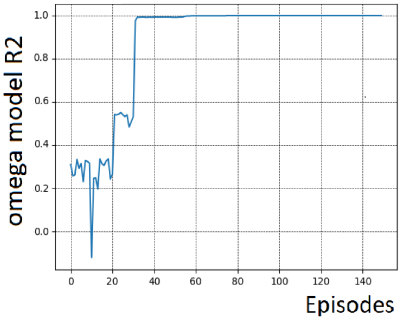
As explained in section 4.2, the coefficient of determination R^2 is calculated for each learned Bayesian regression model to evaluate their predictability. The algorithm depends on the learned regression model only if its coefficient of determination is larger than 0.5. For example, figure 6.7 shows the coefficients of determination for the three Bayesian models h_{v_x}, h_{v_y} and h_ω learned for discrete area (1,1) in the first 150^{th} episodes. The figure shows that the coefficients of determinations for each model increase with time during the training. The coefficients of determination of the h_{v_x}, h_{v_y} and h_ω model are larger than 0.5 after the 5^{th}, 18^{th} and 25^{th} episodes respectively. Therefore, at the discrete area (1,1), the algorithm utilizes the learned kinematic model only after the 25^{th} episode. A similar analysis holds for other discrete areas. The performance of the extended Dyna-Q algorithm is detailed in table 6.4.



(a) The coefficient of determination R^2 of the regression model used to predict v_x of the discrete area (1,1).



(b) The coefficient of determination R^2 of the regression model used to predict v_y of the discrete area (1,1).



(c) The coefficient of determination R^2 of the regression model used to predict ω of the discrete area (1,1).

Figure 6.7: The coefficient of determination R^2 of the regression models h_{v_x}, h_{v_y} and h_ω of the discrete area (1,1).

6.1.5 Algorithms comparison

Table 6.5 shows the performance measures of each simulated algorithm. It is noticeable that all model-based RL algorithms are significantly faster than the standard Q-learning which proves that adopting model-based RL algorithms speeds up the convergence. Moreover, the way models are utilized by the algorithm has a significant influence on the convergence speeds. When the algorithm gives higher priorities to more interesting state-action pairs, the algorithm convergence significantly faster with higher success rates. On the other hand, depending on learned kinematic model to predict the robot motion helps to accelerate the training but has less success rates than the standard Dyna-Q. The reason for this is that if the model is wrong,

state-action pairs are updated to wrong action-values resulting in following sub-optimal policies.

Measure	standard Q	Dyna-Q	Smart Planning	Extended Dyna-Q
The cumulative long-term reward	-7	-7	-7	-6
Optimal policy length (actions)	17	15	17	17
Convergence speed (episodes)	3465	1768	211	317
Success rate (%)	23.275	93.6	93.2	55

Table 6.5: Algorithms comparison table.

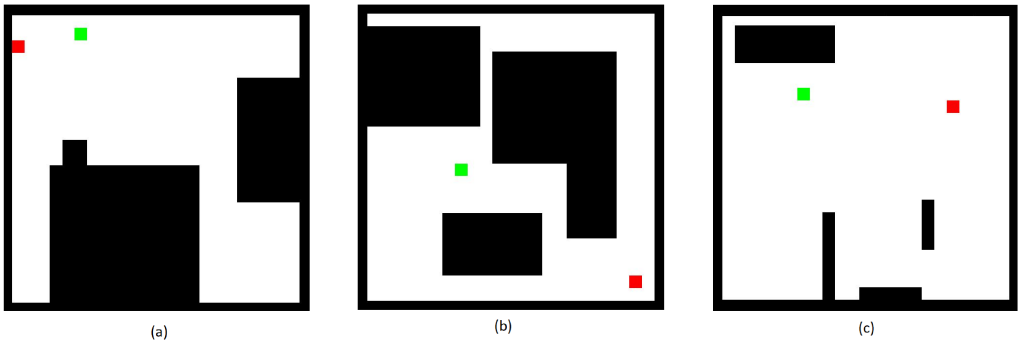
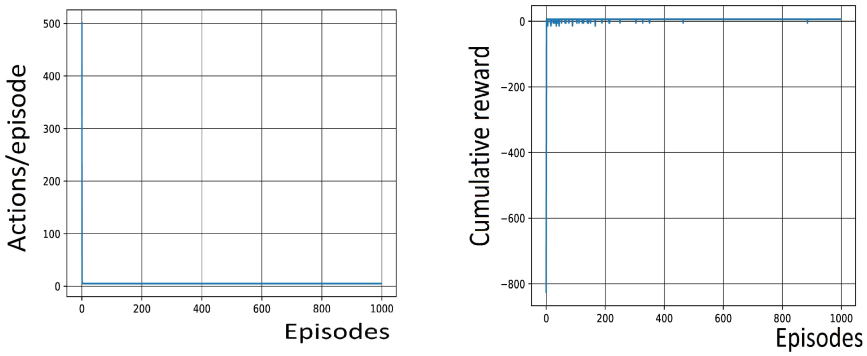


Figure 6.8: Different maps configurations to validate the algorithm robustness.

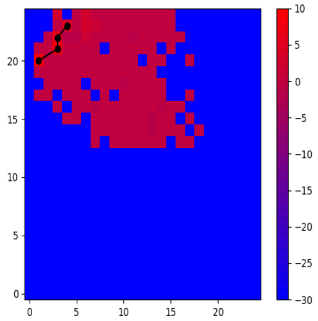
6.1.6 Algorithm robustness

The Dyna-Q algorithm combined with prioritized smart planning has been proved to be powerful in learning an approximate optimal policy in a significantly-short training period. It is important to assess the algorithm in new maps with different obstacles configurations to ensure the repeat-ability of the behaviour in changing navigation environments. Therefore the same smart planning Dyna-Q learning algorithm has been tested in the maps shown in figure 6.8. The results shown in figure 6.9, 6.10 and 6.11 suggest that the speed of convergence of the algorithm depends on the complexity of the obstacles configuration between the initial and the target locations. In case of map a, the initial state is close to the target location and the optimal path between them is obstacles-free. Therefore, the algorithm discovers the optimal trajectory significantly fast (in 3 episodes). It is also remarkable that the map is not fully explored. This is due to the fact that the Q-learning algorithm is guided by an $\epsilon - greedy$ policy with decreasing ϵ. Therefore, the algorithm becomes biased with time towards the exploitation and not the exploration. And since the agent finds the target state without the need to explore most of the map, after the quick convergence, the agent tends to follow the optimal trajectory at each episode without exploring the rest of the map.



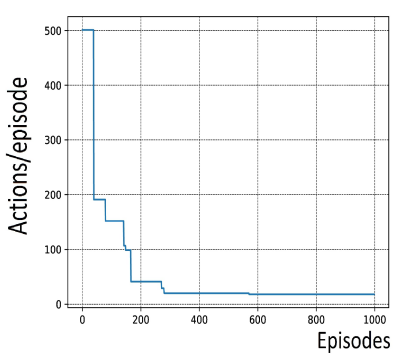
(a) The number of executed actions for each training episode for the smart planning algorithm.

(b) The cumulative rewards for each training episode for the smart planning algorithm.

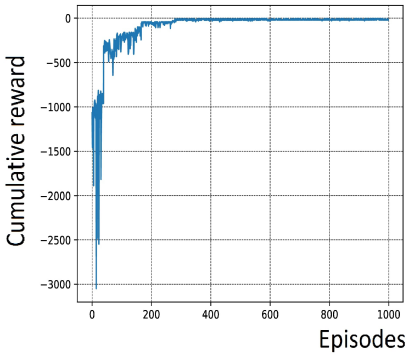


(c) The discovered map and the sub-optimal discovered trajectory. Each cell in the map has a color representing its value according the right-hand-side color map.

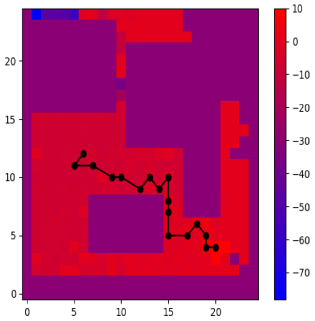
Figure 6.9: Training results for applying the smart planning Dyna Q-learning algorithm for map a.



(a) The number of executed actions for each training episode for the smart planning algorithm.



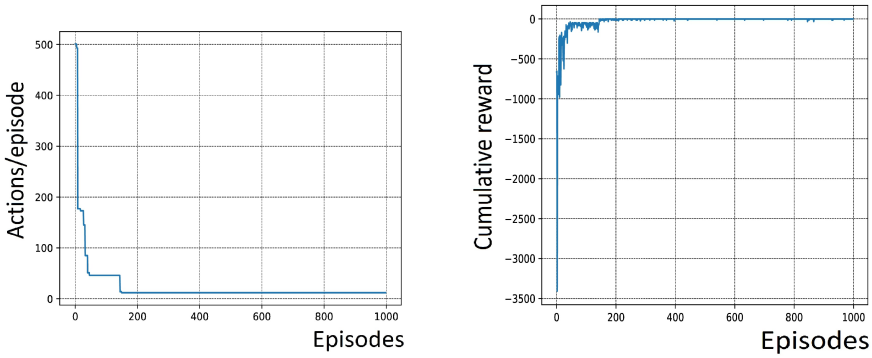
(b) The cumulative rewards for each training episode for the smart planning algorithm.



(c) The discovered map and the sub-optimal discovered trajectory. Each cell in the map has a color representing its value according the right-hand-side color map.

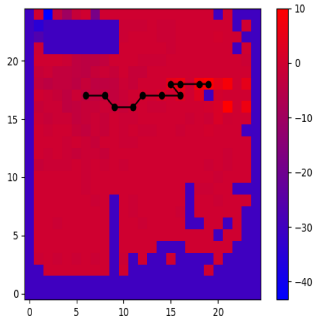
Figure 6.10: Training results for applying the smart planning Dyna Q-learning algorithm for map b.

In contrast to map a, map c is fully explored. The reason it that the obstacles configurations between the initial and the target states are more complex. The convergence occurs after 566 episodes when the map has been fully explored to find the target. The smart planning algorithm succeeds in finding the optimal navigation trajectory for map c after 138 training episodes which is reasonably fast. These results demonstrate the smart planning algorithm ability to find optimal trajectories significantly fast regardless of the map's obstacles configurations. It was also clearly shown that the convergence speed is highly co-related with the complexity of the obstacles distribution between the initial and the target state.



(a) The number of executed actions for each training episode for the smart planning algorithm.

(b) The cumulative rewards for each training episode for the smart planning algorithm.



(c) The discovered map and the sub-optimal discovered trajectory. Each cell in the map has a color representing its value according the right-hand-side color map.

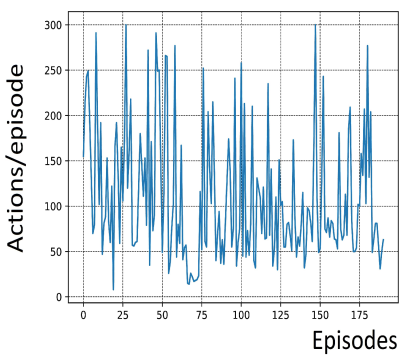
Figure 6.11: Training results for applying the smart planning Dyna Q-learning algorithm for map c.

6.2 Real-time experiments results (Slippery surface)

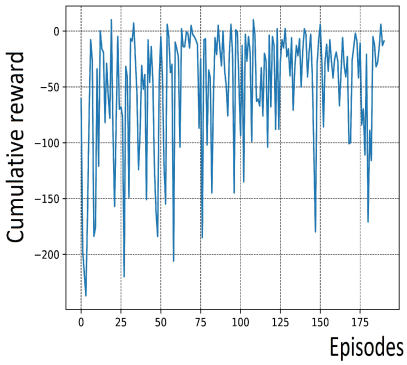
6.2.1 Q-learning algorithm

The Q-learning algorithm has been tested for the same number of episodes (190). The convergence plot (figure 6.12a) shows that the algorithm has a tendency to diverge from sub-optimal policies during the whole training period even at reduced exploration factors. The reasons for that are the noisy rewards and relatively high exploration factor of 20%. As explained in previous sub-sections, noisy rewards occur due to skidding in the robot motion. Skidding makes the same state-action pairs receive different next-states and consequently different next-rewards. In case of model-based algorithms, the model will reduce the effect of noisy less probable rewards because it will re-update the q-values using the most probable rewards. On the other hand, in the standard Q-learning algorithm, these noisy rewards will update the $q(s,a)$ to wrong values. These wrong values will not be updated correctly until the same state-action pair is revisited. On the other hand, the number of successful trials significantly increases from 35% to 55% after the exploration factor is reduced to 20% after the 60^{th} episode. The cumulative rewards graph (figure 6.12b) shows that the harvested rewards are larger after the 60^{th} episode since the majority of the episodes are successful. Finally, figure 6.12c shows that discrete states close to the goal have higher values than the states close to the obstacles and states far from it. However, the gradient of change of the values has some wrong discontinuities. Such

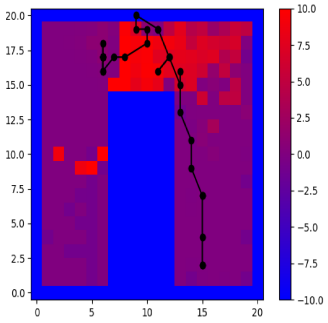
discontinuities are resulted from noisy rewards given by the environments that happen along the training periods and prevent the convergence of the q-values to constant unique values The figure also shows one of the discovered sub-optimal trajectories. Despite avoiding the obstacle, the robot does not follow a correct greedy policy over the states values due to the ongoing exploration.



(a) The number of executed actions for each training episode of the standard Q-learning algorithm.



(b) The cumulative rewards for each training episode of the standard Q-learning algorithm.



(c) The discovered map and the sub-optimal discovered trajectory. Each cell in the map has a color representing its value according the right-hand-side color map.

Figure 6.12: Training results of the Q-learning algorithm.

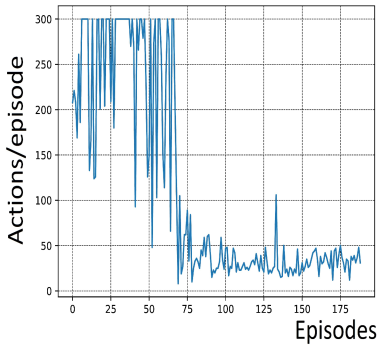
Measure	Value
The maximum cumulative reward	10
Number of action of the shortest trajectory	31 actions
Convergence speed	N.A.
Success rate	55%

Table 6.6: Training results for applying the standard Q-learning algorithm.

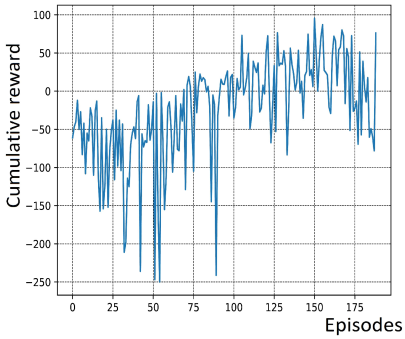
6.2.2 Smart planning algorithm

The results of training does not show a convergence to a unique optimal policy but to a set of sub-optimal policies. In the initial training episodes up to the 60^{th} episode where the exploration rate is high, the robot explore the environment to build the probabilistic model that describes the state transitions. Therefore, it is able to reach the goal in only 10 episodes. After that,

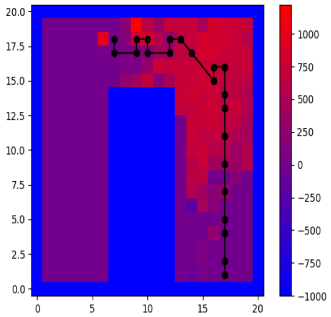
the exploration rate drops to 20% and the robot starts depending on the model for updating the Q-values and for calculating an orientation-based reward as explained in section 4.3. Accordingly, the robot starts to follow the greedy policy defined over the action-values. Therefore, the number of actions needed to reach the goal decreases significantly after the 60^{th} episode (figure 6.13a). However, the robot is not following the exact same trajectory and is not reaching the goal successfully for every episode after the 60^{th}. There are three reasons to explain that. Firstly, the robot does not adopt a fully greedy policy but an $\epsilon - greedy$ one with an exploration rate of 20%. Secondly, the robot is reset if a local minima is detected (when it remains in the same discrete locations for successive 5 trials). The third reason is that the reward received is not stationary but significantly depends on the skidding of the robot. Therefore, the next-state and reward experienced by the robot after applying the same action at the same state may differ with time. Thus, if a less probable next-state is experienced, the reward received consequently will influence the action-value of the pair even if the learning rate is lowered to 0.8. This influence is reduced using the online-created multinomial model. Since the model builds a distribution over all the next states for each state-action pair, the effect of the less probable state will be eliminated by hypothetically simulating a transition to more probable next states during the hypothetical training.



(a) The number of executed actions for each training episode of the smart planning prioritized Dyna-Q algorithm.



(b) The cumulative rewards for each training episode of the smart planning prioritized Dyna-Q algorithm.



(c) The discovered map and the sub-optimal discovered trajectory. Each cell in the map has a color representing its value according the right-hand-side color map.

Figure 6.13: Training results of the smart planning prioritized Dyna-Q algorithm.

In the cumulative reward curve (figure 6.13b) the negative values dominate in the beginning of the training since the majority of the episodes are not successful. After that, the influence of

finding the goal shifts the cumulative rewards to higher positive values. However, the influence of unsuccessful episodes decreases the cumulative reward even in last training episodes. Figure 6.13c shows the values of the discrete states the algorithm converges to. Discrete states close to the goal have higher values than the states close to the obstacles and states far from it. Similar to the model-free Q-learning, the state values do not converge to constant unique values and have some wrong values due to the effect of noisy rewards. Figure 6.13c shows one of the obtained sub-optimal trajectories. Despite avoiding the obstacle, the robot does not follow the greedy policy over the states values due to the continuous exploration.

Measure	Value
The maximum cumulative reward	95.3
Number of action of the shortest trajectory	31 actions
Convergence speed	60 episodes
Success rate	62.3%

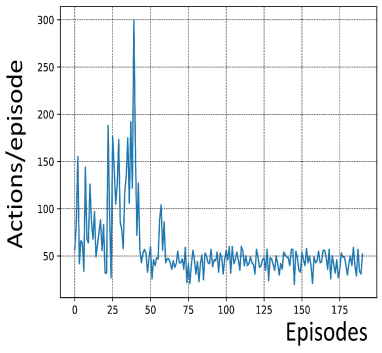
Table 6.7: Training results for applying the prioritized smart planning Dyna-Q algorithm.

6.2.3 Extended Dyna-Q algorithm

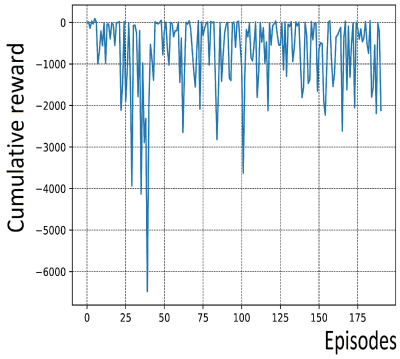
Three Bayesian regression models h_{v_x}, h_{v_y} and h_ω are created to represent the robot kinematic model as detailed in section 4.2. Because the navigation surface of the experiment is homogeneous, its coefficient of friction is assumed to be constant. Thus, we assume it is sufficient to learn a single kinematic model and use it to make predictions at any location in the map. As explained in section 4.2, the coefficient of determination R^2 is calculated for each learned Bayesian regression model to evaluate their predictability. Figure 6.15 shows the coefficients of determination for h_{v_x}, h_{v_y} and h_ω models. The coefficients are larger than 0.5 after the first training episode. Therefore, the algorithm depends on the learned kinematic model from the beginning of the training. Extending the Dyna-Q algorithm with a learned kinematic model for the robot helps to improve the behaviour in initial training episodes (figure 6.14a). The reason is that the algorithm learns the effect of applying each action on the velocity of the robot and uses the learned effect to infer the influence of actions on non-visited states. Therefore, only one training episode reaches the maximum allowable 300 trials. Similar to the prioritized Dyna-Q, after the 60^{th} episode, the algorithm follows an $\epsilon - greedy$ policy with a reduced exploration factor of 20%. The algorithm has some unsuccessful episodes after the 60^{th} episodes because of the reasons mentioned in the previous section and because the length of the trials is reduced to the average of the episodes lengths which is 59. The effect of the unsuccessful episodes is shown in the cumulative reward negative values along the training. Figure 6.14c shows one of the sub-optimal trajectories found. It is sub-optimal in the sense that it is not following the states with higher values perfectly. Similar to the Dyna-Q algorithm, the states close to the targets have higher values than the far ones or states around the obstacle with wrong discontinuities.

Measure	Value
The maximum cumulative reward	91.1
Number of action of the shortest trajectory	29 actions
Convergence speed	60 episodes
Success rate	60.7%

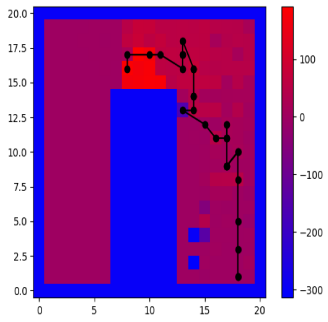
Table 6.8: Training results for applying the extended Dyna-Q algorithm.



(a) The number of executed actions for each training episode of the extended Dyna-Q algorithm.



(b) The cumulative rewards for each training episode of the extended Dyna-Q algorithm.



(c) The discovered map and the sub-optimal discovered trajectory. Each cell in the map has a color representing its value according the right-hand-side color map.

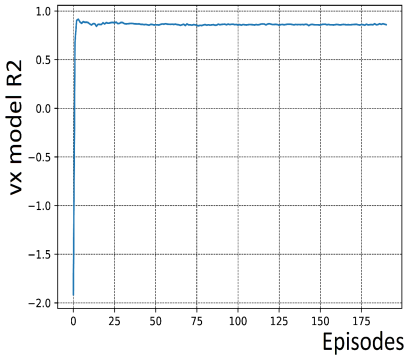
Figure 6.14: Training results of the extended Dyna-Q algorithm.

6.2.4 Algorithms comparison

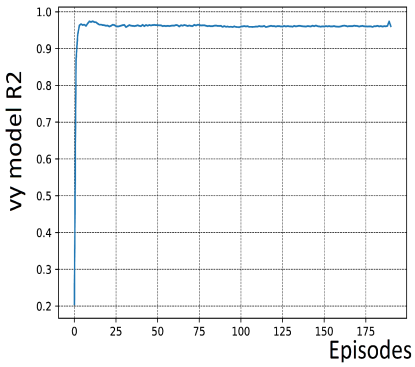
Table 6.9 shows the performance measures of each tested algorithm. None of the algorithms achieves a convergence to a unique optimal policy. However, model based algorithms stick better to the sub-optimal policies discovered after the exploration factor is lowered to 20% at the 60^{th} episode. Therefore, they have higher success rates than the model-free Q-learning since they rarely diverge from the sub-optimal behaviour when the exploration is reduced. The reasons why the model-based algorithms have better performance than the model-free Q-learning is detailed in the following sub-section.

Measure	standard Q	Smart Planning	Extended Dyna-Q
The maximum cumulative reward	10	95.3	91.1
Shortest policy length (actions)	31	31	29
Convergence speed (episodes)	N.A.	60	60
Success rate (%)	55	62.3	60.7

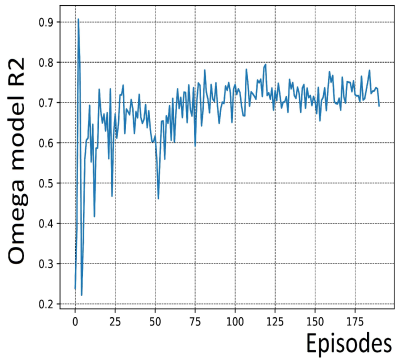
Table 6.9: Algorithms comparison table.



(a) The coefficient of determination R^2 of the regression model used to predict v_x.



(b) The coefficient of determination R^2 of the regression model used to predict v_y.



(c) The coefficient of determination R^2 of the regression model used to predict ω.

Figure 6.15: The coefficient of determination R^2 of the regression models h_{v_x}, h_{v_y} and h_ω.

6.3 Real-time experiments results (Rough surface)

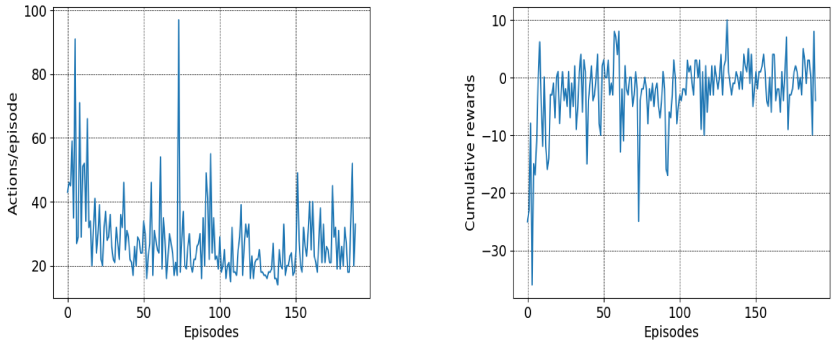
The skidding effects dominate the dynamics of skid-steering mobile robots specially when they change orientation [18]. These effects are the main reason to the stochastic nature of the SSMRs motion. Therefore, the designed algorithms have been examined using a surface with a rougher material to examine the algorithms in reduced slippage conditions (figure 6.16).

Figure 6.16: The changed material of the experiment surface.

Using a rough material for the experiment surface has sharply reduced the skidding effects and consequently a remarkable improvement has been achieved as will be explained in the following subsections.

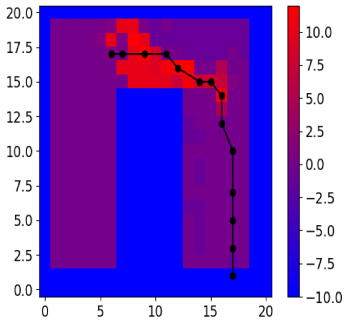
6.3.1 Model-free Q-learning algorithm

The convergence plot (figure 6.17a) shows a significant improvement in the performance of the model-free Q-learning algorithm. The figure 6.17a shows only one divergence from the discovered sub-optimal trajectories during the exploitation phase. This performance is much better than what is exhibited in the slippery surface case (figure 6.12a) in terms of the number of actions needed to reach the target for each training episode. The longest episode took less that 100 actions to reach the goal therefore, the running time of the algorithms was significantly smaller than the slippery surface case. Because the skidding effects are greatly reduced, noisy rewards which wrongly update the action-values are also reduced, therefore, the algorithm sticks better to the discovered sub-optimal policies and has very high success rate of 99.44%.



(a) The number of executed actions for each training episode of the Q-learning algorithm.

(b) The cumulative rewards for each training episode of the Q-learning algorithm.



(c) The discovered map and the sub-optimal discovered trajectory. Each cell in the map has a color representing its value according the right-hand-side color map.

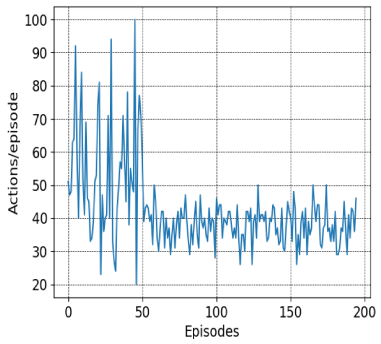
Figure 6.17: Training results of the Q-learning algorithm.

The better performance can also be seen in the cumulative reward plot (figure 6.17b) which increases exponentially from negative values to saturate around zero. The plot has less divergences to negative values than the slippery surface case (figure 6.12b). The analysis of the discovered map (figure 6.17c) shows no significant difference from the slippery surface case. The states that are more close to the target location has higher values than further states. One of the sub-optimal trajectories is plotted in figure 6.17c. The trajectory follows an $\epsilon - greedy$

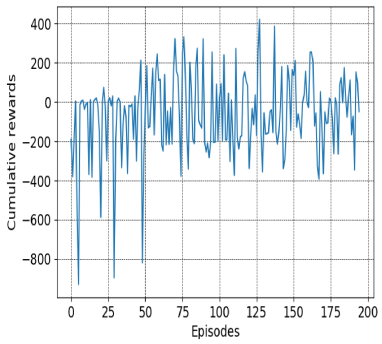
policy defined over the action-values, as can be seen, it follows states whose color gradient is increasing until reaching the target state.

6.3.2 Model-based Q-learning algorithms

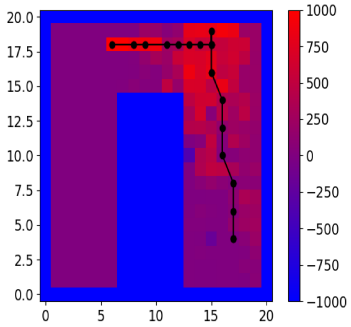
Since the skidding effects were reduces, the variance in the values of next state of every state-action pair is consequently reduced. Therefore, the algorithm needed shorter episodes to learn the environment model. As can be seen in figure 6.18a, the longest training episode of the smart planning algorithm took 100 actions, therefore, the training time needed to learn the model is significantly shorter than the slippery surface case (figure 6.13a) where the algorithm took 60 episodes taking 300 action per episode to learn the environment model. The performance is further improved when the dyna-Q is extended with the learned robot kinematic model where the longest training episode lasted for 55 actions (figure 6.19a). After the exploration is reduced at the 50^{th} episode, the agent sticks to sub-optimal trajectories with an average length of 40 actions for the smart planning algorithm and 30 actions for the extended Dyna-Q algorithm (figure 6.19a). The agent does not diverge to an unsuccessful episode to the end of the training.



(a) The number of executed actions for each training episode of the smart planning algorithm.



(b) The cumulative rewards for each training episode of the smart planning algorithm.

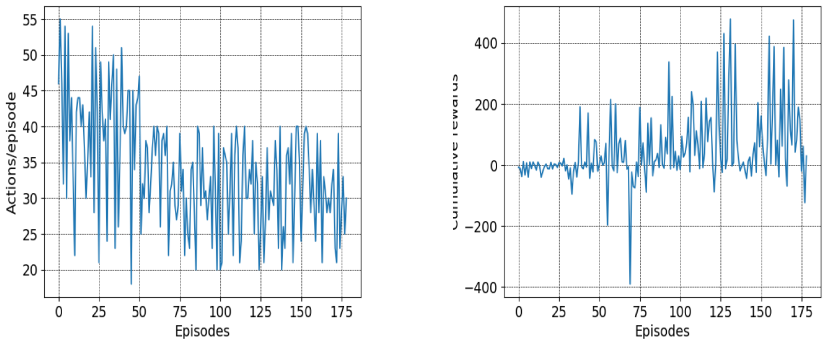


(c) The discovered map and the sub-optimal discovered trajectory. Each cell in the map has a color representing its value according the right-hand-side color map.

Figure 6.18: Training results of the smart planning algorithm.

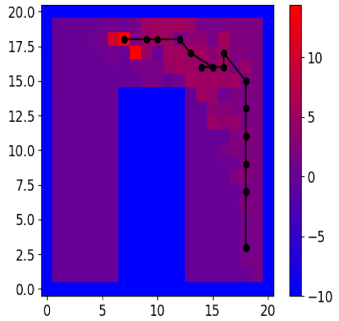
Similar to the model-free Q-learning algorithm, the cumulative reward plot (figure 6.18b) increases exponentially and saturates at a positive average value. Finally, one of the sub-optimal trajectories is plotted in figure 6.18c and figure 6.19c. The trajectory is following an $\epsilon - greedy$

policy over the action values. Despite the existence of some discontinuities, the state values are more smooth than the state values in the model-free Q-learning case.



(a) The number of executed actions for each training episode of the extended Dyna-Q algorithm.

(b) The cumulative rewards for each training episode of the extended Dyna-Q algorithm.



(c) The discovered map and the sub-optimal discovered trajectory. Each cell in the map has a color representing its value according the right-hand-side color map.

Figure 6.19: Training results of the extended Dyna-Q algorithm.

6.4 Discussion

By comparing the performance of the designed algorithms in real-time to simulations the following remarks can be concluded:

1. The tested reinforcement-learning algorithms in real-time experiments don't converge to a single unique optimal policy like simulations for the following reasons:

    - **Discretization errors:** it has been elaborated in section 3.1.1 that representing discrete states of the robotic system reduces the Markovian proprieties of the decision process which is a necessary condition for the Q-learning algorithm to converge. As discussed in section 4.1.3, choosing the discretization levels is a design parameter that needs to be optimized. It should not be so large that the algorithm needs undesirable long time to visit all the states and it should not so low that the system become non-Markovian. The discretization levels have been chosen via trial-and-error to be 20 for each $[x, y, \theta]^T$ dimension. Accordingly, the resolution of the gird is $2.5cm \times 3.1cm \times 18°$. Therefore, to follow a discovered optimal trajectory, the robot

is expected to deviate from it influenced by the discretization errors that it accumulates with time.

- **Skidding effects:** skidding effects dominate when the robot changes its orientation [18]. The skidding effects introduce stochasticity to the movement of the robot in the sense that applying the same action to the same state results in a transition to a set of possible nonidentical next states. Therefore, the reward received from the environment to the same state-action pair is time-variant since it depends on the next state the robot moves to. Moreover, it is hard to assume that the stochastic distributions that guide these transitions are stationary. Therefore, the algorithm may take a greedy action based on an optimal policy learned in a particular distribution but it will not be greedy if this distribution varies with time. In the simulation, the skidding effects were added to the robot kinematic model assuming the assumptions introduced in section 5.1.1. However, the robot can hardly maintain the ideal behaviour expected by the assumptions.
- **Local minima:** Local minima are completely undesirable since they may consume most of the trials available for each episode wasting the time resource of the learning experiment. The local minima problem arises as a consequence of coarse discretization of the state space. When the state space is coarsely discretized, applying different actions on the same state can result in a transition to the same next-state. Thus, the two action will share the same action-value even though they have different effect on the robot motion. This causes the robot to get stuck in local minima. The minimum exploration factor is kept relatively large (20%) for real-time experiments to prevent local minima. Therefore, the algorithm will always follow an $\epsilon - greedy$ policy not a greedy one. In simulations, it was possible to simulate relatively long episodes (2000 or 4000) and checking the greedy behaviour of the learned policy by reducing the exploration to only 1%. However, difficult and time consuming to run such long training experiments in real-time.

2. Model-based algorithms behaves better than the standard Q-learning for the following reasons:

   - **Robustness to noisy rewards:** The Dyna-Q algorithms stores all the next-states experienced by robot at a given state-action pair. It updates the q-value of the pair based on the next-state that has the highest probability. Therefore, if the robot experiences a noisy reward through a less probable next-state, the model will correct its effect by simulation experiences using a more probable next-state. On the other hand, the model-free Q-learning algorithm is more sensitive to noisy rewards since their effects are not corrected until this state-action pair is revisited.
   - **Better long-term behaviour:** After the exploration is reduced, the model-based algorithms follow sub-optimal policies that are shorter. In addition, model-based algorithms have been proven to have higher success rates than the standard Q-learning algorithm.
   - **Shorter training time:** The sub-optimal policies followed by the model-based algorithms are generally shorter than the others followed by the model-free Q-learning. This limits the time of the episodes making the training faster.

3. By changing the material of the navigation surface using a rough material, the performance of the learning algorithm is greatly improved. By increasing the friction between the mobile robot and the ground, the stochastic effects generated by the sliding are sharply reduced. This reduction results in less noisy rewards and therefore, less wrong updates in the action-values for both the model-free and the model-based algorithms.

Therefore, by using a rougher surface, the divergence from learned sub-optimal policies rarely occurs unlike the slippery surfaces case.

6.5 Reflection

In this section the strengths and the weaknesses of the proposed methods are analyzed based on the obtained results. The proposed solution provides a discrete representation of state-action spaces which has various advantages as well as some drawbacks. Representing discrete state spaces helps to avoid wrong predictions of function approximators during extrapolations [25]. However, it is practical to depend on the discrete representation of the state space of relatively small size, otherwise, the convergence of the Q-learning algorithms becomes significantly small. The size of the discrete state space depends on the chosen quantization levels to discretize each $[x\ y\ \theta]$ states. Thus, representing a low-sized state space implies choosing low quantization levels which lead to quantization errors. This consequently leads to sub-optimal behaviours. Similarly, representing a discrete action space introduces a constraint to the optimality of the discovered solution. Accordingly, the discovered optimal trajectories are all discontinues because they are defined with respect to discrete actions, therefore they can be considered sub-optimal. It is possible to solve for smooth navigation trajectories only if the action space is continuous [8]. If the action space is continuous, the reinforcement learning problem can be solved using policy search methods which is different from the value-function-based methods adopted by this thesis.

The proposed reinforcement learning solution is model-based. Constructing online models are beneficial since they allow executing hypothetical planning experiences to update the action-values. This helps to propagate the effect of updating the value of any state-action pair to all connected state-action pairs which accelerates the convergence of the Q-learning algorithm. This has been validated since the model-based algorithms converged significantly faster than model-free algorithms in both simulation and real-time. Moreover, the simulated hypothetical experiences are simulated around the shortest discovered trajectories which was proven to accelerate the convergence. These constructed models are probabilistic, therefore, they simulate the transition to the most probable states learned from the experience. This helps to relieve the effect of noisy rewards if a transition to a less probable state takes place. Therefore, the success rates of model-based algorithms are higher than model-free algorithms since model-based algorithms reduce the divergence from the discovered sub-optimal trajectories. One drawback of these models appears if the state space is very large. Since the models are tabular, it is less feasible to construct them for large-sized state spaces. Another drawback is related to extending the Dyna-Q platform with the robot kinematic model. The proposed method assumes an obstacle-free map while simulating applied actions to non-visited states. Therefore, the predictability of the robot depends on the obstacles distributions in the map.

Although the model accounts for the stochasticity generated due to the skidding effects, it assumes that these stochastic properties are stationary. This assumption can not be guaranteed, therefore, the model-based updates will be not be accurate if the stochastic effects change with time. Moreover, implementing the reinforcement learning in real-time suffered from local minima problems. Thus, the minimum exploration factor of the algorithms was kept at significantly large value leading to an $\epsilon - greedy$ policies. Accordingly, the agent will keep applying exploratory actions even after the convergence resulting in following sub-optimal navigation paths.

7 Conclusion

This project aimed to structure a novel model-based reinforcement learning solution to the autonomous navigation of skid-steering mobile robots in unknown environments. The solution was based on the Dyna-Q platform and is realized by extending the standard model-free Q-learning with two interacting models. The first model learns the state transition probabilities of the robot in a given environment with unknown static obstacles, and the other model learns a parametric kinematic model of the robot motion. It has been proven that the state transition probabilities of the system can be modeled using a multinomial probabilistic model that incorporates the stochastic skidding dynamics. Moreover, a parametric kinematic model of the robot motion can be learned online using a Bayesian ridge regression approach. The parametric kinematic model maps the applied wheels velocities to the resultant robot's COM velocities while taking the slippage effects into account.

The online-structured models have been used to simulate hypothetical motion experiences to update the Q-values in the background through a process termed as planning update. The multinomial probabilistic model has been used to simulate transitions from already-visited states and the robot parametric kinematic model was used to infer the effect of applied actions to non-visited states. The simulated updates was powerful in propagating the effect of any change in the action-value of any state-action pair to connecting pairs. It has also been proved that the sequence by which these hypothetical planning simulations are organized has significant influence on the convergence speed of the algorithm. Accordingly, executing these simulated learning experiences was prioritized around the shortest discovered navigation trajectories between the initial and the target states. To validate such ideas, a simulation and a real-time reinforcement-learning platforms have been designed and implemented in a Python-based environment. The designed algorithms were compared based on three optimality criteria: the convergence speed, the optimal trajectory length and the success rate.

It has been experimentally proven that the model-based RL algorithms are more efficient than the model-free Q-learning. In simulations, model-based algorithms can converge 16 times faster than the model-free Q-learning. It also increases the success rate of the algorithm for up to 70%. There is no significant difference between the lengths of the optimal trajectories for all the designed algorithms. On the other hand, validating the designed algorithms in real-time experiments has shown that the designed algorithms converges not to a single unique optimal trajectory but to a set of sub-optimal ones. As it has been detailed, combination effects of discretization errors, time-variant skidding and local minima limit the convergence ability of the algorithm to a unique optimal solution. However, model-based algorithms still have better learning performance than the model-free Q-learning. Utilizing the models to simulate learning experiences makes the robot more robust to noisier rewards caused by stochastic effects. Therefore, model-based algorithms have better long-term behaviours in terms of less divergences from sub-optimal trajectories than the model-free algorithm. Consequently, model-based algorithms have higher success rates of 12% and shorter training episodes which reduces the total time of the RL experiments.

The performance of all the RL algorithms was significantly improved when a rougher material was used for the navigation surface. The real-time experiments showed that increasing the friction between the ground and the wheels has greatly reduce the sliding stochastic effects. Accordingly, the examined RL algorithms did not exhibit any significant divergences from the

learned sub-optimal policies which led to increasing the success rates to double the case of slippery surfaces.

7.1 Future research

The next step in the research should focus on representing continuous state spaces. Relying on a continuous state space is expected to eliminate the effects introduced by discretization errors, thus, increase the Markovian properties of the system. Instead of the tabular representation of the action-values presented in this thesis, the continuous space representation implies using an appropriate function approximation algorithm to learn a parametric action-value function [28]. Neural networks have been combined effectively with Q-learning as function approximators [29] [17] [34] [30]. In addition, the stochasticity can be integrated into the learned continuous value function using Gaussian processes [14] [15]. Another important track that can be further investigated is representing continuous action spaces. This implies that the agent needs to select multiple actions (each wheel velocity) simultaneously [32]. Continuous parametric action spaces are expected to learn smoother trajectories [8]. Parametric action spaces require solving the reinforcement learning problem using a different approach called *policy search* [12]. It has been proven that policy search methods cope well with robotic problems with high-dimensional state and action spaces[8].

A Appendix

This appendix shows the codes used to run reinforcement learning algorithms in simulation and in real-time.

A.1 Simulation codes

A.1.1 SSMR class

```
class Robot(object):
    """Defines basic mobile robot properties"""
    def __init__(self):
        self.pos_x  = 0.0
        self.pos_y  = 0.0
        self.theta  = 0.0
        self.plot   = False
        self._delta = 0.01   #sample time

    # Movement
    def step(self):
        """ updates the x,y and angle """


        self.deltax()
        self.deltay()
        self.deltaTheta()
    def move(self, seconds):
        """ Moves the robot for an 's' amount of seconds"""
        for i in range(int(seconds/self._delta)):
            self.step()
            if i % 3 == 0 and self.plot: # plot path every 3 steps
                self.plot_xya()

    # Printing-and-plotting:
    def print_xya(self):
        """ prints the x,y position and angle """
        print ("x = " + str(self.pos_x) +" "+ "y = " + str(self.pos_y))
        print ("a = " + str(self.theta))

    def plot_robot(self):
        """ plots a representation of the robot """
        plt.arrow(self.pos_x, self.pos_y, 0.001
                  * cos(self.theta), 0.001 * sin(self.theta),
                  head_width=self.length, head_length=self.length,
                  fc='k', ec='k')

    def plot_xya(self):
        """ plots a dot in the position of the robot """
        plt.scatter(self.pos_x, self.pos_y, c='r', edgecolors='r')



class SSMRobot(Robot):
    """Defines a SSMR drive robot"""

    def __init__(self):
        Robot.__init__(self)
        # geometric parameters

        self.r = 0.0225
        self.c = 0.06
```

```
53          self.a = 0.05
54          self.b = 0.05
55          self.Xicr = 0.05
56          self.length = 0.1
57
58          #states
59          self.omega = 0
60          self.vx = 0
61          self.vy = 0
62
63
64
65          self.omega_r = 0.0
66          self.omega_l = 0.0
67
68          #noise
69          self.noise_co = {}
70          self.Xices = {}
71          self.shape_slippage()
72
73
74      def deltax(self):
75          # calculate vx and vy
76          self.omega = self.r*(self.omega_r-self.omega_l)/(2*self.c)
77          self.vx = self.r*(self.omega_r+self.omega_l)/2
78          self.vy = -self.Xicr*self.omega
79          # calculate X_dot
80          X_dot = cos(self.theta)*self.vx - sin(self.theta)*self.vy
81          self.pos_x += self._delta *X_dot
82
83          if self.pos_x > 0.62:
84              self.pos_x = 0.62
85          elif self.pos_x < 0:
86              self.pos_x = 0
87
88
89
90      def deltay(self):
91          # calculate vx and vy
92          self.omega = self.r*(self.omega_r-self.omega_l)/(2*self.c)
93          self.vx = self.r*(self.omega_r+self.omega_l)/2
94          self.vy = -self.Xicr*self.omega
95          #calculate Y_dot
96          Y_dot = sin(self.theta)*self.vx + cos(self.theta)*self.vy
97          self.pos_y += self._delta * Y_dot
98
99          if self.pos_y > 0.505:
100             self.pos_y = 0.505
101         elif self.pos_y < 0:
102             self.pos_y = 0
103
104
105     def deltaTheta(self):
106         # calculate omega
107         self.omega = self.r*(self.omega_r-self.omega_l)/(2*self.c)
108         self.theta += self._delta * self.omega
109
110     def reset(self,player):
111         # given 3mx3m arena discretized to 25x25
112
113         self.pos_x = player[0]*0.62/24      #discrete level 5
114         self.pos_y = player[1]*0.505/24     #discrete level 5
115         self.theta = 0
```

```
#


    def optimal_action(self,action):

        self.action = action



    def take_step(self):
        self.eta = self.noise_co[self.get_discretized_state_large_level()[0:2]]
        if self.action ==0 :
            self.omega_l = 1.7 +random.uniform(-self.eta,0)  # Add noise in the omegas!!
            self.omega_r = 1.7 +random.uniform(-self.eta,0)
        elif self.action ==1 :
            self.omega_l = -1.7+random.uniform(0,self.eta)
            self.omega_r = -1.7+random.uniform(0,self.eta)
        elif self.action ==2 : # left
            self.omega_l = -1.7+random.uniform(0,self.eta)
            self.omega_r = 1.7+ random.uniform(-self.eta,0)
        else: #right
            self.omega_l = 1.7+ random.uniform(-self.eta,0)
            self.omega_r = -1.7+random.uniform(0,self.eta)
        self.move(self._delta)

    def get_discretized_state(self):
        x_discrete = floor(((self.pos_x )/0.62)*24)
        y_discrete = floor(( (self.pos_y) /0.505)*24)
        theta_discrete = np.arctan2(np.sin(self.theta), np.cos(self.theta))
        #print(theta_discrete*180/np.pi)
        theta_discrete = floor( theta_discrete/(2*np.pi) *20)

        #theta_discrete = 0

        #print((self.action,self.get_IF_velocity() ))
        #print(x_discrete,y_discrete,theta_discrete)

        return (x_discrete,y_discrete,theta_discrete)


    def get_discretized_state_large_level(self):
        x_discrete = floor(((self.pos_x )/0.62)*6)
        y_discrete = floor(( (self.pos_y) /0.505)*6)
        theta_discrete = np.arctan2(np.sin(self.theta), np.cos(self.theta))
        #print(theta_discrete*180/np.pi)
        theta_discrete = floor( theta_discrete/(2*np.pi) *20)

        return (x_discrete,y_discrete,theta_discrete)


    def shape_slippage(self):

        for x_state in range(-40,40):
            for y_state in range(-40,40):
                if y_state < 11:
                    self.noise_co[(x_state,y_state)] = fabs(x_state/1000) +  fabs(y_state/1000)
                else:
                    self.noise_co[(x_state,y_state)] = fabs(x_state/1000) +  fabs(y_state/1000)
                self.Xices[(x_state,y_state)] = random.uniform(-0.05,0.05)

```

```
176
177
178     def assign_discretized_state(self,x,y):
179         self.pos_x= ((x)/24)*0.62
180         self.pos_y= ((y)/24)*0.505
```

A.1.2 simulation environment

```
1
2 class PathFindingGame_kinematic(PathFindingGame):
3     """
4     A PathFinding games
5     state :
6         0 = nothing
7         1 = wall
8         2 = player
9         3 = goal
10    """
11
12    def __init__(self, lines=15, columns=15, *, grid_type="free"):
13        super(PathFindingGame_kinematic,self).__init__(lines, columns,grid_type=
      grid_type)
14        self.robot = SSMRobot()
15        self.robot._delta = 0.1
16        self.sdds = {}
17        self.obstacles = []
18
19    def reset(self):
20
21        self.terminal = False
22
23        self.grid, self.player, self.target = generate_grid(
24            self.shape,
25            grid_type=self.grid_type,
26            generation_seed=self.generation_seed,
27            spawn_seed=self.spawn_seed
28        )
29
30        self.robot.reset(self.player)
31        self.orientation = 0
32        return self.get_state() , self.robot
33
34    def get_state(self):
35        """ return a (n, n) grid """
36        state = np.array(self.grid, copy=True)
37        state[self.player[0], self.player[1]] = 2
38        state[self.target[0], self.target[1]] = 3
39        return state
40
41    def get_sdds(self):
42        return self.sdds
43
44    def step(self, a):
45
46        if self.terminal:
47            return self.step_return(10)
48
49        assert 0 <= a and a < 4
50
51        self.robot.optimal_action(a)
52        for i in range(0,10):
53            self.robot.take_step()
54            next_i, next_j, self.orientation = self.robot.get_discretized_state()
```

```
55              if is_obstacle(self.grid, next_i, next_j):
56                  break
57              self.gather_data()
58          #self.update_robot_model()
59          #print(self.robot.get_IF_velocity())
60
61          next_i, next_j, self.orientation = self.robot.get_discretized_state()
62
63          #print(self.robot.get_discretized_state())
64
65          #print(next_i, next_j)
66          if is_legal(self.grid, next_i, next_j):
67              self.player = (next_i, next_j)
68
69          if is_obstacle(self.grid, next_i, next_j):
70              self.robot.assign_discretized_state(self.player[0],self.player[1])
71              if  (next_i, next_j) in self.obstacles:
72                  self.obstacles.append((next_i, next_j))
73                  return self.step_return(-10)
74              self.obstacles.append((next_i, next_j))
75              return self.step_return(-10) #correct is -5
76
77          if self.player == self.target:
78              self.terminal = True
79              return self.step_return(10) #correct is 10
80
81
82
83          return self.step_return(-1-10*self.sdds[self.robot.
        get_discretized_state_large_level()[0:2]]["var"]) #correct is -1
84
85      def step_return(self, reward):
86
87          return self.get_state(),self.orientation, reward, self.terminal, self.sdds,
         self.robot
88
89      def gather_data(self):
90          mobile_robot = self.robot
91          if mobile_robot.get_discretized_state_large_level()[0:2] not in self.sdds.
        keys():
92                  self.sdds[mobile_robot.get_discretized_state_large_level()[0:2] ] =
         {}
93                  self.sdds[mobile_robot.get_discretized_state_large_level()[0:2] ]["
        omega_l"]=[]
94                  self.sdds[mobile_robot.get_discretized_state_large_level()[0:2] ]["
        omega_r"]=[]
95                  self.sdds[mobile_robot.get_discretized_state_large_level()[0:2] ]["
        v_x"]=[]
96                  self.sdds[mobile_robot.get_discretized_state_large_level()[0:2] ]["
        v_y"]=[]
97                  self.sdds[mobile_robot.get_discretized_state_large_level()[0:2] ]["
        omega"] = []
98                  self.sdds[mobile_robot.get_discretized_state_large_level()[0:2] ]["
        forward_omegas"] = []
99                  self.sdds[mobile_robot.get_discretized_state_large_level()[0:2] ]["
        var"] = 0
100                 self.sdds[mobile_robot.get_discretized_state_large_level()[0:2] ]["
        vx_model_score"]=[]
101                 self.sdds[mobile_robot.get_discretized_state_large_level()[0:2] ]["
        vy_model_score"]=[]
102                 self.sdds[mobile_robot.get_discretized_state_large_level()[0:2] ]["
        omega_model_score"]=[]
```

```
103                 self.sdds[mobile_robot.get_discretized_state_large_level()[0:2] ]["
    omega_l"].append(mobile_robot.omega_l)
104                 self.sdds[mobile_robot.get_discretized_state_large_level()[0:2] ]["
    omega_r"].append(mobile_robot.omega_r)
105                 self.sdds[mobile_robot.get_discretized_state_large_level()[0:2] ]["
    v_x"].append(mobile_robot.vx)
106                 self.sdds[mobile_robot.get_discretized_state_large_level()[0:2] ]["
    v_y"].append(mobile_robot.vy)
107                 self.sdds[mobile_robot.get_discretized_state_large_level()[0:2] ]["
    omega"].append(mobile_robot.omega)
108             else:
109                 self.sdds[mobile_robot.get_discretized_state_large_level()[0:2] ]["
    omega_l"].append(mobile_robot.omega_l)
110                 self.sdds[mobile_robot.get_discretized_state_large_level()[0:2] ]["
    omega_r"].append(mobile_robot.omega_r)
111                 self.sdds[mobile_robot.get_discretized_state_large_level()[0:2] ]["
    v_x"].append(mobile_robot.vx)
112                 self.sdds[mobile_robot.get_discretized_state_large_level()[0:2] ]["
    v_y"].append(mobile_robot.vy)
113                 self.sdds[mobile_robot.get_discretized_state_large_level()[0:2] ]["
    omega"].append(mobile_robot.omega)
114                 if mobile_robot.action == 0:
115                     self.sdds[mobile_robot.get_discretized_state_large_level()[0:2]
     ]["forward_omegas"].append(mobile_robot.omega_l)
116                     self.sdds[mobile_robot.get_discretized_state_large_level()[0:2]
     ]["var"] = np.var(self.sdds[mobile_robot.get_discretized_state_large_level()
    [0:2] ]["forward_omegas"])
117
118     def update_robot_model(self):
119
120         for key in self.sdds.keys():
121             X_data =  np.array( [  self.sdds[key]["omega_l"] , self.sdds[key]["
    omega_r"] ] ).transpose()
122             Vx_data =  np.array(  [self.sdds[key]["v_x"] ] ).transpose().ravel()
123             Vy_data = np.array(  [self.sdds[key]["v_y"] ] ).transpose().ravel()
124             Omega_data =  np.array(  [self.sdds[key]["omega"] ] ).transpose().ravel
    ()
125             X_train = X_data
126             y_train = Vx_data
127
128             clf_vx = BayesianRidge(compute_score=True)
129             clf_vx.fit(X_train, y_train)
130
131             y_train = Omega_data
132
133
134             clf_omega = BayesianRidge(compute_score=True)
135             clf_omega.fit(X_train, y_train)
136
137
138             X_train = Omega_data.reshape(-1,1)
139             y_train = Vy_data
140
141             clf_vy = BayesianRidge()
142             clf_vy.fit (X_train,y_train )
143
144
145             self.sdds[key]["vx_model"] = clf_vx
146             self.sdds[key]["vy_model"] = clf_vy
147             self.sdds[key]["omega_model"] = clf_omega
```

A.1.3 Q-learn Class

```

class QLearn:
    def __init__(self, actions, epsilon, alpha, gamma):
        self.q = {}
        self.epsilon = epsilon  # exploration constant
        self.alpha = alpha      # discount constant
        self.gamma = gamma      # discount factor
        self.actions = actions
        self.v = np.zeros((25,25))-30

    def getQ(self, state, action):
        return self.q.get((state, action), 0.0)

    def learnQ(self, state, action, reward, value):
        '''
        Q-learning:
            Q(s, a) += alpha * (reward(s,a) + max(Q(s') - Q(s,a))
        '''
        oldv = self.q.get((state, action), None)
        if oldv is None:
            self.q[(state, action)] = reward
        else:
            self.q[(state, action)] = oldv + self.alpha * (value - oldv)

    def chooseAction(self, state, return_q=False):
        q = [self.getQ(state, a) for a in self.actions]
        maxQ = max(q)
        i=state[0][0]
        j=state[0][1]
        self.v[25-i,j] = maxQ

        if random.random() < self.epsilon:
            minQ = min(q); mag = max(abs(minQ), abs(maxQ))
            # add random values to all the actions, recalculate maxQ
            q = [q[i] + random.random() * mag - .5 * mag for i in range(len(self.
    actions))]
            maxQ = max(q)

        count = q.count(maxQ)
        # In case there're several state-action max values
        # we select a random one among them
        if count > 1:
            best = [i for i in range(len(self.actions)) if q[i] == maxQ]
            i = random.choice(best)
        else:
            i = q.index(maxQ)

        action = self.actions[i]
        if return_q: # if they want it, give it!
            return action, q
        return action

    def learn(self, state1, action1, reward, state2):
        maxqnew = max([self.getQ(state2, a) for a in self.actions])
        self.learnQ(state1, action1, reward, reward + self.gamma*maxqnew)
```

A.1.4 Reinforcement-learning training

```


def update_world_model(state_prev,state,reward,action):
    if (state_prev,action) in comparator:
```

```
        comparator[(state_prev,action)]["actual"] = state
    if (state_prev,action) in world_model.keys():

        world_model[(state_prev,action)]['next_state'].append(state)
        world_model[(state_prev,action)]['reward'].append(reward)
        world_model[(state_prev,action)]['distribution'][state] = {}
        world_model[(state_prev,action)]['distribution'][state]['reward']= reward
        for state_ in world_model[(state_prev,action)]['next_state']:
            world_model[(state_prev,action)]['distribution'][state_]['prob']=
    world_model[(state_prev,action)]['next_state'].count(state_)/len(world_model[(
    state_prev,action)]['next_state'])
        #print(world_model[(state_prev,action)])
    else:
        world_model[(state_prev,action)]={}
        world_model[(state_prev,action)]['next_state'] = []
        world_model[(state_prev,action)]['next_state'].append(state)
        world_model[(state_prev,action)]['reward'] = []
        world_model[(state_prev,action)]['reward'].append(reward)
        world_model[(state_prev,action)]['distribution']={}
        world_model[(state_prev,action)]['distribution'][state] = {}
        world_model[(state_prev,action)]['distribution'][state]['reward']=reward
        world_model[(state_prev,action)]['distribution'][state]['prob']=1


def hypothetical_experience(k):
    keys = list(world_model.keys())
    state_prev = (random.choice(keys))[0]
    for i in range(0,k+1):
        state_prev = (random.choice(keys))[0]
        action = qlearn.chooseAction(state_prev)
        if (state_prev,action) in world_model.keys():
            index = random.choice(range(0,len(world_model.get((state_prev,action))[
    'reward'])))
            reward = world_model.get((state_prev,action))['reward'][index]
            state = world_model.get((state_prev,action))['next_state'][index]
            qlearn.learn(state_prev,action,reward,state)
        else:
            if episode > 1:
                if get_area(estimate_pose(state_prev)[0],estimate_pose(state_prev)
    [1] ) in sdds.keys():
                    if "vy_model" in sdds[get_area(estimate_pose(state_prev)[0],
    estimate_pose(state_prev)[1] )].keys():
                        state , reward = predict(state_prev,action)
                        qlearn.learn(state_prev,action,reward,state)
                        predictions_checked [episode]+= 1

                        print("kinematic_model")
            else:
                state_prev = (random.choice(keys))[0]




def predict(state_prev,action):
        if action ==0 :
            omega_l = 1.7 +random.uniform(-sdds[get_area(estimate_pose(state_prev)
    [0],estimate_pose(state_prev)[1] )]["var"],0)  # Add noise in the omegas!!
            omega_r = 1.7 +random.uniform(-sdds[get_area(estimate_pose(state_prev)
    [0],estimate_pose(state_prev)[1] )]["var"],0)
        elif action ==1 :
            omega_l = -1.7+random.uniform(0,sdds[get_area(estimate_pose(state_prev)
    [0],estimate_pose(state_prev)[1] )]["var"])
```

```
59              omega_r = -1.7+random.uniform(0,sdds[get_area(estimate_pose(state_prev)
    [0],estimate_pose(state_prev)[1] )]["var"])
60          elif action ==2 : # left
61              omega_l = -1.7+random.uniform(0,sdds[get_area(estimate_pose(state_prev)
    [0],estimate_pose(state_prev)[1] )]["var"])
62              omega_r = 1.7+ random.uniform(-sdds[get_area(estimate_pose(state_prev)
    [0],estimate_pose(state_prev)[1] )]["var"],0)
63          else: #right
64              omega_l = 1.7+ random.uniform(-sdds[get_area(estimate_pose(state_prev)
    [0],estimate_pose(state_prev)[1] )]["var"],0)
65              omega_r = -1.7+random.uniform(0,sdds[get_area(estimate_pose(state_prev)
    [0],estimate_pose(state_prev)[1] )]["var"])
66          state_predicted=  predict_robot_discrete_loc(state_prev,omega_r,omega_l
    ,0.1)
67          comparator[(state_prev,action)]={}
68          comparator[(state_prev,action)]["predicted"] = state_predicted
69          state = ( (state_predicted[0],state_predicted[1]),state_predicted[2] )
70          if state[0] in obstacles or state[0][0]>24 or state[0][0]<0 or state
    [0][1]>24 or state[0][1]<0 :
71              reward = -5
72          else:
73              reward = 0
74          return (state,reward)
75
76
77  def descretize(cont_state):
78      x_discrete = floor(((cont_state[0] )/0.62)*24)
79      y_discrete = floor(( (cont_state[1]) /0.505)*24)
80      theta_discrete = np.arctan2(np.sin(cont_state[2]), np.cos(cont_state[2]))
81      theta_discrete = floor( theta_discrete/(2*np.pi) *10)
82      return (x_discrete,y_discrete,theta_discrete)
83
84  def get_area(x,y):
85          x_discrete = floor(((x)/0.62)*6)
86          y_discrete = floor(( (y) /0.505)*6)
87
88          return (x_discrete,y_discrete)
89
90  score={}
91
92  def predict_robot_discrete_loc(state,omega_r,omega_l,_delta):
93      (x,y,theta) = estimate_pose(state)
94      area = get_area(x,y)
95      if area in sdds.keys():
96          vx = sdds[area]['vx_model'].predict(np.array([omega_l,omega_r]).reshape(1,
    -1) )
97          omega = sdds[area]['omega_model'].predict(np.array([omega_l,omega_r]).
    reshape(1, -1) )
98          vy= sdds[area]['vy_model'].predict(np.array([omega]))
99
100         if np.isnan(vy).any():
101             vy = -omega*0.5
102
103         for i in range(0,10):
104             X_dot = cos(theta)*vx - sin(theta)*vy
105             x += _delta *X_dot
106             Y_dot = sin(theta)*vx + cos(theta)*vy
107             y += _delta * Y_dot
108             theta += _delta * omega
109         return descretize((x[0],y[0],theta[0]))
110
111 def update_robot_model():
112     for key in sdds.keys():
```

```
113         X_data =  np.array( [  sdds[key]["omega_l"] , sdds[key]["omega_r"] ] ).
    transpose()
114         Vx_data =  np.array(  [sdds[key]["v_x"] ] ).transpose().ravel()
115         Vy_data = np.array(  [sdds[key]["v_y"] ] ).transpose().ravel()
116         Omega_data =  np.array(  [sdds[key]["omega"] ] ).transpose().ravel()
117 
118         X_train, X_test, y_train, y_test = train_test_split(X_data, Vx_data,
    test_size=0.7)
119         clf_vx = BayesianRidge(compute_score=True)
120         clf_vx.fit(X_train, y_train)
121         sdds[key]["vx_model_score"].append(clf_vx.score(X_test,y_test) )
122 
123         X_train, X_test, y_train, y_test = train_test_split(X_data, Omega_data,
    test_size=0.7)
124         clf_omega = BayesianRidge(compute_score=True)
125         clf_omega.fit(X_train, y_train)
126         sdds[key]["omega_model_score"].append(clf_omega.score(X_test,y_test)  )
127 
128 
129 
130         X_train, X_test, y_train, y_test = train_test_split(Omega_data.reshape
    (-1,1), Vy_data, test_size=0.7)
131 
132         X_train = Omega_data.reshape(-1,1)
133         y_train = Vy_data
134 
135         clf_vy = BayesianRidge(compute_score=True)
136         clf_vy.fit (X_train,y_train )
137         sdds[key]["vy_model_score"].append(clf_vy.score(np.nan_to_num(X_test),np.
    nan_to_num(y_test)))
138 
139 
140         sdds[key]["vx_model"] = clf_vx
141         sdds[key]["vy_model"] = clf_vy
142         sdds[key]["omega_model"] = clf_omega
143 
144 
145 def estimate_pose(discrete_state):
146     pos_x= (discrete_state[0][0]/24)*0.62
147     pos_y= (discrete_state[0][1]/24)*0.505
148     theta= (discrete_state[1]/10)*2*np.pi
149     return (pos_x,pos_y,theta)
150 
151 def predict_robot_loc(mobile_robot):
152     if mobile_robot.get_discretized_state_large_level()[0:2] in sdds.keys():
153         state = mobile_robot.get_discretized_state_large_level()[0:2]
154         omega_l = mobile_robot.omega_l
155         omega_r = mobile_robot.omega_r
156         vx = sdds[state]['vx_model'].predict(np.array([omega_l,omega_r]).reshape(1,
     -1) )
157         omega = sdds[state]['omega_model'].predict(np.array([omega_l,omega_r]).
    reshape(1, -1) )
158         vy= sdds[state]['vy_model'].predict(np.array([omega]))
159         x = mobile_robot.pos_x
160         y = mobile_robot.pos_y
161         theta = mobile_robot.theta
162 
163         for i in range(0,10):
164             X_dot = cos(theta)*vx - sin(theta)*vy
165             x += mobile_robot._delta *X_dot
166             Y_dot = sin(theta)*vx + cos(theta)*vy
167             y += mobile_robot._delta * Y_dot
168             theta += mobile_robot._delta * omega
```

```
169
170         return (x[0],y[0],theta[0])
171
172
173
174 def trajectory_packup():
175
176
177     for pair in best_experience:
178         state_prev = pair[0]
179         action = pair[1]
180         reward= pair[2]
181         state = pair[3]
182         qlearn.learn(state_prev,action,reward,state)
183
184
185 env.seed(1)
186
187 qlearn = QLearn(actions=range(4), alpha=1 , gamma = 0.9 , epsilon = 0.9)
188
189 initial_epsilon = qlearn.epsilon
190 epsilon_discount = 0.99
191 convergence_list = []
192
193 world_model = {}
194 sdds={}
195 best_experience  = []
196 comparator = {}
197 obstacles = []
198
199 state_predicted = None
200 max_trials = 500
201
202 success_rate = 0
203
204 predicted=[]
205 actual=[]
206 cum_reward = []
207 q = []
208 failure_list = []
209 done = True
210 predictions_checked = {}
211
212
213 for episode in range(200):
214
215     if not done:
216         failure_list.append ((episode,trials))
217
218     done= False
219     if episode>0:
220         convergence_list.append(trials)
221         cum_reward.append(reward_sum)
222         update_robot_model()
223         #plot_prediction(episode)
224     trials = 0
225     observation, robot = env.reset()
226
227     if qlearn.epsilon>0.01:
228         #qlearn.epsilon *=epsilon_discount
229         pass
230
231
```

```
232     state_prev = get_state(observation,done)
233     state_prev= (state_prev,0)
234     experience  = []
235     x_actual = []
236     y_actual = []
237     theta_actual = []
238 
239     x_predicted = []
240     y_predicted = []
241     theta_predicted = []
242 
243 
244     predictions_checked[episode] = 0
245 
246     reward_sum = 0
247 
248 
249 
250 
251     if not len(best_experience)==0:
252         max_trials = len(best_experience)
253         pass
254 
255 
256 
257 
258 
259 
260 
261     for timestep in range(max_trials+1):
262         trials +=1
263 
264 
265         #real experience
266         action = qlearn.chooseAction(state_prev)
267 
268 
269 
270         if not len(convergence_list)==0:
271             if robot.get_discretized_state_large_level()[0:2] in sdds.keys():
272                 if "vx_model" in sdds[robot.get_discretized_state_large_level()
    [0:2]].keys():
273                     state_predicted=  predict_robot_loc(robot)
274                     x_predicted.append(state_predicted[0])
275                     y_predicted.append(state_predicted[1])
276                     theta_predicted.append(state_predicted[2])
277 
278         observation, orientation, reward, done, sdds, robot = env.step(action)
279         reward_sum+= reward
280 
281 
282         state = get_state(observation,done)
283         state = (state,orientation)
284         experience.append((state_prev,action,reward,state))
285 
286         if reward == -10 or reward == -30:
287             obstacles.append(state[0])
288 
289         if not state_predicted == None:
290                 x_actual.append(robot.pos_x)
291                 y_actual.append(robot.pos_y)
292                 theta_actual.append(robot.theta)
293 
```

```
294         qlearn.learn(state_prev,action,reward,state)
295 
296         #hypothetical experience
297 
298         if qlearn.epsilon>0.2:
299             update_world_model(state_prev,state,reward,action)
300             hypothetical_experience(10)
301 
302         if trials +1 == max_trials:
303             if qlearn.epsilon>0.01:
304                 qlearn.epsilon = 0.99*qlearn.epsilon
305 
306 
307 
308         if not(done):
309             state_prev= state
310         else:
311 
312 
313             plot_v(qlearn.v,episode)
314             print("Episode: ",episode," ","Epsilone: ",qlearn.epsilon,"Trials: ",
    trials)
315             success_rate +=1
316             if len(convergence_list) == 0:
317                 convergence_list.append(trials)
318                 best_experience = experience
319             else:
320                 if trials < min(convergence_list):
321                     best_experience = experience
322                     best_experience.reverse()
323                     if qlearn.epsilon>0.01:
324                         qlearn.epsilon = 0.99*qlearn.epsilon
325                qlearn.learn(state_prev,action,reward,state)
326 
327                     #print(qlearn.q[pair])
328             print(len(best_experience))
329 
330 
331             break
```

A.1.5 Plotting codes

```
1 
2 
3 def plot_v(v,episode):
4     pyplot.figure()
5     cmap2 = mpl.colors.LinearSegmentedColormap.from_list('my_colormap',
6                                             ['blue','red'],
7                                             256)
8     img2 = pyplot.imshow(v,interpolation='nearest',
9                          cmap = cmap2,
10                         origin='lower')
11    x= []
12    y= []
13    experience.pop(0)
14    for movement in experience:
15 
16        state = movement[0][0]
17        x.append(25- state[0])
18        y.append(state[1])
19 
20 
21    pyplot.plot(y, x, 'ko-')
```

```
22
23      pyplot.colorbar(img2,cmap=cmap2)
24      pyplot.show()
25
26      script_dir = os.path.dirname(__file__)
27      results_dir = os.path.join(script_dir, 'learnedmap_stochastic_model/')
28
29      pyplot.savefig( results_dir+'value'+'_'+str(episode+1)+'.png', dpi=None,
        facecolor='w', edgecolor='w',
30          orientation='portrait', papertype=None, format=None,
31          transparent=False, bbox_inches=None, pad_inches=0.1,
32          frameon=None)
33      pyplot.close()
```

A.2 Real-time codes

A.2.1 Arduino code

```
1
2  // 11 , 12      and 39 , 40
3
4  #define enA 6
5  #define in1A 22
6  #define in2A 26
7  #define enB 4
8  #define in1B 34
9  #define in2B 30
10
11
12 long count_r = 0;
13 long angle_r = 0;
14 volatile int A_r,B_r;
15 byte state_r, statep_r;
16
17 long count_l = 0;
18 long angle_l = 0;
19 volatile int A_l,B_l;
20 byte state_l, statep_l;
21
22
23 char command;
24
25 void setup() {
26   // put your setup code here, to run once:
27   pinMode(enA, OUTPUT);
28   pinMode(in1A, OUTPUT);
29   pinMode(in2A, OUTPUT);
30
31   pinMode(enB, OUTPUT);
32   pinMode(in1B, OUTPUT);
33   pinMode(in2B, OUTPUT);
34
35
36   pinMode(2, INPUT);
37   pinMode(3, INPUT);
38   pinMode(18, INPUT_PULLUP);
39   pinMode(19, INPUT_PULLUP);
40
41   attachInterrupt(digitalPinToInterrupt(18),Achange_r,CHANGE);//interrupt pins are
       declared here
42   attachInterrupt(digitalPinToInterrupt(19),Bchange_r,CHANGE);
43   attachInterrupt(digitalPinToInterrupt(2),Achange_l,CHANGE);//interrupt pins are
       declared here
```

```
44   attachInterrupt(digitalPinToInterrupt(3),Bchange_l,CHANGE);
45
46   Serial.begin(115200);//for debugging
47
48   Serial3.begin(9600);//Initialize communications with the bluetooth module
49   Serial3.println("Ready!!!");//Send something to just start comms. This will never
        be seen.
50
51
52 }
53
54 void loop() {
55   // put your main code here, to run repeatedly:
56
57   angle_r = count_r * 0.0493827;//. count to angle conversion
58   angle_l = count_l * 0.0493827;//. count to angle conversion
59
60     if (Serial3.available())//If there are any bytes then deal with them
61   {
62
63     command = Serial3.read() ;
64     Serial.println(command);
65     Serial3.println(0);
66
67     if (command == 'F'){
68       moveForward();
69       }
70     if (command == 'B'){
71       moveBackward();
72       }
73     if (command == 'L'){
74       moveLeft();
75       }
76      if (command == 'R'){
77       moveRight();
78       }
79
80  //    Serial3.println("Angle_r: " +String(angle_r) + "  Angle_l: "+ String(angle_l)
       );
81     Serial3.println(1);
82     }
83
84
85
86  brake();
87
88
89
90 }
91
92
93 void brake(){
94
95     digitalWrite(in1A, LOW);
96   digitalWrite(in2A, LOW);
97
98
99   digitalWrite(in1B, LOW);
100  digitalWrite(in2B, LOW);
101
102  delay(70);
103
104 }
```

```
105
106
107 void moveLeft(){
108
109   digitalWrite(in1A, LOW);
110   digitalWrite(in2A, HIGH);
111   analogWrite(enA, 200); // Send PWM signal to L298N Enable pin
112
113
114   digitalWrite(in1B, LOW);
115   digitalWrite(in2B, HIGH);
116   analogWrite(enB, 200); // Send PWM signal to L298N Enable pin
117
118 delay(500);
119 }
120
121
122 void moveRight(){
123
124   digitalWrite(in1A, HIGH);
125   digitalWrite(in2A, LOW);
126   analogWrite(enA, 200); // Send PWM signal to L298N Enable pin
127
128
129   digitalWrite(in1B, HIGH);
130   digitalWrite(in2B, LOW);
131   analogWrite(enB, 200); // Send PWM signal to L298N Enable pin
132
133 delay(500);
134
135 }
136
137 void moveForward(){
138
139   digitalWrite(in1A, HIGH);
140   digitalWrite(in2A, LOW);
141   analogWrite(enA, 200); // Send PWM signal to L298N Enable pin
142
143
144   digitalWrite(in1B, LOW);
145   digitalWrite(in2B, HIGH);
146   analogWrite(enB, 200); // Send PWM signal to L298N Enable pin
147
148 delay(500);
149
150 }
151
152 void moveBackward(){
153
154   digitalWrite(in1A, LOW);
155   digitalWrite(in2A, HIGH);
156   analogWrite(enA, 200); // Send PWM signal to L298N Enable pin
157
158
159   digitalWrite(in1B, HIGH);
160   digitalWrite(in2B, LOW);
161   analogWrite(enB, 200); // Send PWM signal to L298N Enable pin
162
163 delay(500);
164
165 }
166
167 void Achange_r()
```

```
{
  A_r = digitalRead(18);
  B_r = digitalRead(19);

  if ((A_r==HIGH)&&(B_r==HIGH)) state_r = 1;// switch...case method used here
  if ((A_r==HIGH)&&(B_r==LOW)) state_r = 2;
  if ((A_r==LOW)&&(B_r==LOW)) state_r = 3;
  if((A_r==LOW)&&(B_r==HIGH)) state_r = 4;
  switch (state_r)
  {
    case 1:
    {
      if (statep_r == 2) count_r++;
      if (statep_r == 4) count_r--;
      break;
    }
    case 2:
    {
      if (statep_r == 1) count_r--;
      if (statep_r == 3) count_r++;
      break;
    }
    case 3:
    {
      if (statep_r == 2) count_r --;
      if (statep_r == 4) count_r ++;
      break;
    }
    default:
    {
      if (statep_r == 1) count_r++;
      if (statep_r == 3) count_r--;
    }
  }
  statep_r = state_r;
}

void Bchange_r()
{
  A_r = digitalRead(18);
  B_r = digitalRead(19);

  if ((A_r==HIGH)&&(B_r==HIGH)) state_r = 1;
  if ((A_r==HIGH)&&(B_r==LOW)) state_r = 2;
  if ((A_r==LOW)&&(B_r==LOW)) state_r = 3;
  if((A_r==LOW)&&(B_r==HIGH)) state_r = 4;
  switch (state_r)
  {
    case 1:
    {
      if (statep_r == 2) count_r++;
      if (statep_r == 4) count_r--;
      break;
    }
    case 2:
    {
      if (statep_r == 1) count_r--;
      if (statep_r == 3) count_r++;
      break;
    }
    case 3:
    {
      if (statep_r == 2) count_r --;
```

```
231       if (statep_r == 4) count_r ++;
232       break;
233     }
234     default:
235     {
236       if (statep_r == 1) count_r++;
237       if (statep_r == 3) count_r--;
238     }
239   }
240   statep_r = state_r;
241 }
242
243
244 void Achange_l()
245 {
246   A_l = digitalRead(2);
247   B_l = digitalRead(3);
248
249   if ((A_l==HIGH)&&(B_l==HIGH)) state_l = 1;// switch...case method used here
250   if ((A_l==HIGH)&&(B_l==LOW)) state_l = 2;
251   if ((A_l==LOW)&&(B_l==LOW)) state_l = 3;
252   if((A_l==LOW)&&(B_l==HIGH)) state_l = 4;
253   switch (state_l)
254   {
255     case 1:
256     {
257       if (statep_l == 2) count_r++;
258       if (statep_l == 4) count_l--;
259       break;
260     }
261     case 2:
262     {
263       if (statep_l == 1) count_l--;
264       if (statep_l == 3) count_l++;
265       break;
266     }
267     case 3:
268     {
269       if (statep_l == 2) count_l --;
270       if (statep_l == 4) count_l ++;
271       break;
272     }
273     default:
274     {
275       if (statep_l == 1) count_l++;
276       if (statep_l == 3) count_l--;
277     }
278   }
279   statep_l = state_l;
280 }
281
282 void Bchange_l()
283 {
284   A_l = digitalRead(2);
285   B_l = digitalRead(3);
286
287   if ((A_l==HIGH)&&(B_l==HIGH)) state_l = 1;
288   if ((A_l==HIGH)&&(B_l==LOW)) state_l = 2;
289   if ((A_l==LOW)&&(B_l==LOW)) state_l = 3;
290   if((A_l==LOW)&&(B_l==HIGH)) state_l = 4;
291   switch (state_l)
292   {
293     case 1:
```

```
294     {
295       if (statep_l == 2) count_l++;
296       if (statep_l == 4) count_l--;
297       break;
298     }
299     case 2:
300     {
301       if (statep_l == 1) count_l--;
302       if (statep_l == 3) count_l++;
303       break;
304     }
305     case 3:
306     {
307       if (statep_l == 2) count_l --;
308       if (statep_l == 4) count_l ++;
309       break;
310     }
311     default:
312     {
313       if (statep_l == 1) count_l++;
314       if (statep_l == 3) count_l--;
315     }
316   }
317   statep_l = state_l;
318 }
```

A.2.2 Camera Calibration

```
1
2  import numpy as np
3  import cv2
4  import glob
5
6  # termination criteria
7  criteria = (cv2.TERM_CRITERIA_EPS + cv2.TERM_CRITERIA_MAX_ITER, 30, 0.001)
8
9  # prepare object points, like (0,0,0), (1,0,0), (2,0,0) ....,(6,5,0)
10 objp = np.zeros((4*5,3), np.float32)
11 objp[:,:2] = np.mgrid[0:5,0:4].T.reshape(-1,2)
12
13 # Arrays to store object points and image points from all the images.
14 objpoints = [] # 3d point in real world space
15 imgpoints = [] # 2d points in image plane.
16
17 images = glob.glob('calibration_images\*.jpg')
18
19 for fname in images:
20     img = cv2.imread(fname)
21     gray = cv2.cvtColor(img,cv2.COLOR_BGR2GRAY)
22
23     # Find the chess board corners
24     ret, corners = cv2.findChessboardCorners(gray, (5,4),None)
25
26     # If found, add object points, image points (after refining them)
27     if ret == True:
28
29         objpoints.append(objp)
30         corners2 = cv2.cornerSubPix(gray,corners,(11,11),(-1,-1),criteria)
31         imgpoints.append(corners2)
32
33         # Draw and display the corners
34         img = cv2.drawChessboardCorners(img, (5,4), corners2,ret)
35         cv2.imshow('img',img)
```

```
        cv2.waitKey(1000)
        print("true")


cv2.destroyAllWindows()

ret, mtx, dist, rvecs, tvecs = cv2.calibrateCamera(objpoints, imgpoints, gray.shape
    [::-1],None,None)



np.savez("B",  mtx=mtx, dist=dist, rvecs=rvecs, tvecs=tvecs)
```

```

class camera():

    def __init__(self, camera_ref = 1 ):

        with np.load('B.npz') as X:
            self.mtx, self.dist, _, _ = [X[i] for i in ('mtx','dist','rvecs','tvecs
    ')]

        self.cap = cv2.VideoCapture(camera_ref)
        self.criteria = (cv2.TERM_CRITERIA_EPS + cv2.TERM_CRITERIA_MAX_ITER, 30,
    0.001)
        self.cart_objp = np.zeros((4*5,3), np.float32)
        self.cart_objp[:,:2] = np.mgrid[0:5,0:4].T.reshape(-1,2)
        self.goal_objp = np.zeros((6*9,3), np.float32)
        self.goal_objp[:,:2] = np.mgrid[0:9,0:6].T.reshape(-1,2)
        self.axis = np.float32([[3,0,0], [0,3,0], [0,0,-3]]).reshape(-1,3)
        self.fourcc = cv2.VideoWriter_fourcc(*'XVID')
        self.out = cv2.VideoWriter('output.avi',self.fourcc, 20.0, (640,480))


    def find_goal(self):
        while(True):
            ret, frame = self.cap.read()
            gray = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)
            ret_, corners_ = cv2.findChessboardCorners(gray, (9,6),None)

            if ret_ :
                corners= cv2.cornerSubPix(gray,corners_,(11,11),(-1,-1),self.
    criteria)
                _, rvecs, tvecs, inliers = cv2.solvePnPRansac(self.goal_objp,
    corners,self.mtx, self.dist)

                return rvecs,tvecs

    def observe(self):
        xmin = -16
        xmax= 15.5
        ymin = -10.8
        ymax = 13.6

        while(True):
            ret, frame = self.cap.read()

            gray = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)
#            a = np.double(gray)
#            gray = a - 10
#            gray= np.uint8(gray)
#
            if ret:
```

```
47                 frame = cv2.flip(frame,0)
48                 self.out.write(frame)
49
50
51 #             corners = []
52             ret_, corners_ = cv2.findChessboardCorners(gray, (5,4),None)
53 #             corners.append(corners_)
54 #             corners_ = random.choice(corners)
55             if ret_ :
56                 corners= cv2.cornerSubPix(gray,corners_,(11,11),(-1,-1),self.
    criteria)
57                 _, rvecs, tvecs, inliers = cv2.solvePnPRansac(self.cart_objp,
    corners,self.mtx, self.dist)
58                 x = floor(tvecs[0]*20/(xmax-xmin))
59                 y = floor(tvecs[1]*20/(ymax-ymin))
60                 theta = np.arctan2(np.sin(rvecs[2]), np.cos(rvecs[2]))
61                 theta = floor(theta/(2*np.pi) *20)
62
63
64
65                 cv2.imshow('frame',gray)
66                 if cv2.waitKey(1) & 0xFF == ord('q'):
67                     break
68                 return(x,y,theta)
69
70     def get_exact_state(self):
71
72         while(True):
73             ret, frame = self.cap.read()
74
75             gray = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)
76
77 #             corners = []
78             ret_, corners_ = cv2.findChessboardCorners(gray, (5,4),None)
79 #             corners.append(corners_)
80 #             corners_ = random.choice(corners)
81             if ret_ :
82                 corners= cv2.cornerSubPix(gray,corners_,(11,11),(-1,-1),self.
    criteria)
83                 _, rvecs, tvecs, inliers = cv2.solvePnPRansac(self.cart_objp,
    corners,self.mtx, self.dist)
84                 x = tvecs[0]
85                 y = tvecs[1]
86                 theta = np.arctan2(np.sin(rvecs[2]), np.cos(rvecs[2]))
87                 return(x,y,theta)
88
89     def conf(self):
90         while(True):
91             ret, frame = self.cap.read()
92             gray = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)
93             ret_, corners_ = cv2.findChessboardCorners(gray, (5,4),None)
94             print(ret_)
95             if ret_ :
96                 corners= cv2.cornerSubPix(gray,corners_,(11,11),(-1,-1),self.
    criteria)
97                 _, rvecs, tvecs, inliers = cv2.solvePnPRansac(self.cart_objp,
    corners,self.mtx, self.dist)
98
99                 self.out.write(gray)
100                cv2.imshow('frame',gray)
101                if cv2.waitKey(1) & 0xFF == ord('q'):
102                    break
103
```

```
104
105                 return( rvecs, tvecs)
106                 #return rvecs[2]*180/np.pi
107     def measure_orientation(self):
108         while(True):
109             ret, frame = self.cap.read()
110             gray = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)
111             ret_, corners_ = cv2.findChessboardCorners(gray, (5,4),None)
112             if ret_ :
113                 corners= cv2.cornerSubPix(gray,corners_,(11,11),(-1,-1),self.
    criteria)
114                 _, rvecs, tvecs, inliers = cv2.solvePnPRansac(self.cart_objp,
    corners,self.mtx, self.dist)
115                 return rvecs[2]*180/np.pi
116
117
118     def release(self):
119         self.cap.release()
120         self.out.release()
121         cv2.destroyAllWindows()
```

A.2.3 Communication server class

```
1
2 class arduino_server:
3
4     def __init__(self):
5         self.port = "COM6"
6         self.bluetooth=serial.Serial(self.port, 9600, timeout = None )
7         self.bluetooth.flushInput()
8
9
10
11     def close(self):
12         self.conn.close()
13
14     def step(self,letter):
15         self.bluetooth.write(letter.encode())
16         while not self.isReady():
17             pass
18         return
19
20     def isReady(self):
21         input_data=self.bluetooth.readline()
22         str_ = input_data.decode(encoding='UTF-8',errors='ignore')
23         if len(str_) ==3:
24             if int(str_)== 1:
25                 return True
26         print("notReady")
27         return False
```

A.2.4 Real environment class

```
1
2 class real_env:
3     def __init__(self,camera_no=1):
4         self.camera = camera(camera_no)
5         self.server = arduino_server()
6         self.targets = [ ]
7         self.distance = 1000
8         self.count = 0
9
```

```
10
11
12          self.target = (-12,10)
13          for i_x in range(10):
14              for i_y in range(10):
15                  self.targets.append((self.target[0]+i_x,self.target[1]-i_y))
16
17          self.terminal = False
18          self.done = False
19          self.get_state()
20          self.state_prev = (-10,-10,-10)
21          self.state_prev_ = (-10,-10,-10)
22
23          xmin = -16
24          xmax= 15.5
25          ymin = -10.8
26          ymax = 13.6
27
28          self.x_min_discrete = floor((xmin+1)*20/(xmax-xmin))
29          self.y_min_discrete = floor((ymin+1)*20/(ymax-ymin))
30          self.x_max_discrete = floor((xmax-1)*20/(xmax-xmin))
31          self.y_max_discrete = floor((ymax-1)*20/(ymax-ymin))
32          self.obstacles = []
33
34
35
36
37      def get_state(self):
38          self.state = self.camera.observe()
39          self.state_ = (self.camera.get_exact_state()[0].item(0),
40                         self.camera.get_exact_state()[1].item(0),
41                         self.camera.get_exact_state()[2].item(0))
42
43      def reset(self):
44          self.terminal = False
45          self.rotate_to(0.2)
46          for i in range(0,8):
47              self.server.step("B")
48              self.rotate_to(0)
49          self.rotate_to(-1.5759)
50          for i in range(8):
51              self.server.step("B")
52              self.rotate_to(-1.5759)
53          self.get_state()
54
55      def rotate_to(self,theta_ref):
56          while True:
57              self.get_state()
58              theta_current  = env.state_[2]
59              print(theta_current)
60              e_theta = theta_current - theta_ref
61              if abs(e_theta) <= 0.3 :
62                  return
63              elif e_theta < 0 :
64                  self.server.step("R")
65              elif e_theta >0 :
66                  self.server.step("L")
67
68
69      def step_return(self,reward):
70          self.count =  0
71          return self.state,reward,self.terminal,"",self.prev_state_,self.state_
72
```

```
73      def augment_obstacle(self,state):
74          x= state[0]
75          y = state[1]
76          for i_y in range(-1,2):
77              for i_x in range(-1,2):
78                  env.obstacles.append((x+i_x,y+i_y))
79
80
81
82      def step(self,a):
83          self.state_prev_prev = self.state_prev
84          self.state_prev = self.state
85          self.prev_state_ = self.state_
86
87          print(self.state)
88          if self.terminal:
89              return self.step_return(10)
90
91          assert 0 <= a and a < 4
92          if a ==0 :
93              self.server.step("F")
94              print("F")
95          elif a ==1:
96              self.server.step("R")
97              print("R")
98          elif a==2:
99              self.server.step("L")
100             print("L")
101         elif a==3:
102             self.server.step("B")
103             print("B")
104
105         self.get_state()
106
107         if self.state[0:2] in self.targets:
108             self.terminal = True
109             return self.step_return(10)
110
111         orienation_reward = calculate_orientation_reward(self.state,2)
112
113         if self.state[0:2] == self.state_prev[0:2]:
114             self.obstacles.append(self.state[0:2])
115
116
117
118         if self.state[0] <= self.x_min_discrete or self.state[1] <= self.
        y_min_discrete or self.state[0] >= self.x_max_discrete or self.state[1] >=
        self.y_max_discrete or self.state[0:2] in self.obstacles:
119             return self.step_return(-1 +orienation_reward)
120
121
122         self.obstacle = False;
123
124         return self.step_return(orienation_reward + 1/math.pow((abs(state[0]-env.
        target[0]) + abs(state[1]-env.target[1])),2))
125
126 def calculate_orientation_reward(state,counter):
127     if counter == 0:
128         return 0
129     current_orienation = state[2]
130     orienation_reward = 0
131     for i in range(current_orienation-6, current_orienation+7):
132             state_prev = state
```

```
            for action in range(1):
                if (state_prev,action) in world_model.keys():
                        index = random.choice(range(0,len(world_model.get((
    state_prev,action))['next_state'])))
                        state = world_model.get((state_prev,action))['next_state'][
    index]
                        reward = calculate_position_reward(state)
                        orienation_reward += reward+calculate_orientation_reward(
    state,counter-1)
    return orienation_reward

def calculate_position_reward(state):
    if state[0:2] in env.obstacles:
        return(-1)
    elif state[0:2] in env.targets:
        return(10)
    return 1/math.pow((abs(state[0]-env.target[0]) + abs(state[1]-env.target[1]))
    ,2)
```

A.2.5 Reinforcement learning training

```

def update_world_model(state_prev,state,reward,action):
    if (state_prev,action) in world_model.keys():
        world_model[(state_prev,action)]['next_state'].append(state)
        world_model[(state_prev,action)]['reward'].append(reward)
        world_model[(state_prev,action)]['number'] += 1
    else:
        world_model[(state_prev,action)]={}
        world_model[(state_prev,action)]['next_state'] = []
        world_model[(state_prev,action)]['next_state'].append(state)
        world_model[(state_prev,action)]['reward'] = []
        world_model[(state_prev,action)]['reward'].append(reward)
        world_model[(state_prev,action)]['number'] = 1

def hypothetical_experience(k):
    keys = list(world_model.keys())
    for i in range(0,k+1):
        state_prev = (random.choice(keys))[0]
        action = qlearn.chooseAction(state_prev)
        if (state_prev,action) in world_model.keys():
            index = random.choice(range(0,len(world_model.get((state_prev,action))[
    'reward'])))
            reward = world_model.get((state_prev,action))['reward'][index]
            state = world_model.get((state_prev,action))['next_state'][index]
            position_reward = calculate_position_reward(state)
            orientation_reward = calculate_orientation_reward(state,2)
            reward= position_reward+orientation_reward
            qlearn.learn(state_prev,action,reward,state)

def plan(env):
    x_target = env.target[0]
    y_target = env.target[1]
    for x_current in range (-20,20):
        for y_current in range(-20,20):
                for theta_current in range(-50,50):
                    theta_ref = np.arctan2(( y_target- y_current), -1*(x_target-
    x_current) )
                    theta_ref = floor( -1*theta_ref/(2*np.pi) *20)
                    e_theta = theta_current - theta_ref

                    if abs(e_theta) == 0 :
                        qlearn.q[((x_current,y_current,theta_current),0)]= 1
```

```
    #                     planned_action[(x_current,y_current,theta_current)] = 0
                        pass
                    elif e_theta < 0 :
                        qlearn.q[((x_current,y_current,theta_current),1)]= 1
    #                     planned_action[(x_current,y_current,theta_current)] = 1
                    elif e_theta >0 :
                        qlearn.q[((x_current,y_current,theta_current),2)]= 1
    #                     planned_action[(x_current,y_current,theta_current)] = 2

def trajectory_packup():
    for pair in best_experience:
        state_prev = pair[0]
        action = pair[1]
        reward= pair[2]
        state = pair[3]
        position_reward = calculate_position_reward(state)
        orientation_reward = calculate_orientation_reward(state,2)
        reward= position_reward+orientation_reward
        qlearn.learn(state_prev,action,reward,state)

env = real_env(1)
qlearn = QLearn(actions=range(3), alpha=0.99 , gamma = 0.99 , epsilon = 0.4)


nitial_epsilon = qlearn.epsilon
initial_alpha = 1
epsilon_discount = 0.9
alpha_discount = 0.9
world_model_inverse = {}
n = 4
explore_factor = 0.001
action_prev = 0

world_model = {}
best_experience = []
cum_reward = []
convergence_list = []
plan(env)
trajectories = []
done = False
max_trials = 300
env.reset()


for episode in range(190):


    applied_actions = {}
    for x_current in range (-20,20):
        for y_current in range(-20,20):
            for theta_current in range(-50,50):
                applied_actions[(x_current,y_current,theta_current)]= []

    if len(best_experience)>0:
            max_trials = np.mean(convergence_list)

    if episode > 0 :
        convergence_list.append(trials)
        cum_reward.append(reward_sum)
        trajectories.append(experience)

    trials = 0
    if episode == 0:
```

```
104         env.get_state()
105     else:
106         if done:
107             env.get_state()
108             env.reset()
109             done= False
110         else:
111             env.get_state()
112     state_prev=env.state
113     state_prev_prev= env.state_prev
114     state_prev_ = env.state_
115 
116     if qlearn.epsilon>0.3:
117         qlearn.epsilon *=epsilon_discount
118     if qlearn.alpha > 0.9 :
119         qlearn.alpha *=alpha_discount
120 
121     experience  = []
122     counter = 0
123     reward_sum = 0
124 
125     for i in range(max_trials):
126         trials +=1
127         action = qlearn.chooseAction(state_prev)
128         applied_actions[state_prev].append(action)
129         last_two_actions = applied_actions[state_prev][-4:-1]
130         print(last_two_actions)
131         if len(last_two_actions )== 3:
132             if last_two_actions [1:] == last_two_actions[:-1]:
133                 rand = random.randint(0,1)
134                 if rand == 0:
135                     env.rotate_to(-1.5759)
136                     action = 0
137                 else:
138                     env.rotate_to(0)
139                     action = 0
140 
141 
142 
143 
144 
145         for action_ in range(0,4):
146             if (state_prev,action_) in qlearn.q.keys():
147 
148                 print(str((state_prev,action_)) + " : "+str(qlearn.q[(state_prev,
    action_)]) )
149 
150 
151 
152         state, reward, done, _ , state_prev_,state_ = env.step(action)
153         reward_sum+= reward
154         update_world_model(state_prev,state,reward,action)
155         print( "reward: " + str(reward))
156 
157         if reward == -10 :
158             obstacles.append(state[0:2])
159 
160 
161         experience.append((state_prev,action,reward,state))
162 
163         qlearn.learn(state_prev,action,reward,state)
164 
165 
```

```
166
167
168
169         #hypothetical experience
170
171         hypothetical_experience(100)
172
173
174
175
176         trajectory_packup()
177         print(done)
178         if not(done):
179             state_prev= state
180             state_prev_prev= state_prev
181             action_prev = action
182         else:
183             print("Episode: ",episode," ","Epsilone: ",qlearn.epsilon,"Trials: ",
        trials)
184             if len(convergence_list) == 0:
185                 best_experience = experience
186             else:
187                 if trials < min(convergence_list):
188                     best_experience = experience
189                     best_experience.reverse()
190
191             break
192     else:
193         continue
```

Bibliography

[1] A. Andrieux, P. O. Vandanjon, R. Lengelle, and C. Chabanon. New results on the relation between tyre–road longitudinal stiffness and maximum available grip for motor car. *Vehicle system dynamics*, 48(12):1511–1533, 2010.

[2] P. Angelov, K. Atanassov, L. Doukovska, M. Hadjiski, V. Jotsov, J. Kacprzyk, N. Kasabov, S. Sotirov, E. Szmidt, and S. Zadrożny. *Intelligent Systems' 2014: Proceedings of the 7th IEEE International Conference Intelligent Systems IS'2014, September 24-26, 2014, Warsaw, Poland, Volume 1: Mathematical Foundations, Theory, Analyses*, volume 322. Springer, 2014.

[3] D. B. Aranibar and P. J. Alsina. Reinforcement learning-based path planning for autonomous robots. In *EnRI-XXIV Congresso da Sociedade Brasileira de Computaç ao*, page 10, 2004.

[4] B. Bakker, V. Zhumatiy, G. Gruener, and J. Schmidhuber. Quasi-online reinforcement learning for robots. In *Robotics and Automation, 2006. ICRA 2006. Proceedings 2006 IEEE International Conference on*, pages 2997–3002. IEEE, 2006.

[5] H. R. Beom and H. S. Cho. A sensor-based navigation for a mobile robot using fuzzy logic and reinforcement learning. *IEEE transactions on Systems, Man, and Cybernetics*, 25(3):464–477, 1995.

[6] H. Borchani, G. Varando, C. Bielza, and P. Larrañaga. A survey on multi-output regression. *Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery*, 5(5):216–233, 2015.

[7] M. B. Christopher. *Pattern Recognition and Machine Learning*. Springer-Verlag New York, 2016.

[8] M. P. Deisenroth, G. Neumann, J. Peters, et al. A survey on policy search for robotics. *Foundations and Trends® in Robotics*, 2(1–2):1–142, 2013.

[9] M. J. Er and C. Deng. Obstacle avoidance of a mobile robot using hybrid learning approach. *IEEE transactions on industrial electronics*, 52(3):898–905, 2005.

[10] M. J. Er and Y. Zhou. Automatic generation of fuzzy inference systems via unsupervised learning. *Neural Networks*, 21(10):1556–1566, 2008.

[11] M. A. K. Jaradat, M. Al-Rousan, and L. Quadan. Reinforcement based mobile robot navigation in dynamic environment. *Robotics and Computer-Integrated Manufacturing*, 27(1):135–149, 2011.

[12] J. Kober, J. A. Bagnell, and J. Peters. Reinforcement learning in robotics: A survey. *The International Journal of Robotics Research*, 32(11):1238–1274, 2013.

[13] K. Kozłowski and D. Pazderski. Modeling and control of a 4-wheel skid-steering mobile robot. *International journal of applied mathematics and computer science*, 14:477–496, 2004.

[14] M. Kuss. *Gaussian process models for robust regression, classification, and reinforcement learning*. PhD thesis, Technische Universität, 2006.

[15] M. Kuss and C. E. Rasmussen. Gaussian processes in reinforcement learning. In *Advances in neural information processing systems*, pages 751–758, 2004.

[16] S. Levine. Deep reinforcement learning course.

[17] K. Macek, I. PetroviC, and N. Peric. A reinforcement learning approach to obstacle avoidance of mobile robots. In *Advanced Motion Control, 2002. 7th International Workshop on*, pages 462–466. IEEE, 2002.

[18] S. Mafrica. Advance Modeling of a Skid-Steering Mobile Robot for Remote Telepresence. Master's thesis, Universit'a degli Studi di Genova, Roger Williams University, 2011/2012.

[19] S. L. Miller, B. Youngberg, A. Millie, P. Schweizer, and J. C. Gerdes. Calculating longitudinal wheel slip and tire parameters using gps velocity. In *American Control Conference, 2001. Proceedings of the 2001*, volume 3, pages 1800–1805. IEEE, 2001.

[20] A. Y. Ng, D. Harada, and S. Russell. Policy invariance under reward transformations: Theory and application to reward shaping. In *ICML*, volume 99, pages 278–287, 1999.

[21] J.-J. Park, J.-H. Kim, and J.-B. Song. Path planning for a robot manipulator based on probabilistic roadmap and reinforcement learning. *International Journal of Control, Automation, and Systems*, 5(6):674–680, 2007.

[22] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[23] J. Peng and R. J. Williams. Efficient learning and planning within the dyna framework. *Adaptive Behavior*, 1(4):437–454, 1993.

[24] G. C. M. C. W. C. C. C. A. G. P. G. H. H. J. K. B. K. J. L. A. M. K. O. M. P. D. P. K. P. J. S. S. S. L. T. A. Y. S. Amershi, N. Arksey and R. Yuen. Tools for learning artificial intelligence.

[25] W. D. Smart and L. P. Kaelbling. Practical reinforcement learning in continuous spaces. In *ICML*, pages 903–910, 2000.

[26] W. D. Smart and L. P. Kaelbling. Effective reinforcement learning for mobile robots. In *Robotics and Automation, 2002. Proceedings. ICRA'02. IEEE International Conference on*, volume 4, pages 3404–3410. IEEE, 2002.

[27] R. S. Sutton. Integrated architectures for learning, planning, and reacting based on approximating dynamic programming. In *Machine Learning Proceedings 1990*, pages 216–224. Elsevier, 1990.

[28] R. S. Sutton and A. G. Barto. *Reinforcement learning: An introduction*, volume 1. MIT press Cambridge, 1998.

[29] C. F. Touzet. Neural reinforcement learning for behaviour synthesis. *Robotics and Autonomous Systems*, 22(3-4):251–281, 1997.

[30] X.-T. Truong, H. T. Dinh, and C. D. Nguyen. An efficient navigation framework for autonomous mobile robots in dynamic environments using learning algorithms. *Journal of Computer Science and Cybernetics*, 33(2):107–118, 2018.

[31] A. Turiot. An openai gym implementation for the pathfinding problem.

[32] H. Wang and Y. Yu. Exploring multi-action relationship in reinforcement learning. In *Pacific Rim International Conference on Artificial Intelligence*, pages 574–587. Springer, 2016.

[33] T. Wang, Y. Wu, J. Liang, C. Han, J. Chen, and Q. Zhao. Analysis and experimental kinematics of a skid-steering wheeled robot based on a laser scanner sensor. *Sensors*, 15(5):9681–9702, 2015.

[34] G.-S. Yang, E.-K. Chen, and C.-W. An. Mobile robot navigation using neural q-learning. In *Machine Learning and Cybernetics, 2004. Proceedings of 2004 International Conference on*, volume 1, pages 48–52. IEEE, 2004.

[35] N. Yung and C. Ye. Self-learning fuzzy navigation of mobile vehicle. In *Signal Processing, 1996., 3rd International Conference on*, volume 2, pages 1465–1468. IEEE, 1996.