Master of Science Thesis in Electrical Engineering
Department of Electrical Engineering, Linköping University, 2022

Labyrinth navigation using reinforcement learning with a high fidelity simulation environment

Olle Eriksson & Axel Malmberg

LINKÖPING
UNIVERSITY

Master of Science Thesis in Electrical Engineering

Labyrinth navigation using reinforcement learning with a high fidelity simulation environment

Olle Eriksson & Axel Malmberg

LiTH-ISY-EX--22/5492--SE

Supervisor: **Daniel Arnström**
ISY, Linköpings universitet
Rikard Hagman
Combine

Examiner: **Farnaz Adib Yaghmaie**
ISY, Linköpings universitet

Division of Automatic Control
Department of Electrical Engineering
Linköping University
SE-581 83 Linköping, Sweden

Abstract

This is a master thesis on the subject of navigation and control using reinforcement learning, more specifically discrete Q-learning. The Q-learning algorithm is used to develop a steer policy from training inside of a simulation environment. The problem is to navigate a steel ball through a maze made from walls and holes. This thesis is the third thesis made revolving around this problem which allows for performance comparison with more classical control algorithms. The most successful of which is the gain scheduled LQR used to follow a splined path. The reinforcement learning derived steer policy managed at best 68 % success rate when navigating the ball from start to finish. Key features that had large impact on the policy performance when implemented in the simulation environment were response time of the physical servos and uncertainty added to the modelled forces. Compared to the performance of the LQR, which managed 46 % success rate, the reinforcement learning derived policy performs well. But with high fluctuation in performance policy to policy the control method is not a consistent solution to the problem. Future work is needed to perfect the algorithm and the resulting policy. A few interesting issues to investigate could be other formulations of disturbance implementation and training online on the physical system. Training online could allow for fine tuning of the simulation derived policy and learning how to compensate for disturbances that are difficult to model, such as bumps and warping in the labyrinth surface.

Acknowledgments

Firstly, we would like to thank Rikard Hagman. This thesis would not have been possible without his help as he supplied us with both the hardware and the idea that is the basis for this report.

Secondly, we must thank Daniel Arnström who, dispite joining in at the last leg of the race, contributed invaluable knowledge and assistance to help us drive this project home.

Lastly, but in no way least, Farnaz Adib Yaghmaie. You contributed your time and your knowledge to get us started and took on the role of both examinator and advisor which made us feel supported even in a state of uncertainty. So thank you, and congratulations!

Linköping, May 2022
Olle Eriksson & Axel Malmberg

Contents

Notation

Symbols

Notation	Clarification
c_b	Bounce coefficient
f_r	Friction coefficient
g	Gravity acceleration
J_b	Inertia of the ball
m_b	Mass of the ball
r_b	Radius of the ball

Abbreviations

Abbreviation	Clarification
HW	Hardware
LQR	Linear-Quadratic Regulator
MDP	Markov Decision Process
ML	Machine Learning
PID	Proportional Integral Derivative (Controller)
PWM	Pulse Width Modulated
ROS	Robot Operating System
RPi	Raspberry Pi
RL	Reinforcement learning

1 Introduction

This is a thesis on the subject of reinforcement learning (RL). The technique will be applied to the classic children's game the BRIO labyrinth [2] in an effort to automatically solve the challenge of navigating a steel ball through a maze made out of walls and holes. This work is a continuation of the thesis [4] where an LQR and a splined path from predetermined checkpoints were used to solve the same problem. The game is multi variable and irregularities throughout the maze are challenging for any control algorithm.

1.1 Goal

The main goal of this thesis work is to develop a steer policy using RL, specifically Q-learning [9], to reliably navigate the ball from the start to the goal without falling into holes. For the type of Q-learning that is used in this report a simulation environment for the algorithm to train in will be required. Furthermore this thesis investigates what challenges arise when bringing the steer policy from simulation to reality and explore some tactics to bridge the gap between the simulation model and the true behaviour of the system. After implementation, the performance of the RL derived steer policy will be compared with the results presented in [4].

1.2 Problem formulation

The goal of this thesis can be broken down into the following set of problem formulations that will be answered in this report:

1. **What challenges arise when using a steer policy derived from simulation?**

Whenever a simulation is designed it will be highly dependent on the model of the system. Modelling errors may be exploited by the learner, which leads to poor decisions when controlling the physical system.

2. Is it enough to add uncertainty to the motion model in order to bridge the gap between the simulation environment and the physical system? If uncertainty is added in simulation to mimic model errors and the general unpredictability of the physical system, the machine learning derived policy might perform better on the physical system.

3. What performance benefits, if any, can be observed using the Q-learning method compared to a gain scheduled LQR? The approach of using RL to generate a steer policy will be compared with the path planner and the LQR method used in [4]. It is relevant to make this comparison to find out if machine learning is a viable solution for the BRIO labyrinth game.

1.3 Related research

RL and ML are in general fields that have gotten a lot of attention during the last decades and they have been applied to a multitude of problems, classic games for example. An inspiration to this thesis was a study of how RL can be applied to solve the classic atari games [3]. In the study a deep convolutional neural network was used to find a good policy for 50 classic atari 2600 games. The neural network had access to a 60 Hz video stream of the game screen, the score and the controls to operate the game. A problem that is brought up in this study is the size of the network as a large network takes more time to train than a smaller network. In [3] the amount of training necessary to achieve a well trained policy is decreased by the use of experience replay, further explained in Section 3.1.1, in combination with a deep convolutional network. The study presents good results in several of the atari games after just 200 in game sessions. Even so, training a network for 200 sessions can be daunting if it is cumbersome to reset the game or if a failure results in danger or loss, economic or other.

This thesis will explore an alternative way of getting robust and efficient training, by development of a simulation environment in which the agent can do all or at least most of the training. A simulation environment that contains a model of the game or system has some clear advantages. It allows for faster time stepping than what is possible to achieve live which leads to more training sessions in same amount of time. Additionally, failure inside of a simulation environment will not result in any danger or loss either which in a larger context can be desirable. In [10] the method of performing extensive training inside a simulation environment has been used to tune decision making of automated vehicles.

This thesis is a continuation of the work presented in [4], which solved the same problem of navigating the BRIO labyrinth by using a few types of LQRs to handle the control instead of ML. Gain scheduling and gain scheduling with obstacle avoidance were used to try and improve on a more classic formulation of LQR.

According to [4] the implementation of the gain scheduled LQR improved the performance when compared to the classic LQR. The implementation of obstacle avoidance, which introduced knowledge about the hole positions, presented a worse performance than the other two LQRs. The obstacle avoidance LQR lost performance due to instability and indecisiveness when following portions of the path close to holes. In these situations the regulatory goal of obstacle avoidance did not align with the regulatory goal of following the path. The best success rate presented in [4] was 78.7 % on the medium map achieved by the gain scheduled LQR. This achievement sets a performance goal for the ML approach used in this thesis and serves as a benchmark as to what can be expected performance wise of a good regulator.

2

System Description and Model

The system of interest in this thesis work is a modified BRIO labyrinth [2], which can be seen in Figure 2.1. The components that make up the test platform are:

- A BRIO labyrinth game
- A custom medium difficulty labyrinth plate
- A Raspberry Pi 3B+
- A Raspberry Pi Camera Module v2.1
- 2 x Futaba S9154 Digital High Speed Servo
- An Adafruit 16-Channel 12-bit PWM/Servo Driver - PCA9685
- A 4.8V 2500mAh NiMh Instant cub (4 cell AA)

The labyrinth plane is operated by the two high speed servo motors that are powered by the NiMH battery. The auxillary battery is required to sustain a steady power supply which can not be given by the Raspberry Pi. The Adafruit Servo Driver component creates and distributes the Pulse Width Modulated (PWM) signals to the servos. A Raspberry Pi Camera Module has been mounted above the labyrinth plane in order to enable estimation of the ball position. The Raspberry Pi 3B+ is used to analyze the video stream and sends control signals to the servo driver based on the policy derived from the reinforcement learning.

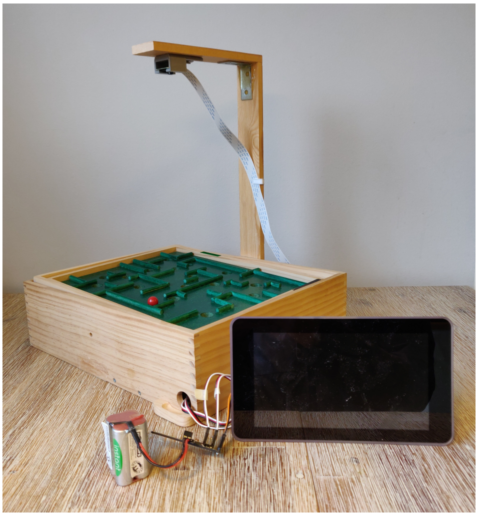


Figure 2.1: *The physical system.*

For a more detailed walkthrough on the construction of the physical system, see [4].

2.1 System Model

The system model for the labyrinth game used in this thesis is presented in [4]. It was derived by considering the two directions, x and y, independently and then apply a ball on beam model for each direction. The assumed coordinate system can be seen in Figure 2.2, where the rectangle is the border of the labyrinth and the measurements have been taken from each side of the labyrinth to the middle.

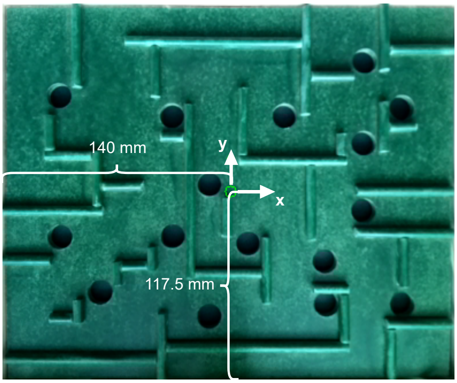


Figure 2.2: *The assumed coordinate system.*

In Figure 2.3 the holes are numbered so that they can be easily referenced. Start and end points are also marked with black and yellow squares respectively.

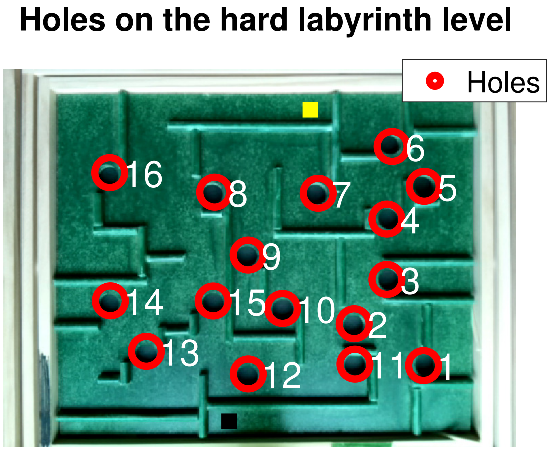


Figure 2.3: *Numbering of the holes on the BRIO labyrinth. The starting point is marked with a black square and the end point is marked with a yellow square.*

The ball on beam model that is used is presented in Figure 2.4. For the x-direction the angle β is the clockwise rotation around the y-axis. For the y-direction, however, the angle is taken to be the counter clockwise rotation around the x-axis. This is done to keep the state-space representation the same for each direction [4]. These so called beam angles are used as control signals in the state-space model since they are controlled by the servos.

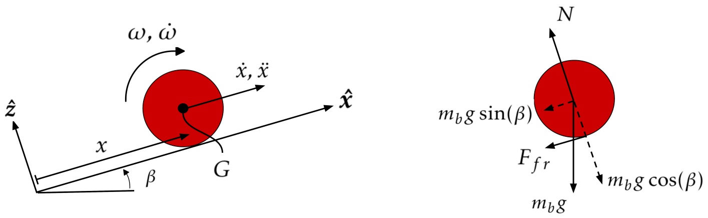


Figure 2.4: *The ball on beam with notation used in the model.*

The model constants have been lifted from [4] and are presented with description and notations in Table 2.1.

Table 2.1: *Model constants*

Notation	Value	Description
g	$9.82[\mathrm{m/s^2}]$	Gravity acceleration
m_b	8×10^{-3} [kg]	Mass of the ball
r_b	6×10^{-3}[m]	Radius of the ball
J_b	$1.152 \times 10^{-7}[\mathrm{kgm^2}]$	Inertia of the ball
f_r	0.0004	Friction coefficient

For the state-space formulation, derived below, the variables used are summarized in Table 2.2. The variables F_{fr}, ω and $\dot{\omega}$ are the frictional force, the angular velocity and the angular acceleration respectively.

Table 2.2: *The variables used in the state-space representation.*

Notation	Description
x, y	Positional coordinates in each direction.
$\dot{x}$, $\dot{y}$	Velocity in each direction.
$\ddot{x}$,$\ddot{y}$	Acceleration in each direction.
u_x,u_y	Control signals corresponding to the tilt angles of the labyrinth plane, i.e, $u_x = \beta$ in Figure 2.4.

Euler's first and second law of motion applied to the situation in Figure 2.4 can be written as

$$\begin{aligned} \hat{x} : \quad & -m_b g sin(u_x) - F_{fr} = m_b \ddot{x} && (2.1) \\ \hat{z} : \quad & -m_b g cos(u_x) + N = 0 && (2.2) \\ G \circlearrowleft : \quad & F_{fr} r_b - f_r N r_b = J_b \dot{\omega}, && (2.3) \end{aligned}$$

where $f_r N r_b$ has been introduced as rolling resistance torque and f_r is the rolling resistance coefficient. Equations (2.1) and (2.2) inserted into (2.3) yields

$$\begin{aligned} & - m_b g \sin(u_x) - f_r m_b g \cos(u_x) - \frac{J_b \dot{\omega}}{r_b} = m_b \ddot{x} \Rightarrow \left\{ \dot{\omega} r_b = \ddot{x} \right\} \Rightarrow \\ \Rightarrow & m_b (1 + \frac{J_b}{m_b r_b^2}) \ddot{x} = -f_r m_b g \cos(u_x) - m_b g \sin(u_x) \Rightarrow \\ \Rightarrow & \left\{ k_b := (1 + \frac{J_b}{m_b r_b^2}), |u_x| < 1.5 \frac{\pi}{180} \right\} \Rightarrow \ddot{x} \approx -f_r \frac{g}{k_b} - \frac{g}{k_b} u_x \end{aligned} \qquad (2.4)$$

In the last step the small angle approximation is used as the labyrinth plane angles are 2 degrees or smaller. The state space representation for the y-direction is reached in the same way.

As the labyrinth plane is rotated around the x- and y-axis the ball is subject to compensatory forces resulting from this rotation. The Euler force, the Centrifugal force and the Coriolis force are of some interest. However, previous work has chosen to disregard these forces as their dependencies,$\dot{\beta}$ and $\ddot{\beta}$ and y-direction equivalents, are deemed to be small [4]. This is reasonable as long as the change in labyrinth plane angle happens at a slow enough pace. To avoid sharp transients of the labyrinth plane angles that would lead to large values on $\dot{\beta}$ and $\ddot{\beta}$ the control signals are low pass filtered before passed to the physical system.

To able to use (2.4) in the simulation environment to approximate the movement of the ball it needs to be converted to a discrete-time model. By assuming constant acceleration for a small time fraction Δt and then integrating two times the complete motion model can be written as

$$\hat{x}: \quad \ddot{x}_i = -f_r \frac{g}{k_b} - \frac{g}{k_b} u_x \tag{2.5a}$$

$$\dot{x}_{i+1} = \dot{x}_i + \ddot{x}_i \Delta t \tag{2.5b}$$

$$x_{i+1} = x_i + \dot{x}_i \Delta t + \frac{\ddot{x}_i \Delta t^2}{2} \tag{2.5c}$$

$$\hat{y}: \quad \ddot{y}_i = -f_r \frac{g}{k_b} - \frac{g}{k_b} u_y \tag{2.5d}$$

$$\dot{y}_{i+1} = \dot{y}_i + \ddot{y}_i \Delta t \tag{2.5e}$$

$$y_{i+1} = y_i + \dot{y}_i \Delta t + \frac{\ddot{y}_i \Delta t^2}{2}, \tag{2.5f}$$

where f_r can be found in Table 2.1 and k_b can be found in (2.4). The index i denotes the current arbitrary time instance and the index $i+1$ denotes the time instance after one time fraction Δt.

2.1.1 Servo modelling

The system model assumes a constant servo angle during one time frame. However, the servos are not capable of switching between angles instantaneously due to physical limits. Therefore an average servo angle during one time frame needs to be found. To calculate the mean servo angle the servo rise speed has to be calculated first.

To investigate the servo rise speed a step response of amplitude 1.2° was recorded using an accelerometer and an Arduino Nano. The raw accelerometer data is presented in Figure 2.5. The values on the vertical axis in Figure 2.5 correlates with the angle of the labyrinth plane but the unit is arbitrary. Since the interest lies in the time passed during the servo step, the physical interpretation of these values is not necessary since it is known that the step in angle represents 1.2°.

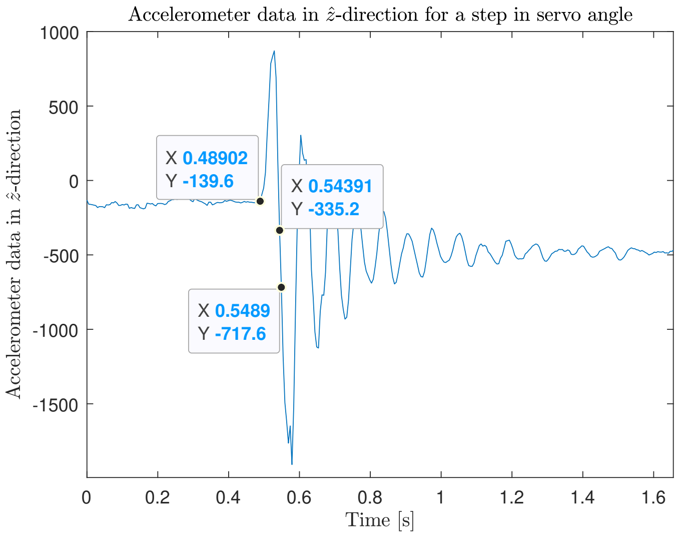


Figure 2.5: *Step response measurements from the accelerometer during an angle step of* 1.2°.

As can be seen in Figure 2.5 the measurement data is oscillating. It is suspected that these oscillations are a result of poor measurement equipment and it is hard to precisely determine the time instances where the step began and where the step reached it's resting value of about −500. However, the accelerometer seemed to detect the beginning of the step at 0.48902 s and the first time where the accelerometer detected the resting value lies between the points $[0.54391, -335.2]$ and $[0.5489, -717.6]$ in Figure 2.5. Interpolating between these points the resting value can be estimated to occur at time 0.54606.

The approximate step time is $0.54606 - 0.48902 \approx 0.05704$ s which yields the approximate rise speed of $1.2/0.05704 \approx 21°/\mathrm{s}$. In section 5.1.4 a sensitivity analysis of the servo rise speed is carried out and different values are compared using the success rate on the hardware.

The servo rise speed then gave an upper limit on how large the change in servo angle can be after one time frame Δt. This upper limit is denoted

$$u_{max} \approx 21\Delta t \quad [°]$$

in Figure 2.6. The time frame Δt is a measure of how much time that passes between each call to the motion model during simulation. By increasing this time the simulation progresses faster as the new state is calculated fewer times over the course of the simulation but if Δt is too long the simulation can become unpredictable in the sense that wall collisions might not be handled correctly.

A general desired steer angle $u_{desired}$ can then be thought of in two different cases. In Figure 2.6, u_1 demonstrates a case where the desired change in steer angle is larger than what the servos can deliver in Δt s and u_2 demonstrates a case where the desired steer angle is within what the servos can deliver.

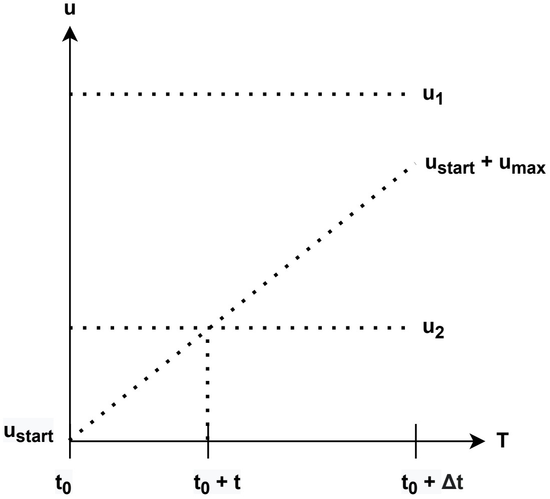


Figure 2.6: *The principal cases in the servo implementation.*

Because the desired angle u_1 is larger than what the servo can deliver in the time frame Δt, the final angle, u_{final}, of the servo can not be u_1 but is in this case set to $u_{final} = u_{start} + u_{max}$.

This is where the servo modeling would stop if the time stepping could be arbitrarily fast, using arbitrarily small time steps, but to improve simulation time the time stepping is only fast enough to sufficiently handle collision with walls. So, to represent what the servos are doing during each Δt as accurately as possible the average servo angle, $\bar{u}$, is calculated and used as the defacto servo angle during one Δt. The formula for $\bar{u}$ can be derived from the general expression for the mean of a function over a period of time.

$$\bar{u} = \frac{1}{\Delta t} \int_0^{\Delta t} u \, dt \tag{2.6}$$

The integral in (2.6) can be split into two parts, a linear part, from t_0 to $t_0 + t$ in Figure 2.6, and a constant part, from $t_0 + t$ to $t_0 + \Delta t$ in Figure 2.6. This can be written as

$$\bar{u} = u_{start} + \frac{1}{\Delta t}\left(\frac{t\Delta u}{2} + (\Delta t - t)\,\Delta u\right). \tag{2.7}$$

Where Δu is defined as $\Delta u = u_{final} - u_{start}$.

The time $t_0 + t$ in Figure 2.6 is where the steer angle reaches its final value and cross over from rising linearly to being constant. This time was found using the maximum angular velocity of the servos

$$t = \Delta t \frac{|\Delta u|}{u_{max}}. \tag{2.8}$$

Insertion of (2.8) into (2.7) yields

$$\begin{aligned} \bar{u} &= u_{start} + \frac{1}{\Delta t}\left(\Delta t \frac{|\Delta u|}{u_{max}} \frac{\Delta u}{2} + \left(\Delta t - \Delta t \frac{|\Delta u|}{u_{max}}\right) \Delta u\right) = \\ &= u_{start} + \Delta u \left(1 - \frac{|\Delta u|}{2u_{max}}\right). \end{aligned} \tag{2.9}$$

Equation (2.9) is used in the simulation model to find the average servo angle during each Δt.

3 Theoretical background

The theoretical background to this thesis can be split into three major parts. A description of the Q-learning algorithm that is used to retrieve a steer policy. A presentation of the Kalman filter that estimates ball position and velocity from camera images. And lastly, an introduction to Linear Quadratic Regulator since the Q-learning derived steer policy will be compared to the results of the LQR developed in [4].

3.1 Q-learning

The reinforcement learning algorithm that is used in this thesis to explore and solve the BRIO labyrinth is known as a Q-learning algorithm described in [9]. The Q-learning algorithm in itself does not require a model since it is built upon a generic network of discrete states and discrete actions and its goal is to find the action in each state that maximises the reward for the remaining session. It does not use any underlying mathematical model of the system but instead learns by interacting with the environment and getting a reward for a certain outcome given a taken action in a certain state. A reward can be positive but also negative to allow penalizing unwanted behavior. A Q-learning algorithm can be used to solve the problem with either a discrete or a continuous representation of states, but in this thesis we solely use the discrete representation.

A system that can be described by a number of discrete states $s \in \mathbb{R}^n$, each with a corresponding reward, that are intertwined via actions, each with a set of outcomes that have probabilities $P(S_f|S_i, a_j)$, is a Markov Decision Process (MDP). The variable $P(S_f|S_i, a_j)$ is the probability to end up in state S_f when action a_j is taken in state S_i where the indices f for final, i for initial and j as a numerator for the taken action. The rewards connected to each state is $R(S_i) = r_i$. An example MDP is visualized in Figure 3.1, where edges represent transition probability and nodes represent states and actions.

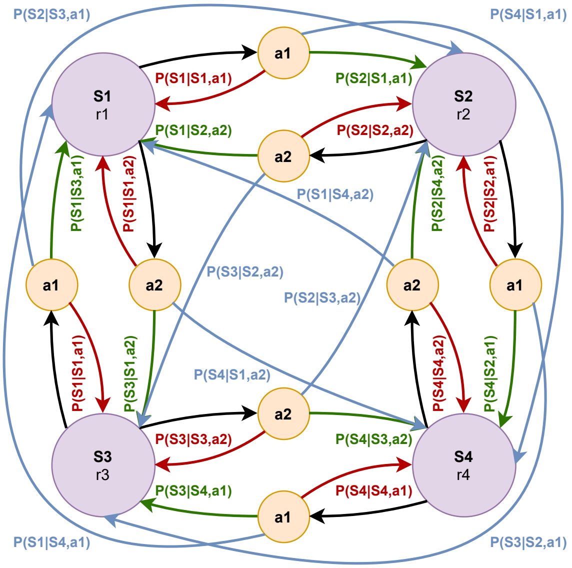


Figure 3.1: *A visualisation of a Markov Decision Process.*

For example if action $a1$ is taken in state $S1$, it can be seen in Figure 3.1 by following the black arrow from $S1$ to $a1$, there are three scenarios. Firstly, with a probability $P(S4|S1, a1)$, the new state is $S4$ and a reward of $R(S4) = r4$ is acquired. Secondly, with a probability $P(S2|S1, a1)$, the new state is $S2$ and a reward of $R(S2) = r2$ is acquired. Thirdly, with a probability $P(S1|S1, a1)$, the new state is $S1$ and a reward of $R(S1) = r1$ is acquired.

When a number of actions are taken in sequence it results in a state sequence. The total reward for any given state sequence in an MDP can be retrieved by summing up the individual rewards. A discount factor γ is used to model urgency of completion, since a solution of infinite actions is not of interest. The total discounted reward for any given state sequence in an MDP is

$$R_{tot} = \sum_{t=0}^{T} \gamma^t R(S_t),$$

where $0 \leq \gamma \leq 1$ is the discount factor which keeps the sequence finite and makes immediate rewards weigh in more than future rewards. Following a policy π, which is a mapping from a state to an action, results in a sequence of states

$[S_0, S_1, S_2, ..., S_T]$, which in turn yields a particular cost R_{tot}. Starting in an arbitrary state s, the value function associated with the policy π is

$$V^{\pi}(s) = \mathbb{E}\left[\sum_{t=0}^{T} \gamma^t R(S_t)|S_0 = s\right]. \tag{3.1}$$

It then follows that the optimal value function is given by the same expression for an optimal policy π which maximises the value function according to

$$\begin{aligned} V(s) &= \max_{\pi} V^{\pi}(s) \\ &= \max_{\pi} \mathbb{E}\left[\sum_{t=0}^{\infty} \gamma^t R(S_t)|S_0 = s\right] \\ &= \max_{\pi} \mathbb{E}\left[R(s) + \sum_{t=1}^{\infty} \gamma^t R(S_t)|S_1 = s'\right] \\ &= \max_{\pi} \mathbb{E}\left[R(s) + \gamma V(s')\right], \end{aligned} \tag{3.2}$$

which is a recursive function where the optimal policy π and the optimal costs $V(s)$ and $V(s')$ are unknown. Equation 3.2 is commonly known as the Bellman optimality equation and is often used for dynamic programming [1]. The parameter s' is the state in which the ball ends up in by following the optimal policy for state s. If the optimal cost function is the value expressed in (3.2), with regards to the policy π, the optimal policy itself can be retrieved as

$$\pi^* = \arg\max_{\pi} \mathbb{E}\left[R(s) + \gamma V(s')\right].$$

By inserting the definition of expectation, into (3.2), the expression in (3.3) can be acquired.

$$V(s) = \max_{\pi} \mathbb{E}\left[R(s) + \gamma V(s')\right] \tag{3.3}$$

$$= \max_{\pi} \sum_{s'} (R(s', s, \pi(s)) + \gamma V(s'))P(s'|s, \pi(s)) \tag{3.4}$$

The sum of s' is due to the probabilistic nature of the MDP where an action $a = \pi(s)$ in state s has a chance to end up in different states s'. Note that our policy is still deterministic but that there is a probability that we do not end up in the discrete state that is desired.

If we introduce the implicit function $Q(s, a)$, usually called quality function, state-action value function or Q-function, as

$$Q(s,a) = \mathbb{E}\left[R(s',s,a) + \gamma V(s')\right]$$

it follows that

$$V(s) = \max_a Q(s,a) \tag{3.5}$$

$$\pi(s) = \arg\max_a Q(s,a), \tag{3.6}$$

given (3.3) and (3.1). Thus Q can also be expressed as

$$\begin{aligned} Q(s,a) &= \mathbb{E}\left[R(s) + \gamma \max_a Q(s',a)\right] \\ &= \sum_{s'} (R(s',s,a) + \gamma \max_a Q(s',a))P(s'|s,a), \end{aligned}$$

so that it can be calculated without knowing the optimal cost function V. This expression is however still dependant on the probability $P(s'|s,a)$, which is unknown in this case. The Q-values are instead approximated iteratively for each state and action pair with a trial and error method and the cost function and policy is then acquired from (3.5) and (3.6). An effective way of ordering the Q-values is by storing them in a matrix with respect to states and actions. This matrix is called the Q-matrix or tabular Q-function and is visualised in Table 3.1 for a simple example with two states and two actions.

Table 3.1: *A representation of the tabular Q-function for a general case with two states and two actions.*

a\s	s_1	s_2
a_1	$Q(s_1,a_1)$	$Q(s_2,a_1)$
a_2	$Q(s_1,a_2)$	$Q(s_2,a_2)$

Updating the Q-matrix for a certain training instance in which state s and the taken action a^* result in state s', the estimated Q-matrix entry for (s,a^*) is updated as

$$\hat{Q}(s,a^*) = R(s',s,a^*) + \gamma \max_a \hat{Q}(s',a).$$

As it is not desired to completely erase experience from previous iterations, a learning rate is used to contain parts of the old experiences. This is done according to the expression

$$\begin{aligned}\hat{Q}_{k+1}(s,a^*) &= \hat{Q}_k(s,a^*) + \alpha(R(s',s,a^*) + \gamma \max_a \hat{Q}_k(s',a) - \hat{Q}_k(s,a^*)) \\ &= (1-\alpha)\hat{Q}_k(s,a^*) + \alpha(R(s',s,a^*) + \gamma \max_a \hat{Q}_k(s',a)), \end{aligned} \tag{3.7}$$

where α denotes the learning rate, γ denotes the discount factor, s denotes a state, a denotes an action, s' denotes the state retrieved by taking action a^* in state s which results in the reward $R(s',s,a^*)$.

The learning rate can be initiated with a high value in the beginning of the learning process to learn as much as possible from the exploration phase but can be decreased linearly as the gathered experience becomes more reliable. This is done to prevent large changes to the expected rewards and by extension to prevent large changes to the policy.

During the learning process the algorithm is usually initiated with a high probability to choose a random action and during the process lean more into experience based actions. This transition can be done in a multitude of different ways. A simple tactic is to use a linearly decreasing probability that a random action is chosen. The general idea is that the probability of choosing a random action decrease as the learning process progresses.

3.1.1 Experience Replay Q-learning

The Q-learning algorithm presented in Section 3.1 takes a step and learns from the state transition and received reward once. The state transition is then forgotten and thus information from a certain rare experience could likely be overwritten over time. Additionally the steps taken in sequence are correlated, which makes the state transitions sequentially biased. This implementation of the Q-learning algorithm could thus be sensitive to biases entrenched in the data.

The Experience Replay version of Q-learning uses (3.7) to learn from sampled data, but uses another method to decide which sampled experience is used to learn from. The experience replay feature is further explained in [11]. Experience in this context is a collection of a state s, action a taken in this state, reward r, terminal flag t and state s' resulting from the action a in state s. The algorithm saves each experience in memory so that each experience can be learned from several times. It also allows the Q-learning algorithm to learn from the experiences in a random order, rather than the sequential order they chronologically occurred in. The memory contains m experiences, from which a batch of b random experiences are retrieved from the memory to train on each iteration.

This method is most commonly applied when Q-learning is used to estimate a parametric model rather than the approach of finding the best possible action in a specific state, as is done in this project. However, positive rewards are sparsely distributed and the multitude of states and actions makes a specific state transition rare. Additionally the expected future reward changes as the Q-learning algorithm progresses, which suggests that experience replay may lead to faster convergence.

3.2 Discrete Kalman filter

A Kalman filter is used to estimate unknown states of a linear state space model using model based predictions and measurements. The Kalman filter tries to recursively predict states using the state space model. Measurements are then used to adjust these predictions to correspond more with the actual states. The Kalman filter in this thesis is used to improve the positioning of the ball on the physical system since the image processing yields a noisy approximation of the ball's position. Additionally, the speed cannot be measured directly and the Kalman filter is used to approximate this state from the position measurements. The Kalman filter in this thesis needs a discrete-time linear state space model which can be written in the form

$$x_{k+1} = F_k x_k + G_{u,k} u_k + G_{v,k} v_k \tag{3.8a}$$

$$y_k = H_k x_k + D_k u_k + e_k, \tag{3.8b}$$

where x are the unknown states, u are known control signals, v_k is process noise and e_k is measurement noise. The process noise v_k is modeled to be normally distributed with expectation 0 and covariance Q_k. Similarly, the measurement noise e_k is modeled to be normally distributed with expectation 0 and covariance R_k. The index k is a time index.

The standard steps performed in the Kalman filter is described in [6] and is given by

Time update:

$$\hat{x}_{k|k-1} = F_k \hat{x}_{k-1|k-1} + G_{u,k} u_{k|k} \tag{3.9a}$$

$$P_{k|k-1} = F_k P_{k-1|k-1} F_k^T + G_{v,k} Q_k G_{v,k}^T \tag{3.9b}$$

Measurement update:

$$\epsilon_k = y_k - H_k \hat{x}_{k|k-1} - D_k u_k \tag{3.10a}$$

$$S_k = H_k P_{k|k-1} H_k^T + R_k \tag{3.10b}$$

$$K_k = P_{k|k-1} H_k^T S_k^{-1} \tag{3.10c}$$

$$\hat{x}_{k|k} = \hat{x}_{k|k-1} + K_k \epsilon_k \tag{3.10d}$$

$$P_{k|k} = P_{k|k-1} - K_k H_k P_{k|k-1}, \tag{3.10e}$$

where (3.9) is the prediction part, often called the time update, whereas (3.10) adjusts the prediction using measurements, often called the measurement update. The iterations start with the initial state estimate $\hat{x}_{1|0}$ and covariance $P_{1|0}$. Q_k is the covariance matrix of the process noise v_k and R_k is the covariance matrix of the e_k. When considering an update of a variable, for example $\hat{x}_{i|j}$, the index i represents the time instance of the approximation and the index j represents up to which time instance measurements have been used.

3.3 Linear Quadratic Regulator

The Linear Quadratic Regulator (LQR) is presented in [5]. An LQR is designed to minimize a quadratic criteria that is based on the control error, e, and the size of the control signal, u. For a discrete problem this can be written as

$$\min(\|e\|_{Q_1}^2 + \|u\|_{Q_2}^2) = \tag{3.11a}$$

$$\min \sum_k e_k^T Q_1 e_k + u_k^T Q_2 u_k. \tag{3.11b}$$

where k is a time index and Q_1 and Q_2 are positive semi definite weight matrices that can be seen as design variables for the regulator characteristic. The LQR relevant to this thesis requires the system on standard discrete state space form,

$$x_{k+1} = Ax_k + Bu_k + Nv_{1,k} \tag{3.12a}$$

$$z_k = Mx_k \tag{3.12b}$$

$$y_k = Cx_k + v_{2,k}. \tag{3.12c}$$

where v_1 and v_2 are white noises with intensities

$$\begin{bmatrix} R_1 & R_{12} \\ R_{12}^T & R_2 \end{bmatrix}.$$

An LQR relies on the observed state of a Kalman filter and the optimal linear controller is given by

$$u_k = -L\hat{x}_k \tag{3.13a}$$

$$\hat{x}_{k+1} = A\hat{x}_k + Bu_k + K(y_k - C\hat{x}_k) \tag{3.13b}$$

where K is the time invariant version of the K_k presented in Section 3.2. The feedback is determined by

$$L = (B^T SB + Q_2)^{-1} B^T SA. \tag{3.14}$$

Where S is the positively semi definite solution to the matrix equation

$$S = A^T SA + M^T Q_1 M - A^T SB(B^T S + Q_2)^{-1} B^T SA$$

The system needs to meet a few criteria for the technique to be applicable.

- (A, B) is stabilizable
- (A, C) is detectable
- (A, M^T, Q_1, M) is detectable
- R_2 is symmetric and positively semi definite
- $R_1 - R_{12}R_2^{-1}R_{12}^T$ is positively semi definite

- R_{12} is constant

3.3.1 Gain scheduled LQR

A gain scheduled LQR consists of two or more LQRs that are designed to prioritize different control objectives. In [4] one LQR is designed to prioritize travelling along the direction of a given path. Another LQR is instead prioritizing to reduce the distance to the same path. The control signals from each LQR are then weighted to construct the effective control signal. The further away from the path the ball is, the more the second LQR is allowed to influence the effective control signal. If the ball is in close proximity to the path, the first LQR gets more influence instead.

4 Implementation

The implementation was made in two stages. The first stage was the software implementation where a Q-learning algorithm learns a steer policy through exploration. The second stage was the implementation of the steer policy on the hardware that allowed the system to use the steer signals given by the policy low-pass filtered to promote smooth operation.

4.1 Simulation software

Applying reinforcement learning on hardware directly can be time consuming. Therefore, a simulation environment was developed in Python. It consists of five parts: the disturbance model, the map, the agent, the reinforcement learning and post-processing of the retrieved policy.

4.1.1 Disturbance

Disturbance was introduced in the simulation to deal with the fact that the model probably is not a perfect representation of the true system. Even in the best of scenarios a perfect match between model and a physical system is impossible. In addition, when investigating the physical system, three sources of uncertainty were observed that seemed to impact the behaviour of the ball.

1. The rolling resistance seemed to vary a lot.
2. The coupling between the servos and the labyrinth plane are a bit loose which makes it difficult to reliably reach an exact desired steer angles.
3. The labyrinth surface is uneven and tilts in random directions.

To deal with the first observation and general model errors, the calculated acceleration is altered by a random amount. This is achieved by randomly generating a percentage factor from a uniform distribution and then multiplying the ideal acceleration by this random factor in each time step. An alternative choice is an added disturbance, but that does not correspond to the behavior of the physical system at stationarity. The choice of a uniform distribution instead of a normal distribution was made to train the network on large disturbances more often.

To deal with the other random disturbances present in the physical system another disturbance source was added. As the ball is respawned at the starting point a random tilt is generated from a uniform distribution and added to the steer angle selected by the agent. The choice of a uniform distribution as opposed to a normal distribution is that the uniform distribution is strictly limited by default and the agent is exposed to large disturbances at the same rate as it is exposed to small disturbances. The aim is to cause the agent to be more careful in places where a random tilt in a certain direction can cause the ball to end up in a hole.

In Chapter 5 the two disturbance models are compared and evaluated to find a good combination of the two.

4.1.2 The map

The map keeps track of the position of walls, holes and the goal, and calculates the reward or penalty for a certain step. To make these operations easy, the map was split into tiles of 5x5 mm, every tile is marked with a number declaring its characteristic. Table 4.1 specifies which number that corresponds to which characteristic.

Table 4.1: *Tile characteristics and their corresponding numerical value.*

Numeric value	Tile characteristic
0	Free to move space, small negative reward
1	Goal, large positive reward
2	Wall, no reward
3	Hole, very large negative reward
4	Close proximity to hole, medium negative reward

The agent uses the map to determine if a collision has occured with a wall or if the ball has ended up in a hole, the goal or outside the map. Additionally, the agent also uses the map to calculate the reward for a certain state transition, based on the reward modifiers described in Section 4.1.4.

A visual representation of the map in the simulation environment is presented in Figure 4.1. The green hollow squares represent walls, red hollow squares represent holes and orange hollow squares represent proximity to holes. The filled squares represent goal and start position, where the black one is the start position and the yellow one is the goal position.

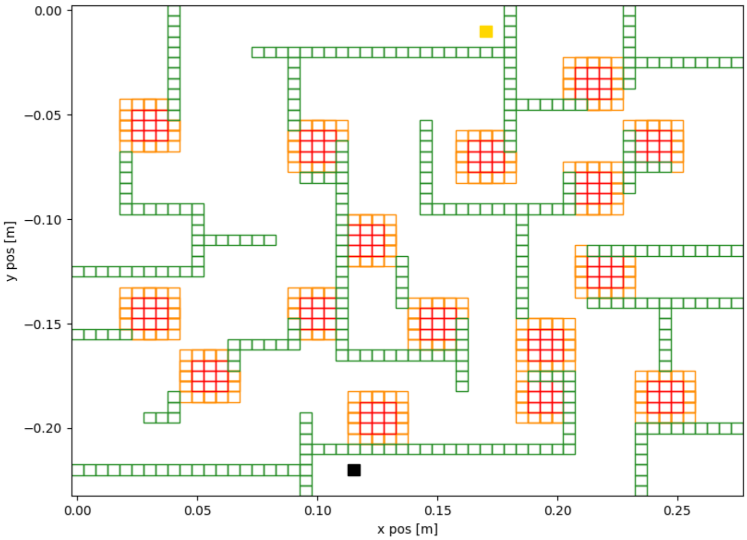
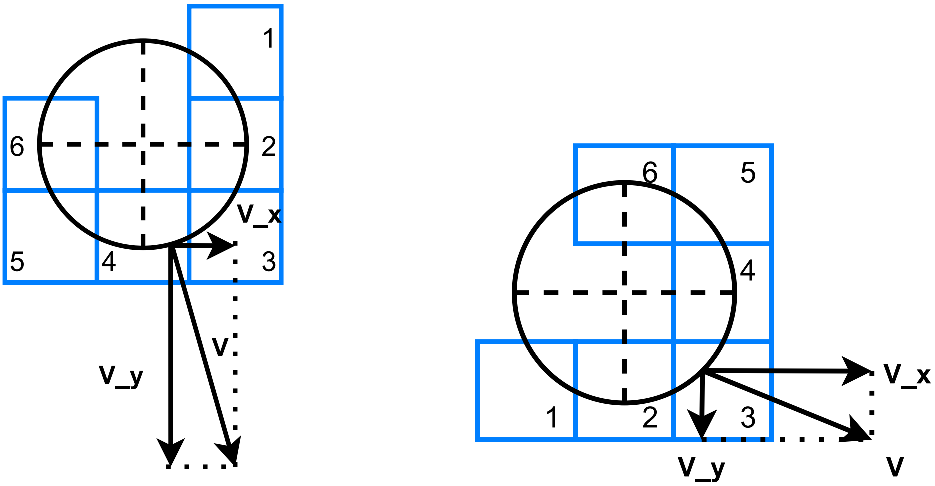


Figure 4.1: *Visualization of the map in the simulation environment.*

4.1.3 The agent

Using a limited set of actions in two dimensions, each dimension for each of the physical system's two servos, the agent steers the ball through the map to gather knowledge about the environment. The ball's state is expressed in a continuous space, to achieve a realistic movement, which the agent then maps onto a discrete state representation due to the Q-learning algorithm defined in 3.1. For each action the ball is moved in small incremental steps such that collisions with walls are not missed. For each of these incremental steps six chosen tiles are checked for collision. The checked tiles are chosen with respect to the ball's current state. It is useful to define primary and secondary directions as they play an integral role in the way the checked tiles are chosen. The primary direction is $\pm\hat{x}$ or $\pm\hat{y}$ along which the ball has its largest velocity component. The secondary direction is the direction in which the ball has its smallest velocity component. Presented in Figure 4.2a is the case where the ball has $+\hat{y}$ as the primary direction and $+\hat{x}$ as secondary direction. Presented in Figure 4.2b is the case where the ball has $+\hat{x}$ as primary direction and $+\hat{y}$ as secondary direction. Note that these are only two out of eight possible cases. In Figure 4.2 the x component of the velocity is called **V_x** and the y component **V_y**. How the checked tiles are obtained is presented in Table 4.2.



***(a)** Primary y.*

***(b)** Primary x.*

***Figure 4.2:** Two cases with different primary axis of movement and how the checked tiles are chosen in relation to this.*

***Table 4.2:** List of tiles and how they are found.*

Tile	**Obtained by**
Tile 1	Obtained by finding the tile closest to the point received when subtracting $r_b/\sqrt{2}$ from the ball's current position with respect to the primary direction and adding $r_b/\sqrt{2}$ with respect to the secondary direction.
Tile 2	Obtained by finding the tile closest to the point received when adding r_b to the ball's current position with respect to the secondary direction.
Tile 3	Obtained by finding the tile closest to the point received when adding $r_b/\sqrt{2}$ to the ball's current position with respect to both the primary and secondary direction.
Tile 4	Obtained by finding the tile closest to the point received when adding r_b to the ball's position with respect to the primary direction.
Tile 5	Obtained by finding the tile closest to the point received when adding $r_b/\sqrt{2}$ to the ball's current position with respect to the primary direction and subtracting $r_b/\sqrt{2}$ with respect to the secondary direction.
Tile 6	Obtained by finding the tile closest to the point received when subtracting r_b from the ball's current position with respect to the secondary direction.

Depending on which tiles are valid or not for each incremental step the collision is handled in a specific way. The checks are done in the order as they are written in Table 4.3.

Table 4.3: *Collision type given tile invalidity.*

Invalid tiles	**Collision type**
2 & 4	inner-corner
4	head-on primary direction
2 & 3	head-on secondary direction
3	outer-corner
5 & NOT 6	outer-corner
1 or 2	head-on secondary direction

An **inner corner** collision triggered by tile 2 and 4 in combination is handled by rotating the velocity 180 degrees and multiplying it with the bounce coefficient c_b. It is handled first to make sure that this situation is not overlooked when 4 is checked individually.

A **primary direction** collision triggered by tile 4 is handled by flipping the primary velocity and multiplying it with the bounce coefficient c_b. It is handled early on as collisions happen more often in the primary direction of travel and by catching this scenario early the collision control can be completed sooner.

A **secondary direction** collision triggered by tile 2 and 3 in combination, by tile 2 alone or by tile 1 alone is handled by flipping the secondary velocity and multiplying it with the bounce coefficient c_b.

An **outer-corner** collision triggered by tile 3, or by tile 5 under the condition that tile 6 is valid, is handled by bouncing the ball on the tangent line T to the ball in the contact point it makes with tile 3 or 5. Figure 4.3 displays one possible situation where an outer-corner collision occurs.

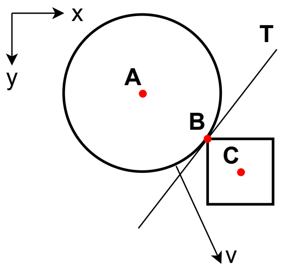


Figure 4.3: *The ball colliding with tile 3.*

The line T is the tangent to the ball in point B. Bouncing the ball on this line is done by mirroring the velocity v in T and multiplying it with the bounce coefficient c_b. The process of mirroring a two dimensional vector in a line is a known problem and it is described in [7]. The goal is to find the velocity's normal component to the tangent line and subtracting that twice from v:

$$\begin{aligned} v_{\perp T} &= v_{\|\vec{AB}} = \frac{v\vec{AB}}{|\vec{AB}|^2}\vec{AB} \implies \\ v_{mirror} &= v - 2v_{\perp T} = \\ &= v - 2\frac{v\vec{AB}}{|\vec{AB}|^2}\vec{AB} \end{aligned} \tag{4.1}$$

The point B is not known, which makes $\vec{AB}$ ill-defined. However, point A, point C and the side length of the tiles are known identities. And so, by adding half of the tile's side length to point C along both x- and y-axis towards point A, point B can be found. Thus $\vec{AB} = \vec{AC} + \vec{CB}$ can be inserted in (4.1). The resulting equation for handling outer corner collisions is then:

$$v_{res} = c_b \left(v - 2\frac{v\left(\vec{AC} + \vec{CB}\right)}{|\vec{AC} + \vec{CB}|^2}\left(\vec{AC} + \vec{CB}\right)\right). \tag{4.2}$$

4.1.4 Reinforcement learning

Reinforcement learning is one of the major parts of the thesis and uses the simulation environment, i.e., the map and agent, described in Section 4.1.3 and 4.1.2 to learn the best possible policy for the model. It is based on the Q-learning algorithm described in Section 3.1.

The Q-learning algorithm described in Section 3.1 is built upon discrete states. This discretization can be done in a multitude of ways. The states are position and velocity, both in $\hat{x}$ direction and $\hat{y}$ direction separately. The position discretization, consisting of tiles, is mentioned in Section 4.1.2. The discretization of speed is discretized with regards to two parameters: how many degrees of freedom that is manageable in the Q-matrix and the maximum velocity that is common on the physical system. More degrees of freedom in the representation of the ball velocity can give a more precise policy but leads to a larger system to train, which takes time or may not converge at all. In Figure 4.4 the Kalman filter estimate of the speed during one run on the physical system is shown to motivate why discretizing the speed over 0.08 m/s is not useful, the ball will only on rare occasions travel faster than that.

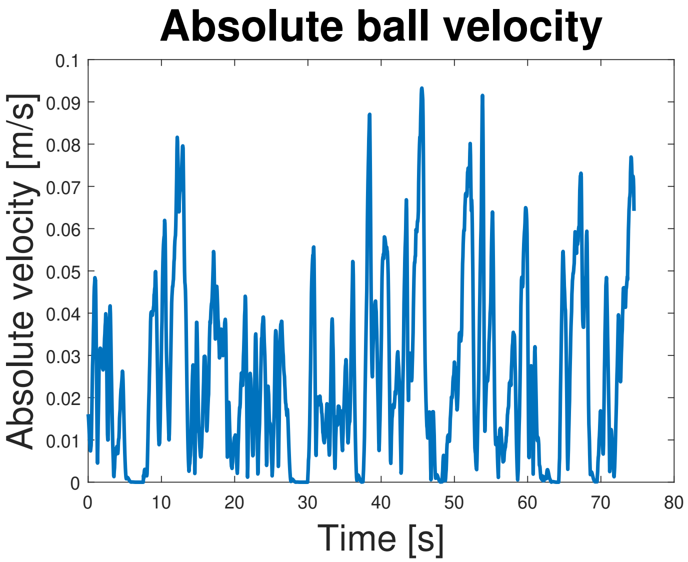


Figure 4.4: *The absolute ball velocity during a run on the physical hardware.*

The states are not the only parameters that needs to be discretized. The action space needs to be considered as well. This discretization needs to be precise enough to navigate accurately through the labyrinth in narrow passages, but also aggressive enough to overcome frictional forces and imperfections in the labyrinth surface. This is done with respect to degrees of freedom and control angle variety. More degrees of freedom in the action space can give a more precise policy but, once again, leads to a larger system to train.

The algorithm is iterated until a certain amount of moves, from now on called learning operations, has been made in total. The total amount of learning operations is the learning operation limit which defines the amount of state transitions that are performed. Each learning operation is either done by taking a random action or exploiting previous knowledge to decide on the best action for the current state. The probability to take a random action is decreased linearly as the learning process progresses. This is due to the common principle of exploring early on and gradually start to trust the gathered experience.

The learning rate is decreased linearly throughout the learning process from a defined start value to a defined end value. During the exploration phase of a Q-learning algorithm it is important to gather as much information as possible from the learning operations since all information is new. Therefore a higher learning rate is used in the beginning. As the learning process progresses the learning rate is decreased to avoid adjusting the learned path drastically based on one extreme case rather than the summed experience from the previous exploration.

To mimic the fact that a game can be either lost or won, a so called terminal state is implemented. Reaching a terminal state ends a specific session. Usually when Q-learning is applied these sessions are called episodes and they are restarted by resetting the ball's position to the start position. However, in this project the agent often terminates a session after a low amount of learning operations due to ending up in a hole. This early termination makes the objective of reaching goal difficult to achieve. To mitigate this shortcoming, two complementary reset handlers were implemented to reset the ball when a terminal state occurred.

The first handler can be described as a full reset handler. It restarts the ball from the start position, given that a certain amount of learning operations has been taken since the last full reset or that the terminal state is also the goal state. The session from one full reset to another full reset is called an episode in this project and the number of learning operations in one episode are called episode step limit.

The second handler can be described as a partial reset handler. It restarts the ball from a position that the ball recently had, given that the terminal state was not the goal state and a certain amount of learning operations has not been taken. The session from one partial reset to another partial reset is called an attempt in this project. Thus an episode consists of one or more attempts.

The experience retrieved from each learning operation is stored in an experience replay buffer. When an experience is added to the buffer the oldest experience is removed if the buffer already contains the maximum amount of experiences. A random subset of experiences are sampled from the buffer to learn from in the Q-learning algorithm. One of these samples will always be the newest sample to guarantee that each experience is used at least once in the learning process.

To reduce the rate at which the Q-values increases in the Q-matrix, since a high growth rate might lead to instability where suboptimal policies are enforced, the Q-matrix was complemented with a secondary Q-matrix. During training the primary Q-matrix is used to select the current best action for a certain state. The secondary Q-matrix is used to determine the expected future reward. The secondary Q-matrix is then updated by low-pass filtering the expected future reward of the primary network.

Reward design

A central part of reinforcement learning is to determine how much reward should be given for a given state-action pair. In this thesis the reward is determined by the ball's velocity and the trait of the end tile. The rewards are set in relation to the reward of taking an action.

The main objective is to reach goal without falling into a hole, in a non infinite time. Therefore, taking an action should not be that expensive as a longer and safer route is better than a short route with a high risk of failure. However, if it is free to take an action there is a risk that the learner easily finds the local optimum to just stay in start without moving at all which is not desired. Doubling the reward for a step would be equivalent to halving the other rewards.

The reward for going into a hole was set with respect to the local optimum that can occur if it is cheaper for the agent to run into a hole than to stay still for an entire episode. To avoid that an optimal path is to run straight in to a hole, the reward for ending up in a hole is about twice as expensive as staying still for an entire episode.

The reward for reaching the goal was set with the idea that the reward needed to be large enough to cover the expenses of travelling through the labyrinth. This value was highly dependent on the map as the ball does not go the same speed or the same distance on every map. The reward for staying still during an entire episode was used as reference.

The reward scaling based on velocity was set after the preliminary tests on the hardware. It became clear that the policy was too aggressive in certain areas. As the speed is discretized there is a maximum value on the speed that the policy can sense and be trained for. On rare occasions the ball reached higher levels of speed than what the policy was trained for and that led to poor control. To mitigate this two types of rewards in relation to speed were tested.

Firstly, a reward to the square of the speed was implemented. Going slow was not a problem, going faster than the discretization could detect was. Therefore a quadratic scaling seemed fitting. As the speeds are in order $10^{-1}[m/s]$ an amplification factor was needed to ensure the speed penalty was not overshadowed by the reward of just taking an action.

Secondly, a constant reward was applied when going over or under certain velocity thresholds. The reward for taking an action was set to zero for this implementation since penalizing the agent with a negative reward under a certain velocity threshold is sufficient to avoid an optimal path of not moving. The two implementations are tested and compared with each other in Chapter 5.

Lastly, in an effort to promote paths that are not in close proximity to holes a negative reward was set on tiles adjacent to holes. This was done because the agent would steer close to holes despite the addition of disturbance. If the reward was too low the policy would behave nervously when tested on the hardware. If the reward was not low enough there was no change in behavior.

4.1.5 The policy smoothing

Once the training has been completed there are certain grid coordinates that are valid positions but have not been reached in the simulation environment due to the radius of the ball being larger than the chosen grid size defined in chapter 4.1.2. This is a problem because the position estimation on the hardware is subject to errors that may position the ball in a coordinate where the steer policy never has been trained and thus is not a product of simulation experience, but rather a random initialization. This is why a policy smoothing script has been implemented. The method described in this section aims to extend the policies to the tiles in proximity to walls which have not been explored.

The technique is inspired by a simple moving average, described in [8], which smooths noisy data by creating a data subset of sequential averages. In a data set where all data points are equal in the sense that they have the same quality, this is useful as is. In our environment this is not the case. A state which has been visited several times by the agent will correspond to an action that has proven to be the most beneficial over and over again, while a state that has not been visited at all will correspond to a randomly initiated action. Therefore the term "well trained" is introduced as a way of differentiating between which actions that are reliable and can be used when smoothing the policy. An action is only deemed to be in need of smoothing if it does not meet the requirement of being well trained.

First an evaluation of which tiles are considered 'well trained' is done by extracting the highest expected future reward, with respect to the action space, for each tile position under the condition that the ball has zero velocity. If the expected future reward is below a certain threshold Q_T in a tile, it is considered poorly trained and is thereby a candidate that should be smoothed. Walls, holes and tiles in proximity to walls will always be considered poorly trained, due to the fact that the ball can not be positioned in these tiles during simulation, and thus always be subject to smoothing.

Iteratively the policy in each poorly trained tile is retrained by looking at the surrounding tiles and their corresponding actions. This is done separately for every velocity combination. In Figure 4.5 a visualisation of which tiles are considered during the retraining process of a poorly trained tile for one velocity combination. The actions considered in the neighboring tiles are the ones for the same velocity combination as in the investigated poorly trained tile that is being retrained. The purple tiles represent the poorly trained tiles, the darker purple tile is the one currently being retrained. The green tiles represent walls, whereas the red tiles represent part of a hole. The numbers represent expected future reward. The tiles with pronounced black borders are tiles considered but not deemed well trained enough to be a candidate to learn from. A poorly trained tile requires at least two neighbouring well trained tiles to be retrained. The blue tiles with blue borders are deemed well trained enough to learn from.

A neighboring tile's action is relevant to the estimation of the new smoothed action if itself is not poorly trained, is not a wall, hole or outside of the map. If at least two candidates are found, a new action can be generated. This is done by first averaging the candidate actions. The averaged action's closest match in the discrete action space is then used. In Figure 4.6 the expected future reward for zero velocity is shown for a policy. The white areas are considered to be well trained, as there is at least one move that is expected to give a reward larger than the threshold Q_T.

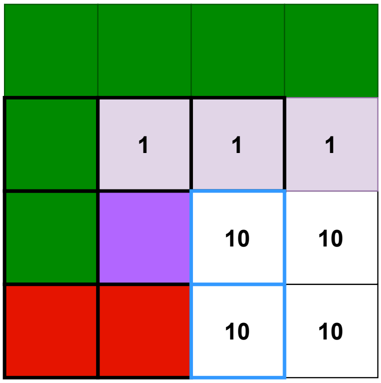


Figure 4.5: *A visualisation of the retraining process of a poorly trained tile. In this example, the threshold Q_T is between 1 and 10. Green tiles represent walls, red tiles represent holes, purple tiles represent poorly trained tiles. The dark purple tile is retrained, where tiles with black pronounced borders are candidates deemed not well trained enough to use in training and blue borders are deemed well trained enough.*

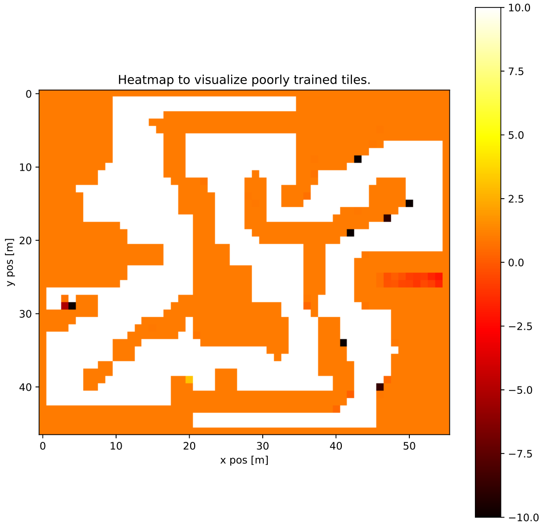


Figure 4.6: *heatmap before smoothing. The heat intensity represents the expected future reward in a tile for zero velocity.*

In Figure 4.7 the policy from Figure 4.6 has been smoothed. The remaining poorly trained tiles are either walls, center point of holes or unreachable areas with less than two well trained neighbouring tiles.

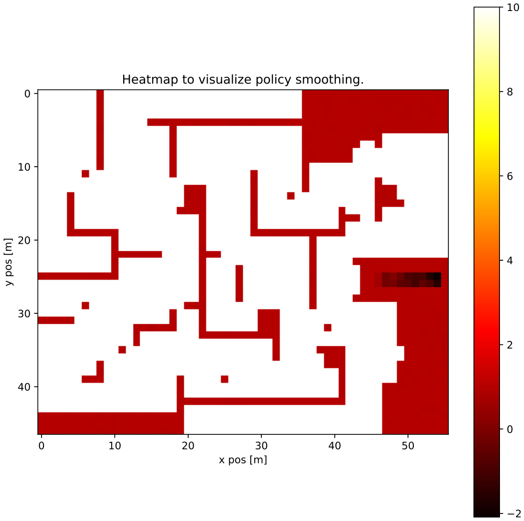


Figure 4.7: *heatmap after smoothing. The heat intensity represents the expected future reward in a tile for zero velocity.*

4.2 Software adjustments for policy deployment on the physical system

To apply the policy learned in the simulation environment to the physical system some adjustments have been done to the physical system's software. The software running the labyrinth is split into three parts: ball tracking, controller and servo.

The three parts cooperate using the Robot Operating System (ROS) framework so that the different scripts can work independently and synchronise information between the scripts using so-called topics. Most of the software regarding controlling the labyrinth was reused, with minor adjustments, from [4]. In Figure 4.8 it is shown how the software on the physical system communicates between modules. Sharp edged boxes represent a piece of hardware and rounded boxes represents software modules. The dotted arrow from simulation environment to the Controller is not an actual connection but rather that the simulation environment produces a policy that is extracted into the controller. This policy is then used to map actions from positions and estimated velocities.

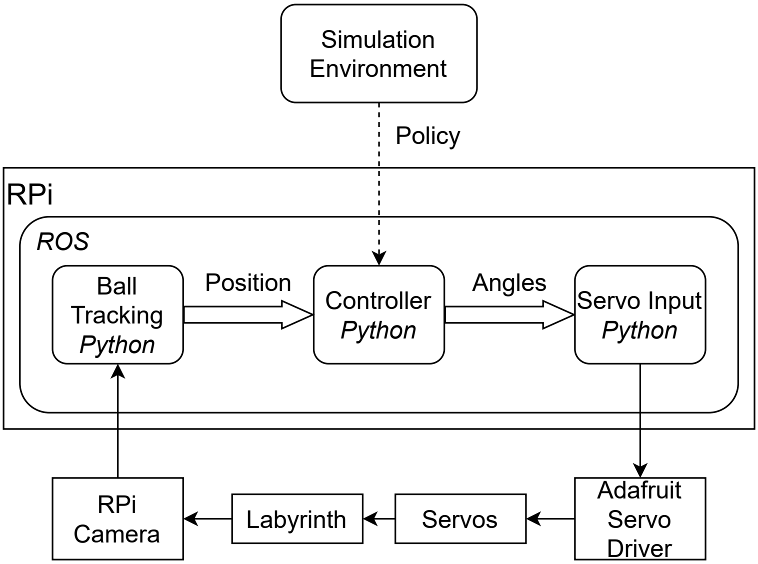


Figure 4.8: *A schematic showing how different parts of the physical system cooperates, both in hardware and in software.*

4.2.1 Ball tracking

The task of the ball tracking script is to estimate the position of the ball using the Raspberry Pi camera available on the hardware and the computer vision library OpenCV that is available for Python.

The ball tracking script uses the CIELAB color space, often called LAB color space, to ease the detection of the ball in an image. The L in the abbreviation stands for the perceived lightness, A for the color grade red to green whereas B is for the color grade blue to yellow. Due to the previous project using the A color grade to find a red colored ball in the labyrinth the same approach was used in this project with the added feature of blurring the image before converting it from RGB to LAB. This allowed for a less aggressive erode and dilation filtering compared to the previous implementation. The erode and dilation filters are applied after the masking in the A dimension of the LAB color space to smooth contours and remove remaining grainy features. In addition the labyrinth was colored green to contrast the red ball further.

4.2.2 Policy Mapping Controller

The task of the controller is to, determine the current state such that an optimal angle can be retrieved from the policy. The current state is given by a Kalman filter which, based on the positions from the ball tracking script, estimate the ball velocity and position. Based on the estimated position and velocity the closest discrete state can be determined and best discovered action mapped out. Applying a new angle to the labyrinth plane up to 25 times per second may end up in a twitchy behaviour, therefore a forgetting factor was applied to make the control smoother. The cost of the forgetting factor implementation is a less responsive controller.

If an action is done and no movement is detected an unstuck procedure is activated. To directly angle the plane aggressively in a direction is risky due to the possibility that the ball in close proximity to a hole. However, an impulse of the plane angle is still desirable in order to get the ball moving again. Thus the unstuck procedure produces two distinct steps in the direction of the current steer signal. The first step with half the maximum steer angle, the second step with maximum steer angle. If the ball is still stuck after the two impulses the unstuck procedure starts to alternate between two directions. The directions are chosen to be ±26° beside the initial direction and the amplitude of the steer signal is still the maximum steer angle. The unstuck procedure is immediately stopped and a normal operation is recovered once a velocity over a certain threshold has been detected.

4.2.3 Servo

The servo script retrieves desired outputs, expressed in angles, from the controller and translates these to usable PWM signals using the Adafruit Servo Driver library for Python. The reason this is separate from the controller script is for historical reasons regarding the previous project using C++ in the controller script. To still allow the old project to be run on the hardware this part was not integrated into the new controller script.

5 Results

As some of the simulation parameters were set ad hoc and in relation to each other, a sensitivity analysis was performed as a basis of discussion for the chosen parameter values. The results are split into two phases. Phase one investigates the sensitivity of simulation parameters that are independent of the full map. Phase two investigates the sensitivity of the rest of the simulation parameters that might depend on the full map.

Phase one and two contains a baseline around which the sensitivity analyses were performed. A summary of the analyses can be found in Table 5.1, which also includes the phase in which a parameter is tested.

5.1 Phase one

The sensitivity analyses of the simulation parameters in this section was done on a small section of the medium map to improve simulation times but still have the physical system to validate the policy on. The start and goal points of the small section are marked out in Figure 5.1 with the start point in black and the goal point in yellow.

Table 5.1: Overview of parameter sensitivity analysis.

Simulation parameter	Section
Phase 1:	5.1
Discretization of speed	5.1.1
Discretization of actions	5.1.2
Disturbance factors	5.1.3
Servo response time	5.1.4
Time between actions	5.1.5
Bounce coefficient	5.1.6
Phase 2:	5.2
Episode step limit	5.2.1
Reward goal	5.2.2
Reward hole	5.2.2
Reward velocity	5.2.2
Reward hole proximity	5.2.2
Primary learning rate	5.2.3
Secondary learning Rate	5.2.4
Learning operations limit	5.2.5

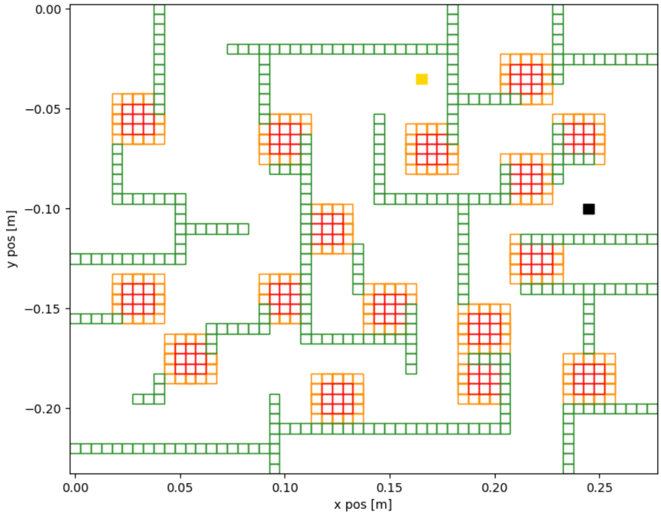


***Figure 5.1:** Visualization of the small map section used in Phase one.*

All parameters are, however, not independent of the map size and can not be tested on a small section of the map and these will be investigated in phase two. The performance of each policy was evaluated by following it 50 times on the physical system. If one parameter test indicated particularly poor performance 25 runs were deemed sufficient to approximate the success rate. The success rate, denoted HW for hardware success rate, is the percentage of runs reaching goal. The baseline parameters for phase one are presented in Table 5.2. These parameter values are used for all parameters that are not tested in a specific sensitivity analysis during phase one.

Table 5.2: *Simulation parameters used as baseline in phase one.*

Parameter	**Value**
Speed discretization	$[0, \pm 0.02, \pm 0.04, \pm 0.06]$ m/s
Action discretization	$[0, \pm 0.2, \pm 0.4, \pm 0.8, \pm 1.2]$ °
Proportional disturbance	20 %
Tilt disturbance	0 °
Servo rise speed	24 °/s
Time between actions	0.08 s
Bounce coefficient	0.35
Episode step limit	2000
Reward goal	400
Reward hole	−4000
Reward velocity	−800
Reward hole proximity	−20
Primary learning rate	Linear 0.8 to 0.5
Secondary learning rate	0.8
Learning operations limit	40×10^6

5.1.1 Analysis of the speed discretization

The discretization of speed, mentioned in Section 4.1.4, is a trade off between fast training and giving the Q-learning algorithm precise knowledge of how the ball is moving which could lead to better control on the physical system. Observations on the physical system, using the Kalman filter estimates, shows that the ball rarely reaches speeds above 0.08 meters per second [m/s]. To discretize speeds above this level would not be efficient.

Nine different parameter sets were designed to determine the sensitivity in discretization of speed. This sensitivity analysis compares three different maximal detectable speeds, 0.08 [m/s], 0.06 [m/s] and 0.04 [m/s], and three levels of discretization, 5 points, 7 points and 9 points. For example, the discretization with maximal detectable speed 0.06 [m/s] and 7 points corresponds to the following discretization: $[-0.06, -0.04, -0.02, 0, 0.02, 0.04, 0.06]$ [m/s]. For each discretization option the success rate of the policy on the physical system is presented in Table 5.3.

Table 5.3: *Resulting success rates for the different sets of speed discretizations.*

	Speeds [m/s]	**HW [%]**
Option 1	[0, ±0.02, ±0.04]	64
Option 2	[0, ±0.03, ±0.06]	56
Option 3	[0, ±0.04, ±0.08]	60
Option 4	[0, ±0.0134, ±0.0268, ±0.0402]	68
Baseline	[0, ±0.02, ±0.04, ±0.06]	90
Option 5	[0, ±0.0267, ±0.0534, ±0.0801]	64
Option 6	[0, ±0.01, ±0.02, ±0.03, ±0.04]	68
Option 7	[0, ±0.015, ±0.03, ±0.045, ±0.06]	72
Option 8	[0, ±0.02, ±0.04, ±0.06, ±0.08]	68

By comparing options 1, 3 and 5 to the others it seems like a too sparse discretization results in poor performance. By comparing options 6 through 8 to the baseline there is a slight decline in performance which might be due to a lack of training, as these many points of discretization leads to a higher number of states. Additionally, option 1 performs rather poorly but has the same sparsity as the baseline. However, it lacks a representation of higher velocities which might be the reason to the performance loss.

5.1.2 Analysis of the action discretization

The discretization of the action space, mentioned in Section 4.1.4, is a trade off between fast training and giving the Q-learning algorithm a wider range of angles that can be utilized by the agent to move.

The different action spacing configurations and their corresponding success rates can be found in Table 5.4.

Table 5.4: *Resulting success rates for the different sets of action discretizations.*

	Action [°]	**HW [%]**
Option 1	[0, ±0.2, ±0.3, ±0.6]	76
Option 2	[0, ±0.2, ±0.4, ±0.8]	64
Option 3	[0, ±0.2, ±0.5, ±1.0]	64
Option 4	[0, ±0.2, ±0.3, ±0.6, ±0.9]	56
Baseline	[0, ±0.2, ±0.4, ±0.8, ±1.2]	84
Option 5	[0, ±0.2, ±0.5, ±1.0, ±1.5]	72
Option 6	[0, ±0.2, ±0.3, ±0.6, ±0.9, ±1.2]	76
Option 7	[0, ±0.2, ±0.4, ±0.8, ±1.2, ±1.6]	56
Option 8	[0, ±0.2, ±0.5, ±1.0, ±1.5, ±2.0]	40

It is difficult to draw any clear conclusions from the results. Options 1, 4 and 6 all have the same action sparsity, yet option 4 performs worse. This is surprising for two reasons. Option 1 lacks a strong steer input that can be used to avoid holes by accelerating quickly in an opposite direction which should lead to poor performance. Option 6 has a larger amount of states which should lead to unreliable training and thus poor performance. Option 4 should then be a good compromise but that is not the case. By comparing options 7 and 8 to the rest it can be seen that these perform worse, which could be due to the lack of training as a result of more states. However, option 6 contradicts this hypothesis. This raises the suspicion that the policy generated by option 6 could be a statistical outlier. The baseline performs the best and based on the results in Section 5.1.1 seems reliable.

5.1.3 Analysis of disturbance

The disturbance is added to simulate model errors and the unpredictability of the physical system. This is done with disturbance in two stages as described in Section 4.1.1, one proportional disturbance and one tilt disturbance.

The proportional disturbance randomly scales the anticipated acceleration with a uniformly distributed percentage. The maximum percentage is the proportion parameter tested in this section.

The tilt disturbance adds a uniformly distributed tilt offset to the simulated labyrinth plane that holds for the duration of an episode. The maximum tilt offset is the tilt disturbance parameter that is tested in this section.

The different values on the disturbance variables are tested both in unison and isolated. The success rates of the disturbance configurations are presented in Table 5.5.

Table 5.5: *Resulting success rates for the different sets of disturbance combinations.*

	Proportional [%]	**Tilt disturbance [°]**	**HW [%]**
Option 1	15	0	68
Baseline	20	0	90
Option 2	25	0	84
Option 3	20	0.05	80
Option 4	20	0.1	68
Option 5	20	0.15	76
Option 6	0	0.15	78

From Table 5.5 it seems like the semi static disturbance does in fact not increase policy performance in presence of the proportional disturbance and isolated it does not achieve the same performance level as the proportional disturbance. In addition the results that this disturbance model produces are not consistent, that there would be a local minima for the tilt disturbance 0.1 seems unlikely and might stem from fluctuations in policy to policy performance. In conclusion, tilt disturbance is not used in the baseline. When comparing option 2 with the baseline both perform quite well. The reason why option 2 was not used as baseline is that adding more disturbance does not seem to further increase performance.

5.1.4 Analysis of servo rise speed

Three different values for the servo rise time, mentioned in Section 2.1.1, will be tested around the approximated servo rise time for a servo angle step. The tested values are 40 ms, 50 ms and 60 ms. The rise times gives three sets of servo rise speeds that will be used in the simulation environment. The step amplitude was presented in 2.1.1 as 1.2° which for the tested rise times gives the servo rise speeds 1.2/0.04 = 30°/s, 1.2/0.05 = 24°/s and 1.2/0.06 = 20°/s. The results of the servo rise speed sensitivity analysis are presented in Table 5.6.

Table 5.6: *Resulting success rates for the servo rise speeds.*

	Rise speed [°/s]	**HW [%]**
Option 1	30	74
Baseline	24	88
Option 2	20	86

When comparing option 2 and the baseline there is no significant difference in performance and they both perform well. Option 1 performs worse which suggests, in comparison to the other 2, that the rise speed is too quick in this configuration. The baseline still performs the best.

5.1.5 Analysis of time between actions

The time between actions, mentioned in Section 2.1.1, effectively determines a time horizon during which the desired steer angle will not change. A too short time between actions has the effect that all actions more or less leads to the current state as the state of the ball will not have time to change. A too long time between actions, in consideration to the labyrinth environment where the distance to a wall is usually short, will make several actions lead to the same states. Some state where the ball has the velocity zero next to a wall, for example a state where the ball is stuck in a corner. Both of these cases cause issues due to the fact that the Q-learning algorithm needs to be able to find a preferable action, which is difficult if many actions lead to the same end state. The sensitivity analysis of time between actions is presented in Table 5.7.

Table 5.7: Resulting success rates for the different times between actions.

	Time [s]	HW [%]
Option 1	0.04	72
Option 2	0.06	36
Baseline	0.08	88
Option 3	0.1	84
Option 4	0.12	48

It is difficult to draw any clear conclusions from the results as they fluctuate. Option 3 and baseline performs the best which suggests that time between actions in the baseline is an acceptable choice.

5.1.6 Analysis of the bounce coefficient

The bounce coefficient, mentioned in Section 4.1.3, determines how much energy is retained when the ball hits a wall, and by extension how the agent will use the walls as it travels through the labyrinth. Different values on this parameter was tested around the baseline to find out how sensitive the end result is to changes in said parameter.

The results of testing the sensitivity of the bounce coefficient are presented in Table 5.8.

Table 5.8: Resulting success rates for the different bounce coefficients.

	Bounce coefficient	HW [%]
Option 1	0.25	80
Option 2	0.3	82
Baseline	0.35	88
Option 4	0.4	82
Option 5	0.45	85

The results are very similar and the differences are statistically insignificant, and thus the baseline is an acceptable choice.

5.2 Phase two

The sensitivity testing of the learning parameters in this section was done on the full medium map. The parameters tested in this section are not independent of the map size, therefore the full map was used to determine their effect on the resulting success rates. The baseline parameters for phase two are presented in Table 5.9. These parameters are used for all parameters that are not tested in a specific sensitivity analysis during phase two.

Table 5.9: Simulation parameters used as baseline in phase two.

Parameter	Value
Speed discretization	[0, ±0.02, ±0.04, ±0.06] m/s
Action discretization	[0, ±0.2, ±0.4, ±0.8, ±1.2] °
Proportional disturbance	20 %
Tilt disturbance	0 °
Servo rise speed	24 °/s
Time between actions	0.08 s
Bounce coefficient	0.35
Episode step limit	20000
Reward goal	10000
Reward hole	−40000
Reward velocity	−800
Reward hole proximity	−50
Primary learning rate	Linear 0.8 to 0.5
Secondary learning rate	0.8
Learning operations limit	650×10^6

5.2.1 Episode step limit

The episode step limit, mentioned in Section 4.1.4, is a measure of how deep the agent explores the labyrinth. If the value is too small it may not be able to reach the set goal point, and if excessively large it makes the learning process slower.

Three different tests were designed to determine the sensitivity with regards to the episode step limit parameter. The results of the testing are presented in Table 5.10.

Table 5.10: Resulting success rates for the different episode step limits.

	Episode step limit	HW success rate [%]
Option 1	15000	24
Baseline	20000	50
Option 2	25000	28

The results suggest that the baseline is a good choice. It is surprising that option 2 is drastically worse than the baseline, it was anticipated that as long as the goal point is reliably reachable the performance should be similar. The reason for the drastically worse performance might be due to inefficient training.

5.2.2 Rewards

The rewards, mentioned in Section 4.1.4, are the tools that are available to shape the behaviour of the derived policy and, to some extent, the training efficiency.

The agent needs to know what is good and what is bad to be able to evaluate a certain move in a certain state. If the evaluation yields unclear results the simulation might not converge to any optimal policy which, of course, is not desired.

Four different tests were designed to explore the sensitivity of each reward around the baseline. A policy was derived for each configuration and then tested on the physical system to determine its effectiveness.

Goal

The result of the sensitivity analysis performed on the goal reward is presented in Table 5.11.

Table 5.11: *Resulting success rates for the different goal rewards.*

	Goal reward	**HW success rate [%]**
Option 1	7500	48
Baseline	10000	50
Option 2	12500	46

The performance of the policies resulting from the different goal rewards is roughly the same. The baseline can therefore be considered an acceptable choice.

Hole

The result of the sensitivity analysis performed on the hole reward is presented in Table 5.12.

Table 5.12: *Resulting success rates for the different hole rewards.*

	Hole reward	**HW success rate [%]**
Option 1	-20000	34
Baseline	-40000	36
Option 2	-60000	24
Option 3	-80000	10

It was first thought that as long as the hole reward was negative enough it would not affect policy performance to a large extent. The results suggests otherwise and it is clear that a too large value results in poor performance. It is, however, not clear as to why this loss in performance occurs. One possible explanation could be that a too negative reward affects the state close to the hole too much. The effect is that states that could be good candidates along the trajectory are not valued as much to reach the goal, since the reward for them are lower. Another interesting observation is that the baseline also had a significant decrease in performance compared to the results in 5.11. In other words the policy performance seem to fluctuate from training to training, even though the parameters are the same for both training sessions.

Hole proximity

The result of the sensitivity analysis performed on the hole proximity reward is presented in Table 5.13.

Table 5.13: *Resulting success rates for the different rewards on hole proximity.*

	Hole proximity reward	HW success rate [%]
Option 1	0	28
Option 2	-15	26
Option 3	-20	26
Option 4	-25	44
Option 5	-30	28
Option 6	-35	24
Option 7	-40	50
Baseline	-50	46
Option 8	-60	36

In this analysis it is not as clear which value on the reward that is the best option. Option 4 and option 7 might be contenders. Comparing the result of the baseline in this section to the result in section 5.2.2 and 5.2.2 it seems like the result can vary between separately trained policies. Therefore the contending options 4 and 7 were retrained and evaluated a second time yielding the results in Table 5.14.

Table 5.14: *Reevaluation of options four and seven.*

	Hole proximity reward	HW success rate [%]
Option 4	-25	40
Option 7	-40	36

The reevaluation confirms that option 7, like the baseline, fluctuates in success rate whereas option 4 seems rather consistent and could be a better option than the baseline. There is no guarantee that option 4 would continue to yield a consistent success rate if evaluated a third time. However, the agent showed some desired characteristics when following the policy from option 4 when compared to the baseline. Primarily, the agent behaved less aggressively near multiple holes and was not as nervous when it needed to travel past individual holes. Option 4 could be useful to generate a more consistent policy.

Velocity

The result of the sensitivity analysis performed on the velocity reward is split into two parts corresponding to the two implementations. The first results, presented in Table 5.15, is for the implementation where the baseline is used and the velocity is rewarded with a negative quadratic scaling of the reward.

Table 5.15: *Resulting success rates for the different velocity rewards of quadratic implementation.*

	Velocity reward	**HW success rate [%]**
Option 1	-600	48
Option 2	-700	20
Baseline	-800	50
Option 3	-900	50

The second results, presented in Table 5.16, corresponds to the implementation where the velocity has a negative reward under the absolute velocity of 0.01 [m/s] and over 0.07 [m/s]. These limits were set ad hoc but with two things in mind. Firstly, 0.01 [m/s] is the threshold where the unstuck sequence activates on the hardware. Secondly, the highest velocity that is with certainty correctly fit into the speed discretization is 0.07 [m/s] as the highest value in the discretization is 0.06 [m/s] and the spacing is 0.02 [m/s] which means that a velocity infinitely close to 0.07 [m/s] would be correctly approximated by 0.06 [m/s] if the discretization were to continue.

Table 5.16: *Resulting success rates for the different velocity rewards of threshold implementation.*

	Velocity reward	**HW success rate [%]**
Option 1	-15	20
Option 2	-25	8
Option 3	-35	20

When inspecting the success rates in Table 5.16, it is clear that something has happened to the performance of the resulting policy when compared to the first implementation of velocity reward. One possible reason to the performance decline could be that it is harder for the Q-learning algorithm to distinguish between actions if the immediate reward for different actions are the same. For example, if the ball is travelling in the $\hat{x}$ direction with velocity 0.04 the difference in immediate reward between doing nothing and deaccelerating slightly is non existent. However, if there is a hole in the direction of travel deaccelerating would be preferable which the implementation of negative quadratic scaling rewards immediately.

5.2.3 Primary learning rate

When updating the values of the primary Q-matrix the primary learning rate, mentioned in Section 4.1.4, states the ratio between how much of the new information that is learned and how much of the values in the secondary Q-matrix that are retained. The primary learning rate can be a static value, but in this report it is linearly declining as the training progresses to fine tune the policy towards the end of simulation. To explore the sensitivity of the primary learning rate, nine policies were tested on the physical system. The result of the sensitivity analysis is presented in Table 5.17.

Table 5.17: *Resulting success rates for the different primary learning rate configurations.*

	Primary learning rate	HW success rate [%]
Option 1	[0.7, 0.4]	24
Option 2	[0.7, 0.5]	32
Option 3	[0.7, 0.6]	44
Option 4	[0.8, 0.4]	24
Baseline	[0.8, 0.5]	46
Option 5	[0.8, 0.6]	20
Option 6	[0.9, 0.4]	40
Option 7	[0.9, 0.5]	32
Option 8	[0.9, 0.6]	0

By comparing the different options it seems that the primary learning rates with a mean value of 0.65 performs rather well, that is option three, six and baseline. For the other configurations a probable cause for the performance loss is underfitting or overfitting. The baseline is an acceptable choice according to the sensitivity analysis. In Figures 5.2 and 5.3 the training measurements for two different primary learning rate setups are shown.

In the figures 5.2 and 5.3 "Total reward per iteration" is the accumulated reward for each attempt throughout the training process. "Steps per iteration" is the total amount of steps taken during each attempt. The third graph, "Total reward divided by step per iteration", represents the total reward for an attempt divided by the amount of steps in the same attempt. All graphs are smoothed with a moving average in order to make trends easier to observe. As an effect, the initial and end values may be exaggerated, for example steps per iteration graph in Figure 5.3.

The setup in Figure 5.2 is Option 6 from Table 5.17 and the setup in 5.3 is Option 8. In Table 5.17 it is clear that these options are very different when it comes to performance and when we compare the training data the graphs do differ as well. Looking at the end of the "Total reward per iteration" and "Steps per iteration" graphs, an exponential behaviour can be observed in 5.3 that is not pronounced in Figure 5.2. This could be a sign of overfitting which could explain the poor performance of Option 8.

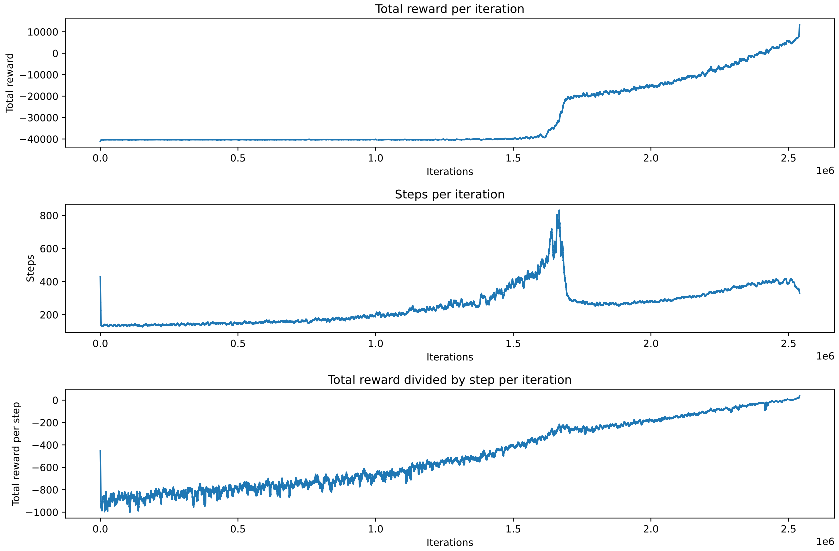


Figure 5.2: *The resulting statistics from training the policy with primary learning rate [0.9, 0.4].*

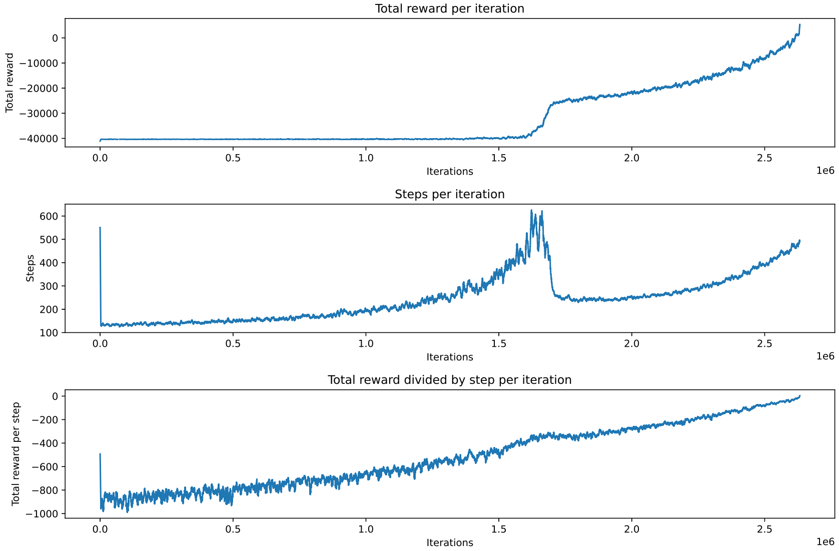


Figure 5.3: *The resulting statistics from training the policy with primary learning rate [0.9, 0.6].*

5.2.4 Secondary learning rate

To prevent the expected rewards in the primary Q-matrix from growing too fast from a good or a bad move a secondary learning rate, mentioned in Section 4.1.4, in combination with a secondary network is utilized. To explore the sensitivity of the secondary learning rate a few adjacent values to the baseline were tested on the physical system. The result of the sensitivity analysis is presented in Table 5.18.

Table 5.18: *Sensitivity analysis result of the secondary learning rate.*

	Secondary learning rate	**HW success rate [%]**
Option 1	0.65	24
Option 2	0.7	40
Option 3	0.75	36
Baseline	0.8	28
Option 4	0.85	12
Option 5	0.90	16
Option 6	0.95	4

This result shows once again that the performance of the baseline policy is not consistent. In this sensitivity analysis option two and three shows promising results that are close to the previously displayed results of the baseline. It could prove useful to decrease the secondary learning rate when developing a high performance policy, however not too much as option one suggests.

5.2.5 Learning operations limit

The learning operations limit, mentioned in Section 4.1.4, is a measure of how much training the agent does in the simulation environment. Too few learning operations and the path does not converge. Too many learning operations on the other hand, makes the policy overfitted. In both cases performance suffers. The results of the sensitivity analysis of the learning operations limit are presented in Table 5.19.

Table 5.19: *Sensitivity analysis result of the learning operations limit.*

	Learning operations limit	**HW success rate [%]**
Option 1	450×10^6	32
Option 2	500×10^6	48
Option 3	550×10^6	40
Option 4	600×10^6	36
Baseline	650×10^6	50
Option 5	700×10^6	52
Option 6	750×10^6	68
Option 7	800×10^6	60
Option 8	850×10^6	40

By comparing the results it seems that more training results in better policy performance to a certain degree. Option eight performs poorly compared to option six and seven. This could either be due to overfitting or a statistical variation. Baseline and option five performs to a similar degree as has been observed previously from the baseline. Option one also performs poorly, which instead could be to underfitting or a statistical variation. Ideally it seems that the learning operations limit could benefit from being set higher.

5.3 Experience replay evaluation

Experience replay, mentioned in Section 3.1.1, uses past state transitions multiple times to extract more information about the environment. This is done by sampling a random subset of experiences stored in a memory. The memory stores information for 300000 state transitions and each sampled subset consists of b state transitions. In Table 5.20 the success rate for experience replay with two different sample sizes are presented side by side with the results of the baseline without experience replay. That is, a sample size of 1 means no experience replay is used.

Table 5.20: *Success evaluation of experience replay.*

	Sample size	HW success rate[%]
Baseline	1	44
Option 1	2	8
Option 2	3	0

The results from this test shows that experience replay cannot be used without extensive complimentary modifications. Lower learning rate and larger sample sizes in combination with less training could improve the results as that would decrease the risk of overfitting which may be the reason to the poor performance.

5.4 The best performing policy achieved

The sensitivity analyses in Section 5.1 and 5.2 showed that the baseline could benefit from some adjustments. A new policy was developed with adjusted parameters in an effort to improve performance further. In Table 5.21 the adjusted parameter set is presented with adjusted parameters marked by an asterisk.

Table 5.21: *Simulation parameters used in an effort to boost performance.*

Parameter	**Value**
Speed discretization	$[0, \pm 0.02, \pm 0.04, \pm 0.06]$ m/s
Action discretization	$[0, \pm 0.2, \pm 0.4, \pm 0.8, \pm 1.2]$ °
Proportional disturbance	20 %
Tilt disturbance	0 °
Servo rise speed	24 °/s
Time between actions	0.08 s
Bounce coefficient	0.35
Episode step limit	20000
Reward goal	10000
Reward hole	−40000
Reward velocity	−900
Reward hole proximity*	−25
Primary learning rate	Linear 0.8 to 0.5
Secondary learning rate*	0.7
Learning operations limit*	750×10^6

The distribution of the failed runs, referring to which hole the run was terminated in, are presented in Figure 5.4. The holes are identified by the numbers declared in Figure 2.3. It is clear that the policy derived from the RL algorithm struggled the most with hole six. This was partly due to the plane being slightly warped which was most evident around the top right and bottom left corner in Figure 5.5.

Figure 5.5 shows the trajectory of a successful run on the hardware where the ball can be observed rolling in the wrong direction in the passage by hole six several times, due to the warped plane, before successfully passing.

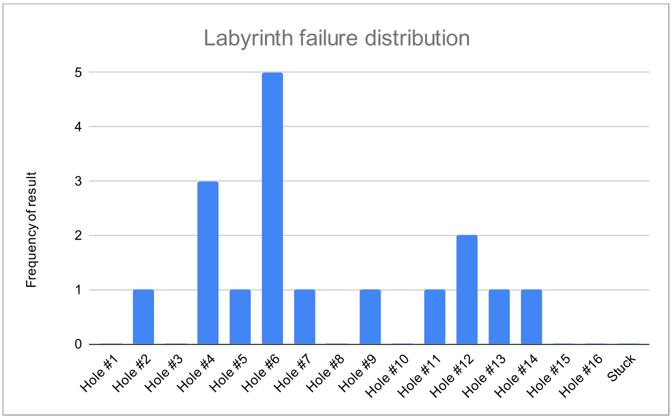


Figure 5.4: *The distribution of the holes that the ball fell in out of 50 rounds.*

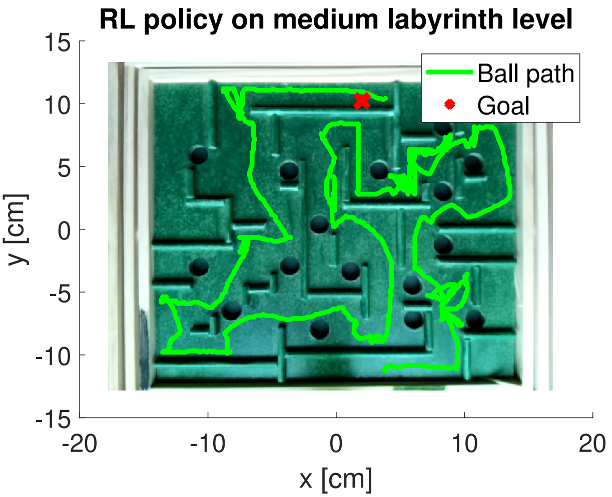


Figure 5.5: *The trajectory of a successful run on the physical system using RL derived policy.*

In Figure 5.6 then simulated trajectories are presented to show the expected trajectory. Comparing with Figure 5.5 it can be seen that the navigation in the simulation environment is better. This is of course due to the fact that the policy is trained to control the model and the modelled disturbances implemented in the simulation environment, not the more unpredictable physical system. When applied to the physical system it still works well, but some issues occur in certain areas like the area in between hole one and two, and then in the passage between hole five, six and seven. It is not clear from Figure 5.5, but the policy also struggles to overcome the labyrinth plane tilt in the bottom left corner of the map.

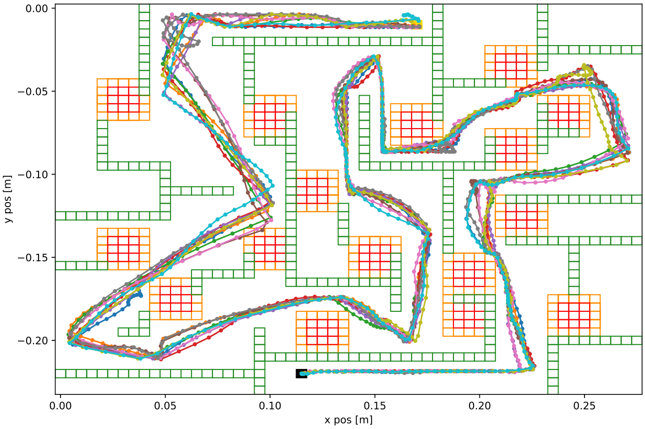


Figure 5.6: *The resulting paths following the policy in the simulation environment.*

In Figure 5.7 the reward progression during the training is presented. Perhaps the most important thing to note is that the total reward per iteration increases linearly towards the end of training and not exponentially which can be a sign of overfitting.

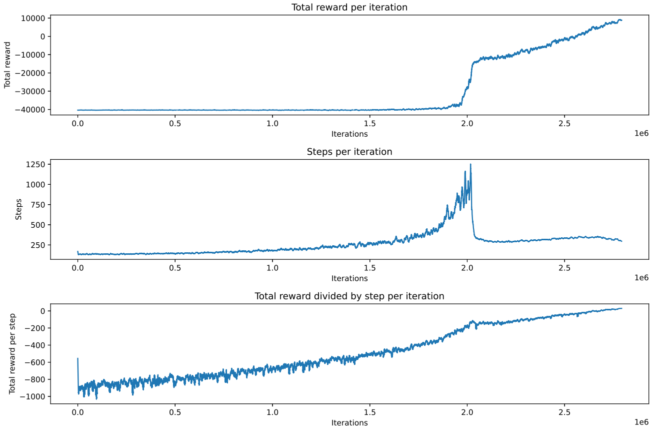


Figure 5.7: *The reward progression during simulation.*

5.5 Gain Scheduling LQR evaluation

The LQR with gain scheduling had the best performance when [4] was conducted, then the control method managed 78.2 % success rate. However, the physical system is made of wood, a material that is sensitive to changes in humidity and temperature that over time could affect the the way the physical system behaves. There are also plastic linkages between the servos and the labyrinth plane that could have gotten worn out from repeated tests and investigations since the 78.2 % success rate were achieved. Another factor that might affect the performance is that the wooden plane has been colored, which might have warped the labyrinth plane or introduced irregularities on the surface. Thus the gain scheduled LQR was tested again to find out if the physical system has gotten harder to control since [4] were conducted.

The hole statistics are presented in Figure 5.8 and, judging by this figure, the irregularities seems to have increased over time. In Figure 5.9 the trajectory of a successful run on the hardware can be seen. From this trajectory it can be seen that the LQR struggles in the same places that the RL derived policy in 5.4 do which suggests that the tilt working against the control algorithms is stronger in these areas. Specifically the area around hole 1 and in the bottom left corner by hole 13 there seem to be a lot of tilt to overcome. Another area that is difficult for both algorithms is the passage between hole six and four. The RL derived policy drives the ball back and forth here for some reason and the LQR struggles to avoid hole 6. The place where the LQR loses in comparison to the RL derived policy is around hole 12 where the ball falls in 8 out of 50 times instead of 2 times like the RL policy. A comparison in success rate between the different control methods is presented in Table 5.22.

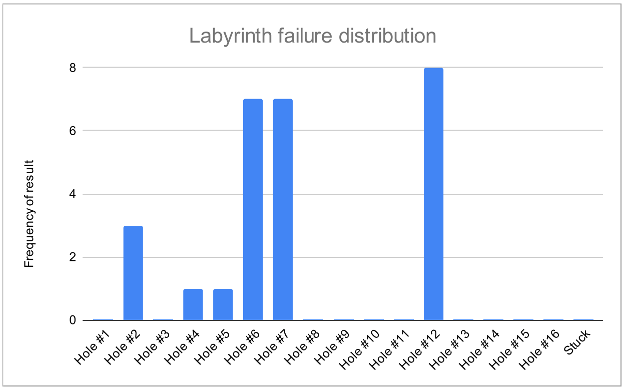


Figure 5.8: *The distribution of the holes that the ball fell in. Out of 50 runs the ball ended up in a hole 27 times.*

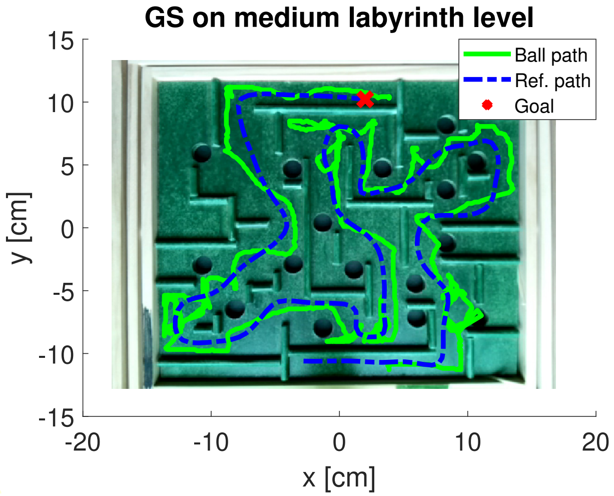


__Figure 5.9:__ The trajectory of a successful run on the physical system using LQR.

__Table 5.22:__ Comparison of success rates in solving the BRIO labyrinth using different control methods.

Control method	**Success rate [%]**
Gain scheduled LQR (2020)	78.2
Gain scheduled LQR (2022)	46
Best performing RL policy	66

6

Challenges and future work

The clear challenge that was found when trying to use a simulation derived policy on a physical system was to find an accurate representation of the disturbances present. The warp of the labyrinth plane proved to be a real challenge to overcome. The warping was so large in certain areas that it completely nullified the steer inputs provided by the policy. This is a perfect area to investigate in future work. The policy provided by the simulation could be a basis for an online policy estimation that tweaks the steer signals while running on the physical system with the goal of mitigating the effects of the warped labyrinth plane.

Further on the subject of warping, it has probably gotten worse since the work in [4] was performed. This is suspected since a far worse performance of the gain scheduled LQR was recorded during this thesis than what was presented in [4]. During this thesis the labyrinth plane was painted green to improve contrast between the ball and the plane, this could be one contributing factor to worsen the warp of the plane. Other factors could be the time that has passed since the last thesis and the fact that the physical system is made from wood. Wood is largely affected by air humidity and could also be affected by uneven drying when the plane is heated by sunlight. This could also be a contributing factor to the wide variance in policy performance between individual training sessions.

As previously mentioned, the modelling of disturbance is a big challenge and there was one type of disturbance that would have been interesting to implement if there were more time: observatory disturbance. Giving the Q-learner less accurate knowledge of where the ball is may be a closer match to the ball positioning available on the physical system. This disturbance could improve the respect for holes since the Q-leaner would not have perfect knowledge of where the ball is. It is possible that the other disturbance implementations may benefit from tweaking as well if a new type of disturbance is added.

Other ML algorithms could have been tried as well. By coloring the walls in a bright colour to distinguish them from the rest of the labyrinth plane it would be interesting to see what could be achieved with a deep Q-learning algorithm that trains online using the video feed as input, perhaps similar to the one in [3].

7 Conclusion

The problem formulations stated in Section 1.2 will be collected and summarized in this section.

1. What challenges arise when using a steer policy derived from simulation? The policy that is derived in the simulation environment is hard coded on to the physical system and that shows. Misalignment of the labyrinth plane, beginning a session without the labyrinth plane in level and other varying factors affect the performance of the policy and it has no way of compensating for these disturbance factors. Despite this shortcoming the Q-learner manages to plan such a safe path that it more often than not can travel through the entire labyrinth. To not be dependent on a path to follow and to be able to take advantage of the walls to stay at a safe distance from holes is the algorithms greatest advantage. One oversight that was noticed by the end of the thesis work was the observational disturbance. As the labyrinth plane tilts the positioning of the ball becomes faulty. To either model this dynamic as part of the model or by introducing a random observational disturbance might make the Q-learner even more cautious and by extension more successful. The observational disturbance, however, is small.

2. Is it enough to add uncertainty to the motion model in order to bridge the gap between the simulation environment and the physical system? The gap has not fully bridged by introduction of disturbance. However, as seen in section 5.1.3 to include disturbance in the simulation environment improves the performance of the derived policy greatly. The observable effect is that walls are used to a larger extent when navigating through the maze. This makes for safer driving and in turn a higher success rate. However, the walls are not being used to the extent that they could be. If this is because there is an even better disturbance model out there or if the rewards can be set differently to promote this behaviour are subjects for closer investigation. Investigating if it is possible to model the warping of the labyrinth plane could increase the fidelity of the simulation environment and lead to improved performance. To get a complete representation, fine-tuning online on the physical system could be required.

3. What performance benefits, if any, can be observed using the Q-learning method compared to a gain scheduled LQR? As seen in Section 5.4 and 5.5 the RL derived policy outperforms the more classical path following gain scheduled LQR by some margin, 68 % success rate compared to 46 %. But the LQR still have some benefits over the RL derived policy, the biggest of which is the ability to compensate for disturbances that are present on the physical system. There is a possibility that the two techniques could be combined to achieve even better performance. By doing path planning in the simulation environment to plan a safe route, and then follow that route with the gain scheduled LQR to be able to compensate for disturbances, an even higher success rate might be possible.

Bibliography

[1] Richard Bellman. Dynamic programming. *Science*, 153(3731):34–37, 1966. doi: 10.1126/science.153.3731.34. URL `https://www.science.org/doi/abs/10.1126/science.153.3731.34`.

[2] BRIO, (accessed: 30.12.2021). URL `https://www.brio.se/produkter/alla-produkter/roll-lek-spel/labyrinth/`.

[3] Volodymyr Mnih et al. Human-level control through deep reinforcement learning. *Nature*, 2015. URL `https://web.stanford.edu/class/psych209/Readings/MnihEtAlHassibis15NatureControlDeepRL.pdf`.

[4] Emil Frid and Fredrik Nilsson. Path following using gain scheduled LQR control. *diva*, 2020. URL `http://liu.diva-portal.org/smash/get/diva2:1451989/FULLTEXT01.pdf`.

[5] Torkel Glad and Lennart Ljung. *Reglerteori - Flervariabla och olinjära metoder*. Studentlitteratur AB, 2003.

[6] Fredrik Gustafsson. *Statistical Sensor Fusion*. Studentlitteratur AB, 2018.

[7] Ulf Janfalk. *Linjar algebra*. MAI LiU, 2014.

[8] MathWorks. Filtering and smoothing data, (accessed: 03.05.2022). URL `https://se.mathworks.com/help/curvefit/smoothing-data.html?fbclid=IwAR1jdXDb4_N7B0g5yCPNiJNAWvWriMSbvT53ZZjqa593tOp2LEwsk5PrQHE`.

[9] Christopher J.C.H. Watkins and Peter Dayan. Technical note q-learning. *Kluwer Academic Publishers*, 1992. URL `https://link-springer-com.e.bibl.liu.se/content/pdf/10.1023/A:1022676722315.pdf`.

[10] Yingjun Ye, Xiaohui Zhang, and Jian Sun. Automated vehicle's behavior decision making using deep reinforcement learning and high-fidelity simulation environment. *Transportation Research Part C: Emerging Technologies*, 107:155–170, 2019. ISSN 0968-090X. doi: https://doi.org/10.1016/

j.trc.2019.08.011. URL https://www.sciencedirect.com/science/article/pii/S0968090X19311301.

[11] Shangtong Zhang and Richard S. Sutton. A deeper look at experience replay. *CoRR*, abs/1712.01275, 2017. URL http://arxiv.org/abs/1712.01275.