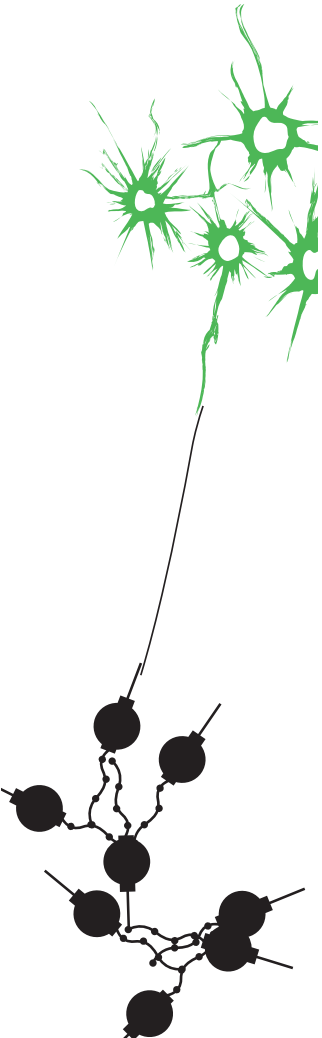


RAM ROBOTICS AND MECHATRONICS

LEARNING TO EXPLORE AND MAP WITH EKF SLAM AND RL

B.T. (Bastian) van Manen

MSC ASSIGNMENT

Committee:
dr. ir. J.F. Broenink
N. Botteghi, MSc
dr. M. Poel

August 2021

056RaM2021
Robotics and Mechatronics
EEMathCS
University of Twente
P.O. Box 217
7500 AE Enschede
The Netherlands

UNIVERSITY OF TWENTE. | TECHMED CENTRE UNIVERSITY OF TWENTE. | DIGITAL SOCIETY INSTITUTE

Summary

A fundamental aspect of research in mobile robots is autonomous navigation. When Global Positioning Systems (GPS) is not available, Simultaneous Localization And Mapping (SLAM) provides a technique to infer the robot position and build a map of the unknown environment. Extented Kalman Filter SLAM is one of the most popular filter SLAM methods.

This thesis implements a novel Reinforcement Learning (RL) based EKF SLAM algorithm for exploration of static unknown environments. The proposed approach uses two hierarchically structured RL policies for the generation of sensing locations and trajectories parameterized by 3^{rd} order Bézier curves. Additionally, the thesis focuses on exploring how EKF and RL can be combined in a single framework to minimize the EKF uncertainties and improve the map accuracy. While navigating in an unknown environment, the EKF SLAM combined with a Light Detecting And Ranging (LIDAR) sensor provides the robot with an estimate of its current pose along with the estimated position of the environment landmarks. A map is build in the form of an occupancy grid by merging the estimated robot pose with the LIDAR data.

Afterwards, the high-level RL network selects informative sensing locations based on the occupancy grid and the actual LIDAR output. The high-level policy assists the robot with obstacle avoidance by only selecting sensing near the robot. To reach these sensing locations, a second RL network is trained to compute an obstacle free trajectory that minimizes the uncertainty in the EKF estimates.

The low-level RL agent manages the shape of the trajectory by modifying the positions of the Bézier control points. The thesis explores two continuous reward functions to minimize the EKF uncertainty. The first reward function computes the sum of the diagonal elements of the Kalman gain. Whereas, the second reward function directly computes the sum of the diagonal elements of the EKF covariance matrix holding the variances in the EKF estimates. To cover obstacle avoidance in the learned policy, a final reward function is studied including a discrete penalty upon collision with an obstacle.

At last, a path planner discretizes the chosen trajectory into a fixed amount points serving as short term goals for a PD controller to compute appropriate motor control commands. The RL and SLAM-based approach is implemented and tested in a 2D Python environment. Results show that the low-level policy does manage to select trajectories that reduce the EKF uncertainty. But the policy has more difficulty handling obstacle avoidance. The high-level policy did not manage to perform better than random sensing locations when evaluating the map coverage. The occupancy grids as well as the reward function likely did not provide enough information for the high-level policy to converge. Longer training is also necessary. The trajectory generation can be improved by adding a reward function based on the ground-truth positions of the robot and the landmarks. Additionally, obstacle avoidance can better be incorporated into the reward function using an exponential of the distance between the robot and the closest obstacle as a penalty. The high-level policy can further be improved by providing large occupancy grids to the RL network. Large occupancy grids provide better accuracy of the location of obstacles and unexplored areas. A reward function based on ground-truth maps is also thought to be more effective compared than the simple visited landmark percentage utilized until now.

Contents

Acronyms

ANN Artificial Neural Network

CVPR Computer Vision and Pattern Recognition

D4PG Distributed Distributional Deep Deterministic Policy Gradient

DDPG Deep Deterministic Policy Gradient

DQN Deeq Q-Network

EKF Extended Kalman Filter

GPS Global Positioning System

IMU Inertial Measurement Unit

LIDAR Light Detection And Ranging

LSTM Long Short-Term Memory

MDP Markov Decision Process

MLP Multi Layer Perceptron

OU Ornstein Uhlenbeck

POMP Partially Observable Markov Decision Process

RL Reinforcement Learning

SEIF Sparse Extented Information Filter

SLAM Simultaneous Localization And Mapping

TD Temporal Difference

TD3 Twin Delayed Deep Deterministic Policy Gradient

1 Introduction

1.1 Autonomous navigation for mobile robots

Over the years it has become essential for any mobile robot to have a navigation system that tells the robot how to move around in an environment and reach a certain destination. Autonomous navigation refers to the robot's ability to perform all navigational tasks itself without human intervention. The robot is in that case able to self-steer from one place to another based on on-board computers and sensors. Autonomous navigation has already proven its usefulness during the years and is currently widely used in factories, storage facilities and agriculture for instance. Still, the market is expected to grow significantly in the next few years with projections ranging from US$ 8.5 billion by the end of 2025 (ReportLinker, 2021) to US$ 13.5 billion in 2030 with demands coming from both commercial and military sectors (MarketsandMarkets, 2019). Additionally, upcoming technologies such as self-driving cars or extra-planetary exploration robots all benefit from improved navigational skills. At last, with the ever more increasing number of robots introduced in almost all fields of the economy, it is clear that autonomous navigation in mobile robots is and will still be a relevant area of research in the near future.

Nonetheless, in order to navigate in an environment, the robot must know its position at all times. Most often this is done using a combination of GPS (Global Positioning System) and IMU (Inertial Measurement Unit) data. But GPS requires an active signal between the robot and the four GPS satellites in orbit around the Earth. When navigating in enclosed environments, GPS signal is not always available. Think of large buildings, bunkers, cave systems or even underwater environments as areas where GPS signal is not present. Therefore, another method to determine the robot's precise location is necessary that works in enclosed and remote environments. This is where Simultaneous Localization And Mapping (SLAM) comes in.

Unlike GPS, SLAM does not require external aid from satellites, instead it uses an on-board sensor to capture the state of the environment around the robot from which it can determine the relevant information. Most commonly used sensors are cameras and Light Detection and Ranging (LIDAR) equipment. SLAM was first introduced in the early 1990's by Smith (1986). Since then many algorithms have been proposed with each their advantages and disadvantages such as EKF SLAM (Bailey et al., 2006), FastSLAM (Montemerlo, 2002), GraphSLAM (Grisetti et al., 2010) and OrbSLAM (Elvira, 2020).

Exploring with autonomous mobile robots has several key benefits over remote-controlled mobile robots. Firstly, it permits to eliminate driving errors by the human operator affecting the map accuracy. Secondly, it allows to explore dangerous or remote areas with long communication delays or no communication possibilities at all. Long cave systems or distant planetary exploration are great examples where autonomous navigation is essential. Furthermore, it is a crucial step in exploring unknown environments as the user has no a priori knowledge about the environment. Effectively making it impossible for the user to plan trajectories.

1.2 Exploring unknown environments with RL and EKF SLAM

The approach proposed in this thesis uses Reinforcement Learning (RL) to navigate the robot through the environment. Combined with EKF SLAM and a LIDAR sensor to infer the robot pose and create a map of the unknown environment.

Reinforcement Learning is a type of Machine Learning where an algorithm (also called an agent) is trained to choose correct decisions by receiving either a reward when the decision is beneficial or a penalty when it is not.

The benefit of Reinforcement Learning over other branches of Machine Learning such as Su-

pervised Learning is that it does not require ground-truth data. Considering path planning, this is a huge benefit when learning to navigate in continuous environment since it is not tractable to find the ground-truth trajectory for every possible position of the robot in the environment.

The Extented Kalman Filter (EKF) is a version of the Kalman Filter but extended to include non-linear state transitions and measurement models. By approximating non-linearity's using Taylor Series Expansion, EKF can proceed according to the linear Kalman Filter method. The EKF SLAM algorithm is composed of two main steps, namely a prediction step and a correction step. In the first step, the algorithm predicts the robot pose given its motion model. The second step corrects the estimated robot pose using the measurement model and updates the map. The EKF map consists of a list of landmarks extracted from the LIDAR data.

Three main advantages make EKF SLAM an interesting choice for autonomous navigation. First, EKF is known to be particularly fast especially for smaller maps. Second, when handling obstacle avoidance, the EKF map along with the robot pose and the LIDAR data can be passed to the Reinforcement Learning agent instead of an occupancy grid. This yields a reduced state vector and thus avoids the need for deep Neural Networks in the policy network. Third, the EKF SLAM algorithm computes the uncertainty in its estimates as well as the Kalman gain. Both metrics can be used to improve the decision making of the Reinforcement Learning agent.

The drawbacks of using EKF is that it becomes computationally demanding for large maps when updating the covariance matrix (Paz and Neira, 2006). Yet, there exists methods and variations of the EKF algorithm such as the Sparse Extended Information Filter (SEIF) SLAM to improve scalability (Eustice et al., 2005).

1.3 Previous work

Even though, the combination of Reinforcement Learning and EKF SLAM to improve navigation is an active field of research, this report focuses on the approach proposed by Kollar and Roy (2008). In the paper, the authors use EKF SLAM together with a RL model to generate trajectories that minimize the EKF uncertainty. A key take-away from the paper is that sharp turns in the robot's trajectory result in mapping errors. For this reason, the authors stress the importance of choosing a smooth parameterization for the generated trajectories. Thus, they define the action space of the RL agent as a set of 3^{rd} order parametric polynomials in the x and y directions from the robot pose to the next sensing location. Here, sensing locations define positions of interest in the environment that the agent should visit while exploring the environment. They proceed by training the agent in simulation before it is tested in the real-world. In both the simulations and in the real-world test, results show a smaller error in the estimated robot position for the trajectories learned with Reinforcement Learning compared to the shortest path. Consequently, the paper validates the benefit of using RL as a path planning method to improve map accuracy.

Nevertheless, the research from Kollar and Roy (2008) assumes that the robot knows the position of the sensing locations beforehand and does not cover the exploration of unknown environments. In addition, they do not deal with obstacle avoidance, an imperative part in autonomous navigation. Instead, the authors choose the best obstacle free trajectory.

In the state-of-the-art, most if not all of the exploration algorithms for mobile robots employing RL, either focus on obstacle avoidance or on maximizing map coverage without optimizing for map accuracy. Moreover, when it comes to mobile robots, the state-of-the-art still uses discrete actions for their RL agent in charge of navigation instead of trajectories like in Kollar and Roy (2008). Discrete actions significantly simplify navigation but reduce the map accuracy since the overall robot trajectory is less smooth. All in all, what is lacking in the current state-of-the-art for exploration in mobile robots, is an algorithm that both maximizes map accuracy and

map coverage while taking into account obstacle avoidance. The thesis seeks to create such an algorithm by merging EKF SLAM with RL to generate obstacle free smooth trajectories as well as optimal sensing location for full map coverage.

1.4 Research questions and thesis contribution

To study the effectiveness of combining EKF SLAM with RL in creating an algorithm that maximizes map accuracy and map coverage, the following main research question and four subquestions are considered.

1. How can we tightly incorporate RL and EKF in a single framework?
    (a) How can Reinforcement Learning be used to select informative target positions and generate informative trajectories?
    (b) How can we hierarchically structure the RL policies?
    (c) How can we shape the reward function?
    (d) What is the benefit of the proposed Reinforcement Learning approach?

The work presented in this thesis is novel considering that it proposes a new algorithm for exploration with mobile robots. The approach combines sensing location generation for optimal map coverage together with trajectory generation for maximum map accuracy in static unknown environments. The coverage planner is integrated into the algorithm as a high-level policy and chooses adequate sensing locations governing the overall movement direction of the agent. In turn, the path planner works as a low-level policy. It generates trajectories deciding how the agent reaches the sensing locations in such a way that minimizes errors in the map. This novel exploration algorithm is not only unique considering that it takes into account map coverage and map accuracy simultaneously, but also includes obstacle avoidance. Additionally, trajectory generation is a topic that is vastly utilized for aerial vehicles and robotic arms but remains understudied for ground-based exploration in the field of mobile robotics.

1.5 Thesis outline

The proposed algorithm consists of two Reinforcement Learning agents (the coverage planner and the path planner), hierarchically structured. Together they form the brain of the robot and decide where and how the robot should explore the unknown environment. The coverage planner makes decisions based on information it receives from the EKF SLAM block. While the path planner sits hierarchically lower and uses information from the coverage planner as well as the EKF SLAM module. Hence, the success of the proposed EKF SLAM and RL algorithm depends on how well both RL agents can be trained to find optimal trajectories and sensing locations. As such, the training of the RL networks also pose the most challenge in this thesis.

Due to the current COVID-19 pandemic, the research is only executed in simulation considering no real-life experiments could be performed. Additionally, the proposed algorithm is implemented and tested in Python 3.8.5 because of three main reasons. First, Python gives access to a large amount of Reinforcement Learning and Machine Learning libraries with greater documentation support compared to C++. Second, to provide the computational power (CPU and GPU) required to train two RL networks, the University's HPC (High Performance Computing) cluster must be utilized. The HPC cluster has very good support for Python scripts. Third, a RL framework was already in place at the University which could be reused to speed up the creation of a suitable RL training environment.

Finally, the proposed algorithm is independent of the chosen method of implementation and thus could also very well be implemented in C++ or ROS. The algorithm is also independent of the chosen SLAM techniques once both RL networks have been trained.

2 Background

2.1 Reinforcement Learning

In RL the process of learning a certain task requires the presence of an agent and an environment on which it can act. The agent is the decision making body and usually consists of a neural network. On the other hand, the role of the environment is to react on the agent's actions and provide feedback on how beneficial the action is by means of a reward, see Figure 2.1. The goal of the agent is to maximize the accumulated reward in a single episode. Repeated interaction between agent and environment permits the agent to gain progressively more experience and thus improve its decision making skills.

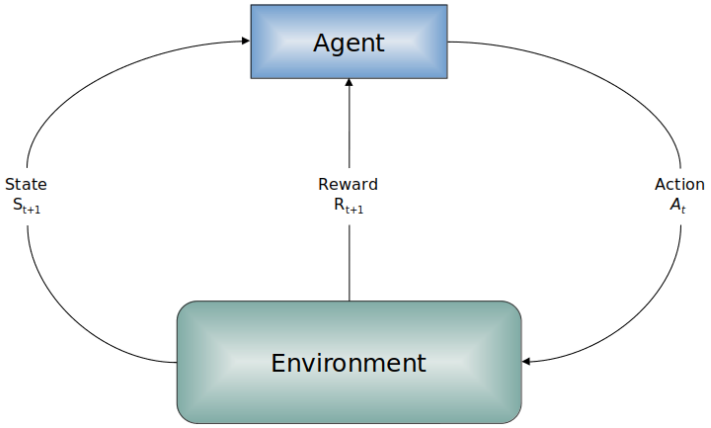


Figure 2.1: Reinforcement Learning cycle. The agent in blue generates a new action based on previously acquired state and reward. The environment takes in the action and returns the corresponding state and reward.

On top of giving a reward, the environment must also tell the agent in which state it lays such that the agent knows what effect its action has on the environment. In essence, the reward is only used during training as a measure of performance feedback for the agent. Whereas the state describes the current condition of the environment. In the context of robotics, the environment is in many cases a simulation of real-life physics.

Nonetheless, the reward function is a key factor that ultimately determines if the agent adopts the desired behavior. It often consists of a combination of positive and negative rewards, the latter also called a penalty. The reward function should penalize bad behavior while encouraging desirable behavior. Therefore, an adequately shaped reward function is equally important as a well defined environment.

Although, Figure 2.1 describes the main idea behind the learning process of an RL agent, every problem to be solved using RL must be formulated mathematically first. To do so, the following section introduces the decision process at the root of most RL problems and named after the Russian mathematician Andrey Markov, the Markov Decision Process (MDP).

2.1.1 Markov Decision Process

A MDP is a decision process to model the interaction between agent and environment. It is usually characterized by the 5-tuple $< S, A, T, R >$ where S is a set of states, otherwise also called a chain. Each state is uniquely described from anything like a set of coordinates to any other feature uniquely defining the state. Together they encompass all possible configurations in the environment.

Furthermore, A represents a set of actions containing the complete range of actions available to the agent. Given all possible states S and actions A, the probability that action a in state s at time t will lead to state s' at time t+1 is then designated by the transition function T.

Finally, R is a set of real numbers that simply amounts to the reward function that the agent tries to maximize.

Using this definition of an MDP, many real-world problems can be modeled such that an agent can learn an optimal policy. To clarify, a deterministic policy, denoted by $\pi(a|s)$, maps any action to a corresponding state. While in the probabilistic case, the policy returns the probability of taking action a in state s. Hence, the goal of the agent is to find the best policy that maximizes the accumulated reward.

At last, it is important to realize that the power of the MDP comes from the fact that each state only depends on the state immediately prior to it and its action, rather than all previous states. Often called the Markov Assumption, it drastically increases computational efficiency.

2.1.2 Partially Observable Markov Decision Process

The MDP assumes the current state can be fully observed at all times. This may be true for environments comparable to that of a game of chess, but in many cases the agent can only partially observe the state of the environment. In that case, the MDP transforms into a Partially Observable Markov Decision Process (POMP).

At this instant, the environment is not fully observable anymore and the agent is not able to view the current state directly. Instead, it receives an observation, usually a sensory measurement, that hints towards the current state of the environment. For example, a self-driving vehicle only knows the state of the environment around itself based on the data it gathers from its various sensors. But it is not able to see the environment past the range of its sensors.

POMPD models are harder to solve than a regular MDP. The partial observability also means that Figure 2.1 must be altered since the agent receives observations instead of states. Consequently, decisions are taken based upon a probability distribution over all states rather than the current state. The probability distribution must be updated after each new action and observation.

With this in mind, a key element of the RL process is the agent considering it is the trainable body. In particular, the learning aspect of the agent is handled by an artificial neural network. Although there are different types of neural networks depending on the application, the simplest form is the Multilayer Perceptron (MLP).

2.1.3 Multilayer perceptron

First and foremost, an MLP is a feedforward artificial neural network composed of multiple layers of perceptrons. In turn, a perceptron is a single link between an input and an output node. Every perceptron gives a certain weight to the input and caries an activation threshold. Activation thresholds are defined by activation functions. These functions can be relatively simple such as a linear function or more complex like the sigmoid or hyperbolic tangent function. Multilayer perceptrons are therefore simply multiple perceptrons linked together to form a network. Figure 2.2 shows an example of a 3 layer deep MLP.

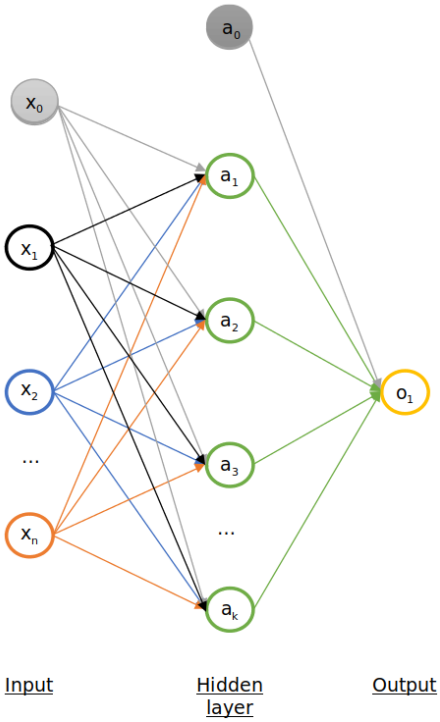


Figure 2.2: 3 layer deep multilayer perceptron. Input to the network is characterized by x_1 to x_n while the output is denoted by o_1. a_1 to a_k represent the output of the hidden layer. x_0 and a_0 are the bias nodes for the input and hidden layer. The colours are added to improve the visibility of the different connections in the MLP.

Data enters the network at the input layer and propagates through the hidden layer towards the output layer. To further tune the network such that it accurately fits the data, a bias is also added at each layer. The bias works as an offset where the only difference between any other node is that its input is always one.

Additionally, each arrow represents a data stream and holds a corresponding weight. If a node has multiple contributions, the sum over all weighted inputs is taken and subsequently passes through the activation function to yield an output. The output serves as input for the next layer of nodes. This process is called a forward pass and is described by Equation 2.1 for the first layer.

$$a_j = f_j(\sum_{i=1}^{n} w_{j,i}^{(1)} * x_i + w_{j,0}^{(1)}) \quad with \quad j = 1,2,3,...,k \tag{2.1}$$

Where a_j and f_j are the output and the activation function for the j^{th} node in the hidden layer respectively. Conversely, $w_{j,i}^{(1)}$ is the weight for the connection between the i^{th} input and the j^{th} node of the first hidden layer whereas x_i is the i^{th} input. Likewise, the output of the entire MLP follows Equation 2.2.

$$o_1 = g_1(\sum_{j=1}^{k} w_{1,j}^{(2)} * a_j + w_{1,0}^{(2)}) \tag{2.2}$$

In a similar fashion, the function g_1 is the activation function of the output node.

2.1.4 Deep Deterministic Policy Gradient

Although, for trajectory and sensing location generation RL provides a solution, it is still important to choose the correct RL algorithm that best suits the problem. Therefore, one should take into account the type of environment and the type of action space required. To closely match the real-world, this thesis considers a continuous partially observable environment. First, the continuity of the environment indicates that there exists an infinite amount of possible states in which the environment can exist. Second, despite the fact that the environment is at all times only partially observable by the LIDAR, the EKF provides a state estimate of the entire environment. As a result, the exploration problem can be modelled as a MDP rather than a POMPD.

Next, contrarily to most research in autonomous navigation with mobile robots, the agent does not directly choose the motor control commands but computes a trajectory in the environment. This means that the action space of the RL agent becomes continuous instead of a predefined set of discrete actions. Likewise, the sensing location generation algorithm also requires a continuous action space seeing that the environment itself is continuous. Hence, in both cases a Deep Deterministic Policy Gradient (DDPG) is an appropriate initial choice.

Originally created by Lillicrap et al. (2015), DDPG is a model-free, off-policy, actor-critic type algorithm that combines techniques from policy based and value based algorithms such as Deep Q-Network (DQN). The term model-free refers to the fact that DDPG does not try to predict state transitions or rewards and thus does not receive or learn a model of the environment. Generally, model-free RL algorithms are easier to implement and tune.

Additionally, considering that DDPG is an actor-critic type algorithm it uses two separate neural networks, one to compute a new action (actor network $\mu(s \mid \theta^{\mu})$) and the other to evaluate how beneficial this action is by estimating a Q-value (critic network $Q\left(s, a \mid \theta^{Q}\right)$). The bigger the Q-value the more beneficial the action is. Here s and a are the state and action respectively, while θ^{μ}, θ^{Q} correspond to the weights of the actor and critic network.

For both the actor and the critic, DDPG also introduces a target network to improve the training stability. Each target network is a time-delayed copy of the original network (critic $Q'\left(s, a \mid \theta^{Q'}\right)$ or actor $\mu'\left(s \mid \theta^{\mu'}\right)$) thelike also sharing the same weight initialization and architecture.

Yet, DDPG suffers from similar unstable behaviour when directly using Artificial Neural Networks (ANN) as other Q-learning algorithms due to the false assumption that the data samples are independently distributed. This assumption is no longer valid as the environment is sequentially explored. Experience replay can overcome this issue by randomly choosing a mini-batch at each time step t from a replay buffer to effectively remove the time correlation between each sample. This mini-batch is then used during the training. The replay buffer itself consists of a collection of past experiences $e_t = (s_t, a_t, r_t, s_{t+1})$ stored in a finite-sized set $D_t = [e_1, e_2, ..., e_t]$. Once the replay buffer is full the oldest experience is replaced by the most recent one.

Moreover, by using a replay buffer the agent (actor network) can now use all previous experiences to update its current policy. Off-policy networks such as DDPG have the advantage that the actor policy is detached from the target policy. The disconnection allows to add noise to the actor policy and continue exploration following Equation 2.3 while learning an optimal policy.

$$\mu_{expl}(s_t) = \mu\left(s_t \mid \theta_t^{\mu}\right) + \mathcal{N} \tag{2.3}$$

Here μ_{expl} is the exploration policy, θ_t^{μ} the parameters for the actor network at time t and $\mathcal{N}$ the added noise. Although, the original authors of the DDPG paper (Lillicrap et al., 2015) use the Ornstein Uhlenbeck process (OU noise) as noise mode, according to successive algorithms

like TD3 and D4PG ((Dankwa and Zheng, 2019), (Barth-Maron et al., 2018)) a Gaussian Noise works just as well. Consequently a Gaussian noise is chosen in this thesis to increase exploration since it is easier to implement compared to an OU process.

To better understand how DDPG works, Figure 2.3 shows the pseudo-code of the DDPG algorithm as described by the original authors of the algorithm (Lillicrap et al. (2015)).

Algorithm 1 DDPG algorithm

Randomly initialize critic network $Q(s,a|\theta^Q)$ and actor $\mu(s|\theta^\mu)$ with weights θ^Q and θ^μ.
Initialize target network Q' and μ' with weights $\theta^{Q'} \leftarrow \theta^Q$, $\theta^{\mu'} \leftarrow \theta^\mu$
Initialize replay buffer R
for episode = 1, M **do**
  Initialize a random process $\mathcal{N}$ for action exploration
  Receive initial observation state s_1
  for t = 1, T **do**
    Select action $a_t = \mu(s_t|\theta^\mu) + \mathcal{N}_t$ according to the current policy and exploration noise
    Execute action a_t and observe reward r_t and observe new state s_{t+1}
    Store transition (s_t, a_t, r_t, s_{t+1}) in R
    Sample a random minibatch of N transitions (s_i, a_i, r_i, s_{i+1}) from R
    Set $y_i = r_i + \gamma Q'(s_{i+1}, \mu'(s_{i+1}|\theta^{\mu'})|\theta^{Q'})$
    Update critic by minimizing the loss: $L = \frac{1}{N}\sum_i (y_i - Q(s_i, a_i|\theta^Q))^2$
    Update the actor policy using the sampled policy gradient:

$$\nabla_{\theta^\mu} J \approx \frac{1}{N}\sum_i \nabla_a Q(s,a|\theta^Q)|_{s=s_i, a=\mu(s_i)} \nabla_{\theta^\mu}\mu(s|\theta^\mu)|_{s_i}$$

    Update the target networks:

$$\theta^{Q'} \leftarrow \tau\theta^Q + (1-\tau)\theta^{Q'}$$
$$\theta^{\mu'} \leftarrow \tau\theta^\mu + (1-\tau)\theta^{\mu'}$$

  end for
end for

Figure 2.3: DDPG algorithm (Lillicrap et al., 2015)

After initialization, the training loop starts by computing a new action based on the current exploration policy μ_{expl}. The action is then applied to the environment which returns a new state and reward. The new experience tuple of the agent that is subsequently stored in the replay buffer.

The critic network is updated by minimizing the loss function in Equation 2.4 defined as the mean squared error of the difference between the temporal difference (TD) target and the current Q-value.

$$L(\theta^Q) = \frac{1}{N}\sum_i \left[y_i - Q\left(s_i, a_i \mid \theta^Q\right)\right]^2 \tag{2.4}$$

Note that the iterator i refers to a summation over the samples in the mini-batch. In turn, the TD target y_i is obtained by the Bellman Equation 2.5.

$$y_i = r_i + \gamma Q'\left(s_{i+1}, \mu'\left(s_{i+1} \mid \theta^{\mu'}\right) \mid \theta^{Q'}\right) \tag{2.5}$$

Where γ is the discount factor. It tells the agent how much it should care about future rewards compared to immediate rewards.

Considering that the assumption of independently distributed samples is now validated with the replay buffer, the loss function can be minimized using traditional gradient descent methods. However, remark that the new Q-value in Equation 2.5 comes from the target critic net-

work while the current Q-value in Equation 2.4 originates from the critic network. If this would not be the case and the same network with the same weights would be used in both Equation 2.5 and 2.4, the interdependence in the parameters would make the gradient of the loss function prone to diverging during gradient descent. Hence, a time-delayed copy of the original critic network is used to compute the new Q-value.

Besides, the goal of the actor network is to find actions that maximize the expected return. The objective function for the actor network can thus be written as Equation 2.6.

$$J(\theta) = \mathbb{E}\Big[\, Q(s,a)|_{s=s_t, a_t=\mu(s_t)} \Big] \tag{2.6}$$

Consequently, the gradient of the objective function becomes the sum of the gradients over the entire mini-batch like described in Equation 2.7.

$$\nabla_{\theta^\mu} J(\theta) \approx \frac{1}{N} \sum_i \Big[\, \nabla_a Q\left(s, a \mid \theta^Q\right)\Big|_{s=s_i, a=\mu(s_i)} \nabla_{\theta^\mu} \mu\left(s \mid \theta^\mu\right)\Big|_{s=s_i} \Big] \tag{2.7}$$

Finally, the weights of the actor network are then updated following Equation 2.8, where gradient ascent is performed.

$$\theta^\mu_{t+1} = \theta^\mu_t + \alpha \nabla_{\theta^\mu} J(\theta) \tag{2.8}$$

With α corresponding to the learning rate.

Since DDPG uses a deterministic policy, the actor network maps directly from state to action instead of a distribution over actions as is the case with stochastic policies. In addition, like the critic network, a target actor network is also used while updating the actor policy.

At last, the parameters of both target networks are kept mostly the same throughout training, only slowly updating according to Equation 2.9.

$$\begin{aligned} \theta^{Q'} &\leftarrow \tau\theta^Q + (1-\tau)\theta^{Q'} \\ \theta^{\mu'} &\leftarrow \tau\theta^\mu + (1-\tau)\theta^{\mu'} \end{aligned} \tag{2.9}$$

With a typical value of $\tau = 0.01$ one can see that the target weights gradually merge with the current weights.

2.1.5 Twin Delayed Deep Deterministic Policy Gradient

Granted DDPG would solve most RL problems but it still suffers from overestimation of the Q-value. After all, if the Q-value is overestimated, the agent policy is thought to be better than it is in reality. In return, this leads to sub-optimal solutions which is of course not desirable. To solve the overestimation issue, Fujimoto et al. (2018) propose a successor to the DDPG with three major improvements, called the Twin Delayed Deep Deterministic Policy Gradient.

First, two critic networks are used instead of one to reduce the overestimation. TD3 computes two Q-values for each action but uses the lesser of the two during training. Second, due to the overestimation of a poor policy, the training of the agent can diverge. More stability is introduced by reducing the frequency at which the actor updates. While the critic updates every training step, the actor does so only after a certain amount of steps. Lastly, target networks in the DDPG have tendencies to generate high-variance Q-values by over-fitting errors in the Q-function. Given these points, action smoothing is introduced to smooth out changes in the action and reduce the variance by adding a small amount of clipped noise to the target network. Clipping the noise is necessary so to keep the new target value close to the original one.

2.2 Simultaneous Localization And Mapping

This section starts off with a small introduction in SLAM after which it continues with an extensive explanation about how EKF SLAM works.

2.2.1 Introduction to Simultaneous Localization And Mapping

SLAM solves a problem that at first hand seems paradoxical. The agent, in order to create a map of the unknown environment, must know its position in this environment. Reciprocally, it also needs a map to infer its position in the first place. Still, several algorithms exist that solve the SLAM problem like those mentioned in Section 1.1 for example.

The initial estimate of the robot pose can be derived using an internal model of the system and a given control input or from the odometry. Additionally, much like humans trying to navigate in an unknown city, SLAM algorithms look at landmarks in the environment to derive the robot's position while it is exploring. It is then the job of the chosen sensor to find and extract suitable landmarks. Good landmarks are features in the environment that are easily distinguishable and re-observable at a later time. The latter is especially important to perform data association and consequently loop closure. Loop closing refers to the event where the robot, after a certain amount of time has passed, navigates in an area that it had previously already explored. It is an important aspect of SLAM as it is crucial to reduce the drift that occurs in the estimated trajectory over time.

2.2.2 Extended Kalman Filter -Simultaneous Localization And Mapping

Extended Kalman Filter -Simultaneous localization and mapping (EKF SLAM) was the standard SLAM algorithm until the introduction of FastSLAM (Montemerlo, 2002) and is still widely used today. The input state of the EKF SLAM algorithm consists of a collective state space of the robot pose $\hat{\mathscr{R}}$ and a list of landmarks called the map $\hat{\mathscr{M}}$, see Equation 2.10.

$$\hat{X} = \begin{bmatrix} \hat{\mathscr{R}} \\ \hat{\mathscr{M}} \end{bmatrix} = \begin{bmatrix} x \\ y \\ \theta \\ \mathscr{L}_1^{(x_1,y_1)} \\ \mathscr{L}_2^{(x_2,y_2)} \\ \vdots \\ \mathscr{L}_n^{(x_n,y_n)} \end{bmatrix} \tag{2.10}$$

Additionally, the EKF keeps track of the uncertainties in the estimated robot pose and the n landmarks with positions (x_1, y_1) to (x_n, y_n) in the form of the symmetric covariance matrix P, Equation 2.11.

$$P = \begin{bmatrix} \sigma_{xx} & \sigma_{xy} & \sigma_{x\theta} & \sigma_{xx_1} & \sigma_{xy_1} & \sigma_{xx_2} & \sigma_{xy_2} & \dots & \sigma_{xx_n} & \sigma_{xy_n} \\ \sigma_{yx} & \sigma_{yy} & \sigma_{y\theta} & \sigma_{yx_1} & \sigma_{yy_1} & \sigma_{yx_2} & \sigma_{yy_2} & \dots & \sigma_{yx_n} & \sigma_{yy_n} \\ & & & & \vdots & & & & & \\ \sigma_{x_nx} & \sigma_{x_ny} & \sigma_{x_n\theta} & \sigma_{x_nx_1} & \sigma_{x_ny_1} & \sigma_{x_nx_2} & \sigma_{x_ny_2} & \dots & \sigma_{x_nx_n} & \sigma_{x_ny_n} \end{bmatrix} \tag{2.11}$$

Each element represents the standard deviation between the robot pose and the n landmark coordinates as well as the error in the estimates themselves. For instance, σ_{xx} and σ_{yy} are the standard deviations associated to the x- and y-coordinate in the estimated robot pose $\hat{\mathscr{R}}$ while σ_{xx_1} is the difference between the robot pose x-coordinate and the x-coordinate of the first landmark.

The covariance matrix P can also be written more concisely according to Equation 2.12 where

$\sigma^2_{\mathscr{R}}$ and $\sigma^2_{\mathscr{M}}$ are the variance in the robot pose and in the map respectively. While $\sigma^2_{\mathscr{R}\mathscr{M}}$ is the variance of the robot pose with respect to the map and vice versa for $\sigma^2_{\mathscr{M}\mathscr{R}}$.

$$P = \begin{bmatrix} \sigma^2_{\mathscr{R}} & \sigma^2_{\mathscr{R}\mathscr{M}} \\ \sigma^2_{\mathscr{M}\mathscr{R}} & \sigma^2_{\mathscr{M}} \end{bmatrix} \tag{2.12}$$

That is to say $\sigma^2_{\mathscr{R}}$ and $\sigma^2_{\mathscr{M}}$ can be interpreted as the uncertainty in the robot pose and in the map respectively. While $\sigma^2_{\mathscr{R}\mathscr{M}}$ is the uncertainty of the robot pose with respect to the map and vice versa for $\sigma^2_{\mathscr{M}\mathscr{R}}$.

However, the EKF algorithm does not compute the variances in Equation 2.12 directly and instead gives an estimate of the P matrix in the prediction step. The EKF SLAM algorithm consists of two steps, a prediction step to predict a new state given a control input and a correction step to perform data association and update the belief of the estimated positions.

The prediction step requires the previously estimated state $\hat{X_{t-1}}$, covariance matrix $\hat{P_{t-1}}$, the new motor control commands u_t and a given state covariance C_x. Depending on the control vector u the motion model in the EKF SLAM may vary. Here the control vector is assumed to contain a linear and angular velocity in the form of $u = (v, \omega)$.

In simulation, process noise is added to the control vector u, modelling disturbances between the digital control signal and the motor speed. The covariance matrix Q of the process noise is a diagonal matrix with the variances in the velocity and yaw rate respectively, see Equation 2.13.

$$Q = \begin{pmatrix} \sigma^2_v & 0 \\ 0 & \sigma^2_\omega \end{pmatrix} \tag{2.13}$$

At first, a motion model of the robot movement is created to predict the next robot state given the new motor control command vector u. The motion model presented in Equation 2.14 is a simple model where the control vector $U = u^T$ is added to the previous robot state $\hat{\mathscr{R}}_{t-1}$.

$$\hat{\mathscr{R}}_t = F\hat{\mathscr{R}}_{t-1} + BU \tag{2.14}$$

Here F is the identity matrix, $\hat{\mathscr{R}}$ the new robot pose and B the matrix mapping velocity to position. Substituting these matrices in Equation 2.14 yields Equation 2.15.

$$\begin{bmatrix} \hat{x}_{t+1} \\ \hat{y}_{t+1} \\ \hat{\theta}_{t+1} \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \hat{x}_t \\ \hat{y}_t \\ \hat{\theta}_t \end{bmatrix} + \begin{bmatrix} \Delta t\cos(\theta_t) & 0 \\ \Delta t\sin(\theta_t) & 0 \\ 0 & \Delta t \end{bmatrix} \begin{bmatrix} v_t \\ w_t \end{bmatrix} \tag{2.15}$$

Notice that in Equation 2.15 the state vector only contains the robot pose seeing that the control vector U has no effect on the position of the landmarks. Δt is the simulation tick time, $\Delta t = 0.1s$ is used here.

Second, the prediction step of the EKF SLAM algorithm continues by estimating the covariance matrix $\hat{P}$ following Equation 2.16.

$$\hat{P}_t = G^T\hat{P}_{t-1}G + F_x^T C_x F_x \tag{2.16}$$

Where C_x is the known state covariance matrix and G the jacobian described by Equation 2.17. The matrix F_x (Equation 2.18) is needed to map from the 3 dimensions of the robot pose to

the $2n+3$ (n landmarks with 2 coordinates and the robot pose) dimensions of the covariance matrix P.

$$G_t = I + F_x^T \begin{pmatrix} 0 & 0 & -\Delta t v_t \sin(\theta) \\ 0 & 0 & \Delta t v_t \cos(\theta) \\ 0 & 0 & 0 \end{pmatrix} F_x \tag{2.17}$$

$$F_x = \begin{pmatrix} 1 & 0 & 0 & 0 & \cdots & 0 \\ 0 & 1 & 0 & 0 & \cdots & 0 \\ 0 & 0 & 1 & 0 & \cdots & 0 \end{pmatrix} \tag{2.18}$$

C_x is a 3 by 3 diagonal matrix holding the variances in the x, y coordinates as well as in the robot orientation. Expressed in Equation 2.19, C_x represents the uncertainty in the state and is added at each prediction step to the system uncertainty characterized by P. C_x can be thought of as the process noise for the robot pose, modelling in simulation disturbances that affect the robot pose.

$$C_x = \begin{pmatrix} \sigma_x^2 & 0 & 0 \\ 0 & \sigma_y^2 & 0 \\ 0 & 0 & \sigma_\theta^2 \end{pmatrix} \tag{2.19}$$

From Equation 2.16 it can be observed that the covariance matrix P is only dependant on the covariance of the robot pose C_x.

Lastly, a correction step is required to perform data association on new landmarks, compute the Kalman gain K and correct the state and covariance estimate. At each new cycle in the EKF SLAM algorithm, a new observation vector z_t is created. z_t contains the range and bearing from the landmark to the robot position as measured by the onboard LIDAR sensor for n observed landmarks, similar to Equation 2.20.

$$z_t = \begin{pmatrix} \begin{pmatrix} range_1 \\ bearing_1 \end{pmatrix} & \cdots & \begin{pmatrix} range_n \\ bearing_n \end{pmatrix} \end{pmatrix} \tag{2.20}$$

Considering that the landmarks are used to correct the EKF state estimate, proper data association is critical. Hence, all landmarks within a fixed Mahalanobis distance from a known landmark in the map are considered to be the same. For each observation, re-observed landmarks are used to correct the prior belief (loop closure) while new landmarks are added to the state EKF vector X and the covariance matrix P. To keep track of their position as well.

The Mahalanobis distance is the distance between two points in mutlivariate space. Unlike the Euclidean distance, the Mahalanobis distance can be calculated for correlated multi variable points. The Mahalanobis distance between two observations with innovation covariance matrix S is given by Equation 2.21.

$$D_{Mahalanobis} = \sqrt{(z_1 - z_2)^T * S * (z_1 - z_2)} \tag{2.21}$$

The innovation covariance matrix S is computed similarly as for the Kalman gain which is shown next.

To find the Kalman gain K and the innovation covariance S, the expected measurements $\hat{z}$ must initially be found. Given an observed landmark $\mathscr{L}_i$, the LIDAR sensor returns its position in the

form of a range and bearing. The conversion from polar coordinates to Cartesian coordinates for the i^{th} landmark follows Equation 2.22.

$$\mathscr{L}_i = \begin{pmatrix} \mathscr{R}_x + range_i * cos(\mathscr{R}_\theta + bearing_i) \\ \mathscr{R}_y + range_i * sin(\mathscr{R}_\theta + bearing_i) \end{pmatrix} \tag{2.22}$$

The difference in position between the estimated robot pose $\hat{\mathscr{R}}$ to landmark $\mathscr{L}_i$ is given by Equation 2.23.

$$\Delta_i = \begin{pmatrix} \Delta_{i,x} \\ \Delta_{i,y} \end{pmatrix} = \mathscr{L}_i - \hat{\mathscr{R}}_{x,y} \tag{2.23}$$

From which the distance from the estimated robot pose to the i^{th} landmark is found from Equation 2.24.

$$q_i = \sqrt{\Delta_{i,x}^2 + \Delta_{i,y}^2} \tag{2.24}$$

Using the nonlinear observation, the expected measurements $\hat{z}$ given the current belief then becomes Equation 2.25.

$$\hat{z}_i = \begin{pmatrix} q_i & atan2(\Delta_{i,y}, \Delta_{i,x}) - \mathscr{R}_\theta \end{pmatrix} \tag{2.25}$$

Subsequently, comparing the measured observation z_i to the expected measurement $\hat{z}_i$ yields an innovation factor I for landmark i.

The innovation factor evaluates the accuracy of the predicted robot pose since $\hat{z}_i$ is derived from the estimated state vector. Equation 2.26 describes the innovation factor I for the i^{th} landmark.

$$I_i = z_i - \hat{z}_i \tag{2.26}$$

The Kalman gain K serves to balance how much of the innovation should be added in the corrected state vector $X_t^{corrected}$ based on the uncertainty in the measurement.

Nevertheless, due to the nonlinearity of the observation $\hat{z}$, the jacobian of $\hat{z}$, H, is used to compute the innovation covariance S and the Kalman gain K. H is found through Equation 2.27.

$$H = J * T \tag{2.27}$$

$$J = \begin{pmatrix} \frac{-\Delta_{i,x}}{q} & \frac{-\Delta_{i,y}}{q} & 0 & \frac{\Delta_{i,x}}{q} & \frac{\Delta_{i,y}}{q} \\ \frac{\Delta_{i,y}}{q^2} & \frac{-\Delta_{i,x}}{q^2} & -1 & \frac{-\Delta_{i,y}}{q^2} & \frac{\Delta_{i,x}}{q^2} \end{pmatrix}$$

$$T = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \cdots \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \cdots \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \cdots \\ 0 & 0 & 0 & 0 & 0 & \cdots & 1 & 0 & 0 & 0 & \cdots \\ 0 & 0 & 0 & 0 & 0 & \cdots & 0 & 1 & 0 & 0 & \cdots \end{pmatrix}$$

The transformation matrix T has as many columns as there are observed landmarks in the estimated EKF state vector $\hat{X}$. T is needed to compute the linear approximation only for the robot pose and the i^{th} landmark.

Finally, substituting the jacobian of the measurement model H, the covariance matrix P and the

covariance matrix of the measurement noise R for a specific time t in Equation 2.28, provides the innovation covariance S.

$$S = H_t \hat{P}_t H_t^T + R_t \tag{2.28}$$

And thus the Kalman gain follows in Equation 2.29.

$$K = \hat{P}_t H_t^T S^{-1} = \frac{\hat{P}_t H_t^T}{H_t \hat{P}_t H_t^T + R_t} \tag{2.29}$$

R_t is known and defines the amount of noise present in the measurement. Its value usually depends on the type of sensor used for the measurements.

From Equation 2.29, one can see that if the uncertainty in the measurement is very low compared to the current estimate ($R_t << P_t$), the Kalman gain will tend towards $K = 1$. This means that it adds all the innovation to the current estimate as it completely trusts the measurement. In contrary, very high uncertainty in the measurement ($R_t >> P_t$) leads to a very low kalman gain and thus little inclusion of innovation in the current estimate.

Both the state estimate $\hat{X}_t$ and the covariance estimate $\hat{P}_t$ are corrected with the Kalman gain, with the state estimate following Equation 2.30.

$$\hat{X}_t^{corrected} = \hat{X}_t + (K I_n) \tag{2.30}$$

And the corrected covariance estimate specified by Equation 2.31.

$$\hat{P}_t^{corrected} = (I_n - K H_t)\, \hat{P}_t \tag{2.31}$$

3 Related work

In recent years, there has been many advancements in exploration and mapping with RL and SLAM. But also in related areas such as combining EKF and RL as well as different exploration policies and path tracking controllers. Progress in these field are equally important to create accurate maps since they tackle specific problems that arise when exploring with RL and SLAM. In this section, previous studies are reviewed to position the thesis in the context of past research.

3.1 Research in exploration with RL

RL can be used to generate trajectories similar to other conventional approaches for path planning such as Dijkstra or A*. The main advantage of RL is its flexibility to maximize any criterion that suits the purpose of the robot, being either distance, energy efficiency or map accuracy. Though, multi-objective RL algorithms have been proposed capable of learning several optimal policies simultaneously ((Barrett and Narayanan, 2008), (Nguyen et al., 2020)).

On top of the study of Kollar and Roy (2008) discussed in the introduction, many more research has been done dealing with exploring and mapping static unknown environments ((Zhang et al., 2017), (Ramezani and Lee, 2018), (Mustafa et al., 2019), (Chen et al., 2019), (Placed and Castellanos, 2020), (Botteghi et al., 2020)). Previous research has mostly focused on shaping the reward function to incorporate obstacle avoidance by adding knowledge about the map or prove the benefit of adding memory. Either short-term with Long Short-Term Memory (LSTM) networks and memory-based multi layer Q-networks or long-term with external memory. Except for the work of Kollar and Roy (2008), none of the found literature attempted path planning with RL through trajectory generation, instead discrete actions are computed for the robot to execute. Arguably, learning to explore with trajectory generation also means the agent is unable to adjust its path as often as with discrete actions. Consequently, this method makes it harder for the agent to navigate through the environment and reduces the frequency at which it updates its policy during training. Both factors contribute to a decrease in the sample efficiency when using trajectory generation.

Furthermore, the aforementioned literature focused especially on improving model convergence during training, computational efficiency and generalization for different environments rather than maximizing the accuracy of the map. Similarly to map accuracy, ensuring full map coverage is also a feature not always taken into account.

3.2 Neural SLAM

Automatically finding optimal targets is a necessary step towards fully autonomous exploration. The purpose of the sensing locations is to ensure a full map coverage of the unknown environment. Kollar and Roy (2008) discuss the use of a coverage planner that finds suitable locations for the agent to visit. In their research however, they consider them to be fully observable during the exploration phase. They also assume such a coverage planner exists and instead provide a predefined set of destination points that the agent must visit to satisfy the coverage constraint. However, advancements in SLAM algorithms like Active Neural SLAM from Chaplot et al. (2020) show very promising results using a high-level policy picking suitable sensing locations. Their algorithm achieves 94.8% coverage on the Gibson database of 3D spaces. Which is 15% more than other exploration algorithms according to their research. The Active Neural SLAM is based on visual observation and divides the coverage problem in long-term and short-term goals. The long-term goal can be seen as the high-level policy (RL algorithm) de-

termining the location the agent should drive. The high-level policy is subsequently trained with ground-truth maps. Next, a path planner creates short-term goals in order to reach this destination. Although this method achieves great coverage, as it is the winner of the CVPR 2019 habitat challenge, it is important to realize that Active Neural SLAM only focuses on the coverage and not the map accuracy. In contrast to the approach proposed in this thesis which deals both with coverage and map accuracy.

Besides, Chen et al. (2019) provide another interesting approach to optimize full map exploration using information gain in the form of Shannon entropy as a metric to represent map completeness. They then train an RL agent to learn an optimal policy that maximizes the information gain for each discrete action taken by the agent. Their results show an increase in computational efficiency and accuracy compared to other state-of-the-art approaches.

3.3 Towards dynamic environments

Nevertheless, even though the thesis concentrates on static environments, in most cases the real-world environment is not static but dynamic which adds a new set of difficulties. For instance, when an obstacle blocks a pathway that was previously accessible, the estimated SLAM location of the robot becomes less accurate. Several papers propose modified algorithms that integrate obstacle detection and sudden changes in the environment ((Arana-Daniel et al., 2011), (Guevara-Reyes et al., 2013), (Wen et al., 2020)). During the exploration phase these algorithms work similarly as for static environments. However, when an obstacle is detected that was previously not there, the agent uses its long-term memory or experience to avoid the obstacle. If the agent is not successful, a new path is generated to circumvent the obstacle and reach the desired sensing location. Although dynamic environments are not the aim of this thesis, it is important to understand the complexities that they bring in order to facilitate the generalization to dynamic environments in future work.

3.4 Towards multi-agent exploration

In addition to dynamic environments, one can also expand to a multi-agent case where multiple agents explore and build together a map of the environment. Several studies have already considered such a scenario and tackle some of the issue that arise ((Pei et al., 2020), (Dinnissen et al., 2012), (Luviano Cruz and Yu, 2014)). Though, multi-agent exploration and mapping certainly comes with it own set of difficulties like map merging, multi-agent path planning and collaboration between the agents, it would also significantly speed up the exploration of large environments.

4 The proposed algorithm

This chapter describes the proposed trajectory-based EKF SLAM algorithm. In a first instance, a general overview of the algorithm is given after which the chapter dives deeper in all the individual modules.

4.1 Trajectory-based EKF SLAM architecture

The trajectory-based EKF SLAM algorithm consists of five main components governed by a high- and low-level policy. Figure 4.1 shows the algorithm architecture from the sensor input to the output signal controlling the movement of the agent.

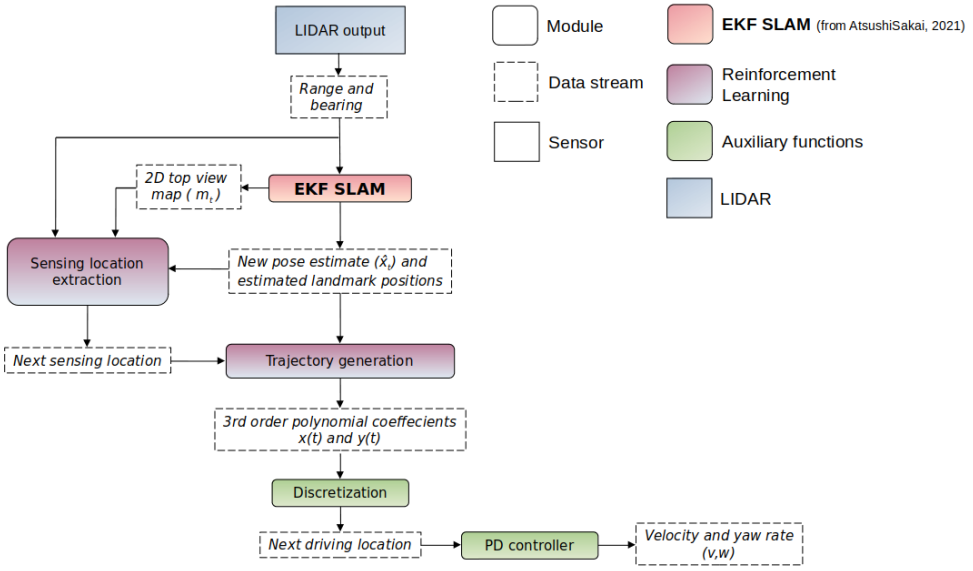


Figure 4.1: Trajectory-based EKF SLAM. Sensor outputs have a blue color while the different modules are represented in rounded rectangles with a unique color for differentiation. The information exchange is expressed by doted rectangles. Modules denoted in bold represent parts of the algorithm that are based on open-source code.

The first decision making module is the sensing location generation. Also called the high-level policy, it oversees the general movement of the agent. Afterwards, the role of the low-level policy is to compute an appropriate trajectory to reach these sensing locations.

Initially, the trajectory-based EKF SLAM algorithm starts with the EKF SLAM module in charge of mapping and localization of the agent. It takes as input the initial known robot pose as well as the actual LIDAR readings. Given the newly estimated robot pose, the algorithm can update the occupancy grid indicating which areas of the environment have already been explored and conversely which areas remain unexplored.

After, the occupancy map is inserted into the high-level policy to select a new sensing location acting as target location for the trajectory generation. Two third order polynomials can then be generated, one for the x coordinate and one for y coordinate. The high-level policy chooses the sensing locations such to minimize the amount of unexplored area in the occupancy map. Whereas the trajectory created by the low-level policy aims to reach the sensing location with an obstacle free path that maximizes the map accuracy.

However, in order to properly move around the environment, the agent still requires motor control commands.

Hence, both third order polynomials are discretized into a finite set of coordinates. These coordinates form together the trajectory and constitute short term goals for the PD controller. Consequently, the PD controller returns appropriate velocities and yaw rates to reach these short term goals and travel along the trajectory defined by the low-level policy.

At last, the EKF SLAM module and the LIDAR provide new outputs every time the agent moves from one position to another. This means that both modules along with the PD controller continuously update their outputs for every step along the chosen trajectory. The high- and low-policy on the contrary, only change their output if the target sensing location is reached. Computing a new sensing location and a new trajectory.

4.2 EKF SLAM implementation

The algorithm makes use of a Light Detecting And Ranging (LIDAR) sensor in combination with EKF SLAM to determine the robot pose, landmark locations and occupancy grid. Though, the LIDAR can also be replaced by any other sensor such as a camera as long as it provides the distances and angles to nearby obstacles and landmarks with respect to the robot pose. In such a case, the observation function inside the EKF SLAM module can be skipped.

4.2.1 Overview

Primarily, the tasks of the EKF SLAM module is to estimate the position of the agent and subsequently update the map of the environment given the LIDAR data. Yet, also other tasks are taken care of, including landmark observation and data association. Figure 4.2 shows an overview of the EKF SLAM module with the various data exchanges between the submodules.

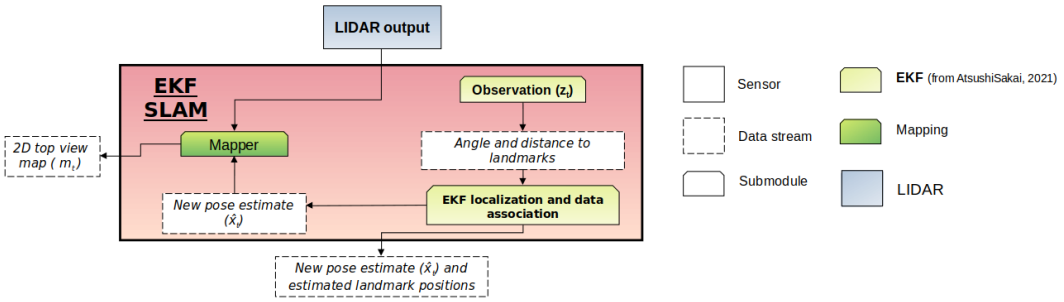


Figure 4.2: EKF SLAM algorithm. Sensor inputs are represented in blue while the different submodules are characterized by faded yellow and green rectangles with indented corners. The exchanged data is shown in dotted rectangles. Submodules denoted in bold represent parts of the algorithm that are based on open-source code.

The only input to the EKF SLAM module is the LIDAR sensor. It tells the agent at which angle and distance objects are located. In general, both the EKF localization and data association submodule as well as the mapper require this information to properly function. The mapper uses the LIDAR for object detection and keeps track of the occupied areas around the agent as it moves around. In contrast, the localization submodule needs the LIDAR data to extract meaningful landmarks. Note however, in Figure 4.2 one can see that there is no information exchange between the LIDAR output and the observation submodule. Instead, to simplify the algorithm it is assumed that the agent knows the position of the landmarks if they come within range of the LIDAR sensor. Effectively disregarding the LIDAR landmark extraction.

All known landmark coordinates are transformed to a range and bearing for further processing in the EKF localization and data association submodule. Consecutively, this data along with previously estimated positions is used to update the estimated robot pose. Finally, data association is a necessary step that incorporates loop-closure detection and the clustering of a group

of landmarks into a single landmark.

At this point, the mapper can map the corresponding LIDAR data to the correct position in the occupancy map according to the estimated robot pose from the EKF.

4.2.2 Observation

Landmarks are a crucial part of the localization method. Good landmarks must be static and easily recognizable between successive steps taken by the agent. In the context of this thesis, both the training and testing environment in which the EKF SLAM operates consists of rectangular shaped objects and walls. Therefore, the only viable option to find fitting landmarks is to use the corners of the walls and objects in the environment. If the environment has no walls, the landmark coordinates are defined during the experiment setup.

With this in mind, to simulate a LIDAR feature extraction submodule, the environment in which the agent moves provides a list of landmarks where each landmark represents a corner in the surroundings. However, the agent's field of view is bounded by the maximum operating range of the LIDAR sensor. Only landmarks within the LIDAR range can be considered valid.

Hence, the observation function computes the Euclidean distance between all landmarks in the environment and the robot pose, see Equation 4.1. Both the landmarks and the robot pose used in the observation submodule are the true positions as given by the environment.

$$\begin{aligned} \|d_{\mathscr{L}_i^{true}-\mathscr{R}^{true}}\| &= \sqrt{(d_{i,x})^2+(d_{i,y})^2} \quad for\ i\ landmarks \\ d_{i,x} &= \mathscr{L}_{i,x}^{true} - \mathscr{R}_x^{true} \\ d_{i,y} &= \mathscr{L}_{i,y}^{true} - \mathscr{R}_y^{true} \end{aligned} \tag{4.1}$$

The observation algorithm evaluates which landmarks are within the LIDAR range using the Euclidean distance. Nonetheless, a simple distance threshold is not sufficient to accurately replicate a LIDAR feature extraction submodule. Indeed, LIDAR beams cannot traverse solid obstacles such as walls. Therefore, in order for a landmark to be observable by the agent, it must both be within range of the LIDAR sensor and have an obstacle free, straight path between the landmark and the agent. In this manner the agent only observes landmarks that are accessible by the LIDAR laser beams. The landmarks that fulfill both conditions are subsequently propagated to the data association submodule for loop closure detection and clustering.

Next, along with the Euclidean distance, the observation function also finds the angle difference between the agent and all landmarks with respect to the positive x-axis like Equation 4.2, where $\mathscr{R}_\theta^{true}$ is the true orientation of the agent and i the i^{th} landmark.

$$\Delta\phi_{\mathscr{L}_i^{true}-\mathscr{R}^{true}} = [(atan2(d_{i,y}, d_{i,x}) - \mathscr{R}_\theta^{true} + \pi) \mod (2\pi)] - \pi \tag{4.2}$$

At last, before returning the observation matrix, noise is added in simulation to the motor control commands, the Euclidean distance and the angle variation in the form of process and sensor noise. The process noise discussed in Section 2.2.2 is added to the motor control commands $u = [v, \omega]$. u consist of a velocity and yaw rate that are fed into the robot motion model. The addition of process noise follows Equation 4.3.

$$u^{noisy} = u + \mathscr{N} * Q \tag{4.3}$$

With $\mathscr{N}$ a randomly normally distributed sampled number with 0 mean and variance 1 and the process noise covariance matrix Q given by Equation 2.13.

The process noise influences the accuracy and time lag of the EKF estimates. On the contrary, the measurement noise covariance matrix R represents the noise characteristics of the sensor,

hence it is added to both the range and the bearing of the LIDAR sensor in Equation 4.1 and Equation 4.2. Equation 4.4 and 4.5 show the resultant noisy distance and angle variation denoted by $\|d_{\mathscr{L}_i^{true}-\mathscr{R}^{true}}\|^{noisy}$ and $\Delta\phi^{noisy}_{\mathscr{L}_i^{true}-\mathscr{R}^{true}}$ respectively.

$$\|d_{\mathscr{L}_i^{true}-\mathscr{R}^{true}}\|^{noisy} = \|d_{\mathscr{L}_i^{true}-\mathscr{R}^{true}}\| + \mathscr{N} * R[0,0] \tag{4.4}$$

$$\Delta\phi^{noisy}_{\mathscr{L}_i^{true}-\mathscr{R}^{true}} = \Delta\phi_{\mathscr{L}_i^{true}-\mathscr{R}^{true}} + \mathscr{N} * R[1,1] \tag{4.5}$$

with R expressed in Equation 4.6.

$$R = \begin{bmatrix} \sigma^2_{range} & 0 \\ 0 & \sigma^2_{bearing} \end{bmatrix} \tag{4.6}$$

Where σ^2_{range} and $\sigma^2_{bearing}$ are the variances in the range and bearing of the LIDAR sensor respectively.

The introduction of noise into the system is essential to better mimic real-life conditions, but most importantly to insert small errors in the EKF estimates so they can be minimized by the trajectory generation module. In practice Q and R can be difficult to find and often depend on the system at hand, the manufacturer and the type of sensor used. For simulation purposes, Q and R are usually tuned to an appropriate size depending on the data. A good practice is to start with the identity matrix multiplied by a scalar that is less than one. But care must be taken in this step considering that the EKF output is sensitive to changes in the process and measurement noise.

Proceeding with the observation submodule, all that remains is to gather Equations 4.4 and 4.5 for every valid landmark into a single observation matrix according to Equation 4.7.

$$\begin{aligned} z &= [z_1, z_2, z_3, ..., z_i] \quad for\ i\ landmarks \\ z_{1,...,i} &= [\|d_{\mathscr{L}^{true}_{1,...,i}-\mathscr{R}^{true}}\|^{noisy}, \Delta\phi^{noisy}_{\mathscr{L}^{true}_{1,...,i}-\mathscr{R}^{true}}, 1, ..., i] \end{aligned} \tag{4.7}$$

4.2.3 Localization and data association

The EKF localization and data association submodule is the core function of the EKF SLAM module. As the name suggests, it not only handles both EKF steps (prediction and correction) but also loop-closure detection and landmark clustering.

These activities are performed by the algorithm in a specific order portrayed by the pseudo-code shown in Algorithm 2.

Algorithm 2: EKF prediction and correction step with data association

Input : $\hat{X}_{t-1}^{corrected}$, $\hat{P}_{t-1}^{corrected}$, u^{noisy}, z, C_x, Mahalanobis distance threshold, dimensions of $\mathscr{R}$, dimensions of $\mathscr{L}$, time step dt
Output: $\hat{X}_t^{corrected}$, $\hat{P}_t^{corrected}$, Kalman gain

Calculate G and F_x according to Equation 2.17 and 2.18
Initialize measurement jacobian H and Kalman gain K
Predict robot pose $\hat{\mathscr{R}}$ in $\hat{X}_t$ and covariance of robot pose in $\hat{P}_t$ according to Equation 2.14 and 2.16
for *observation vector* **in** *observation matrix z* **do**
    Calculate Mahalanobis distance to all known landmarks
    if *Mahalanobis distance bigger than Mahalanobis distance threshold* **then**
        Compute landmark position using $\hat{X}_t$ and observation vector
        Add landmark coordinates to $\hat{X}_t$ and $\hat{P}_t$
    end
    Calculate innovation factor according to Equation 2.26
    Calculate Kalman gain according to Equation 2.29
    Correct state vector $\hat{X}_t$ and covariance matrix $\hat{P}_t$ following Equation 2.30 and 2.31
end
return $\hat{X}_t^{corrected}$, $\hat{P}_t^{corrected}$, K

The process is straight forward and follows exactly the equations discussed in Subsection 2.2.2.

The succeeding section deals with building the map of the unknown environment in the form of an occupancy grid. The role of the occupancy grid is to keep track of the positions of all solid obstacles detected by the LIDAR sensor. The grid is in many ways the end goal of the SLAM algorithm since it represents a map of the previously unknown environment.

4.2.4 Mapper

The mapper translates the coordinates of known obstacles detected by the LIDAR to a row, column index in the occupancy grid. In it simplest form, the occupancy grid is a matrix where each element is either equal to 0, 0.5 or 1. The matrix can further be interpreted as an image with black, grey and white pixels. Table 4.1 discloses the meaning behind each pixel color in the occupancy grid.

Pixel value	Color	Meaning
0	black	occupied
0.5	grey	unknown
1	white	free

Table 4.1: Value, color and meaning of each pixel in the occupancy grid. An occupied pixel refers to a location that the agent cannot drive through. As opposed to white pixels which correspond to free space. Grey pixels serve to depict undiscovered areas where the algorithm does not know if the region is free or occupied.

Before actually creating the occupancy grid, the mapper must compute the coordinates of each object detected by the LIDAR. The module receives from the LIDAR sensor a range and a bearing characterizing the precise location of an object. When combining the EKF and LIDAR data

one can easily infer the location of the obstacle in the world coordinates. Equation 4.8 shows an example for an arbitrary LIDAR angle and distance pair.

$$\begin{bmatrix} obstacle_x \\ obstacle_y \end{bmatrix} = \begin{bmatrix} \hat{X}_t^{corrected}[0] \\ \hat{X}_t^{corrected}[1] \end{bmatrix} + \begin{bmatrix} d_{lidar} * cos(\hat{X}_t^{corrected}[2] + \hat{\Theta}_{lidar}) \\ d_{lidar} * sin(\hat{X}_t^{corrected}[2] + \hat{\Theta}_{lidar}) \end{bmatrix} \tag{4.8}$$

The variable $obstacle_{x,y}$ is the newly computed set of coordinates of the object detected by the LIDAR. $\hat{X}_t^{corrected}[i]$ with $i = 0, 1, 2$ is the estimated robot pose by the EKF (x,y coordinates and orientation to the positive x-axis). Finally, d_{lidar} and Θ_{lidar} stand for the received LIDAR data in the form of a distance and angle.

At the beginning, the occupancy grid starts off as a completely grey image, centered around the current position of the agent. The initial size of the image is chosen large enough to encompass the entire unknown environment.

Despite the fact that the coordinates of the obstacles are known, a conversion step to pixel coordinates is still necessary seeing that the occupancy grid has different dimensions than the environment itself. For that reason, each pixel symbolizes an area of the environment. The size of the area each pixel embodies depends on the resolution of the image.

Essentially, the received LIDAR data by the mapper is arranged in two separate categories, the laser beams and individual points. The area covered by the laser beams is considered free space because no obstacle is detected at that specific location. The same can be said for angles where the LIDAR sensor did not return any echo. As opposed to a point, which represents occupied space.

Every time the mapper updates the occupancy grid, the location of the obstacles and the robot pose must be converted to pixel coordinates. Equation 4.9 presents the conversion for a point in the world coordinate to a pixel coordinate, according to the conversion factor $c_{image} << 1$. In this case, a pixel coordinate is a row and column index meaning it must be rounded to the nearest integer value.

$$\begin{bmatrix} image_x \\ image_y \end{bmatrix} = \begin{bmatrix} x/c_{image} \\ y/c_{image} \end{bmatrix} \tag{4.9}$$

Conversely, the laser beams of the LIDAR sensor are modelled in the occupancy grid according to the Bresenham's line drawing algorithm. Bresenham's algorithm requires the start and the end point of the line, after which the algorithm iterates through all columns in-between. For each column or pixel x-coordinate, it then estimates the closest pixel center to the line. The y-coordinate is thus given by Equation 4.10.

$$y^{pixel} = \frac{y_1^{pixel} - y_0^{pixel}}{x_1^{pixel} - x_0^{pixel}} * (x^{pixel} - x_0^{pixel}) + y_0^{pixel} \tag{4.10}$$

In Equation 4.10, (x^{pixel}, y^{pixel}) are the pixel coordinates of the closest pixel center to the line for a given column number x^{pixel}. In contrary, (x_0^{pixel}, y_0^{pixel}) and (x_1^{pixel}, y_1^{pixel}) are the start and end coordinates of the line respectively.

In addition, for angles where no echo was received by the LIDAR sensor (no obstacles), the maximum LIDAR range is in that case used to compute an endpoint for Bresenham's line drawing algorithm. All obstacles are given a two pixel big width to make them more visible in the map as well.

A great start for implementing the LIDAR conversion to occupancy grid can be found in the Python Robotics library of AtsushiSakai (2021).

Second, despite the fact that at this point the mapper has the ability to map obstacles and LI-

DAR beams in the occupancy grid, it remains important to choose a proper image resolution in order to create a useful map. From Equation 4.9 it is clear that the conversion factor c_{image} determines the resolution of the image. Decreasing c_{image} would increase the value of the corresponding pixel coordinate. In other words, for a similar change in world coordinates, there is a bigger change in pixel coordinates compared to a larger conversion factor c_{image}.

Starting with a conversion factor of $c_{image} = 0.0025$, the resulting image presented in the Appendix Figure A.1 is too large. A large occupancy grid is far from favorable because larger maps increase the computation cost considerably. Additionally, the smaller conversion factor also means there are more unknown pixels between successive LIDAR beams or obstacle points. This leads then again to more noise in the overall map. Therefore, it is desirable to choose a larger conversion factor, yet not too big to avoid loosing detail in the map.

A conversion factor that is too big, like $c_{image} = 0.01$, has better coverage by the LIDAR and decreases computational cost due to its smaller size. The resulting image can be observed in the Appendix Figure A.2. Though, one could argue that the map is in this case too small and that too much detail is lost in the process. One pixel represents a larger area in the world coordinates which explains the zoomed out look that arises.

Thus, an adequate conversion factor lays in-between, comparable to $c_{image} = 0.005$. The new occupancy grid appears to be the best of both worlds. The image is less sparse while keeping smaller details in the map. Likewise, walls are clearly identifiable. Still, the occupancy grid can be improved by applying a small medium blur to the image as can be seen in Figure 4.3.

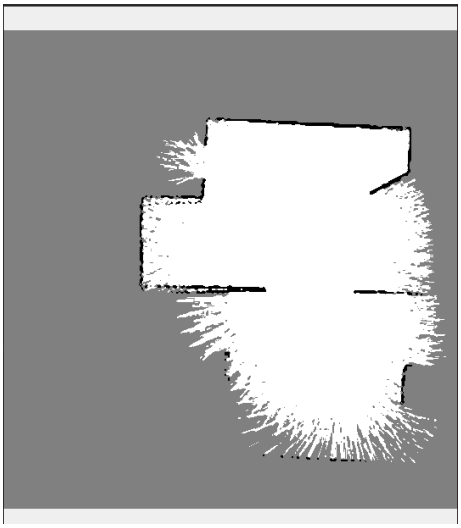


Figure 4.3: Occupancy grid with conversion factor $c_{image} = 0.005$. A median blur is applied with kernel size 3x3. White areas represent obstacle free space while black pixels portray obstacles in the environment. Grey pixels instead are unknown regions.

The median blur computes the average pixel value in a fixed window size to remove sparsity in the original image. In Figure 4.3 the kernel size is 3x3. Sections of free space now become a solid white patch whereas walls are clearly identifiable as a straight black line.

Moreover, if the angle resolution of the LIDAR sensor is too big, it can happen that an obstacle is located exactly in-between two laser beams. In that case, part of the obstacle can be seen as free space by the mapper and thus overwritten due to the beam width. In order to avoid this error, any occupied pixel in the occupancy grid is binding, meaning it can not be converted back to free space. Even though, this means that the mapper cannot revise the positions of obstacles in subsequent passes. This choice is deliberate because of the following three factors. First, if the mapper was allowed to overwrite the position of obstacles in the map in successive runs,

the mapper could introduce errors instead of correcting them. Second, based on the previous point, it is the goal of the low-level policy to choose trajectories that minimize the uncertainty in the EKF estimates. Hence, after a single pass the map should already not contain many errors. Finally, considering that the high-level policy works as a coverage planner, the agent should not visit the same location in the environment very often in the first place. Note that to further minimize the chance of obscuring small objects in the environment, a fine LIDAR angle resolution of 1 degree is chosen.

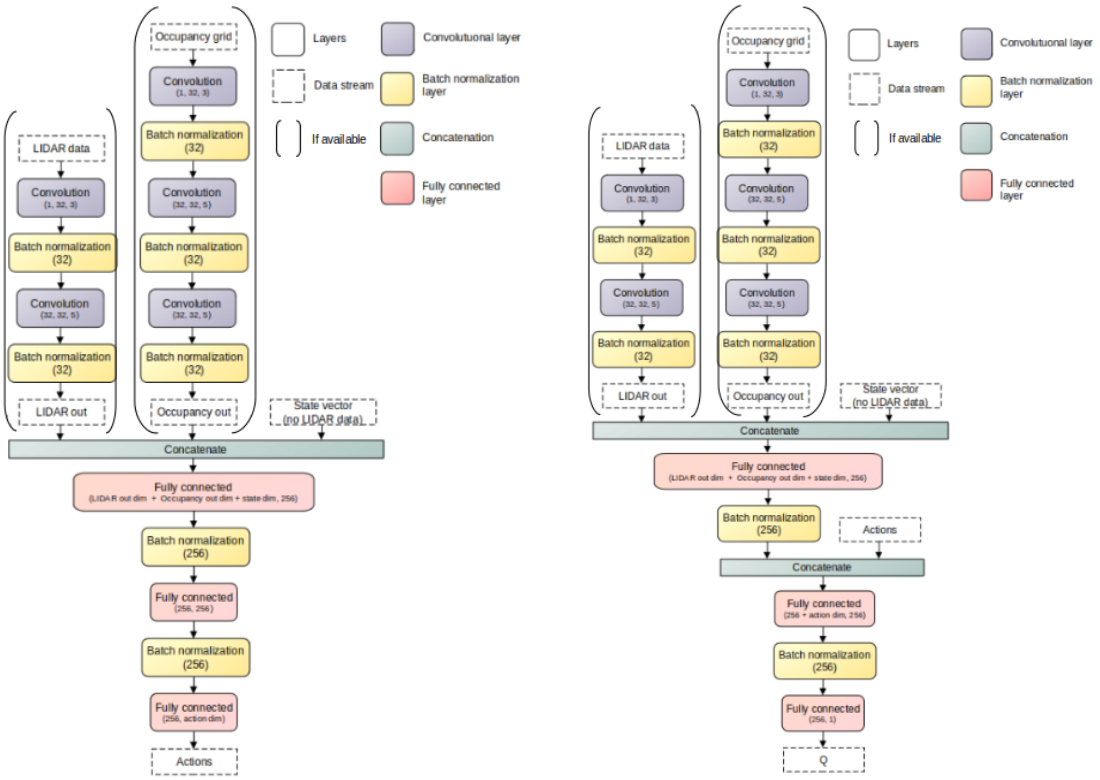
4.3 Sensing location extraction

The sole purpose of the high-level policy is to ensure the agent explores the full map, preferably as efficiently as possible. Formally, this can be translated to visiting all the landmarks in the environment. A high-level policy refers to a decision making body that governs the generation of sensing locations.

Accordingly, the high-level policy contains a TD3 RL network trained to minimize the amount of unvisited landmarks. It receives the current LIDAR data as well as the EKF state vector holding the robot pose and the observed landmark coordinates. Exploration is considered finished if all landmarks have been visited.

4.3.1 Network architecture

With the RL algorithm for the sensing location generation being a TD3 network, a suitable network architecture is defined for both the actor and the critic in Figure 4.4.



(a) Actor network architecture

(b) Critic network architecture

Figure 4.4: High-level policy network architectures for the actor and the critic in the TD3 algorithm.

Both the actor and the critic network have very similar architectures. The key differences is that the critic has an additional concatenation step where the actions are added to the data before entering the last two fully connected layers. Two or three convolutional layers are sufficient to extract the required features from the LIDAR and the occupancy grid. More convolutional layers do not provide more benefits. The same can be said for the fully connected layers.

Note that the size of the network inputs (LIDAR data and occupancy grid) are hyper-parameters while the final outputs (actions and Q-value) are fixed in size. For a good feature extraction, the LIDAR data and the occupancy grid are passed through a series of convolutional layers separately. The resulting latent space is then joined together before being processed in the fully connected layers.

4.3.2 Action space

Considering the continuous nature of the environment in which the agent moves, the action space of the high-level policy is continuous with dimension $\mathbb{R}^2$. Instead of directly computing Cartesian coordinates for the sensing location, the high-level policy chooses polar coordinates to select the desired sensing location.

Selecting directions rather than coordinates still gives the high-level policy enough freedom to explore the map but simplifies the decision making process. It reduces the amount of possible actions and avoids the necessity for conversion between the occupancy grid and the world coordinates. Furthermore, because obstacle avoidance is a difficult task for the low-level policy to handle alone, the module eliminates directions where the distance between the robot and an object is smaller than a pre-defined safety margin of 0.1 meters. This means sensing locations are always located at least 0.1 meters away from any obstacle. The safety margin is set at 0.1 meters considering the agent has a size if 0.018 meters in simulation. Figure 4.5 illustrates the action space of the high-level RL agent.

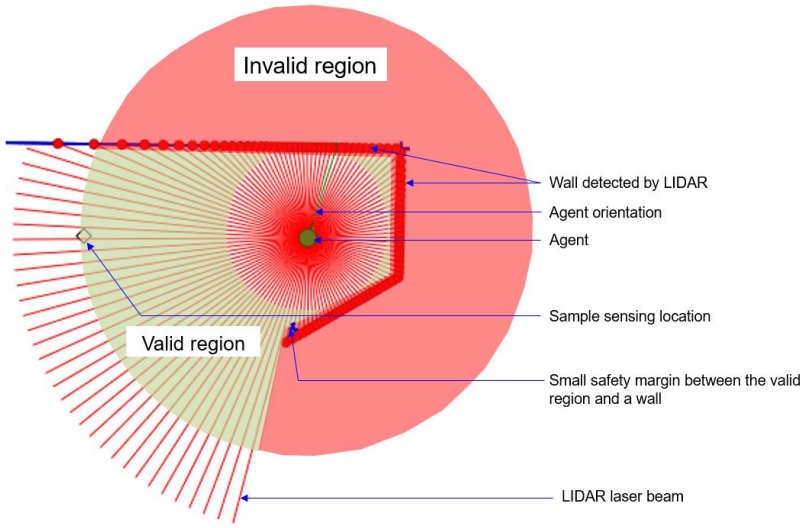


Figure 4.5: Invalid and valid sensing location regions for the high-level policy in the case the agent is surrounded by walls on three sides. A LIDAR sensor displays the position of the walls and the free space. Polar coordinates are chosen such that the sensing location resides in the valid region.

From there on, the module picks a distance and an orientation to define the polar coordinates of the sensing location. If the RL network chooses a direction outside the valid region, the closest valid angle is selected instead. Similarly, the chosen distance is clipped if it exceeds the boundary of the valid region. In addition, it is good to know that the high-level RL agent receives a fixed sized occupancy grid while it is exploring. This is imperative to avoid having to

deal with variable size inputs to the convolutional layers in the actor and the critic.

These coordinates are then converted to Cartesian coordinates before being used for trajectory generation. From Figure 4.5, one can see that the valid region fits inside the LIDAR range to assist with obstacle avoidance. With the current approach, the low-level policy has an obstacle free straight path to the next sensing location, without obstacles hidden in blind spots. The maximum allowed distance a sensing location can be placed from the agent depends on the range of the LIDAR sensor. The agent is only able to perform local obstacle avoidance relying on the immediate LIDAR data. Thus, sensing locations must be placed well within the LIDAR range to give sufficient leeway to the low-level policy. If the sensing location is too close to the LIDAR range, the agent may compute trajectories that exit the LIDAR range and collide with unobserved obstacles. The lower boundary of the valid sensing location region defines the smallest distance the agent can travel. This depends very much on the type of environment that must be explored. Large environment permit long trajectories while small and compact environments require at times short trajectories..

4.3.3 Reward function

The goal of the high-level policy is to ensure full map coverage of the unknown environment. Hence, to teach the network to select informative sensing locations, the percentage of discovered landmarks is used as training metric. Given the total number of landmarks in the environment (L_{all}), the amount of undiscovered landmarks ($L_{undiscovered}$) and an environment state s with action a^*, the reward function for the high-level RL agent can be written as in Equation 4.11.

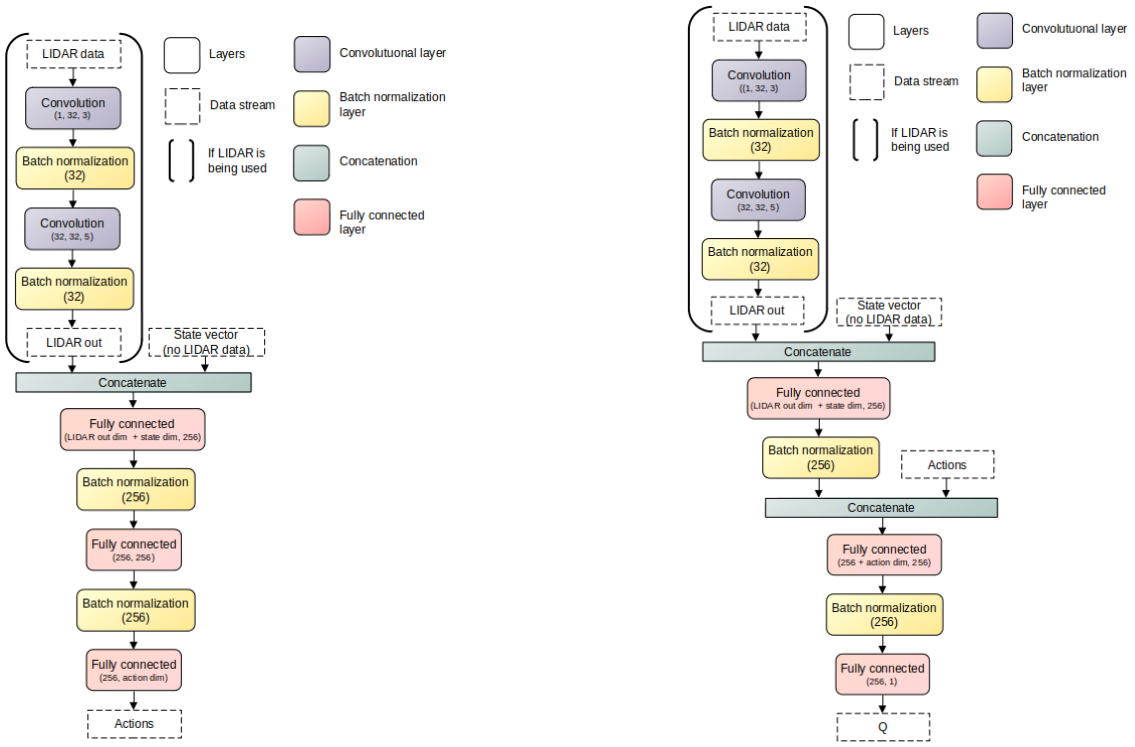
$$R_{hl}(s,a^*) = 100 - \frac{L_{undiscovered}}{L_{all}} * 100 \tag{4.11}$$

4.4 Trajectory generation

The low-level policy consists of a TD3 RL agent with a continuous normalized action space between $[-1,1]$. The agent computes 3 actions to select a specific trajectory towards the sensing location.

4.4.1 Network architecture

Very similar to the high-level RL agent, the low-level RL agent uses an arrangement of 2 convolutional layers to extract features from the LIDAR data. Followed by 3 fully connected layers after concatenation with the remaining RL state vector. Figure 4.6 shows the network architecture for the actor and the critic used in the low-level RL agent.



(a) Actor network architecture **(b)** Critic network architecture

Figure 4.6: Low-level policy network architectures for the actor and the critic in the TD3 algorithm.

Here again the critic concatenates the actions only after the LIDAR data and the RL state vector went through a fully connected layer. This step can be seen as a pre-processing step to gradually extract features as more data is added to the network. The brackets around the 2 layer deep convolutional network means this portion of the actor or critic network can be removed if there is no LIDAR data.

The RL state vector that the TD3 network receives is characterized in Equation 4.12.

$$s = [\hat{X}, \mathscr{S}_1, \mathscr{S}_2, action1, action2, action3, \mathscr{S}_x^{next}, \mathscr{S}_y^{next}, range_1, ..., range_n] \tag{4.12}$$

Where $\hat{X}$ is the estimated robot pose and landmark locations by the EKF module, $\mathscr{S}_x$ and $\mathscr{S}_y$ are the Cartesian coordinates of the sensing location associated with the trajectory defined by the agent's actions $action_1$, $action_2$ and $action_3$. $\mathscr{S}_x^{next}$, $\mathscr{S}_y^{next}$ and $range_1, ..., range_n$ represent the next target sensing locations and the range values of the n LIDAR beams respectively. The number of LIDAR beams depends on the specified LIDAR angle resolution. The bearing information is not needed since the position of each range in the state vector is fixed.

4.4.2 Action space

In all cases, the policy class consists of a set of two trajectories that define the parameterized trajectory from the robot pose to the next sensing location. The thesis further compares two different parameterization methods. The first method is the standard cubic polynomial like initially proposed by Kollar and Roy (2008). The second method looks at Bézier curves instead, due to their smoothness and control simplicity.

Both the standard cubic polynomial and the Bézier curves can be used to create smooth trajectories between two points. Cubic polynomials have already been proven to be effective in creating smooth trajectories by Kollar and Roy (2008). 3^{rd} order Bézier curves on the other hand, are known to be particularly smooth and are widely used by car designers and in com-

puter graphics to generate smooth curves. Their shape is interpolated using four control points which is thought to give a more intuitive control to the low-level policy.

Bézier curves

Starting with the derivation of the Bézier parameterization. The coordinates of the four control points are used in Equation 4.13 to form the Bézier parameterization for the x- and y-coordinates.

$$B_x(t) = (1-t)^3 * p_{c1}[0] + 3(1-t)^2 t * p_{c2}[0] + 3(1-t)t^2 * p_{c3}[0] + t^3 * p_{c4}[0] \tag{4.13}$$
$$B_y(t) = (1-t)^3 * p_{c1}[1] + 3(1-t)^2 t * p_{c2}[1] + 3(1-t)t^2 * p_{c3}[1] + t^3 * p_{c4}[1]$$

Where the four control points are denoted by $p_{c1}(x_{p1}, y_{p1})$ to $p_{c4}(x_{p4}, y_{p4})$ and t the simulation tick time.

Two of the four control points (point 1 and point 4) are fixed since these are the initial position (robot pose) and final position (sensing location) of the trajectory. Hence, the first and last control point can already be defined by Equation 4.14.

$$p_{c1} = (\mathscr{R}_x, \mathscr{R}_y) \tag{4.14}$$
$$p_{c4} = (\mathscr{S}_x, \mathscr{S}_y) \tag{4.15}$$

The so-called easing function is known for its smooth curves and therefore it is a great option for trajectory generation. To start generating Bézier curves, only two more control points are needed. But the agent cannot directly choose coordinates for the two control points seeing that it is always in movement which makes it impossible to have a fixed action range. Rather, the agent computes an offset for the x and y direction which are later translated into coordinates for the two control points.

The second control point is related to the initial position of the trajectory. The farther away the point is from the initial position, the bigger the speed at t=0. Where exactly control point 2 is located will decide the orientation at the start of the trajectory. Like is the case for the standard cubic polynomial, one must constraint the initial orientation to enforce smooth transitions between successive trajectories. Thus, control point 2 is always located in front of the agent, with only the distance to the agent acting as a free parameter. The coordinates of p_{c2} are shown in Equation 4.16.

$$p_{c2} = (\mathscr{R}_x + free_{action1} * cos(\mathscr{R}_\theta), \mathscr{R}_y + free_{action1} * sin(\mathscr{R}_\theta)) \tag{4.16}$$

Finally, control point 3 is completely free for the agent to choose. The agent computes two offsets from the target sensing location $(\mathscr{S}_x, \mathscr{S}_y)$ in order to set the third control point. The coordinates of control point 3 are in Equation 4.17.

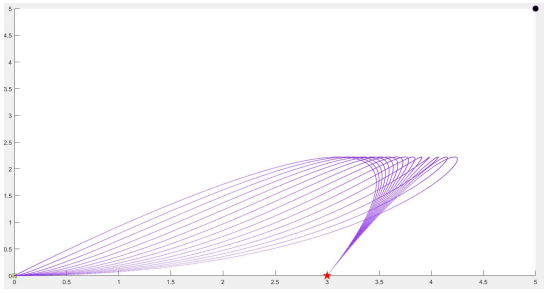
$$p_{c3} = (\mathscr{S}_x + free_{action2}, \mathscr{S}_y + free_{action3}) \tag{4.17}$$

After all, the Bézier curve approach gives very similar control to the agent as the standard cubic polynomial. A change in the distance of a control point with respect to the start or end position results in a larger corresponding speed. Varying the angle between the control point and the

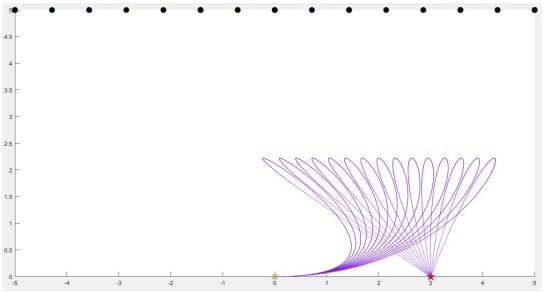
positive x-axis changes the arrival orientation of the trajectory.

With all the control points now defined, the trajectory is parameterized with two 3^{rd} order Bézier curves as seen in Equation 4.13.

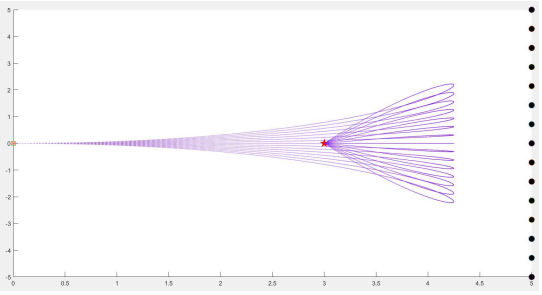
Varying each action of the agent, i.e. the positions of control point 2 and 3 influences the shape of the curve in different manners. Figure 4.10 emphasizes the impact each parameter has on the Bézier curve.



(a) 15 trajectories with varying control point 2. Control point 3 is fixed and shown as a black dot.



(b) 15 trajectories with varying x coordinate in control point 3 shown by a black dot. Control point 2 is fixed.



(c) 15 trajectories with varying y coordinate in control point 3 shown by a black dot. Control point 2 is fixed.

Figure 4.7: 15 trajectories with corresponding control points. In each subfigure only one parameter is varied. Control point 3 is shown with black dots. The orange square is the initial position at (0,0) while the red star shows the target sensing location at (3,0).

Increasing the initial speed creates an overshoot which is clearly visible on Subfigure 4.7a. Shifts along the x- and y-axis are distinguishable in Subfigure 4.7b and 4.7c as control point 3 is slowly altered. The trajectories become even more interesting if both control points are modified simultaneously. As an example of the complete action space, Figure 4.8 plots a range of sample trajectories.

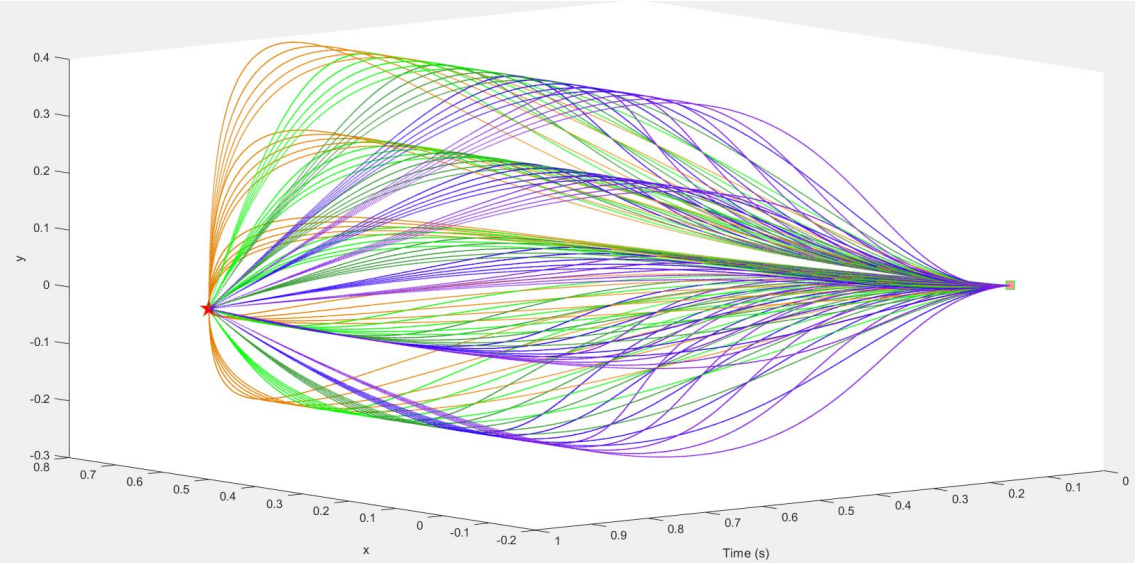


Figure 4.8: Bézier curves for varying control points 2 and 3. The colors represent different horizontal positions on x-axis for control point 3. The orange square represents the initial position (0,0,0) whereas the red star is the target sensing location (3,1,0). Units are in meters.

Figure 4.8 shows a portion of all the possible trajectories that the agent can compute using the Bézier parameterization. The action space of the agent is continuous with dimensions $\mathbb{R}^3$.

Bézier curves versus standard cubic polynomials

Comparing the parameterization using the Bézier approach versus the standard cubic polynomial shown in the Appendix section A.2, it can be observed that Bézier curves provide slightly smoother trajectories between the robot pose and the target sensing location. In addition, the parameterization is considerably easier to implement since the low-level policy directly selects the 2 adjustable control points. As a result, the low-level policy in the trajectory-based EKF SLAM algorithm uses the Bézier parameterization to compute appropriate trajectories that minimize the map and robot pose uncertainty.

Though, the TD3 agent generating the trajectories has a normalized action range between [-1,1], the three actions are scaled to their true range before computing the Bézier curve. A normalized action range is chosen to improve the convergence of the RL agent and to give more flexibility to the true action ranges. The final action range for the 3 free parameters depends on the scale and complexity of the environment. Large environments may require larger speeds. Similarly, a bigger action range leads to more intricate trajectories necessary to explore complex environments. Scaling the normalized action range to [-0.5,0.5] turned out to be adequate for the environments used in this thesis.

4.4.3 Reward functions

The reward function governing the desired behaviour of the agent consists of a combination of a continuous and a discrete penalty. The continuous portion of the reward function serves to guide the agent towards computing trajectories that minimize the map uncertainty. Whereas, the discrete reward handles the obstacle avoidance by penalizing the agent for picking any trajectory that leads to a collision.

Regarding the uncertainty minimization, two continuous reward functions are proposed to minimize the map uncertainty, either based on the Kalman gain K or on the EKF uncertainty matrix P. From Subsection 2.2.2, Equation 2.30 reveals that a reduction in the Kalman gain translates to a higher confidence in the current state estimate. Remember that the EKF state

estimate $\hat{X}$, contains both the estimated robot pose and the estimated landmark positions. Therefore, reducing the Kalman gain is one way to decrease the uncertainty in the EKF estimates. Yet, Equation 2.29 shows that K only decreases if the covariance matrix P or the jacobian of the measurement function H declines. Seeing that the H matrix exclusively holds the derivatives of the measured distance between the robot pose and the landmark and has no direct relation to the agent's action. H is not a viable metric to reduce the map uncertainty. Hence, minimizing the Kalman gain comes down to minimizing the uncertainty matrix P. This makes sense since P represents the variances in the robot pose and in the map. Both K and P are considered as candidate metric in the reward function to minimize the map uncertainty. Given a RL state s, an action a and neglecting obstacle avoidance for now, Equation 4.18 presents the two proposed continuous reward functions, denoted by $R(a)_{c1}$ and $R(a)_{c2}$.

$$\begin{aligned} R(s,a)_{c1} &= -\sum trace(K) \\ R(s,a)_{c2} &= -\sum trace(P) \end{aligned} \tag{4.18}$$

The trace function sums the elements on the diagonal of the square matrix. In both K and P, this refers to the elements relating to the robot pose and the map. The summation in Equation 4.18 spans over the entire trajectory to create a cumulative penalty for the agent.

Next, the continuous reward function that best trains the agent to minimize the map uncertainty is combined with a discrete penalty to include obstacle avoidance when selecting trajectories. Equation 4.19 describes the final reward function used to train the low-level policy within the framework of the trajectory-based EKF SLAM algorithm.

$$R(s,a) = \begin{cases} -2000, & \text{if collision.} \\ -\sum trace(K \; or \; P), & \text{otherwise.} \end{cases} \tag{4.19}$$

The size of the discrete penalty in case of a collision depends on the size of the continuous reward. The discrete penalty should be big enough such that the agent quickly learns to stay away from collisions. While the continuous reward serves as a gradient to learn trajectories that minimize the map uncertainty.

4.4.4 Evaluation metric

The accuracy of the occupancy grid depends on the accuracy of the EKF estimates. To evaluate the performance of the low-level policy, the Root Mean Square Error (RMSE) between the true and estimated robot pose $\hat{\mathscr{R}}$ and landmark positions $\hat{\mathscr{L}}$ is calculated. Equation 4.20 shows how the RMSE is computed for the robot pose and for n observed landmarks.

$$\begin{aligned} RMSE_{\hat{\mathscr{R}}_{x,y}} &= \sqrt{\frac{(\mathscr{R}_x - \hat{\mathscr{R}}_x)^2 + (\mathscr{R}_y - \hat{\mathscr{R}}_y)^2}{2}} \\ RMSE_{\hat{\mathscr{R}}_\theta} &= \sqrt{(\mathscr{R}_\theta - \hat{\mathscr{R}}_\theta)^2} \\ RMSE_{\hat{\mathscr{L}}} &= \sqrt{\frac{\sum_{j=1}^{n} (\mathscr{L}_{j,x} - \hat{\mathscr{L}}_{j,x})^2 + (\mathscr{L}_{j,y} - \hat{\mathscr{L}}_{j,y})^2}{n}} \end{aligned} \tag{4.20}$$

Where $\mathscr{R}$ is the true robot pose and likewise $\mathscr{L}$ the true landmark positions.

Despite the fact that the RMSE provides a better picture of the trend in the error signal compared to the Mean Square Error (MSE), it is still subject to a lot of variance. For that reason, the average RMSE over the entire N steps that compose the trajectory is determined, following Equation 4.21.

$$
\begin{aligned}
RMSE^{trajectory}_{\hat{\mathcal{R}}_{x,y}} &= \frac{\sum_{k=1}^{N} RMSE_{\hat{\mathcal{R}}_{x,y},k}}{N} \\
RMSE^{trajectory}_{\hat{\mathcal{R}}_{\theta}} &= \frac{\sum_{k=1}^{N} RMSE_{\hat{\mathcal{R}}_{\theta},k}}{N} \\
RMSE^{trajectory}_{\hat{\mathcal{L}}} &= \frac{\sum_{k=1}^{N} RMSE_{\hat{\mathcal{L}},k}}{N}
\end{aligned}
\tag{4.21}
$$

Though, the agent must not only minimize the map uncertainty but also discriminate between trajectories that interfere with obstacles. To measure the agent's ability to avoid obstacles, the environment keeps track of the average number of obstacle free trajectories per episode during training.

4.5 Discretization

In order for the agent to follow the parameterized trajectory, Equation 4.13 must be discretized in a finite amount of points l with corresponding time step $\Delta t = \frac{1}{l}$. This process is best described in a for loop going from 0 to the number of desired points l. In each loop, the time step Δt is added to time variable t in Equation 4.13, starting at $t = 0$.

These points serve as short term goals for the path follower in Section 4.6. The amount of points characterizing the trajectory between the robot pose and the sensing location can be fixed but a suitable amount of points must be chosen. Too many points will slow down the algorithm substantially without providing any more benefits. Too little and the 3^{rd} order polynomial is not accurately approximated anymore. The thesis uses 25 points for each trajectory. Figure 4.9 shows an example of a discretized trajectory.

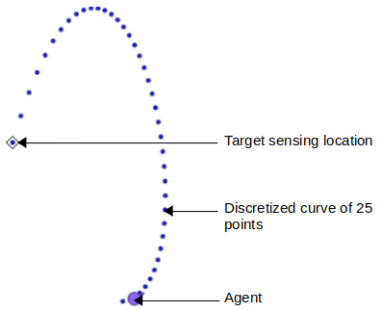


Figure 4.9: Example trajectory after discretization. Each blue point in the trajectory denotes a short term target for the PD controller. The diamond shaped marker is the target sensing location while the agent is represented in light blue on the bottom.

The 25 points that make up the trajectory show the path the agent takes to reach the target sensing location in Figure 4.9.

It is important to realize that the trajectory implementation considers a fixed time, from $t = 0s$ to $t = 1s$ and a fixed amount of discretization points per trajectory. However, this means that there is currently no set limit for the robot speed. Sensing locations that are set far away from the agent will lead to faster speeds achieved by the robot. In contrary, close-by sensing locations lead to very slow speeds.

4.6 PD controller

The EKF SLAM module requires a velocity and a yaw rate in the form of the motor control command $u = (u, \omega)$. Given the current estimated robot pose $\hat{\mathscr{R}}$ and the short term goals received from the low-level policy, the path follower module consists of two separate PD controllers, one for the velocity and one for the yaw rate. It gives the proper motor controls to accurately follow the trajectory.

In both cases, an error signal must be created. Firstly, the module calculates the difference in position between the current robot pose and the target position, see Equation 4.22.

$$\begin{aligned} \Delta X &= target_x - \hat{\mathscr{R}}_x \\ \Delta Y &= target_y - \hat{\mathscr{R}}_x \\ e_{pos} &= \sqrt{\Delta X^2 + \Delta Y^2} \end{aligned} \tag{4.22}$$

The same approach is done for the change in angle as described in Equation 4.23.

$$\begin{aligned} \Phi &= (arctan2(\Delta Y, \Delta X) + 2 * \pi) \mod (2 * \pi) \\ \Delta\Phi &= (\Phi - \hat{\mathscr{R}}_{theta}) \end{aligned} \tag{4.23}$$

Mark that the $arctan2$ function returns an angle in the range of $[-1, 1]$, for that reason 2π is added and subsequently removed to map the result back into the $[0, 2\pi]$ range of the robot pose.

Another key point is the loop back behaviour of the unit circle when calculating the signed difference between the robot orientation and the target angle. For the PD controller, the smallest angle difference between two angles is needed. A simple subtraction works fine in most cases but not when the 0 deg angle is in-between the two angles. A simple solution to solve this problem is to add or subtract $2pi$ if $\Delta\Phi$ is smaller than $-\pi$ or bigger than $+\pi$ respectively.

The error signal for both proportional controllers are now ready. Proceeding with the derivative control, the time derivatives of the error signals in the angle and position are inferred from Equation 4.24 using the simulation tick time $dt = 0.1s$.

$$\begin{aligned} de_{pos} &= \frac{e_{pos}}{dt} \\ d\Delta\Phi &= \frac{\Delta\Phi}{dt} \end{aligned} \tag{4.24}$$

At this point, the position and derivative error signals are fed into Equation 4.25 for the PD controller of v and ω.

$$\begin{aligned} v &= Kp_v * e_{pos} + de_{pos} * Kd_v \\ \omega &= Kp_\omega * \Delta\Phi + d\Delta\Phi * Kd_\omega \end{aligned} \tag{4.25}$$

Kp_v, Kd_v, Kp_ω and Kd_ω are the specific gains for the proportional and derivative control of the velocity and yaw rate PD controller. They must be tuned appropriately for the PD controller to function.

Both PD controllers are tuned separately starting with proportional control only. Once the proportional gain is satisfactory, the derivative gain is adjusted very slightly for each PD controller.

The proportional gain makes the system respond faster, for that reason the yaw rate proportional gain is easiest to tune in the corner of a trajectory. A good indication of a too low proportional gain is when the agent takes the corner too late. On the other hand, to tune the velocity proportional gain one must evaluate if the agent reaches the target locations. If the velocity proportional gain is too low, the agent will finish slightly in front of the target location after each step. These errors accumulate over the trajectory, meaning that the agent will end up short at the end of the trajectory.

The derivative control is tuned by looking at how the PD controller reacts to error changes. This is especially noticeable after the agent overshoots a target location. A good derivative gain brings the agent back on the trajectory and is able to recover from the overshoot. This holds for both the velocity and yaw rate derivative control.

4.7 Summary

The proposed trajectory-based EKF SLAM architecture consist of 5 different modules arranged hierarchically. The first module is the EKF SLAM. It starts by retrieving the range and bearing of the landmarks within the agent's observation range. When dealing with obstacle avoidance, landmarks are located on the corner of an obstacle. From there, the EKF performs its prediction and correction step as described in Section 2.2.2. Subsequently, the estimated robot pose is combined with the LIDAR data to create an occupancy grid. The occupancy grid keeps track of which areas in the environment are occupied space or free space.

Next, the high-level RL network uses the occupancy grid, the LIDAR data and the estimated robot pose to pick a new sensing location in polar coordinates. To aid with obstacle avoidance, the TD3 network can only choose sensing locations located in a defined valid area. Additionally, a reward function based on the amount of discovered landmarks is used to incite maximum map coverage.

On the other hand, the generated trajectories are parameterized with the 3^{rd} order Bézier curves. The shape of the Bézier curve is interpolated from 4 different control points, where the first and last control points are the start and end positions of the trajectory. This second TD3 RL agent is tasked to decide on the shape of these Bézier curves by changing the location of control point 2 and 3. The thesis initially proposes two continuous reward functions to minimize the EKF uncertainty. The first continuous reward sums the diagonal coefficients of the Kalman gain K. While the second continuous reward function sums the diagonal coefficients of the EKF covariance matrix P. To account for obstacle avoidance, a discrete penalty of -1000 is handed to agent upon collision with an obstacle.

At last, to satisfy the EKF's need for motor control commands, the computed Bézier curves are discretized into 25 short term target locations. A PD controller then handles the path following by finding the appropriate velocity and yaw rate to reach all 25 short term goals forming the trajectory.

5 Simulation experiments and results

The Experimental Design Chapter starts off by describing the process used to create custom environments for training and testing the proposed RL and SLAM-based framework. The two main experiments are presented. In the first experiment, the low-level policy is trained to minimize the map uncertainty without the presence of obstacles in the environment. This to validate the reward functions proposed in Equation 4.18. In the second experiment, it is researched how obstacle avoidance can be included as a goal for the low-level policy together with the minimization of the map uncertainty. Both experiments include the training and testing of the high-level policy. Finally, a discussion with recommendations and a summary conclude the chapter.

5.1 Experimental setup: 2D Python environment

In order to train the high- and low-level policy as well as test the trajectory-based EKF SLAM implementation, a 2D Python environment is created. The environment is based on the Gym library. Since the proposed algorithm is still in the prototyping phase, a simple developing environment such as Python and 2D physics is preferred. In a latter stage the algorithm can however be expanded to more complex and realistic environments like the robotic simulator Gazebo (Gazebo, 2021).

5.1.1 Custom environment

To quickly design versatile environments, the thesis makes use of the Python library Shapely. Shapely allows to create a wide range of shapes and objects with which a RL agent can interact with.

Figure 5.1 shows an example of a complex custom environment created with Shapely that can be used to train a RL agent.

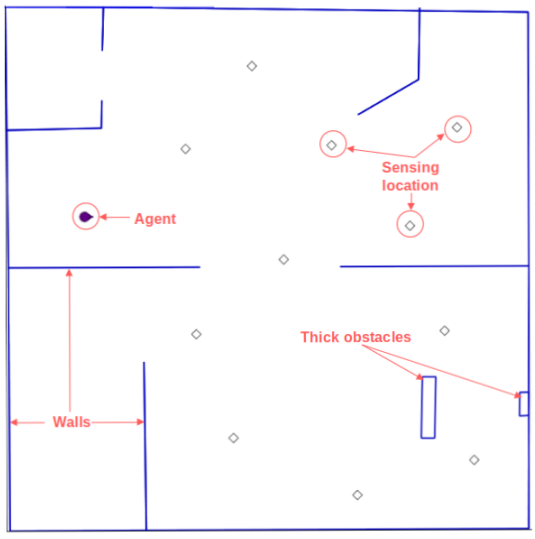


Figure 5.1: Example of a custom Python environment for training RL agents.

This way of creating custom environments is simple and sufficient for training the high- and low-level RL networks in the proposed SLAM algorithm. All custom environments entirely consist of a collection of polygons and individual points. In Figure 5.1 for instance, one can see thin polygons representing walls, placed either perpendicularly or diagonally. Polygons with a larger width are used to simulate internal obstacles.

In addition, the custom environment must at least consist of one polygon representing the boundary of the environment and one point denoting the starting position of the RL agent. All other polygons, other than the boundary, are considered obstacles that the agent cannot traverse. The custom environment is also normalized to have a fixed window size.

In the end, the environment changes state each time the agent moves from one position to another. Accordingly, the training of the low-level policy is divided into a finite number of episodes, where one episode coincides with one attempt to explore the environment. Such an episode can be terminated early in the event the agent collides with a wall. If no collision occurs, the episode finishes once a fixed amount of trajectories is reached (with one full trajectory corresponding to 25 steps). At each start of a new episode, the environment is reset to its initial state.

5.1.2 Generation of random environments

Introducing sufficient randomization in the environment is important for the creation of a well generalized high- and low-level policy. Without randomizing the training conditions, the RL agent would quickly over-fit the training environment. As a result, the learned policy would have a low operational capability and would only be useful in the exact conditions experienced during training.

Generalization can be increased by randomizing the environment after every new training episode. Therefore, the initial starting position of the robot is randomized as well as the location of the walls and landmarks. The walls are of random length and orientation. Figure A.4 in the Appendix Section A.3 shows an example of a custom environment with 15 random walls.

Note that in this particular example, fixed sensing locations (diamond markers in Figure A.4) are used since the high-level policy is not yet trained.

It is clear that increasing the amount of random walls in the environment increases the exploration difficulty for the robot. The more random walls are spawned, the likelier the chance of generating intricate structures.

Moreover, the RL agent generalizes better if the replay buffer contains disparate states. Care must be taken that the agent's starting orientation does not face a nearby wall. Otherwise the agent has no choice but to collide after its first step. To resolve this issue, random x- and y-coordinates are chosen for the agent that lay within the world boundary and that do not interfere with any walls or obstacles. At these initial x- and y-coordinates, a valid starting orientations can be found by evaluating the vector from the agent to nearby walls for all angles in the unit circle. Doing so eliminates all orientations pointing towards close-by walls. A valid starting orientation can thus randomly be chosen from the remaining angles. Appendix Section A.3, Figure A.5 demonstrates how every agent is spawned facing away from the nearby wall.

5.1.3 LIDAR simulator

LIDAR data is required in the EKF SLAM module to train the high- and low-level policy and test the trajectory-based EKF SLAM implementation. For each laser beam of the LIDAR sensor, a range and a bearing is returned. Given an angle resolution for the LIDAR, one can compute the corresponding amount of laser beams. Laser beams that do not encounter any obstacles return the maximum LIDAR range instead along with their bearing.

In the simulation, the Python library Shapely is used to create the desired amount of laser beams. Each laser beam is represented as a line. The LIDAR simulator then checks for any intersections between the laser beams lines and the obstacles in the environment. If an intersection is detected, a red point is placed at the position of intersection to illustrate the laser beam hitting the obstacle. The range to an obstacle is calculated as the Euclidean distance

between the true robot pose and the position of intersection. Figure 5.2 illustrates the LIDAR sensor in the simulation rendering for a 3 degree angle resolution.

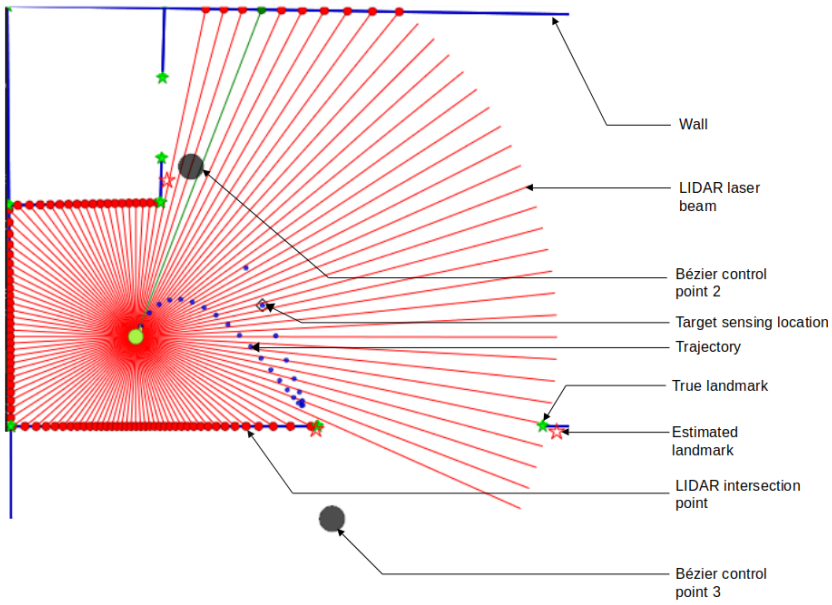


Figure 5.2: LIDAR simulation rendering with the remaining elements of the trajectory-based EKF SLAM algorithm. The agent is displayed as a yellow dot at the center of the LIDAR sensor. The red lines simulate the LIDAR beams. The green LIDAR beam denotes the current robot orientation.

On top of the LIDAR, Figure 5.2 also shows the rendering of all modules inside the trajectory-based EKF SLAM algorithm.

5.2 Training high- and low-level RL models without obstacle avoidance

All experiments are performed in simulation due to the current COVID-19 pandemic. The main goal of this section is to assess and compare the performance of the reward functions proposed in Equation 4.18. For that reason, this section neglects obstacle avoidance since it affects the performance of the low-level RL agent. The second goal is to evaluate the performance of the high-level policy as a coverage planner.

Moreover, to further reduce the influence of the environment when training the trajectory generation, the high-level RL agent is replaced by pre-defined sensing locations or a simple algorithm that sets a sensing location towards the closest unobserved landmarks. Training both RL agents separately has several benefits. First, it ensures that the performance of the RL agent during training truly reflects the performance of the chosen reward function. Second, it increases the training stability. If both high- and low-level RL agents were to be trained simultaneously, the failure of one RL agent would confuse the training of the second RL agent. Third, it allows to train the high- and low-level policy in parallel thus saving time. All training experiments are done with seed 1 as opposed to testing procedures that utilize five seeds $< 12, 1111, 98, 4321, 123 >$

At the same time, Table 5.1 summarizes the PD controller gains used throughout the experiments of this section.

Table 5.1: Velocity and yaw rate PD controller gains used in all experiments

Kp_v	Kd_v	Kp_ω	Kd_ω
2.75	0.15	4.5	0.28

At last, regarding the process and sensor noise in the simulation, the variances are qualitatively tuned starting from the identity matrix. An indication of too much noise is apparent if the EKF estimates diverge from their true values. The values for the variances in the process and sensor noise Q and R along with the covariance matrix of the EKF state vector C_x are given in Table 5.2.

Table 5.2: Values for the covariance matrix of the EKF state vector, process and sensor noise

Process Q		Sensor R		EKF state		
σ_v^2	σ_ω^2	σ_{range}^2	$\sigma_{bearing}^2$	σ_x^2	σ_y^2	σ_θ^2
0.01	0.0017	0.002	0.00017	0.25	0.25	0.27

5.2.1 Low-level model hyper-parameter tuning

The first step in training the low-level RL agent is finding suitable values for the TD3 hyper-parameters.

Setup

As the experiment only seeks to tune the model hyper-parameters, notably the learning rate, a simple environment with pre-defined sensing locations and landmarks suffices. Figure 5.3 illustrates the environment used in this experiment.

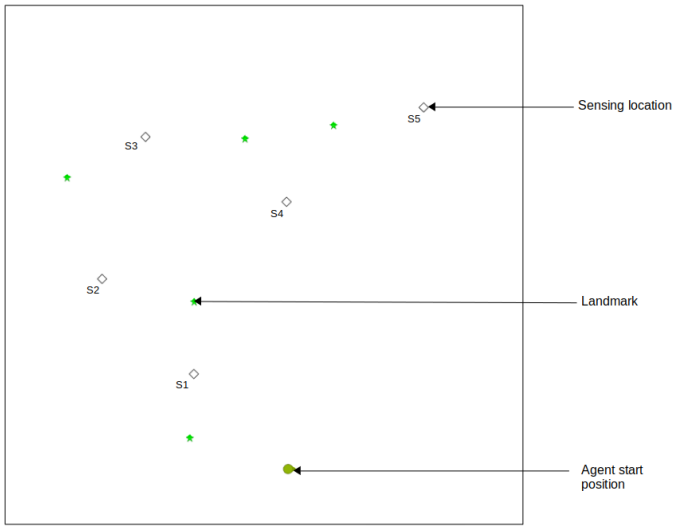


Figure 5.3: Training environment to tune the TD3 network hyper-parameters. The agent is depicted as a green dot, while the landmark and the sensing locations (S1, ..., S5) are displayed as green stars and black diamonds respectively.

For a good comparison between the different sets of hyper-parameters, no randomization is added to the environment. For each episode, the agent starts at the same initial position and moves through the fixed sensing locations from $S1$ to $S5$. The landmarks are placed exactly in-between two successive sensing locations such that each trajectory observes at least one landmark.

The hyper-parameters are tuned with the continuous reward function $R(s,a)_{c1}$ described in Equation 4.18. Note that $R(s,a)_{c2}$ could have been used as well. After performing research in the used hyper-parameters in the original TD3 paper (Fujimoto et al., 2018), the set of hyper-parameters shown in Table 5.3 is chosen as a baseline for further tuning.

Table 5.3: Starting hyper-parameters for the TD3 network

Hyper-parameter	Batch size	γ	τ	Learning rate	Start episode	Policy noise	Noise clip	Policy frequency	Exploration noise
Value	256	0.99	0.005	0.001	50	0.2	0.5	2	0.1

Starting from the learning rate in Table 5.3, the convergence of the average reward curve after training is evaluated qualitatively. Depending on how well the model converges, the learning rate is increased or decreased. The best learning rate is found when the agent accumulates the most reward, in a steady manner throughout the episodes. The agent trains for at least 2000 episodes where a random policy is used for the first 50 episodes.

Results

The first training session with the hyper-parameters listed in Table 5.3 results in a noisy cumulative reward curve. Therefore, a moving average with a window size of 20 episodes is taken to smoothen the curve and make the overall trend more apparent. The experiment yields a moving average reward that increases with approximately 0.5 over 1500 episodes (see Figure A.6 in Appendix Section A.4). Though, a large drop-off can be observed at around 2100 episodes. This abrupt decrease in the average reward is most likely due to a too large learning rate resulting in an unstable learning process. Still, considering the rising nature of the overall average reward, all hyper-parameters except for the learning rate are kept the same in the next experiments.

Next, the learning rate is decreased by one order of magnitude to $lr = 1e-4$. In this case, the average reward curve did not exhibit any increasing or decreasing trend at all (see Figure A.7, Appendix Section A.4). This can only mean that the new learning rate is set too low this time. Hence, the optimal learning must lay in between $lr = 1e-3$ and $lr = 1e-4$. A learning rate of $lr = 0.0004735$ turns out to be near optimal given the experiment conditions. Figure 5.4 shows the resulting moving average reward curve obtained after training for 10000 episodes.

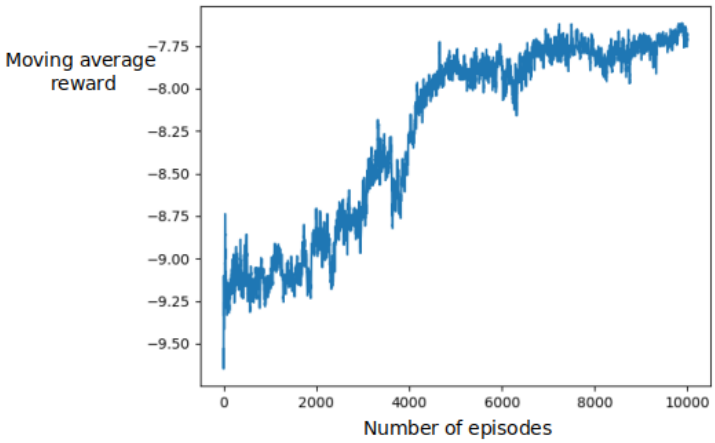


Figure 5.4: Moving average over 20 episodes after training for 10000 episodes and learning rate $lr = 0.0004375$.

Figure 5.4 shows a clear upwards trend in the average reward, which means the agent is learning the desired behaviour.

5.2.2 Reward function comparison

The next step in the process of training the low-level RL agent is to find the best reward function that leads to model convergence. $R(s,a)_{c1}$ and $R(s,a)_{c2}$ are qualitatively and quantitatively compared to find the best reward function for subsequent experiments.

Setup

Given both continuous reward functions of Equation 4.18, two identical low-level RL agents are trained in the environment presented in Figure 5.5. One with the Kalman gain based reward function $R(s,a)_{c1}$ and the other with the covariance matrix based reward function $R(s,a)_{c2}$.

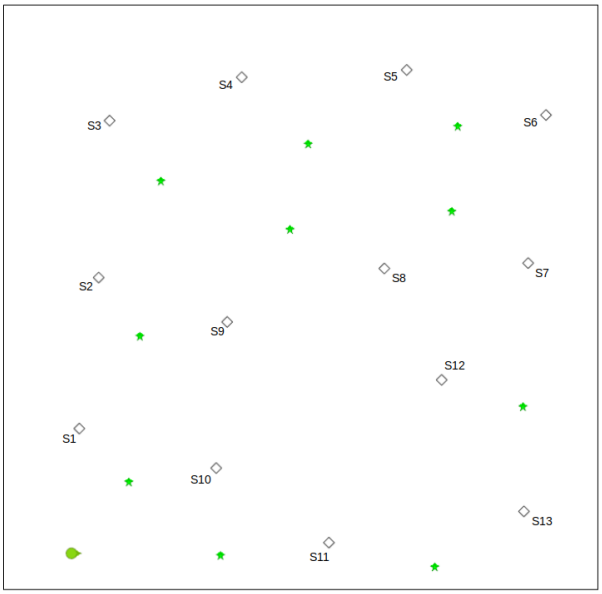


Figure 5.5: Training environment for the reward function comparison experiment. The agent initial position is displayed as a green dot in the bottom left corner. Sensing locations and landmarks can be observed as diamond shaped markers and green stars respectively.

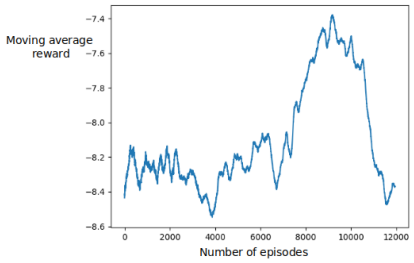
In this experiment, fixed sensing locations and landmarks are chosen again since generalization is not a concern at this stage. In addition, for a good comparison between $R(s,a)_{c1}$ and $R(s,a)_{c2}$, both training runs must be as identical as possible. Figure 5.5 shows the order in which the RL agent moves through the sensing locations.

Besides, the landmarks are located only on one side between successive sensing locations. Hence, the low-level policy should prefer trajectories that pass through these landmarks as it increases the accuracy of the EKF estimates. Both low-level RL agents are trained with the hyper-parameters presented in Table 5.3 but with learning rate $lr = 0.0004375$. The models train for at least 10000 episodes which is equivalent to 160000 trajectories.

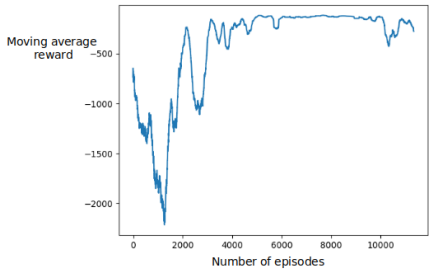
A final qualitative validation is also performed, comparing the trajectories originating from the best learned policy to random trajectories when tested in the environment of Figure 5.5.

Results

The best reward function between $R(s,a)_{c1}$ and $R(s,a)_{c2}$ produces a moving average reward curve that has a continuously upwards trend. This would mean the model is converging towards a global maximum. Figure 5.6 presents the moving average of the received reward by both low-level RL agents during training.



(a) $R(s,a)_{c1}$ using the Kalman gain K



(b) $R(s,a)_{c2}$ using the covariance matrix $Pest$

Figure 5.6: Average reward curves during training for both continuous reward functions proposed in Equation 4.18

Looking at Figure 5.6, it is obvious that the reward function based on the EKF uncertainties ($R(s,a)_{c2}$) managed to converge much better compared to Kalman gain based reward function ($R(s,a)_{c1}$). The most notable difference is that $R(s,a)_{c2}$ reaches a clear asymptote with an average reward around -200, while $R(s,a)_{c1}$ does not.

In general, a model converges easier if the actions of the RL agent have a direct influence on the resulting reward. In this way, the RL network can better approximate the function yielding optimal actions. With this in mind, the results of Figure 5.6 make sense since it was seen from Equation 2.29 that minimizing the Kalman gain boiled down to minimizing the EKF covariance matrix. Yet, the presence of other matrices in the Kalman gain and the presence of the EKF covariance matrix in both the numerator and the denominator complicates the minimization of the ratio.

The moving average RMSE values in the robot position, the robot orientation and the landmark position (Figure A.8, Figure A.9 and Figure A.10 in the Appendix Section A.5) also confirms the idea that $R(s,a)_{c2}$ is a more effective reward function.

Although, for both $R(s,a)_{c1}$ and $R(s,a)_{c2}$ the average RMSE in the landmark and robot positions does decline during training. $R(s,a)_{c2}$ reaches its asymptote, of an RMSE around 0.04m, already after 40000 trajectories compared to 140000 for $R(s,a)_{c1}$. This further validates the superiority of $R(s,a)_{c2}$ as a reward function for trajectory generation.

Finally, given $R(s,a)_{c2}$ a simple test can be done to study the effect of the process and sensor noise on the trajectory shape. The experiment is described in Appendix Section A.6. Results show that the low-level RL agent prefers trajectories with the least amount of loops as the noise gets bigger. The test validates the hypothesis that excessive turns are less desirable.

5.2.3 Low-level policy generalization

In this last experiment without obstacles, a low-level RL agent is trained in a random environment to improve generalization.

Setup

The low-level RL agent trains with the hyper-parameters of Table 5.3 and with learning rate $lr = 0.0004375$. Randomization is added to the training in the form of 30 randomly generated landmarks and a random robot pose initialization before every episode. Since generalizing the model also increases the convergence time, the model is left training for a total of 240000 episodes. Such a large amount of episodes amounts to around 800000 computed trajectories by the low-level policy.

Moreover, the RL agent has an observation range of 0.6 meters. The sensing locations are placed in the direction towards the closest unobserved landmark.

At last, the generalized policy is evaluated 20 times in the same environment with randomization for seeds $< 12, 1111, 98, 4321, 123 >$.

Results

The moving average reward after 240000 episodes is shown in Figure 5.7.

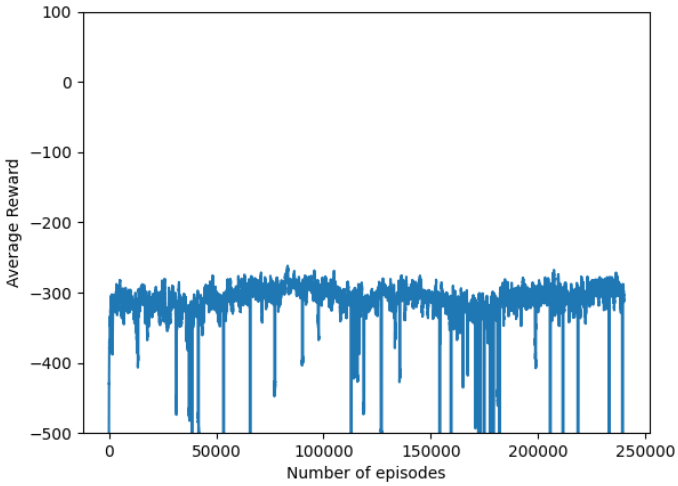


Figure 5.7: Moving average reward of the low-level RL agent while training in the stochastic environment of Subsection 5.2.3.

From Figure 5.7, no clear increase in the average reward can be observed over time. The reward curve keeps oscillating around approximately -290. Many large, downwards peaks, can also be observed. These large spikes in the EKF uncertainty are likely due to moment where the RL agent did not observe any landmarks. Hence, the error accumulates quickly until new landmarks are observed.

Yet, results from testing the trained policy for 20 episodes on 5 different seeds show very promising results. Table 5.4 shows the average RMSE in the robot position, robot orientation and landmark position average over the 20 episodes.

Table 5.4: RMSE during evaluation

	Robot position (m)		Robot orientation (radians)		Landmark position (m)	
Seeds	Random	Policy	Random	Policy	Random	Policy
12	0.0348	0.0317	3.0885	2.9051	0.0767	0.0739
1111	0.0420	0.0327	3.8704	2.8390	0.0831	0.0759
98	0.0344	0.0411	3.4639	3.2164	0.0791	0.0842
4321	0.0392	0.0359	3.6616	3.3738	0.0821	0.0799
123	0.0274	0.0240	3.8329	3.0875	0.0667	0.0599

In 4 out of 5 seeds the average RMSE in the robot pose as well as in the landmarks is smaller than the RMSE using random trajectories. These results validate the effectiveness of the chosen reward function.

5.2.4 High-level policy training

This experiment deals with the training of the high-level RL agent serving as coverage planner.

Setup

The training of the high-level RL agent follows the same setup as the generalization of the low-level policy in Section 5.2.3. Nevertheless, a maximum of 10 trajectories per episode is set in case the high-level policy does not manage to visit all the landmarks. After 10 trajectories the training episode is terminated. As mentioned in Section 5.2, the high- and low-level RL agents are trained separately. Therefore, the trajectory generation consists of a simple straight line between the agent and the chosen sensing location.

Additionally, the landmark coordinates in the estimated EKF state vector should provide the high-level RL agent with enough information to know which areas have already been discovered or not. Thus, as a first test, the high-level RL agent is trained without the occupancy grid. Including the occupancy grip in the RL state vector makes training significantly more difficult as it requires a lot more GPU memory.

Results

After training for 37000 episodes, Figure 5.8 shows the obtained moving average reward curve.

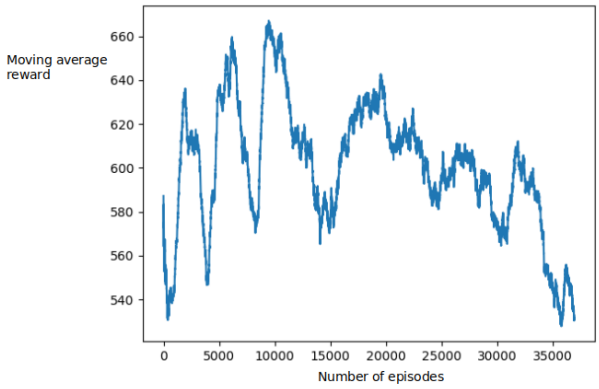


Figure 5.8: Moving average reward of the high-level RL agent during training

From the moving average reward seen in Figure 5.8, an increase in the coverage per episode is observed until 10000 episodes. After this turning point, the high-level policy continuously decreased in performance until the training session ended.

The poor convergence of the high-level policy in such a simple environment suggests that the EKF state vector alone is not sufficient for the high-level RL agent to infer optimal sensing locations. The occupancy grid must be passed to the high-level RL agent as well. It functions as memory, showing which areas of the environment has already been explored. But it also highlights where the unexplored areas are located.

5.2.5 Coverage evaluation

In this last experiment, the trained high- and low-level RL agents are combined in the trajectory-based SLAM algorithm. The goal of the experiment is to evaluate the coverage of the trained high-level policy in different seeds.

Setup

The trained high-level policy is tested for 20 episodes on the seeds $< 12, 1111, 98, 4321, 123 >$. The average coverage over the 20 episodes is calculated for each seed. The high-level agent must visit 15 landmarks, randomly distributed in the environment.

Results

Table 5.5 shows the average coverage percentage obtained during testing of the trained high-level policy and random sensing locations.

	Coverage percentage (%)	Coverage percentage (%)
Seeds	Random	Policy
12	54.0	52.3
1111	57.2	49.8
98	60.7	50.2
4321	68.1	50.5
123	53.7	47.0
Average coverage	58.7	50.0

Table 5.5: Coverage percentage for the trained and random policy. The coverage is averaged over 20 episodes for 5 different seeds.

The results in Table 5.5 validate the findings during training of the high-level policy. Namely, the high-level RL agent did not converge towards a global maximum.

5.3 Addition of obstacle avoidance

The inclusion of obstacle avoidance in the training of both the high- and low-level RL agents is studied in this section.

5.3.1 Merging uncertainty minimization with obstacle avoidance

The goal of this experiment is to study the performance and convergence ability of the low-level RL agent when training with the reward function in Equation 4.19 on the low-level RL agent.

Setup

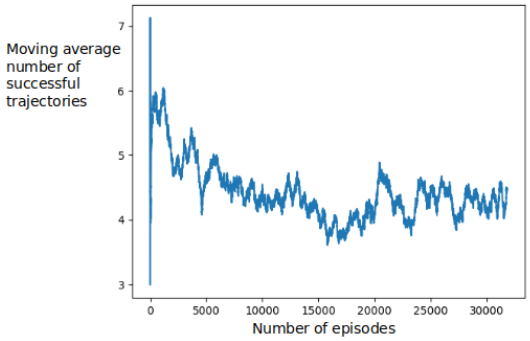
To include obstacle avoidance in the trajectory generation, the low-level RL agent receives a penalty of -2000 if it collides with an obstacle (Equation 4.19). Furthermore, the training environment is kept simple, with only one wall randomly generated in the environment. Starting with a low amount of walls reduces the importance of obstacle avoidance to maximize the episodic reward. This ensures the low-level agent still focuses primarily on minimising the EKF uncertainty.

Following the same approach as in Section 5.2.3, 15 landmarks are randomly generated in the environment. The initial position of the robot pose is also randomized. Note, since walls are present in the environment, the LIDAR data is passed to the low-level RL agent. The LIDAR has a resolution of 3 degrees and a maximum range of 0.6 m.

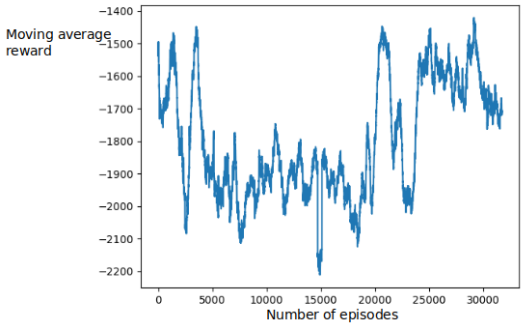
The trained policy is again tested for 20 episodes on seeds $< 12, 1111, 98, 4321, 123 >$ with 1 wall and 15 landmarks.

Results

The training results manifest themselves through the average number of successful trajectories in Figure 5.9a and the moving average reward Figure 5.9b



(a) Moving average number of trajectories without a collision.



(b) Moving average reward for the low-level policy with obstacle avoidance.

Figure 5.9: Moving average reward for the low-level policy while training with obstacle avoidance. Together with the average number of trajectories that are collision free.

The training results in Figure 5.9 show an initial decrease in the received reward which corresponds to a decrease in the number of successful trajectories. Looking at Figure 5.9a, it is clear that the low-level RL agent has difficulties with obstacle avoidance. This was expected as it remains the most challenging problem to solve with the low sample efficiency of trajectory generation. Overall, the model did not manage to truly converge to an asymptote. Nevertheless, the evaluation on the 5 different seeds presented in Table 5.6 gives a better picture of the model performance.

Table 5.6: RMSE during evaluation with obstacle avoidance

	Robot position (m)		Robot orientation (radians)		Landmark position (m)	
Seeds	Random	Policy	Random	Policy	Random	Policy
12	0.0551	0.0156	5.4071	2.3040	0.1001	0.0292
1111	0.0330	0.0132	3.8307	6.9186	0.0598	0.0254
98	0.0551	0.0323	4.1119	6.0664	0.0939	0.0750
4321	0.0553	0.0376	5.0883	9.4193	0.0967	0.0767
123	0.0168	0.0326	2.2977	6.8001	0.0345	0.0640

The evaluation results in Table 5.6 make apparent that the model still minimizes the EKF uncertainties as expected. Except for the robot orientation. It is very likely, considering the long training time in Section 5.2.3, that the model still needs more training time to minimize the robot orientation. Thus, the results in Table 5.6 lead to think that the robot orientation is more difficult to minimize compared to Cartesian coordinates.

5.3.2 High-level policy retraining

In this experiment, the high-level RL agent is trained again but with the addition of obstacles and thus also obstacle avoidance.

Setup

To include obstacle avoidance in the training of the high-level policy, the high-level RL agent receives a penalty of -200 if the robot collides with an obstacle. Furthermore, analogous to Section 5.3.1, the high-level RL agent is trained on 15 randomly generated landmarks and one randomly generated wall in the environment. In order to properly combine the high- and low-level policy, it is also important that both models are trained with the same LIDAR angle resolution

and range (i.e. a resolution of 3 degrees and a range of 0.6 meters).

As opposed to Section 5.2.4, the high-level RL agent receives the LIDAR data as well as the occupancy grid (72px x 72px). The occupancy grid serves two main purposes. First, it provides memory as it keeps track of where the walls are located even if they are out-of-range of the LIDAR sensor. Second, it provides crucial information about which areas of the environment have already been visited (white or black pixels in the occupancy grid) and which areas are still unexplored (grey space). This should help the RL agent find undiscovered landmarks.

Results

Figure 5.10 presents the moving average reward obtained during training for 25000 episodes.

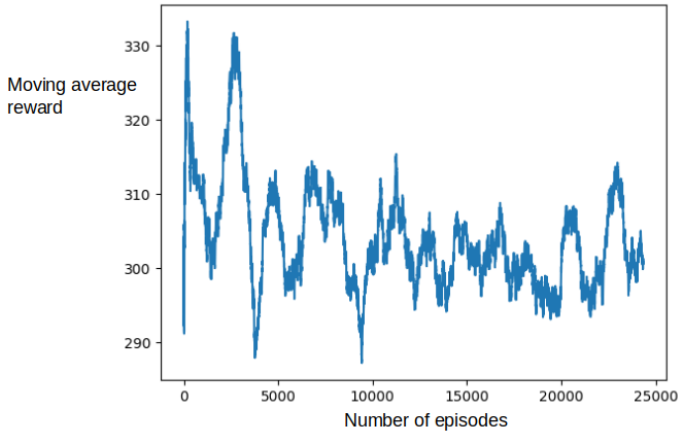


Figure 5.10: Average reward during training of the high-level policy with obstacle avoidance.

From the training results presented in Figure 5.10, it can be seen that the RL agent does not manage to increase its accumulated reward over the number of episodes. Unlike the high-level policy without obstacle avoidance shown in Figure 5.8, the average reward has a clear decreasing trend from the start to around 20000 episodes. After 20000 episodes, a small incline in the curve can be observed. Such a rise indicates the agent is learning a better policy than it currently uses. Therefore, one can assume that the agent requires further training time and 25000 episodes did not suffice for the model to converge.

5.3.3 Trajectory-based SLAM algorithm evaluation

The final experiment evaluates the performance of the high-level RL agent in Section 5.3.2 as a coverage planner. Given the high-level RL agent action space in Figure 4.5, this experiment does not seek to evaluate the obstacle avoidance capabilities of the RL agent. Rather, it assess the affect of the action space when confronted with obstacles on the coverage performance.

Setup

Similarly to the previous coverage planner assessment in the no obstacle case (Section 5.2.5), the algorithm is tested on 5 different seeds <12, 1111, 98, 4321, 123>. The chosen environment consists of a random environment with 15 landmarks and one wall for obstacle avoidance. The position of the landmarks are randomized after every training episode. The assessment is done in the framework of the trajectory-based EKF SLAM algorithm using the trained low-level policy.

Results

The average percentage of landmark coverage over 20 episodes in 5 different seeds is displayed in Table 5.7.

	Random	Policy
Seeds	Coverage percentage(%)	Coverage percentage(%)
12	60.0	41.4
1111	53.3	54.0
98	62.1	35.1
4321	64.9	68.1
123	56.1	52.7
Average coverage	59.3	50.3

Table 5.7: Comparison of a random policy and the trained high-level policy on the landmark coverage for 5 different seeds. Coverage percentages are averaged over 20 episodes per seed.

Comparing the results of Table 5.7 to the results obtained without obstacle avoidance in Table 5.5, one can see that the average coverage for both the random and the trained policy vary less than 0.6%. For the trained policy, the average coverage stays very close to 50% with or without obstacles. At first hand, these results seem great seeing that the high-level policy is able to perform obstacle avoidance without affecting the coverage percentage. However, the same can be said about the random policy. Hence, one wall is insufficient to accurately measure the influence of the chosen action space on the coverage percentage. In all cases, the coverage percentage reached by the high-level policy is too low and must first be improved in order to have a successful trajectory-based EKF SLAM algorithm.

5.4 Discussion and recommendations

In both the experiments with and without obstacle avoidance, the low-level policy is able to minimize the EKF uncertainties and improve the EKF estimates. Even though, the thesis took a slightly different approach regarding the parameterization and the reward functions, the experiment results are consistent with the conclusions of Kollar and Roy (2008).

The covariance matrix based reward function for the low-level policy also showed that it can be merged with a discrete penalty for obstacle avoidance. Nevertheless, the experiment in Section 5.3.1 highlight the difficulty in finding a proper balance between the continuous and the discrete portion of the reward function. As it was not able to achieve good obstacle avoidance performance. Though, the low sample efficiency of the trajectory-based navigation also increases the difficulty of the obstacle avoidance problem.

Experiments regarding the high-level RL agent functioning as coverage planner failed to properly converge given the reward function proposed in Equation 4.11. The first experiment in Section 5.2.4 exhibited a parabolic trend in the average reward curve during training. Although, oscillations in the reward curves are to be expected during training, the curve returns to its initial value of 540 after 37000 episodes. This change in the reward curve is not drastic and shows the RL agent requires more training episodes before it can converge. A similar behaviour is also observed when the experiment is repeated with obstacle avoidance. The coverage planner experiments show that the current reward function for the high-level policy can be improved. Additionally, it indicated the need for a larger experiment with at least 5 walls to increase the effect of obstacle avoidance on the high-level RL agent action space.

It is recommended to migrate from the current 2D Python environment to a more accurate and realistic robotic simulator like Gazebo (Gazebo, 2021) in future experiments. Together with a better motion model for the robot movements, the more accurate physics will help the RL agents converge to global optimum. Furthermore, the low-level policy can be improved by using the ground-truth coordinates of the robot pose and the landmarks during training. For instance, the RMSE can be used as an additional metric along with the EKF covariance matrix to further reduce the error in the EKF estimates. Regarding obstacle avoidance, the continuous reward function proposed by Botteghi et al. (2020) can provide better feedback to the low-level RL agent during training about how close the trajectories come to nearby obstacles.

Improvements regarding the high-level policy can be found in the size of the occupancy grid. The occupancy grid tells the high-level agent where walls are located and where it has never been before. Hence, a larger occupancy grid provides greater accuracy of the position of the objects in the grid itself. Meaning the high-level agent can perform decisions based on more accurate data. Though, larger occupancy grid also lead to more GPU memory usage. GPU memory especially becomes an issue during training due to the large replay buffer and the target networks that must all be loaded in memory. Therefore, the experiments in this thesis were limited to an occupancy grid of 72px x 72px.

Next, having a fixed size occupancy grid is not computationally efficient but it simplified the loading in the convolutional layers of the RL network. A better approach would be to use a variable size occupancy grid that grows as the agent explores. All the while passing a fixed window size of the occupancy grid through the high-level RL network. Doing so would mean that part of the memory provided by the occupancy grid will be lost. Therefore, one must in that case implement recurrency in the RL network in the form of recurrent blocks. The recurrency provides old occupancy grids to the network. Finally, the reward function of the high-level policy can be improved by looking at the ground-truth maps instead of a simple coverage percentage like was used in this thesis. Similarly to the approach proposed by Chaplot et al. (2020). The ground-truth maps are fully explored occupancy grids of the environment. While the agent is exploring the unknown environment, it builds an occupancy grid that should look identi-

cal to the ground-truth occupancy grid after completion. Such a reward function provides much more accurate information about how much of the environment is left to be explored and where precisely the unexplored areas are. The ground-truth maps themselves can easily be created automatically when generating new environments in simulation.

5.5 Summary

Experiments are vital to adequately train and test the proposed RL and SLAM-based framework. All experiments are performed in a 2D Python environment using the Python library Gym to create the structure needed to train and test RL networks. Custom environments can be created using the Python library Shapely. Though, for better generalization of the RL models, random environments are created by randomizing the agent's initial position and the position of obstacles and landmarks.

With obstacle avoidance in mind, a LIDAR sensor is simulated as a series of Shapely lines with fixed lengths, extending out from the agent at a constant angle resolution. Any intersection of a LIDAR line with any other Shapely polygon in the environment returns the range and bearing to the intersection point.

Two main experiments are proposed. The first experiment neglects obstacle avoidance, focusing solely on the validation of the proposed reward functions for the high- and low-level RL agent. At the beginning, the hyper-parameters of the low-level TD3 network are tuned using an environment with fixed landmarks and sensing locations. Once the proper hyper-parameters are found, the low-level policy is trained in a bigger environment, keeping fixed landmarks and sensing locations to ensure good comparison between the two proposed reward functions in Equation 4.18. The best reward function is then used to train the low-level policy in a generalized environment where the agent initial position and the landmarks are randomized. Here the fixed sensing locations are replaced by a high-level policy that selects a sensing location in the direction of the closest unobserved landmark. The low-level policy experiments show that the second reward function $R(s,a)_{c2}$ performs better. Additionally, the low-level policy successfully managed to reduce the EKF uncertainty compared to random trajectories. At last, training of the high-level policy in a random environment proved to be unsuccessful as the coverage remained lower than random sensing locations.

The second main experiment focuses on merging obstacle avoidance with the minimization of the EKF uncertainties as well as on the training of the high-level TD3 network. Experiments are performed with random landmarks positions and initial positions. One wall is randomly placed for obstacle avoidance. The experiment starts off by training the low-level policy. Results show again that the low-level RL agent achieves to reduce the EKF uncertainties. But the policy fails with regards to obstacle avoidance. Conversely, the high-level policy is trained is a similar fashion. Altough, the high-level policy achieved similar performance with or without obstacle avoidance. The coverage is not better than random sensing locations. The high-level RL agent must be further trained to improve results. The last test utilizes the trained low-level policy to move between sensing locations and test the high-level coverage on 5 different seeds.

6 Conclusion

The aim of this thesis was to develop and implement a trajectory based EKF SLAM algorithm and research how RL and EKF SLAM can be combined to improve the map accuracy of the unknown environment.

(a) *How can Reinforcement Learning be used to select informative target positions and generate informative trajectories?*

The thesis shows informative trajectories can be selected using a TD3 RL agent trained using the EKF covariance matrix as reward function. Selecting informative sensing locations using RL proves unsuccessful at this stage.

(b) *How can we hierarchically structure the RL policies?*

The RL policies can be hierarchically structured by having one policy select informative sensing locations that serve as target position for the trajectory generation of the second RL policy. The proposed algorithm shows how both RL policies can be hierarchically structured.

(c) *How can we shape the reward function?*

The best reward function to generate informative trajectories consists of the sum of the trace of the EKF covariance matrix. At this point in the development, no successful coverage reward function has been found for the high-level policy.

(d) *What is the benefit of the proposed Reinforcement Learning approach?*

The proposed RL approach reduces the error in the EKF estimates while exploring. RL policies can be shaped and generalized to navigate in all types of environments while being trained in simulation.

1. *How can we tightly incorporate RL and EKF in a single framework?*

The proposed trajectory-based EKF SLAM algorithm shows how RL and EKF can tightly be incorporated in a single framework. The successful training of the low-level policy proves that EKF can benefit and improve the performance of the RL.

A Appendix

A.1 Mapper figures

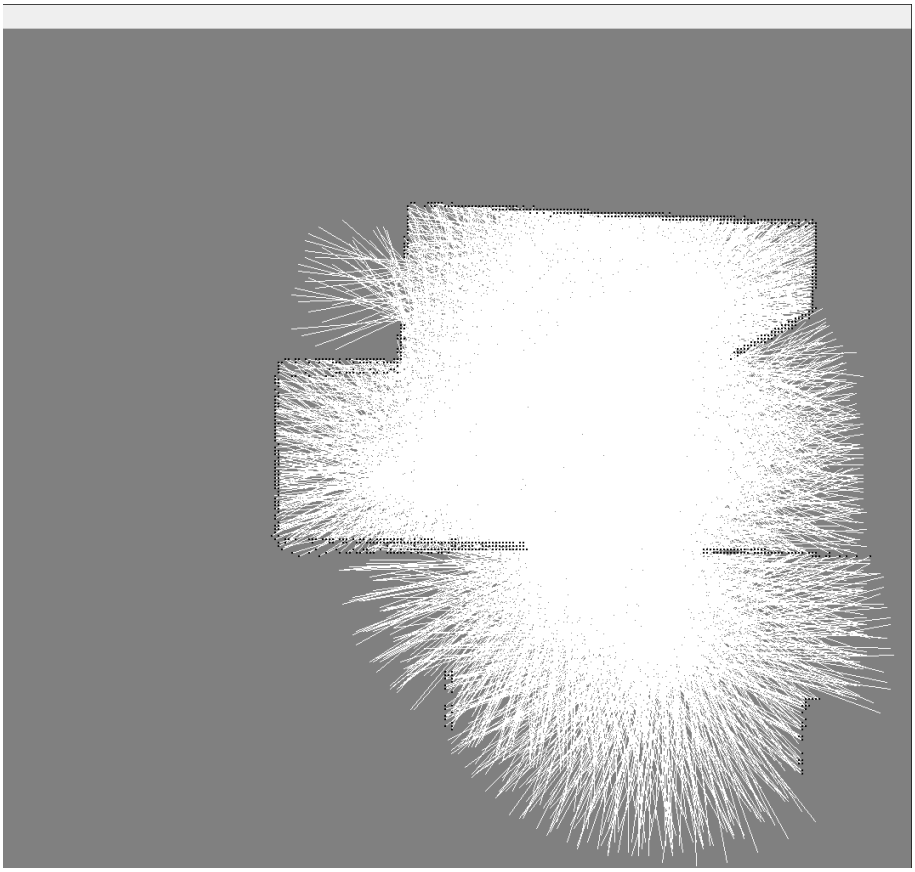


Figure A.1: Occupancy grid with conversion factor c_{image} = 0.0025. White areas represent obstacle free space while black pixels portray obstacles in the environment. Grey pixels instead are unknown regions.

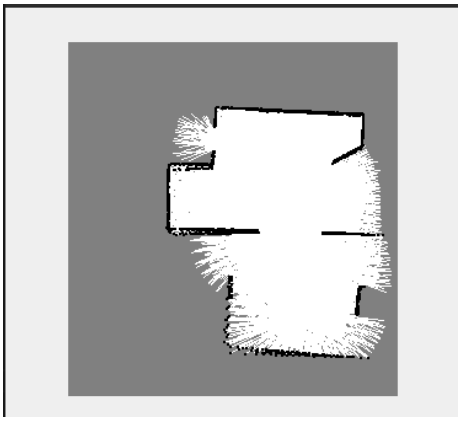


Figure A.2: Occupancy grid with conversion factor c_{image} = 0.01. White areas represent obstacle free space while black pixels portray obstacles in the environment. Grey pixels instead are unknown regions.

A.2 Low-level policy cubic polynomial parameterization

With the standard cubic polynomial, the parameterization of the trajectory between the robot pose and the sensing location follows Equation A.1.

$$\begin{aligned} x(t) &= a_x * t^3 + b_x * t^2 + c_x * t + d_x \\ y(t) &= a_y * t^3 + b_y * t^2 + c_y * t + d_y \end{aligned} \tag{A.1}$$

Where $t \in [0,1]$. The next step is to find the eight coefficients $(a_{x,y}, b_{x,y}, c_{x,y}, d_{x,y})$ that define the polynomial in Equation A.1. From the eight parameters, four of them must be fixed in order for the trajectory to pass through the robot pose and the sensing location.

Next, a second constraint is applied to the orientation at the start of the trajectory, $\mathscr{R}_\theta$. By constraining the initial direction of the agent, a smooth transition between successive trajectories is ensured. Thus, five of the eight parameters are fixed. The last three are free for the agent to choose. The action vector of the agent is the combination of all three free parameters $actions = [free_{action1}, free_{action2}, free_{action3}]$. These are the initial and end speeds of the trajectory together with the end orientation. Now that the fixed and free parameters are defined, the system of equations can be solved to find the eight coefficients from Equation A.1.

The initial conditions, d_x and d_y are easily derived according to Equation A.2.

$$\begin{aligned} x(t=0) &= d_x = \mathscr{R}_x \\ y(t=0) &= d_y = \mathscr{R}_y \end{aligned} \tag{A.2}$$

Similarly, the initial velocities (at $t = 0$) follow Equation A.3.

$$\begin{aligned} \frac{dx(t=0)}{dt} &= c_x = V_x 0 \\ \frac{dy(t=0)}{dt} &= c_y = V_y 0 \end{aligned} \tag{A.3}$$

Where $V_x 0 = free_{action1} * cos(\mathscr{R}_\theta)$ and $V_y 0 = free_{action1} * sin(\mathscr{R}_\theta)$.

Continuing with the derivation, the end position is fixed like in Equation A.4.

$$\begin{aligned} x(t=1) &= a_x + b_x + V_x 0 + \mathscr{R}_x = \mathscr{S}_x \\ y(t=1) &= a_y + b_y + V_y 0 + \mathscr{R}_y = \mathscr{S}_y \end{aligned} \tag{A.4}$$

With $\mathscr{S}_{x,y}$ the coordinates of the target sensing location. Equation A.4 shows four unknown coefficients but only two equations. Two more equations are thus missing, these are the end velocities characterized in Equation A.5.

$$\begin{aligned} \frac{dx(t=1)}{dt} &= 3 * a_x + 2 * b_x + V_x 0 = V_x 1 \\ \frac{dy(t=1)}{dt} &= 3 * a_y + 2 * b_y + V_y 0 = V_y 1 \end{aligned} \tag{A.5}$$

Likewise, $V_x 1 = free_{action3} * cos(free_{action2})$ and $V_y 1 = free_{action3} * sin(free_{action2})$. At this point, one can solve for all four remaining coefficients, see Equation A.6.

$$a_x = 2 * (\mathscr{R}_x - \mathscr{S}_x) + V_x 0 + V_x 1 \tag{A.6}$$
$$a_y = 2 * (\mathscr{R}_y - \mathscr{S}_y) + V_y 0 + V_y 1$$

and therefore b_x and b_y are found as follow, Equation A.7.

$$b_x = 3 * (\mathscr{S}_x - \mathscr{R}_x) - 2 * V_x 0 - V_x 1 \tag{A.7}$$
$$b_y = 3 * (\mathscr{S}_y - \mathscr{R}_y) - 2 * V_y 0 - V_y 1$$

Substituting all the coefficients in Equation A.1 yields the correct cubic parameterization. In the end, the action space $\mathscr{A}$ of the agent is continuous with dimensions $\mathbb{R}^3$.

With this parameterization, the agent can take many different paths to reach a given sensing location. Figure A.3 illustrates in a 3D plot a more extensive example of all the available trajectories for the agent to compute given the set of 3 free parameters constituting the action space $\mathscr{A}$.

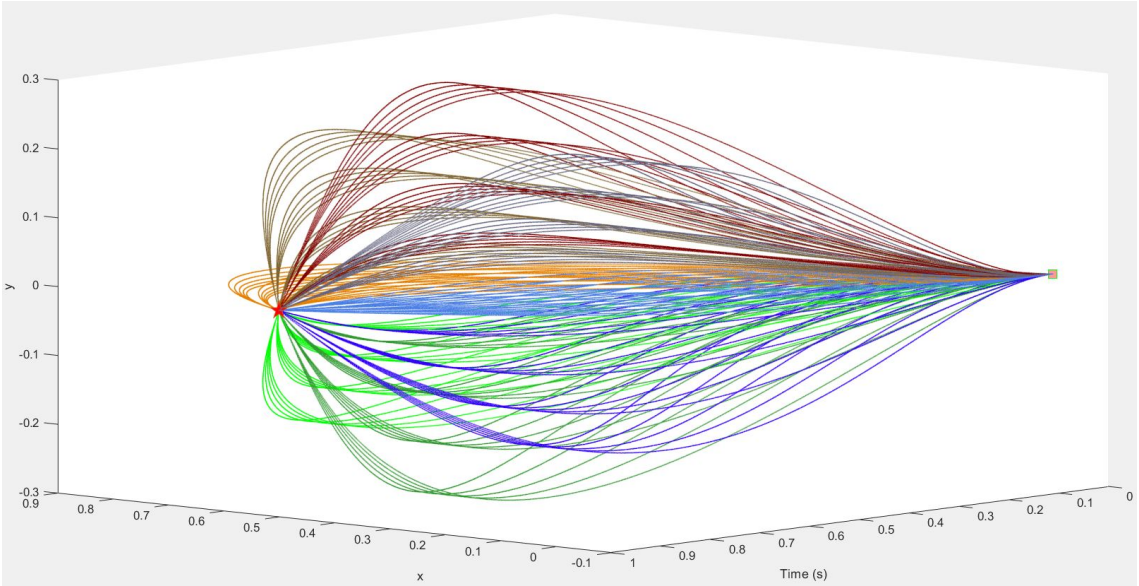


Figure A.3: 225 sample trajectories for the standard cubic polynomial parameterization. The vertical z-axis contains the y-coordinates of the sample trajectories. The x-axis on the horizontal plane depicts the x-coordinates of the sample trajectories while the time $t \in [0,1]$ is shown on the y-axis. The 8 different colors still represent 8 different orientations at t=1. The orange square located at (0,0,0) represents the initial position, i.e. the robot pose. The red start at (3,1,0) is the target sensing location. Units are in meters.

The sample trajectories in Figure A.3 highlight the effect each action has on the generated trajectories. Varying the speed at t=0 creates these shifted trajectories along the positive x-direction, large initial speeds lead to overshooting the target position. Equivalently, changes in the speed at t=1 affect the trajectories in the y-direction in this example, yielding a smaller coil for lower speeds. At last, the trajectories are colored depending on the arrival orientation, the last free parameter of the agent.

Even though Figure A.3 illustrates 225 different trajectories, the actual action space is continuous.

A.3 Randomizing custom environments

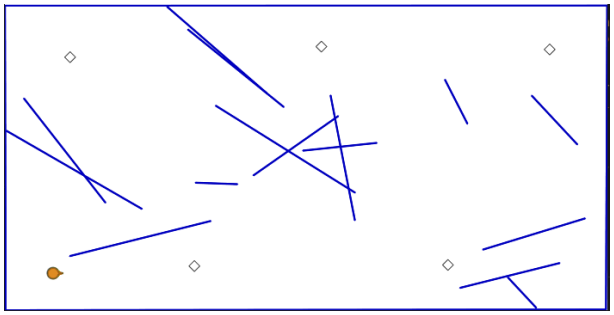


Figure A.4: Custom environment with 15 randomly placed walls. The diamond shaped markers are fixed sensing locations. The agent start position (orange dot) is located in the bottom left corner.

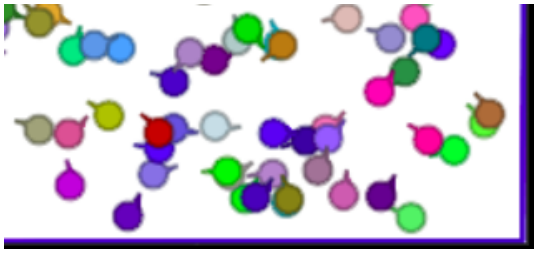


Figure A.5: Custom environment showing no agent spawns facing a wall. The small protrusion on the agent designates the orientation of the robot.

A.4 Learning rate tuning results

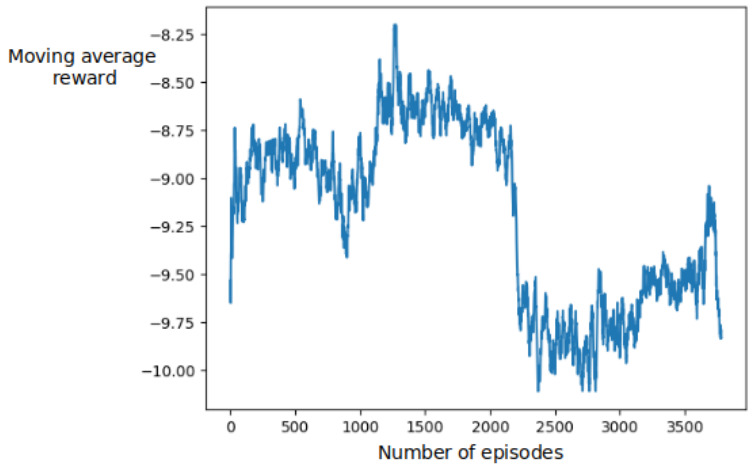


Figure A.6: Moving average reward over 20 episodes after training for a total of 3500 episodes and learning rate $lr = 0.001$.

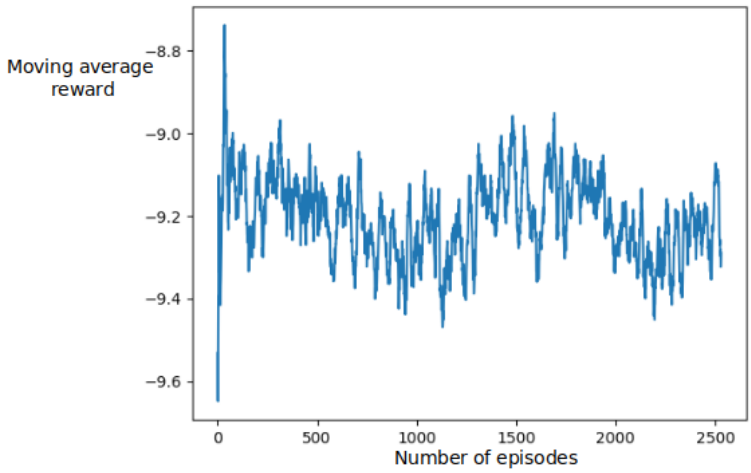
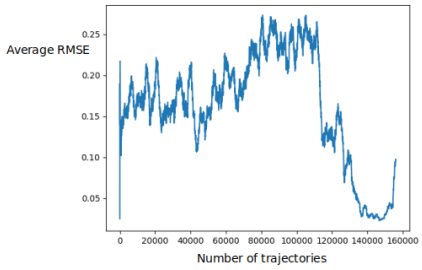
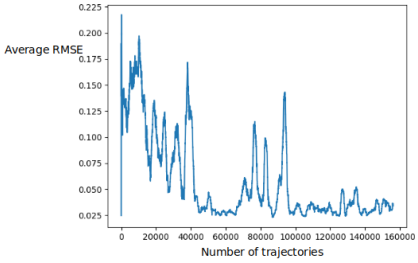


Figure A.7: Moving average reward over 20 episodes after training for a total of 2000 episodes and learning rate $lr = 0.0001$.

A.5 Reward function comparison

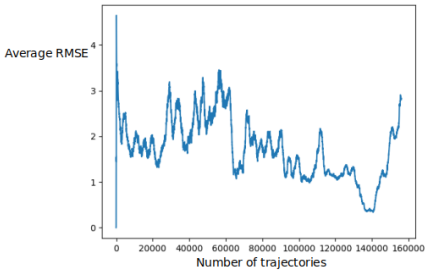


(a) $R(s,a)_{c1}$ using the Kalman gain K

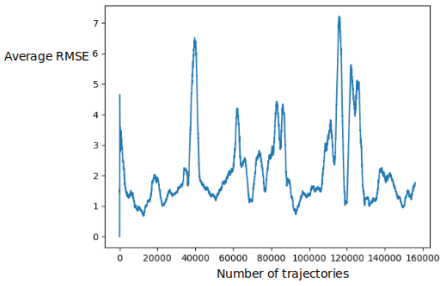


(b) $R(s,a)_{c2}$ using the covariance matrix $Pest$

Figure A.8: Average RMSE in the robot coordinates (in meters) during training of the low-level policy with R_{c1} and R_{c2}.

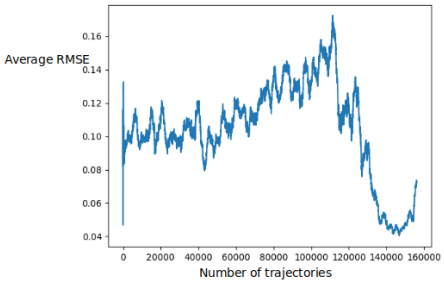


(a) $R(s,a)_{c1}$ using the Kalman gain K

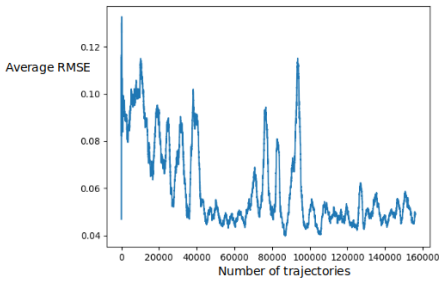


(b) $R(s,a)_{c2}$ using the covariance matrix $Pest$

Figure A.9: Average RMSE in the orientation (in radian) during training of the low-level policy with R_{c1} and R_{c2}.



(a) $R(s,a)_{c1}$ using the Kalman gain K



(b) $R(s,a)_{c2}$ using the covariance matrix $Pest$

Figure A.10: Average RMSE in the landmark coordinates (in meters) during training of the low-level policy with R_{c1} and R_{c2}.

A.6 Effect of the noise on the trajectory shape

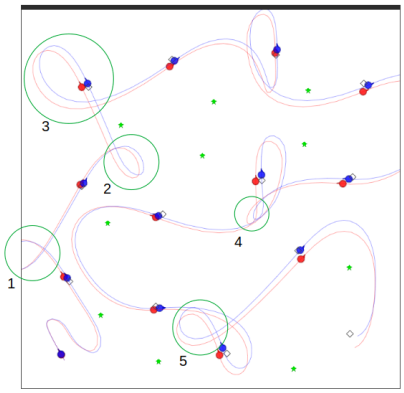
The goal of this experiment is to study the effect of the added process and sensor noise in simulation on the computed trajectories. Too much noise leads to incoherent EKF estimates which results in bad PD control outputs. The effect of the noise on the trajectory shape is studied by gradually decreasing the process and sensor noise.

Setup

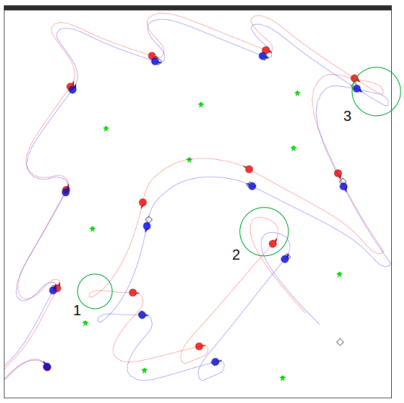
The experiment repeats the experiment from Section 5.2.2 two more times but with $R(s,a)_{c2}$ as the chosen reward function. In each case the amount of process and sensor noise added to the simulation is decreased by a factor of 0.6 and 0.8 respectively.

Results

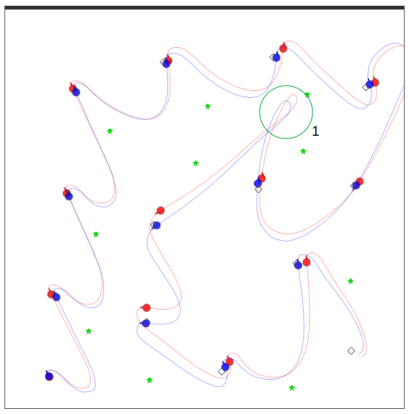
A qualitative evaluation of the generated trajectories for all three process and sensor noise values highlight the differences in the learned policies. Figure A.11 shows the produced trajectories during one test episode.



(a) Trajectories originating from the learned policy with 0.6 times the noise model of Table 5.2. Loops in the trajectory are accentuated with green circles.



(b) Trajectories originating from the learned policy with 0.8 times the noise model of Table 5.2. Loops in the trajectory are accentuated with green circles.



(c) Trajectories originating from the learned policy with 1 times the noise model of Table 5.2. Loops in the trajectory are accentuated with green circles.

Figure A.11: Comparison of generated trajectories for policies trained with 0.6, 0.8 and 1 time the noise presented in Table 5.2. The green circles indicate loops in the trajectories.

Increasing the process and sensor noise in the EKF SLAM module results in a decrease in the number of generated loops. This finding is consistent with Kollar and Roy (2008), who state that the EKF algorithm is sensitive to trajectories with excessive rotation. When the noise is relatively small, the RL agent is not penalized for trajectories with excessive rotation. Furthermore, it is observed that the agent has no preference for trajectories that come particularly close to landmarks as long as the landmark fits within the agents observation range.

Bibliography

Arana-Daniel, N., R. Rosales-Ochoa and C. López-Franco (2011), Reinforced-SLAM for path planing and mapping in dynamic environments, in *2011 8th International Conference on Electrical Engineering, Computing Science and Automatic Control*, pp. 1–6, doi:10.1109/ICEEE.2011.6106563.

AtsushiSakai (2021), Python Robotics.
https://github.com/AtsushiSakai/PythonRobotics/tree/master/SLAM/EKFSLAM

Bailey, T., J. Nieto, J. Guivant, M. Stevens and E. Nebot (2006), Consistency of the EKF-SLAM Algorithm, in *2006 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pp. 3562–3568, doi:10.1109/IROS.2006.281644.

Barrett, L. and S. Narayanan (2008), Learning all optimal policies with multiple criteria, pp. 41–47, doi:10.1145/1390156.1390162.

Barth-Maron, G., M. W. Hoffman, D. Budden, W. Dabney, D. Horgan, D. TB, A. Muldal, N. Heess and T. Lillicrap (2018), Distributional Policy Gradients, in *International Conference on Learning Representations*.
https://openreview.net/forum?id=SyZipzbCb

Botteghi, N., B. Sirmaçek, M. Khaled, M. Poel and S. Stramigioli (2020), On reward shaping for mobile robot navigation: A reinforcement learning and SLAM based approach, Workingpaper, arXiv.org.

Chaplot, D. S., D. Gandhi, S. Gupta, A. Gupta and R. Salakhutdinov (2020), Learning To Explore Using Active Neural SLAM, in *International Conference on Learning Representations (ICLR)*.

Chen, F., S. Bai, T. Shan and B. Englot (2019), Self-Learning Exploration and Mapping for Mobile Robots via Deep Reinforcement Learning, doi:10.2514/6.2019-0396.

Dankwa, S. and W. Zheng (2019), Twin-Delayed DDPG: A Deep Reinforcement Learning Technique to Model a Continuous Movement of an Intelligent Robot Agent, pp. 1–5, doi:10.1145/3387168.3387199.

Dinnissen, P., S. N. Givigi and H. M. Schwartz (2012), Map merging of Multi-Robot SLAM using Reinforcement Learning, in *2012 IEEE International Conference on Systems, Man, and Cybernetics (SMC)*, pp. 53–60, doi:10.1109/ICSMC.2012.6377676.

Elvira, R. (2020), OrbSlam3.
https://github.com/UZ-SLAMLab/ORB_SLAM3

Eustice, R., M. Walter and J. Leonard (2005), Sparse Extended Information Filters: Insights into Sparsification, pp. 3281 – 3288, ISBN 0-7803-8912-3, doi:10.1109/IROS.2005.1545053.

Fujimoto, S., H. van Hoof and D. Meger (2018), Addressing Function Approximation Error in Actor-Critic Methods, in *Proceedings of the 35th International Conference on Machine Learning*, volume 80 of *Proceedings of Machine Learning Research*, Eds. J. Dy and A. Krause, PMLR, pp. 1587–1596.
http://proceedings.mlr.press/v80/fujimoto18a.html

Gazebo (2021), Gazebo simulator.
http://gazebosim.org/

Grisetti, G., R. Kümmerle, C. Stachniss and W. Burgard (2010), A tutorial on graph-based SLAM, *IEEE Transactions on Intelligent Transportation Systems Magazine*, **vol. 2**, pp. 31–43, doi:10.1109/MITS.2010.939925.

Guevara-Reyes, E., A. Y. Alanis, N. Arana-Daniel and C. Lopez-Franco (2013), Integration of an inverse optimal control system with reinforced-SLAM for path planning and mapping in

dynamic environments, in *2013 IEEE International Autumn Meeting on Power Electronics and Computing (ROPEC)*, pp. 1–6, doi:10.1109/ROPEC.2013.6702713.

Kollar, T. and N. Roy (2008), On the Representation and Estimation of Spatial Uncertainty, **vol. 27**, no.2, pp. 175–196, doi:10.1177/0278364907087426.

Lillicrap, T., J. Hunt, A. Pritzel, N. Heess, T. Erez, Y. Tassa, D. Silver and D. Wierstra (2015), Continuous control with deep reinforcement learning, *CoRR*.

Luviano Cruz, D. and W. Yu (2014), Multi-agent path planning in unknown environment with reinforcement learning and neural network, in *2014 IEEE International Conference on Systems, Man, and Cybernetics (SMC)*, pp. 3458–3463, doi:10.1109/SMC.2014.6974464.

MarketsandMarkets (2019), Autonomous Navigation Market by Solution (Sensing System, Software, Processing Unit), Platform (Airborne, Land, Space, Marine, Weapon), Application (Commercial, Military Government), and Region - Global Forecast to 2030.
https://www.marketsandmarkets.com/Market-Reports/autonomous-navigation-market-206053964.html

Montemerlo, M.; Thrun, S. K. D. W. B. (2002), FastSLAM: A factored solution to the simultaneous localization and mapping problem, in *Proceedings of the AAAI National Conference on Artificial Intelligence*, pp. 593–598.

Mustafa, K., N. Botteghi, B. Sirmacek, M. Poel and S. Stramigioli (2019), Towards continuous control for mobile robot navigation: A reinforcement learning and slam based approach, **vol. 42**, no.2/W13, pp. 857–863, ISSN 1682-1750, doi:10.5194/isprs-archives-XLII-2-W13-857-2019, 4th ISPRS Geospatial Week 2019 ; Conference date: 10-06-2019 Through 14-06-2019.
https://www.gsw2019.org/

Nguyen, T. T., N. D. Nguyen, P. Vamplew, S. Nahavandi, R. Dazeley and C. P. Lim (2020), A multi-objective deep reinforcement learning framework, *Engineering Applications of Artificial Intelligence*, **vol. 96**, p. 103915, ISSN 0952-1976, doi:https://doi.org/10.1016/j.engappai.2020.103915.
https://www.sciencedirect.com/science/article/pii/S0952197620302475

Paz, L. and J. Neira (2006), Optimal local map size for EKF-based SLAM, pp. 5019 – 5025, doi:10.1109/IROS.2006.282529.

Pei, Z., S. Piao, M. Quan, M. Z. Qadir and G. Li (2020), Active collaboration in relative observation for multi-agent visual simultaneous localization and mapping based on Deep Q Network, **vol. 17**, no.2, p. 1729881420920216, doi:10.1177/1729881420920216.
https://doi.org/10.1177/1729881420920216

Placed, J. A. and J. A. Castellanos (2020), A Deep Reinforcement Learning Approach for Active SLAM, **vol. 10**, no.23, ISSN 2076-3417, doi:10.3390/app10238386.
https://www.mdpi.com/2076-3417/10/23/8386

Ramezani, A. and D. Lee (2018), Memory-based reinforcement learning algorithm for autonomous exploration in unknown environment, *International Journal of Advanced Robotic Systems*, **vol. 15**, p. 172988141877584, doi:10.1177/1729881418775849.

ReportLinker (2021), Autonomous Navigation Market Research Report by Platform, by Solution, by Application - Global Forecast to 2025 - Cumulative Impact of COVID-19.
https://www.reportlinker.com/p05913497/Autonomous-Navigation-Market-Research-Report-by-Platform-by-Solution-by-App.html?utm_source=GNW

Smith, R.C.; Cheeseman, P. (1986), On the Representation and Estimation of Spatial Uncertainty, **vol. 5**, no.4, pp. 56–68, doi:10.1177/027836498600500404.

https://frc.ri.cmu.edu/~hpm/project.archive/reference.file/Smith&Cheeseman.pdf

Wen, S., Y. Zhao, X. Yuan, Z. Wang, D. Zhang and L. Manfredi (2020), Path planning for active SLAM based on deep reinforcement learning under unknown environments, *Intelligent Service Robotics*, **vol. 13**, doi:10.1007/s11370-019-00310-w.

Zhang, J., L. Tai, J. Boedecker, W. Burgard and M. Liu (2017), Neural SLAM: Learning to Explore with External Memory, *arXiv preprint arXiv:1706.09520*.