Master's thesis

NTNU
Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Engineering Cybernetics

Jan Aleksander Fijalkowski

LiDAR-based Resilient Collision-free Navigation for Aerial Robots in Closed Environments

Master's thesis in Cybernetics and Robotics
Supervisor: Kostas Alexis
June 2021

NTNU
Norwegian University of Science and Technology

Jan Aleksander Fijalkowski

LiDAR-based Resilient Collision-free Navigation for Aerial Robots in Closed Environments

Master's thesis in Cybernetics and Robotics
Supervisor: Kostas Alexis
June 2021

Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Engineering Cybernetics

NTNU
Norwegian University of
Science and Technology

Problem Description

The research conducted in this thesis aims to investigate a reinforcement learning approach for collision-avoidance and navigation that can be applied onto an aerial robot aiming to navigate complex, confined environments. The method should be able to utilize perceptual sensors and observations and function without the need of reconstructing a 3D map of the environment. Hence, the goal is to:

- Develop real-time collision-avoidance systems that ensure the safety of a Micro Aerial Vehicle (MAV) given only minimalistic sensing and computing requirements.
- Explore the effectiveness and robustness of the proposed approach on a suitable test environment.

To undertake this effort, the following set of tools are made available for this project:

- Computer with specification found in Appendix B.1 provided by the Autonomous Robots Lab at NTNU (ARL-NTNU).
- Computer with specification found in Appendix B.2 provided by NTNU.
- OpenAI baselines [25].
- The open-source Rotors Simulator [9], including an appropriate model of an MAV provided by ARL-NTU.
- LiDAR Simulator provided by ARL NTNU.

Preface

This master's thesis was written during the spring semester 2021 at the Department of Engineering Cybernetics at the Norwegian University of Science and Technology (NTNU) in collaboration with the Autonomous Robots Lab (ARL) at NTNU.

The work conducted in this thesis continues the work done during the fall semester 2020, as a part of the course TTK4550 - Specialization Project in Engineering Cybernetics [11]. A considerable effort was put into a literature review and the foundation was laid to further develop the system used in this thesis. Chapter 4 is similar to the theoretical chapter in the unpublished project report but is considerably rewritten and updated to further fit the scope of this thesis.

I would like to thank Prof. Dr. Kostas Alexis for the opportunity to work with such an interesting subject as autonomous navigation and flight, as well as guidance and help during the process. I would also like to thank Huan Nguyen and the rest of the ARL-NTNU lab for all the help provided during my research.

Trondheim, June 2021
Jan A. Fijalkowski

Abstract

The thesis aims to develop and evaluate a navigation controller for quadcopters using deep reinforcement learning frameworks. The quadcopter is deployed in confined environments and should navigate collision-free to a goal region without any prior information or any online construction of a map. For this an end-to-end learning algorithm was used that transforms raw sensory data directly into actuation commands for a flying robot. It does so by extracting relevant data from a 3D LiDAR and odometry sensor and trains the robot using a curriculum-based transfer learning strategy. To train the specific quadcopter, several different environments were created with diversified complexity in Gazebo simulator. After the training stage, the robot was deployed to several different unseen, simulated environments to evaluate the controller. In addition, the proposed method was evaluated and tested in a underground mine structure with an expert planner. The controller was able to generalize to unseen situations and navigate fast through all of the environments.

Sammendrag

Denne oppgaven tar for seg utviklingen og evalueringen av en navigasjons- og kollisjonsunngåelseskontroller for et autonomt multirotorfartøy. For å løse dette ble det brukt et rammeverk som utnytter dyp forsterkende læring. Det autonome fartøyet hadde til hensikt å navigere kollisjonsfritt til et bestemt mål gjennom trange og lukkede miljøer. Dette ble gjort uten å ha noe kjennskap til miljøet fra før eller å aktivt konstruere et 3D-kart. En ende-til-ende læringsalgoritme ble brukt for å transformere rådata fra sensorer til kommandoer brukt til å styre det flyvende fartøyet. Den utviklede metoden gjør dette ved å hente ut viktig data fra en 3D LiDAR og odometrisensor, og bruker dette til å trene en kontroller. Denne kontrolleren blir trent i et spesifikt miljø og overfører den lærte kunnskapen til et nytt, mer komplisert miljø. De forskjellige miljøene ble konstruert i simulatoren Gazebo. For å validere den utviklede metoden ble fartøyet testet i forskjellige miljøer som flyvende roboten ikke hadde blitt eksponert for tidligere. Hensikten med dette var å teste alle aspekter ved den utviklede metoden. Kontrolleren om bord på fartøyet ble også testet i en sjakt, og ble guidet ved hjelp av en algoritmisk baneplanleggerekspert. Metoden var i stand til å løse alle de forskjellige scenarioene den ble eksponert for, og navigerte raskt og effektivt til sitt endemål.

Abbreviations

DDPG	Deep Deterministic Policy Gradient
DoF	Degrees of Freedom
GBPlanner	Graph-based Planner
GNSS	Global Navigation Satellite System
IMU	Inertial Measurement Unit
LiDAR	Light Detection And Ranging
MAV	Micro Air Vehicle
MDP	Markov Decision Process
RL	Reinforcement Learning
RMF	Resilient Micro Flyer
ROS	Robot Operating System
RRG	Rapidly-exploring Random Graphs
SGD	Stochastic Gradient Descent
SOR	Statistical Outlier Removal

Contents

List of Figures

List of Tables

Chapter 1

Introduction

This chapter introduces the motivation, a short introduction to the problem, previous work related to the problem, the structure of this thesis and the contributions to the overall solution.

1.1 Motivation

When disaster strikes, humans rely on first responders to rush in and save lives. The degraded environments that these humans face are beyond dangerous and often impossible to navigate through due to many hazards present. In these *Search and Rescue* (SAR) operations it can be more useful to send in robots, without risking any human life to get better situational awareness. This can pose a huge tactical advantage when doing decision making in difficult situations. However, such operations are often time critical, as any time wasted can be fatal for the victims. Any aid that should be used needs to be efficient and reliable, as there is also a lot of uncertainty tied to such conditions. *Micro Air Vehicles* (MAVs) have the ability to navigate through narrow spaces in a very fast and agile motion. There is already a lot of related work exploring these topics using MAVs. However, what is very common between all of these studies is that fast solutions come at the expense of agile and robust solutions. External sensors or a prior map is usually required in an attempt to have quick and robust autonomous flight. Eventually, one strives for a system that is reliable, fast, and agile and at the same time uses as little prior knowledge as possible, essentially eliminating the need for a prior map. In addition, most of the autonomous quadcopters that navigate beyond the visual line of sight today uses mostly *Global Navigation Satellite Systems* (GNSS). However, such aid does not work in indoor environments. This is problematic, as the navigation solutions should be deployed in challenging environments where humans cannot necessarily reach, and which may as well be underground or in some cave. The alternative is to therefore only relay on onboard sensors, such as *LiDAR*, *Camera*, and

Inertial Measurement Unit (IMU). To achieve this, a robust solution that can handle inaccuracies in data, such as inaccuracies provided by sensors, is required. One would also need to reduce latency by having fast sensors and algorithms. The MAV is limited by its battery life, and any hardware or software used by the MAV should work to the benefit of extending flight time or search area.

Deep learning is a class of techniques that has proven itself to be effective and fast in handling diverse and complex environments. In order to combine this kind of generalization power with the ability to make effective decisions, one needs an algorithmic framework around decision-making, which is what reinforcement learning provides. This thesis presents a fast computational data-driven method that utilizes reinforcement learning and aims to reduce any computational overhead related to construction of a consistent online 3D map of the environment by only processing and using raw sensory data to navigate within structurally complex, confined environments. Consequently, the method presented in this thesis heavily emphasizes on fast algorithms while also being robust and reliable. In brief, the method departs from the current state-of-the-art where collision-free navigation methods in previously unknown environments require the online reconstruction of the map and instead offers an end-to-end solution starting from extremely low-dimensionality range data and providing robot control actions for safe flight.

1.2 Related Work

There is a lot of proposed methods and work tied to fast autonomous mapping, navigation, and exploration. In 2017, Defense Advanced Research Projects Agency (DARPA) launched a subterranean challenge [3], searching for novel approaches where teams compete using robots to do large-scale exploration in unknown underground environments. In the work of Reinhart et al [30] they use a robot platform to perform autonomous exploration that relies on a multimodal localization and mapping solution fusing Light Detection and Ranging (LiDAR) and Inertial Measurement Unit (IMU) data in combination with a graph-based exploration path planner. However, a wide set of methods focuses on learning navigation in simpler environments emphasizing on navigation by heuristic approaches such as neural networks [20] and classical methods. In the problem of navigating from one initial configuration to another, it is also possible to apply simpler geometrical algorithms such as *Line-of-sight* and *Pure Pursuit* guidance laws if one can generate obstacle-free paths. Such path following algorithms rely on simpler control strategies that do not necessarily work well if to be applied in more complex, confined environments. In most problems it is necessary to couple perception and control to have a fully aware agent that can navigate within structurally challenging environments. Such perceptional-aware navigation

can be formulated as an optimization problem by formulating certain action and perception objectives. Thus, it is possible to solve this optimization problem by leveraging a *Model Predictive Control* (MPC) [10].

1.3 Contributions

This thesis aims to investigate a reinforcement learning approach for mapless navigation and collision-avoidance that will be applied onto a simulated aerial robot. The work in this thesis overcomes these challenges by:

- Designing different simulated environments that incentive specific behavior within the agent during training. Thus, subsequently demonstrating that navigation and obstacle avoidance policies can effectively be learned end-to-end through a curricular approach in confined obstacle-filled environments.

The training process is done by leveraging a reinforcement learning framework. The additional contribution within this thesis lies in:

- The development of the feature extraction pipeline.
- Reward shaping.
- Effectively using all available data to systematically train the quadcopter to have the desired behavior such that it can be deployed in a diverse set of simulated environments.

1.4 Structure of the Thesis

The remainder of this thesis is structured as follows:

- Chapter 2 introduces the notation and the theoretical background of the simulator used to simulate the MAV in this project.
- In Chapter 3 a theoretical overview is given of the perceptional part of the system.
- Chapter 4 describes the reinforcement learning framework used in the proposed method.
- Chapter 5 presents the learning strategies applied onto the system and how the proposed method can be combined with a motion planner.
- Chapter 6 describes in depth the proposed method used in the thesis.
- In Chapter 7 the proposed method is evaluated and tested on a diverse set of environments.

- Chapter 8 discusses the results from the previous chapter and how well the proposed method works.
- Lastly in Chapter 9, the findings in this thesis are concluded.

Chapter 2

Simulating the Quadcopter

With the object of applying sophisticated control system to a *Micro Air Vehicle* (MAV) to be stable and fly autonomously, it is necessary to model it in some way. This chapter will therefore cover how to model such vehicle, specifically a quadcopter, so that it can be used in a simulator. The equations and notations are represented using Fossens notation from [12].

2.1 Introducing the System

The simulated quadcopter is modeled after the *Resilient Micro Flyer* (RMF)[6], an aerial robot which is made out of four rotating propellers. The actuators are spun in precise ways to control the RMF in 6 different *degrees of freedom* (6-DoF), 3 translational directions and 3 rotational directions. However, to understand how to model the RMF, it is first necessary to look at the hardware specifications such that the simulated RMF realistically depicts the real version. There are mainly 3 onboard sensors available on one of the latest versions of RMF developed at ARL-NTNU, and the structure of the system implies some modularity, as it is possible to change out the different sensor components. The sensors available are:

- *Inertial Measurement Unit* (IMU): IMU ADIS [7].
- *Light Detector And Ranging* (LiDAR) sensor: OS-1 [27], OS-2 [28] and Velodyne [37].
- *Camera*: mvBlueFOX [38].

In addition, there is an onboard low-level autopilot [8].

All of these components are connected to an onboard computer, NVIDIA TX2 [34] as seen in Figure 2.1.

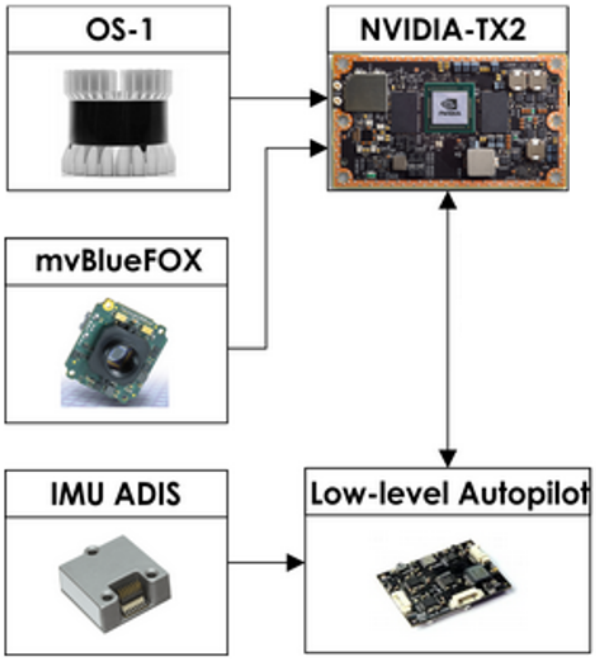


Figure 2.1: The structure of how all of the components are connected. (Source of figures: [27], [38], [7], [34], [8])

2.2 Reference Frames and Notation

Before defining the dynamic of the system, the first thing to look at are the available reference frames. There are two available coordinate systems, one attached to the moving RMF $\{b\}$ and the other one being the non-accelerating coordinate system $\{w\}$. $\{b\} = (b_1, b_2, b_3)$ with origin o_b constitutes the set of unit vectors that describes the body-fixed coordinate frame, and likewise $\{w\} = (w_1, w_2, w_3)$ with origin o_w describes an inertial frame that is fixed to some arbitrary point in the world as seen in Figure 2.2. Note that the z-direction is pointing up. Throughout this thesis the position and orientation of the RMF are described relative to the world reference frame, while the linear and angular velocities of the vehicle are expressed in the body-fixed frame. $\boldsymbol{p}_{wb}^{w} = [x^w, y^w, z^w]^\top \in \mathbb{R}^3$ is the position vector of the center of mass with respect to the world frame. Euler angles, roll (ϕ), pitch (θ) and yaw (ψ) are used to represent rotations. In order to express these in the world frame, a rotation matrix $\mathbf{R}$ that can transform between body and the world frame is introduced.

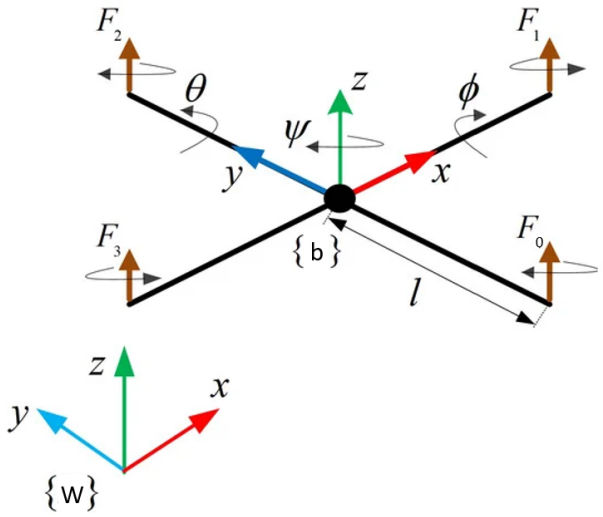


Figure 2.2: Dynamic model of the RMF (Source of Figure: [41])

However, due to representation of singularity of the Euler angles, a four-parameter based unit quaternion $\boldsymbol{q}$ is used instead. A quaternion consists of a real part η and three imaginary parts $\boldsymbol{\epsilon} = [\epsilon_1, \epsilon_2, \epsilon_3]$ as depicted in Equation (2.1)

$$\boldsymbol{q} = \begin{bmatrix} \eta \\ \epsilon_1 \\ \epsilon_2 \\ \epsilon_3 \end{bmatrix}. \tag{2.1}$$

The rotation matrix for the unit quaternion can be expressed as in Equation (2.2) [12, eq.2.70].

$$\mathbf{R}(\boldsymbol{q}_b^w) = \boldsymbol{I}_3 + 2\eta \boldsymbol{S}(\boldsymbol{\epsilon}) + 2\boldsymbol{S}^2(\boldsymbol{\epsilon}), \tag{2.2}$$

where $\boldsymbol{I}$ is the identity matrix and $\boldsymbol{S}$ is the skew-symmetric matrix.

Equation (2.2) can be used to express the body-fixed velocities in the world frame seen in Equation (2.3) [12, eq.2.71]

$$\dot{\boldsymbol{p}}_{wb}^w = \mathbf{R}(\boldsymbol{q}_b^w)\boldsymbol{v}_{\boldsymbol{wb}}^{\boldsymbol{b}}, \tag{2.3}$$

where $\boldsymbol{v}_{\boldsymbol{wb}}^{\boldsymbol{b}} = [u, v, w]^\top$ is the body-fixed linear velocities. Similarly, it is possible to express the angular velocity transformation by the body-fixed angular velocities $\boldsymbol{\omega}_{\boldsymbol{wb}}^{\boldsymbol{b}} = [p, q, r]^\top$ and a transformation matrix $\boldsymbol{T}$ as seen in Equation (2.4)[12, eq.2.77]

$$\dot{\boldsymbol{q}}_b^w = \boldsymbol{T}(\boldsymbol{q}_b^w)\boldsymbol{\omega}_{\boldsymbol{wb}}^{\boldsymbol{b}}. \tag{2.4}$$

Equation (2.3) and Equation (2.4) can be written in a 6-DoF kinematic compact equation form as seen in Equation (2.5) [12, eq.2.83]

$$\begin{bmatrix} \dot{\boldsymbol{p}}_{wb}^{w} \\ \dot{\boldsymbol{q}}_{b}^{w} \end{bmatrix} = \begin{bmatrix} \mathbf{R}(\boldsymbol{q}_{b}^{w}) & \mathbf{0}_{3\times3} \\ \mathbf{0}_{4\times3} & \boldsymbol{T}(\boldsymbol{q}_{b}^{w}) \end{bmatrix} \begin{bmatrix} \boldsymbol{v}_{\boldsymbol{wb}}^{\boldsymbol{b}} \\ \boldsymbol{\omega}_{\boldsymbol{wb}}^{\boldsymbol{b}} \end{bmatrix} \tag{2.5a}$$

$$\dot{\boldsymbol{\eta}} = \boldsymbol{J}_q(\boldsymbol{\eta})\boldsymbol{\nu} \tag{2.5b}$$

where $\boldsymbol{\eta} = [x^w, y^w, z^w, \phi, \theta, \psi]^\top$ is the state vector, $\boldsymbol{\mathcal{V}} = [u, v, w, p, q, r]^\top$ is the generalized velocity vector expressed in the body frame and $\boldsymbol{J}_q(\boldsymbol{\eta}) \in \mathbb{R}^{7\times6}$ is a nonquadratic transformation matrix.

In addition, one should also introduce the external moments and forces that act upon the body of the RMF, denoted by a generalized vector of external forces and moments expressed in the body frame $\boldsymbol{\tau} = [X, Y, Z, J, K, N]^\top$.

The notation can be summarized in the following table (Table 2.1):

Body frame				World frame
DOF		Forces and moments	Linear and angular velocities	Positions and Euler angles
1	Motions in the x_b-direction	X	u	x^w
2	Motions in the y_b-direction	Y	v	y^w
3	Motions in the z_b-direction	Z	w	z^w
4	Rotation about the x_b-axis (roll)	K	p	ϕ
5	Rotation about the y_b-axis (pitch)	M	q	θ
6	Rotation about the z_b-axis (yaw)	N	r	ψ

Table 2.1: Notation of the forces, moments and states of a 6 DoF vehicle (SNAME (1950)[23]).

2.3 RMF Dynamics

The RMF is modeled as a rigid body, and there are mainly three forces and two moments acting on the RMF. These are the thrust force $\boldsymbol{F}_{T_i}$, the drag force $\boldsymbol{F}_{D_i}$, the rolling moment $\boldsymbol{M}_{R_i}$ and the moment of the drag $\boldsymbol{M}_{D_i}$, where i = {0, 1, 2, 3}, acting on each of the four rotors. In addition, there is the gravitational force $\boldsymbol{F}_G$ acting on the center of the gravity of the RMF. By applying Newton's law and Euler's equation it is possible to deduce the *Equations Of Motion* used to simulate the dynamics of the RMF as seen in Equation (2.6) [13, eq.8, eq.9]

$$\sum_{i=0}^{3}(\boldsymbol{R}(\underbrace{\boldsymbol{F}_{T_i}+\boldsymbol{F}_{D_i}}_{\boldsymbol{F}_i}))+\boldsymbol{F}_G=\boldsymbol{M}\dot{\boldsymbol{v}}_{wb}^{w} \tag{2.6a}$$

$$\sum_{i=0}^{3}\boldsymbol{M}_{R_i}+\boldsymbol{M}_{D_i}+\boldsymbol{F}_i\times\boldsymbol{r}_i=\boldsymbol{J}\dot{\boldsymbol{\omega}}_{wb}^{b}+\boldsymbol{\omega}_{wb}^{b}\times\boldsymbol{J}\boldsymbol{\omega}_{wb}^{b}, \tag{2.6b}$$

where $\boldsymbol{M}=\mathrm{m}\boldsymbol{I}_3\in\mathbb{R}^{3\times3}$ is the mass matrix, $\boldsymbol{J}\in\mathbb{R}^{3\times3}$ is the inertia matrix, $\boldsymbol{R}\in\mathbb{R}^{3\times3}$ denotes the rotational matrix from body to world frame and $\boldsymbol{r_i}$ depicts the vector from center of gravity of the RMF to the center of one rotor.

2.4 Actuators

Inspecting Equation (2.6), one notes that the RMF can only induce force in the z-direction relative to the body frame. This also implies that this is an *Underactuated System*, since there are fewer actuators than the degrees of freedom to be controlled. However, the four rotors are independently actuated. Thus, the RMF can maneuver by alternating the rotation speed of each rotor to yaw, pitch, roll and change altitude. It is also important to specify that during the simulation, complex nonlinear dynamics such as wind, vibrations and ground effect are neglected and will not affect the RMF. In an effort to control the RMF, the generalized control force $\boldsymbol{\tau}$ needs to be distributed to the 4 motors. This is done through control allocation as in Equation (2.7) [12, eq.11.2]

$$\boldsymbol{u}=\mathbf{B}^{\mathbf{T}}(\mathbf{B}\mathbf{B}^{\mathbf{T}})^{-\mathbf{1}}\boldsymbol{\tau}, \tag{2.7}$$

where $\boldsymbol{u}=[u_{b,1},\,u_{b,2},\,u_{b,3},\,u_{b,4}]^T$ are the four control inputs - the angular velocities of the motors, and $\boldsymbol{B}$ Equation (2.8) is the allocation matrix. The MAV simulator used in this project is built upon the ETH RotorS implementation [9], and consequently the same allocation matrix is used.

$$B=\begin{bmatrix} C_T & C_T & C_T & C_T \\ 0 & lC_T & 0 & -lC_T \\ -lC_T & 0 & lC_T & 0 \\ -C_TC_M & C_TC_M & -C_TC_M & C_TC_M \end{bmatrix} \tag{2.8}$$

where C_T is the rotor thrust constant, C_M is the rotor moment constant and l is the distance from the center of origin in the body frame to the rotors as seen in Figure 2.2, also referred to as the arm length. It is possible to control the actions of the RMF using low level motor torque commands, one for each of the four rotors. However, this is not a feasible command

structure for this problem. The provided rotors simulator already consists of low-level PID-controllers for controlling the position, attitude and thrust. For the purpose of applying autonomous behavior, it is necessary to abstract from the low-level control scheme and focus on the high-level autonomous problem. Henceforth, it would be more applicable to control the states of the robot, through position, velocity, or the acceleration, and let the low-level controller map those actions to appropriate rotors commands.

2.5 State Estimation

For the RMF to fully behave autonomously, the first step is to understand the current state of the environment. The RMF needs to interpret the sensor data provided by the onboard sensors and put them into a model of the world. In addition, it is necessary to understand which states that are relevant for the system and the solution to the problem. Subsequently, the first step is to do state estimation and derive the geometrical properties of the RMF in the world. As mentioned earlier, the position and the orientation of the RMF is described relative to the world frame. This is estimated by measuring the states with sensors. However, in the simulation these states are provided by plugin sensors. In the real world the quadcopter states can be provided by an *Inertial Measurement Unit* (IMU). The measurements from the IMU come at a very high rate but are also affected by noise and time-varying bias.

IMU

The IMU ADIS [7] has a 3-axis *Accelerometer* which measures linear acceleration and a 3-axis *Gyroscope* that measures angular velocity. The accelerometer can be expressed in {b} as seen in Equation (2.9)[12, eq.14.29]. The linear acceleration $\boldsymbol{a}^{w}_{wm_I}$ is given in the world frame referenced to the IMU measurement frame $\{m_I\}$.

$$\boldsymbol{f}^{b}_{imu} = \boldsymbol{R}(\boldsymbol{q}^{w}_{b})^{\top}(\boldsymbol{a}^{w}_{wm_I} - \boldsymbol{g}^{w}) + \boldsymbol{b}^{\boldsymbol{b}}_{\boldsymbol{acc}} + \boldsymbol{\omega}^{\boldsymbol{b}}_{\boldsymbol{acc}} \tag{2.9a}$$

$$\dot{\boldsymbol{b}}^{b}_{acc} = \boldsymbol{w}^{b}_{b,acc} \tag{2.9b}$$

where $\boldsymbol{f}^{b}_{imu}$ is the IMU specific force (non-gravitational force per unit mass), the gravity vector depicted as $\boldsymbol{g}^{n} = [0,\ 0,\ g]^{T}$, the accelerometer bias is the $\boldsymbol{b}^{b}_{acc}$, and the additive Gaussian white measurement and bias noise are the $\boldsymbol{w}^{b}_{acc}$ and $\boldsymbol{w}^{b}_{b,acc}$, respectively.

The angular velocity measurements $\boldsymbol{\omega}^{b}_{imu}$ also expressed in {b} from the 3-axis rate-gyroscope can be seen modeled in Equation (2.10)[12, eq.14.6 and eq 14.7]

$$\boldsymbol{\omega}_{imu}^{b} = \boldsymbol{\omega}_{wb}^{b} + \boldsymbol{b}_{gyro}^{b} + \boldsymbol{w}_{gyro}^{b} \tag{2.10a}$$

$$\dot{\boldsymbol{b}}_{gyro}^{b} = \boldsymbol{w}_{b,gyro}^{b} \tag{2.10b}$$

where the gyro bias is denoted as $\boldsymbol{w}_{\boldsymbol{gyro}}^{\boldsymbol{b}}$. The measurement noise $\boldsymbol{w}_{\boldsymbol{b,gyro}}^{\boldsymbol{b}}$ and bias noise $\boldsymbol{b}_{\boldsymbol{gyro}}^{\boldsymbol{b}}$ are modelled as zero-mean Gaussian white noise.

Chapter 3

Perception

This chapter presents the theoretical part of perception related to this thesis. An overview will be given followed by the fundamentals of 3D LiDAR and processing of the data. Section 3.6 gives an overview of the feature extraction process from LiDAR data.

3.1 An Overview

An autonomous system is usually said to be made up of four parts which interplay with each other in a close manner to achieve autonomous behavior. These 4 components, as seen in Figure 3.1, are *Sensing*, *Understating*, *Planning* and *Control*. In literature, the *Diagnostic* component does occur as well, which usually refers to autonomously identifying problems within the system itself. However, this was not considered in this thesis, but an example of such could be that the RMF identified that it had low battery life and had to fly back to a recharge station.

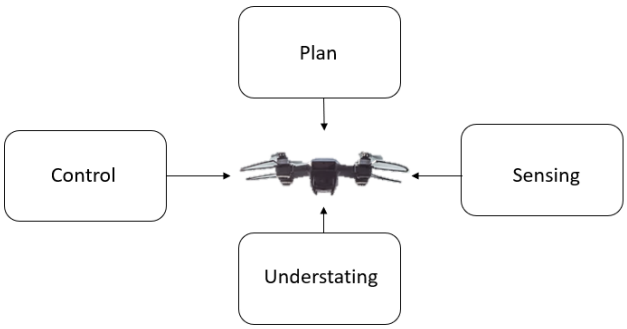


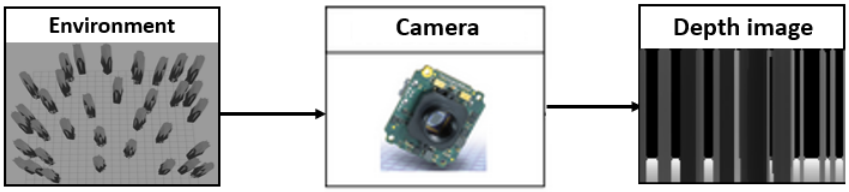
Figure 3.1: The four components in an autonomous system.

One of the use cases for sensing the environment is collision checking

for planning algorithms. In general, the available physical sensors produce measurements that are machine understandable. These measurements are then used and passed through various algorithms for understanding. This will be closer discussed in Section 4.5. When the agent is flying towards some goal, there is also a need to do planning within the environment. This will be closer discussed in Section 5.5. This eventually results in some control commands which are then used by the system, the RMF, to steer towards that goal. Consequently, the first step to produce such commands is to percept the environment. In general, objects are required to be detected in three-dimensional (3D) space. The reason for this is that planning happens in 3D space as well. Hence, the measurements that are the inputs to the navigation algorithm are required to be represented in 3D.

3.2 Different Sensors

There are many ways to produce measurements that are in 3D space, ranging from *Radar*, *Stereo Camera*, *LiDAR* or some fusion where one uses all or some of these different sensors. The quadcopter is equipped with a stereo camera, which can for instance produce a depth map of the environment as seen in Figure 3.2a, and a LiDAR sensor that can register the environment through a large cloud of points as seen in Figure 3.2b.



(a) An environment perceived by a camera generating a depth image.

(b) An environment perceived by a LiDAR registering a point cloud.

Figure 3.2: The same environment sensed with a camera and LiDAR.

It is therefore only natural to compare those two sensors. The advantages of stereo cameras:

- Low-cost.

- High resolution and good at detecting semantics.
- Passive sensing using a low amount of power.
- Lightweight.
- Good at measuring depth in the environment.

However, stereo cameras suffer from:

- Not working well in low-light environments.
- Difficult to use in foggy and dusty environments.
- Short range.

The advantages of LiDAR:

- Produces very accurate, dense distance measurements.
- Works well in low-light environments.
- Can have a very long range.

The weaknesses of LiDAR:

- It is an active sensor and consumes more power compared to cameras.
- Difficult to use in foggy and dusty environments.
- Heavy compared to cameras.
- Expensive.

Both options could suit our problem, but for the benefit of perimetric perception we have for now opted to use a LiDAR.

3.3 Fundamentals of 3D LiDAR

Light Detection and Ranging (LiDAR) is a light ranging sensor and uses time of flight measurement for a laser pulse that is reflected off an object to determine the distance to the said object. It usually scans the environment $[0°, 360°]$ in the xy-plane and due to limitations in most LiDAR sensors in the range $[15°, 165°]$ in the xz-plane. Thus, the environment can be registered through a 3D *Point Cloud* as seen in Figure 3.2b. Each 3D scan from the sensor consists of a time stamp, the reference frame the points are given in, number of points in the given scan, the Cartesian coordinates $(x_{b_i}, y_{b_i}, z_{b_i})$ of each point, and the RGB values of each point. In this case the points should

be tied to the moving RMF body-fixed frame $\{b\}$ with origin o_b. The data received from different LiDAR sensors may vary, and one can for instance receive additional information such as the intensity of the points and the scan angle. Since the measurements are tied to the reflectivity of an object, it can create problems if the RMF is in an environment where targets are absorbing the pulse, or the pulse may return in some way fractionally. In addition, the pulse can be obstructed in a hazardous environment if there is a lot of smoke or rain. What makes LiDAR sensor a very powerful instrument is that it receives data at extremely high frequencies, and one gets fine granularity in the distance measurements. The commercially available LiDAR sensors are also safe for the eyes, but at the cost of not being so powerful which can affect the performance for long distance measurements. Nevertheless, this will not affect this problem much, as one only needs to be very aware of the distances to many obstacles that lie in proximity within a small, confined environment. LiDAR has also been extensively used as a perception sensor in autonomous vehicles and has proven itself as a reliable and stable instrument.

3.4 Preprocessing LiDAR Data

A big problem with sensors used for perception is that there is often too much information within the data that is received, and it can be computational expensive to make use of all the information available. This is especially the case for micro-sized aerial robots with limited payload capabilities. In Figure 3.2b one can see roughly 12800 points from one single scan. This implies that there is a need for doing preprocessing of the data for the purpose of speeding up computation time and give feasible data to the reinforcement learning algorithm. In general, from a downstream perspective, one wants to take in the minimum amount of information which will allow to produce the satisfactory results. Any more than that and there is a waste of CPU cycles and potentially adding noise, which can affect the results in a negative way. There are some different operations that can be used to reduce the complexity, one of which is removing useless data which will not affect the result. Problematic or bad data can also be removed, but this could negatively affect the result. Redundant data could also be removed, which can make the problem less computational expensive. However, with the object of having a robust solution, there should not be done more preprocessing than necessary to produce consistent outputs.

There already exist a lot of different methods for preprocessing LiDAR data. One is *Ranged Based Filtering*, which can be thought of as an example of removing useless or bad data. An example of this are points that are very far away from the sensor are useless because they only provide noise, and no sensible information. By contrast, points that are very close to the sensor may negatively affect the system by providing false positive detections. How-

ever, since the RMF is supposed to operate in confined environments with a lot of obstacles nearby, one would be more reluctant to filter out those kinds of points. One would rather rely on having a good LiDAR sensor that works well near obstacles. Another example of removing bad data is *Angle Based Filtering*. There can be regions in the LiDARs field of view which maybe do not work so well for various reasons. One example of such is that when having multiple LiDAR sensors, the lasers from one LiDAR can cause false positive detections in another LiDAR. Downsampling is usually done in order to remove redundant data. Normally, one can reasonably well represent an object with fewer points. For example, the floor can be represented by few points falling onto a plane rather than a lot of points. The most common approach for downsampling points is by *Voxel Grid* techniques. This is done by subdividing the space into a series of boxes or cubes, and every point that falls into a particular box is represented by a single point that lies in the center of that cube or by approximating the center of all the points within that box. Random sampling can also be done but is generally not recommended because it introduces bias. In addition, it is random, which can especially complicate things in complex environments. Lastly, downsampling can also be done using more complicated techniques by utilizing *Convolutional Neural Networks* or *Variational Autoencoders*. However, these are non-trivial operations since the 3D point cloud or voxels needs to be processed through those networks and largely correspond to open research questions on their own. In addition, even though these are valid techniques, it does to some extent complicate the solution regarding how to interpret the output and the latent space.

Fusing multiple point clouds together into a single consistent representation may also be considered. The reason for doing so is to have a single and clean representation which can also be used by other algorithms within the system. This can also be simpler for interpretation when doing analysis. All that needs to be done is a static transform to put all the points into a common coordinate frame.

3.5 Environmental Noise Filtering

One of the challenges related to navigation in unknown, hazardous conditions is the problem with perceptual degradation. As previously mentioned, LiDAR can be limited by adverse conditions such as mist, fog, and dust. There already exist many intricate and sufficient ways to do noise removal, and one of the simpler and fastest ways of denoising a point cloud is by considering the intensity or the density of the points in the cloud. This will be further discussed in Section 6.4.1.

3.6 Point Cloud Feature Extraction

After preprocessing the data, the next step would be *Feature Selection*. Feature selection is the process of selecting a subset of relevant features that will be most beneficial to the solution. For instance, in an urban environment, line features will be the most dominant feature, as they can be found in common structures. The point cloud consists of a lot of valuable information, and one could most certainly learn object representations in some way by applying a learning algorithm. However, given the geometrical properties of each point in the point cloud, one could also simply sample the closest points and feed the distances, d_i, to the reinforcement learning algorithm. In this way it would be easy to use odometry states with the LiDAR states to create behavior that emphasizes on avoiding occupied space. This will be further discussed in Section 6.5.3. Such an intuitive solution allows for more easily interpretable results from training the RMF. This translates to being easier to implement and understand, in contrast to doing feature extraction by for example a neural net. In addition, by carefully selecting the most relevant distances in a simple manner, the computational overhead is reduced because the computation is minimal. This is very important to keep in mind, as the algorithm is bounded by the computer on the RMF. Hence, the proposed method, further discussed in Section 6.4.1, is more broadly applicable. It provides a solid foundation, which then permits options for adding additional algorithms, which can improve the overall solution in the long run.

3.6.1 Point Cloud Representation

Given the structure of the point cloud, one of the most common ways of expressing the location of a point in 3D is using *Spherical Coordinates* as seen in Figure 3.3.

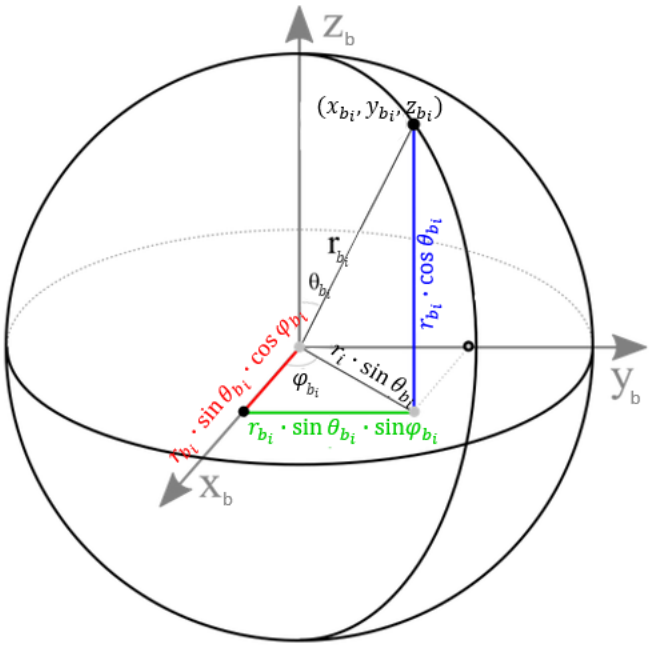


Figure 3.3: A point represented on a sphere. (Source of figure: [1])

This is done by expressing the point by $(r_{b_i}, \theta_{b_i}, \varphi_{b_i})$ in the body-frame. r_{b_i} is the direct distance from the origin out to the point. φ_{b_i} depicts the angle from the positive x-axis in the xy-plane. θ_{b_i} is the angle from the positive z-axis down to the line segment from the origin to the point. The relations between $(r_i, \theta_{b_i}, \varphi_{b_i})$ and $(x_{b_i}, y_{b_i}, z_{b_i})$ can be seen in Equation (3.1)

$$r_{b_i} = \sqrt{x_{b_i}^2 + y_{b_i}^2 + z_{b_i}^2}, \qquad r_{b_i} \geq 0 \tag{3.1a}$$

$$\theta_{b_i} = arctan\left(\frac{\sqrt{x_{b_i}^2 + y_{b_i}^2}}{z_{b_i}}\right), \quad \theta_{b_i} \in [0, \pi] \tag{3.1b}$$

$$\varphi_{b_i} = arctan\left(\frac{y_{b_i}}{z_{b_i}}\right), \qquad \varphi_{b_i} \in [0, 2\pi]. \tag{3.1c}$$

Chapter 4

Reinforcement Learning

This chapter provides the necessary theory to understand the learning algorithm used in this project. The motivation for using deep reinforcement learning will be given. In addition, key concepts and the algorithm that was used, the deep deterministic policy gradient algorithm, will be covered.

4.1 Motivation Behind Deep Reinforcement Learning

Before explaining the core concepts of the reinforcement learning module, it is important to motivate why to use deep reinforcement learning as the baseline of the solution in the first place. There already exists a lot of theory and solutions from classical control that has been rigorously tested and applied to real life problems both in the industry and in the academia. Traditionally, robotics has relied on heavily engineered features as representations. These features might leverage expensive sensors that tell exactly where things are and give the exact location of obstacles. However, one would often need to specify and categorize those features for every specific object or situation. Then, when the robot moves to the next situation, it would have to do it all over again. As the problem gets more complex, the state space becomes sufficiently large, and machine learning would need to be applied because traditional programming techniques, even traditional dynamic programming techniques, are insufficient to solve these problems efficiently. So, one may rather learn representations from very high dimensional inputs like cameras or LiDAR and give the system, the RMF, a perceptional understanding of the environment. As one moves toward the high dimensional control domains, adding such features is not necessarily trivial using traditional methods. Hence, deep reinforcement learning is easier to scale up in terms of state representations. Solving complex navigation problems using classical tools can therefore be impossible, and by using the concept of learning one is also closing the gap between how humans solve a problem and how robots

can solve a problem. In addition, in this case one cannot assume that all the data is present, and since this is a highly nonlinear problem, it may be very hard or impossible to apply any type of nonlinear control for guidance and obstacle avoidance.

4.2 Reinforcement Learning - Key Concepts

Reinforcement Learning (RL) is a branch of machine learning. It can be applied in learning control strategies that can be used to interact with a complex environment. In RL one models a decision-making system as an interaction between an agent and an environment. The environment is usually depicted as a *Markov Decision Process* (MDP). The *Agent*, or the controller, makes decisions in form of *Actions* a and the environment responds to those decisions with observations which are called *States* s and *Rewards* r. Hence, this creates a fully closed loop system as seen in Figure 4.1.

Figure 4.1: A flow diagram describing the general framework in reinforcement learning.

A state is a concrete and immediate situation in which the agent perceives, for instance odometry data, an image feed, or a point cloud. From a given state, an agent will send out output in the form of actions to the environment. The environment will respond with the agent's new state s_{t+1}, which resulted on acting on that previous state, as well as any rewards that may be collected or penalized by reaching that particular state. One of the most fundamental quantities in RL is the idea of learning how to interact with the environment through rewards r_t. The scalar feedback signal r_t indicates how well the agent is doing at a given time step t. However, rewards can be either immediate or delayed. So, even if rewards effectively are evaluating the agent's actions, the rewards might be given a long time into the future. For example, the RMF might take many different actions but only be given a reward if it reaches its intendent goal.

The overall job of the agent will be to maximize the cumulative reward,

by finding certain actions a_t and get as much reward as possible. This process is repeated multiple times over the course of an episode. RL can be finite horizon, which means that this is repeated a fixed number of times, or infinite horizon, which means that the decision-making cycle goes on forever. However, it is very common to consider not just the total return but also the discounted sum of rewards as in Equation (4.1).

$$R_t = \sum_{i=t}^{\infty} \gamma^i r_i = \gamma^t r_t + \gamma^{t+1} r_{t+1} + \cdots + \gamma^{t+n} r_{t+n}, \quad 0 < \gamma < 1. \tag{4.1}$$

The discounting factor is depicted as γ and multiplied by the future rewards that are discovered by the agent. This is done in order to dampen those rewards effect on the agent's choice of action. By doing so, future rewards are made less important than immediate rewards, which enforces a somewhat short-term learning in the agent. The discount factor is typically between 0 and 1.

Another important property related to RL and the reward function is the Q-function. The Q-function is a function that takes as input the current state s_t that the agent is in and the action a_t that the agent takes in that state. Then it returns the expected total future reward that the agent can receive after that point. Thus, the Q-function can be written as Equation (4.2)

$$Q(s_t, a_t) = \mathbb{E}[R_t | s_t, a_t]. \tag{4.2}$$

This can be used to determine, given the state that the agent is currently in, what is the best action to take by simply taking the action that results in the highest expected total return. The actions which the agent produces is based on a strategy which is referred to as a *Policy*, denoted as $\pi(a|s)$. This policy function takes as input the state and it tells what kind of action that should be executed. This can be written as in Equation (4.3) by finding the $argmax$ of the Q-function over all possible actions that one can take at the state. Hence, a greedy policy that maximizes the future reward is chosen.

$$\pi(a_t|s_t) = arg\max_{\theta} Q(s_t, a_t). \tag{4.3}$$

4.2.1 Stochastic and Deterministic Policies

Policies can also be categorized as either *Deterministic Policies* or *Stochastic Policies*. A deterministic policy maps the state to action without any uncertainty, whereas a stochastic policy gives the probability distribution over actions. One hopes that within that set of policies, there will be a good one. Stochastic policies tend to smooth out the optimization problem in the sense of having a distribution of the actions in every state. The distribution can be slightly shifted, and that will only slightly shift the expected sum of rewards.

Having a deterministic policy changing an action in a state can have a very significant change on the outcome. Hence, it is not a very smooth solution. This will be elaborated in Section 4.5.

4.3 Deep Reinforcement Learning Algorithms

Reinforcement learning algorithms are usually divided into two categories, one of which will try to learn the Q-function. This is done by *Value Learning* algorithms. However, it can sometimes be very challenging to understand or intuitively guess what the Q-value is for a given state-action pair, even for humans. They are therefore usually approximated and modeled using deep neural networks, often referred to as *Deep Q Networks* (DQN). The other category of RL algorithms is called *Policy Learning* algorithms because they will try to directly learn the policy instead of using a Q-function to infer the policy. This is a much more direct way of solving the problem, but finding such policy is not necessarily trivial.

There are many advantages of using Q-learning, and there have previously been seen some amazing results using this approach at Atari games [22]. Nevertheless, there are some very important downsides to this technique. The first of which is that it can only handle action spaces which are discrete, and, secondly, it can only really handle them when the action space is small. This is a huge drawback for the autonomous flying vehicle that wants to predict where to go in a small, confined environment with a lot of obstacles. The RMF cannot be limited to certain directions that may, in worst case, not be available. RL is used to learn a continuous action space that is not discretized into bins, but can take any real number within some bound that the actuators can execute, $a = [a_{Low}, a_{High}]$.

The flexibility of Q-learning is also somewhat limited. It is not able to learn policies that can be stochastic and that cannot change according to some unseen probability distribution. This means that they are deterministically computed from the Q-function through the maximum formulation. It will always pick the action that maximally elevates the expected return. Thus, it cannot really learn from these stochastic policies. On the other hand, policy gradient methods address all of these issues.

4.4 Policy Gradient Methods

The key idea of policy learning is to instead of predicting the Q-values, the policy $\pi_{\boldsymbol{\theta}}(a|s)$ will be directly optimized. The control policy is parameterized by the parameter vector $\boldsymbol{\theta} \in \mathbb{R}^d$, depicting the weights in the neural net. The way to think of the policy would be some neural net that processes state information through a few layers and outputs a distribution over possible actions that the agent might want to take. In order to act, the agent samples

from that distribution, observes what happens, processes and repeats as seen earlier in Figure 4.1. This is a lot simpler, since it means that the actions can now be directly sampled from the policy function that the agent can learn. This is also one of the big advantages of using policy-based RL, since it is very easy to extend too high dimensional or even continuous action spaces because the policy is parameterized. Different representations can simply be plugged in which then happens to capture continuous actions. Another advantage of using a policy-based approach is that it has good convergence properties because it is usually only dependent on stochastic gradient decent during optimization. Having a nonlinear function, one will end up in some local optima fairly reliably. This in turns means that policy learning algorithms are quite susceptible to local optima, especially with nonlinear function approximation. Another disadvantage with policy learning is that it is ignoring a lot of information. This means that it might not be making the most efficient use of all the information that is in the data that comes at us. In some sense if the agent wants to be efficient in learning, it should learn all that it can from all the available data, in terms of data efficiency.

4.4.1 Off- and On-Policy Learning

One also needs to differentiate between *Off- and On-Policy Methods*. The main difference lies in how the data is collected. The *Behavior Policy* is used to generate samples from which one can learn and update the Q-values for different actions. The *Target Policy* is used to control the system during exploitation and is essentially the policy that the agent wants to learn. If the target policy is different from the behavior policy, one has off-policy learning and vice versa. This is a very important difference which will be discussed on later in Section 4.5.

4.4.2 Policy Objective Function

To perform policy optimization, the first step is to define an objective $J(\boldsymbol{\theta})$. The RL problem can be treated as a finite horizon problem and as stated earlier, the policy aims to produce actions that give high expected rewards. Thus, the objective can be formulated using the immediate rewards as in Equation (4.4) [35, eq. 13.4]

$$J(\boldsymbol{\theta}) = \mathbb{E}_{\pi_{\boldsymbol{\theta}}}[r(\tau)], \tag{4.4}$$

where the expectation is over states an action and τ denotes the finite state-action sequence s_0, a_0, ..., s_N, a_N. It is also assumed that there is discounting present in this case.

Continuing, with the object of computing the gradient of the expectation in Equation (4.4) it is assumed that the policy is differentiable almost

everywhere. Data samples are used to compute the gradients, but the reward cannot be directly sampled followed by taking the gradient because the samples are just numbers. It will not depend on our parameter $\boldsymbol{\theta}$. Instead, one uses the identity derived from [2, sec. 4.11]. The gradient of this expectation in Equation (4.5) equals the expectation of a gradient, which is the reward times the gradient of the logarithm of the policy. This is helpful, since this gives an expected gradient that can be sampled and then be used in an algorithm.

$$\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) = \mathbb{E}_{\pi_{\boldsymbol{\theta}}}\left[\sum_{t=0}^{N} \nabla_{\boldsymbol{\theta}} log\pi(a_t|s_t) Q_{\pi_{\boldsymbol{\theta}}}(s_t, a_t)\right]. \tag{4.5}$$

4.4.3 Stochastic Gradient Ascent

Now, to optimize the objective $J(\boldsymbol{\theta})$, $\boldsymbol{\theta}$ needs to be found such that it maximizes Equation (4.4). There are of course many ways to do this without using gradients. One can for example use genetic algorithms or evolutionary strategies. For this problem, stochastic gradient ascent will be used, which turns out to be often quite efficient. It is also simple to use with deep neural networks, as one just backpropagate the gradients. Policy gradient algorithms look for a local maximum by locally looking in which the gradient of the policy is ascending. Hence, there is some update to the parameters $\boldsymbol{\theta}$ giving Equation (4.6)

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}), \tag{4.6}$$

where α is the learning rate. However, one would usually prefer a more advanced optimizer, such as RMSprop or Adam optimizer [17]. An important note is that the policy update should be sufficiently small, and one might face problems with policy updates if an update is too big. This can be solved by adding constraints to the optimization problem, such as trust regions [24]. This means that in most cases, the algorithms that employ such trust regions choose a step length α that may guarantee monotonic improvement. This forms the basis of many known algorithms in RL, such as *Proximal Policy Optimization*[32] (PPO) and *Trust Region Policy Optimization*[33] (TRPO).

4.4.4 The Baseline

In practice, Equation (4.5) requires an impractical number of samples to get a good estimate. In addition, using Equation (4.5) might introduce a high variance which in turn slows down the learning. A way to solve this is by introducing a baseline, $b(s)$. This baseline function usually depends on the states and cannot depend on the actions. The variance can be reduced for instance by using a baseline that subtracts the average reward across all actions as seen in Equation (4.7).

$$\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) = \mathbb{E}_{\pi_{\boldsymbol{\theta}}}\Big[\sum_{t=0}^{N} \nabla_{\boldsymbol{\theta}} log\pi(a_t|s_t)(Q_{\pi_{\boldsymbol{\theta}}}(s_t, a_t) - b(s_t))\Big]. \tag{4.7}$$

Thus, if an action has a higher-than-expected reward, one increases the preference of that action, and if it is lower than expected the baseline will decrease it. Thus, by reducing the variance, the learning speed is increased.

4.5 Deep Deterministic Policy Gradient

This section aims to cover the fundamental concepts regarding the implementation of the *Deep Deterministic Policy Gradient* (DDPG) algorithm which also forms the basis of the RL portion of the solution. The theory behind this is heavily inspired by Bengio et al.[19] and Section 4.4 will be used as the foundation to explain this method. For the actual implementation of the algorithm, the DDPG baseline provided by *OpenAI Gym* [25] was used. This is a popular platform for RL, and since everything is open source it will make it easier to benchmark RL ideas.

4.5.1 DDPG - Key Elements

For simple problems with discrete action spaces, one can get away with having a single network where the lower layer is learning the features of the environment, and the upper layer splits off into outputting the critic evaluation as well as the output for the actor network, similarly as in the DQN algorithm. However, as mentioned earlier, this does not apply to this problem as the action space has to be continuous. To solve this the DDPG algorithm is applied. The DDPG algorithm is a type of *Actor-critic Method*, meaning that there are two distinct networks. One is the *Actor* and the other one is the *Critic*. In addition, there are two target networks, one *Target Actor* and one *Target Critic*. Conceptually, the actor, denoted by $\pi_{\boldsymbol{\theta}}(a_t|s_t)$, where $\boldsymbol{\theta}$ is depicting the weights in the neural net, will decide upon which action to take. This is done by using the current state as input $\{s_1, s_2, \ldots, s_n\}$ and outputs action value $\{a_1, a_2, \ldots, a_m\}$. These are continuous numbers that correspond to the direct input from the environment as in a deterministic policy. Similarly, the critic network denoted by $Q(s_t, a_t|\boldsymbol{\phi})$, where $\boldsymbol{\phi}$ denotes the weights and biases of the neural net, will evaluate state-action pairs by taking the current state and the corresponding action $\{s_1, s_2, \ldots, s_n, a_1, a_2, \ldots, a_m\}$, given from the actor, and output a Q-value. In other words, the critic is performing a policy evaluation for the current policy.

4.5.2 DDPG and Off-Policy learning

The DDPG algorithm is leveraging off-policy learning because it uses a *Replay Buffer*. This means that it does as few assumptions as possible concern-

ing the behavior policy, and in this case the behavior policy is stochastic. The target policy should be an approximation to the optimal policy, and in this algorithm it is deterministic. Off-policy learning is preferable because it can reuse old data that comes from a previous version of the policy, for instance from a replay buffer. This is beneficial since it considerably improves the sample efficiency, as it can reuse the same data several times during training. Another benefit with off-policy control is that it has more freedom for exploration. It is possible to differentiate between the policy that is used for exploration and the one that is supposed to be learned. Lastly, being trained off-policy means that it is possible to use the replay buffer, used for learning a particular policy, and transfer this knowledge to different policies used for different tasks. These tasks may require some past experience, and this is often referred to as transfer learning. This will be discussed in Section 5.3.1. Since off-policy learning is being done, the replay buffer may effectively be quite large, and it can store a lot of past information. Using the replay buffer to transfer experience in such way also improves the sample efficiency.

4.5.3 Replay Buffer

In the DDPG algorithm the replay buffer is a finite sized memory bank. When the agent transitions in the environment using some behavior policy, one would sample from it and store the tuple (s_t, a_t, r_t, s_{t+1}, d) in the replay buffer. As information is accumulated over time, the oldest samples are thrown away. d depicts if s_{t+1} is a terminal state. The inspiration for having such a buffer came from the DQN algorithm, and the intention is to reduce correlations between each update, and by doing so reducing the variance of the updates. Aiming to actually train the actor and the critic the algorithm samples a mini-batch uniformly and updates the weights of the deep neural networks. The samples do not correspond to an agent trajectory, but instead one is simply jumping from state to state without considering the next state. Using uniform sampling is beneficial for policy learning because it means that one makes no assumption on the way as the behavior policy collects the samples. There is no exploration issue and there is no bias toward good samples. However, this way of training the RMF is only possible in simulations which will be discussed in Section 5.2.

4.5.4 The Actor Network

The update for the actor network is done by minimizing the loss function as seen in Equation (4.8)

$$L_{actor} = -\frac{1}{N}\sum_{i=1} Q_{\boldsymbol{\phi}}(s_i, \pi_{\boldsymbol{\theta}}(a_i|s_i)) \tag{4.8}$$

where N is the size of the mini-batch. This is somewhat different from the policy gradient in the DDPG paper[19], but since Tensorflow will be used one wants to put the negation of the deterministic policy gradient as Tensorflow optimizes expected loss. One should also note that the critic parameters are treated as constants. Intuitively, the agent should learn a policy that takes the actions that minimizes the loss function. In practice, states will be randomly sampled from the replay buffer and then the actor network will be used to determine what actions it believes it should take based on those states. Similarly, $\boldsymbol{\theta}$ also needs to be updated as discussed in Section 4.4.3.

4.5.5 The Critic Network

The next step is to plug those actions from the actor into the critic network along with the states that are sampled from the buffer. The critic is updated by minimizing the loss function given by Equation (4.9) [19].

$$L_{critic} = \frac{1}{N}\sum_{i}(y_i - (Q_{\boldsymbol{\phi}}(s_i, a_i))^2, \tag{4.9a}$$

$$y_i = r_i + \gamma(1-d)Q_{\boldsymbol{\phi}_{next}}(s_{i+1}, \pi_{\boldsymbol{\theta}_{next}}(a_{i+1}|s_{i+1})), \tag{4.9b}$$

where N is the size of the mini-batch, r_i is the reward from the current time step which is sampled from the buffer, γ is the discount factor, and d indicates if s_{i+1} is a terminal state. Equation (4.9) is often referred to as the *Mean Squared Bellman Error*, where the error between the target y_i and the Q-value is computed for the current state and action. Similarly, as was done with the actor network parameters, one would likewise need to update the critic parameters $\boldsymbol{\phi}$ as discussed in Section 4.4.3. In the beginning of the training, the behavior policy will be random because the target policy is random. When learning the critic goes well, the target policy will improve, then the behavior policy will be biased towards better and better samples because the samples in the replay buffer will be improved. This loop continues as one proceeds with the training. However, due to the problem being a more general policy search problem where the action space is continuous, there is no guarantee that an optimal policy will always converge as discussed in [21].

4.5.6 Noise Based Exploration

The agent starts out knowing nothing about its environment. As it progressively finds how certain states transition from one into another and how certain actions affect those states it builds a model of the environment. However, such model will never be fully accurate. The degree to which the agent takes off-optimal actions is known as the *Explore-exploit Dilemma* which is

present in all RL problems. Taking an off-optimal action is referred to as *Exploration*, and taking the optimal action is called *Exploitation*. In the DDPG algorithm, this will be solved by taking the output of the actor network and apply some noise as seen in Equation (4.10)[19]. In this implementation Gaussian noise $\mathcal{N}(0, 0.1^2)$ was used as it was sufficient enough.

$$\pi'_\theta(s_t) = \pi_\theta(s_t) + \mathcal{N} \tag{4.10}$$

4.5.7 Batch Normalization

If the features given to the networks are on different scales such as the distance measurement and the odometry states, then associated weight parameters will end up taking very different values and the optimization space can end up rather elongated. This can affect stability of the algorithm during the optimization process. The DDPG algorithm solves with the batch normalization. The batchnorm layer will maintain a moving average of the mean and the variance to ensure effective and stable learning.

4.5.8 Target Networks

The DDPG algorithm is also making use of target networks. The reason for this is due to the stability issues in Equation (4.9) caused by the rapidly changing value, y_i. At each time step the weights are getting updated and the evaluation of similar states changes rapidly over the course of the simulation, causing the learning to be unstable. This can be solved by having a target actor network, $\pi'_{\boldsymbol{\theta'}}$, where $\boldsymbol{\theta'}$ depicts the parameters of the target actor network, and a target critic network, $Q'_{\boldsymbol{\phi}}$, where $\boldsymbol{\phi'}$ depicts the parameters of the target critic network. One can slow down these updates through a soft copy every once in a while, which is determined by some time hyperparameter. This soft update can be seen in *Equation* (4.11) [19].

$$\theta' \leftarrow (1 - \tau)\theta' + \tau\theta, \tag{4.11a}$$
$$\phi' \leftarrow (1 - \tau)\phi' + \tau\phi, \tag{4.11b}$$

where $\tau \ll 0$ is a hyperparameter which should be set sufficiently small in order to slowly track the learned networks, which in turn improves stability of the learning. To summarize, the algorithm will randomly sample states, new states, actions, and rewards and then the target actor network will be used to determine the actions for the new states. Then those actions will be plugged into the target critic network to get the y'_i which one wants to shift the estimates towards for the critic. After that, the states and actions are plugged into the critic network. These are the actions the agent actually took that was sampled from the buffer. The full pseudocode of the DDPG baseline provided by OpenAI Gym [25] can be seen in Appendix A.1.

Chapter 5

End-to-End Learning

This chapter focuses on the theory related to how to learn to combine the state information provided by the sensors and how this can be passed through the DDPG algorithm to create a feasible end-to-end policy for collision avoidance. More specifically, it will discuss how the entire system should be trained in simulation. It will focus on learning in the sense of machine learning and how different strategies can be used to train the RMF. Lastly, one shall examine a state-of-the-art motion planner.

5.1 Traditional and End-To-End Control

The traditional approach to control quadcopters typically relies on the separation of state estimation, planning and control. One usually notices in the traditional architecture that these modules are typically designed individually rather than conjointly. In addition, these models run consecutively rather than concurrently while it has already been discussed that perception and actions are to some degree coupled. Furthermore, this modularity introduces latency in the system. One also knows that each module is sensitive to errors in their internal models of the environment, the sensors, and the actuators. This means that small errors can eventually accumulate, which in turn will affect negatively on the performance in each of the other modules. Lastly, the traditional architecture requires extensive tuning, especially for the controller gains. These are the reasons why one instead wants to directly learn an end-to-end controller in the form of a neural network. So, what one has is a controller that maps sensory inputs from LiDAR and IMU directly to control commands in the form of thrust as illustrated in Figure 5.1.

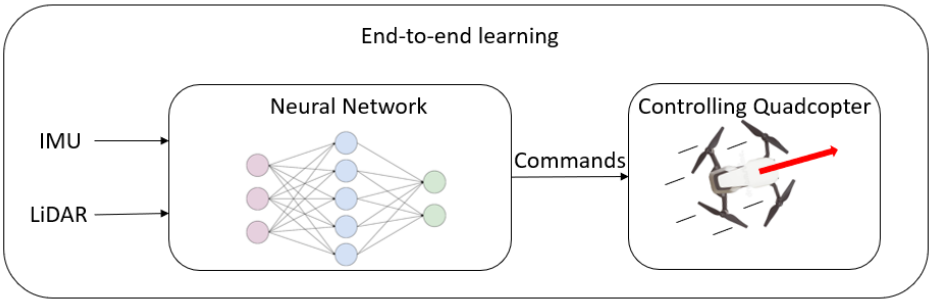


Figure 5.1: Overview of the end-to-end learning architecture.

5.2 Learning in Simulation

In this project, the networks were exclusively trained in simulation. The reasons for training in simulation is that it is fast, cheap, and safe. Furthermore, it does not require any human interference and it also allows the collection of almost unlimited amount of data within a limited time. The key issue when the agent is trained in simulation is that there is no guarantee that the policy would also transfer to the real world. This is because simulation and reality are often very different, and this is often referred to as the *Sim-To-Real Gap*. The simulation is often a simplified version of reality and the real world is something that one cannot fully control in terms of noisy data representations, an unpredictable environment, and other unforeseen events. However, there are several key steps one can take to reduce this sim-to-real gap. The first thing to look at is the data that is used to train the RMF. First, one could use raw sensor data or intermediate representations. In the paper by Zhou et al. (2019) [42] they show that the current state-of-the-art end-to-end approaches, that uses intermediate representations, outperforms approaches that directly map raw image pixels to control actions. A neural network is trained faster, achieves higher task performance, and even generalizes better to previously unseen environments when intermediate representation is used. This can for instance be segmentation masks or depth maps. In addition, data with added noise can be added during training in simulation. Another step one can take to reduce the sim-to-real gap is to match the observation models in simulation with the observation models in the real world. Hence, one should aim to make the simulation photorealistic. This is not necessarily very simple, and given the scope of this thesis, this was not something that one necessarily tried to achieve. Nevertheless, there do exist open source environments where one can get fairly good cave models, for instance provided by the DARPA SubT Challenge [3]. In order to have consistent improvements in the training process, diverse models are needed. It is also necessary to create scenarios of increasing difficulty which the RMF

can learn from, which will be further elaborated in Section 5.3.1. From a practical standpoint, this may be very challenging to implement. However, the environments were made as diverse and dense as possible to better replicate a chaotic and complex environment such as caves. This will be further discussed in Section 6.7.1.

5.3 Learning Strategies

Independent identically distributed data is something that machine learning is founded on and used throughout when training different models, as done in the DDPG algorithm. This is different from human learning, which focuses more on relatively prolonged inspection and consideration of a single subject at a time. If one is to learn about a certain subject and aspire to become an expert in a certain field, one works on it progressively, step by step. One does not sample different facts, but rather learn all about it in the context of all the different aspects of a larger subject. Applying this idea to machine learning, it is possible to learn multiple subjects, but each one deeply and with consideration. Doing so step by step, one might have a fast and reliable way of doing machine learning.

5.3.1 Curriculum Learning

One of the most powerful ideas in deep learning is the concept of *Transfer Learning*, where one can take knowledge that the neural net has learned from one task and apply that knowledge to a separate task. Specifically, for this problem one can use *Curriculum-based Transfer Learning*. When doing standardized training with the DDPG algorithm, mini-batches are randomly sampled from the buffer and fed to the actor and the critic. The weights are updated by SGD at each training iteration. With curriculum learning the networks are trained in a similar way, but the data is organized according to some level of difficulty and the networks are exposed for more intricate data sets as the agent progresses and converges to better results. This is very similar to human learning, where one starts to learn simple ideas, and progresses by increasing the difficulty. This way of learning reduces the complexity and training time compared to learning the final objective from scratch in one training phase. One did this by creating, evaluating, and improving different environments and selecting the environments where the agent had the highest positive learning curve and ended up gaining the most rewards. Doing so with many different environments, it is possible to create a curriculum for the RMF. When the RMF masters one environment, it moves on to a more challenging environment and continues doing so until it has successfully learned its end objective, being point-to-point navigation while avoiding obstacles in confined complex environments. A challenge with this approach is the problem of *Catastrophic Forgetting*. Neural networks tend

to forget how to solve certain tasks if they are not exposed to them. This can be solved by continuously mixing in easier tasks while learning the hard ones.

In practice, when doing curriculum learning, one starts with randomly initialized weights in the networks and train all of the parameters within by SGD. During the first phase of training, the agent is exposed to very simple environments where the main task is to learn to navigate to a waypoint. When having successfully trained the neural net to achieve this task, one changes the environment to a slightly more challenging one and continues with training while keeping the weights and biases in the net. The task of reaching a waypoint is still present, but the agent should now start to learn to avoid very simple obstacles. The process of training in increasingly challenging environments is finished when the behavior converges and there are no more improvements in the policy.

5.3.2 Imitation Learning

Another popular area of interest within deep reinforcement learning is *Imitation Learning*. In imitation learning, an expert provides demonstrations to replace the random neural network initializations with a better initial policy. In this way one can provide some domain knowledge. Since the DDPG algorithm is trained off-policy, imitation learning can be done by storing the state-action pairs that the expert is performing in the replay buffer. Then the policy can be trained using the data from the replay buffer, which can be much more efficient than discovering everything from scratch. Conceptually, this makes a lot of sense. Most of what humans learn is rarely learned from scratch. One is almost always given examples. This idea was founded on that leveraging imitation learning would allow to build scalable deep reinforcement learning solutions that could learn within a reasonable amount of time. Nevertheless, such an approach is limited to only finding provided solutions, which may not necessarily be innovative or very different from the provided performance. This can be a problem if the agent approaches a situation which is very different from what was experienced during training.

5.4 Auxiliary Rewards

Another dimension of end-to-end learning is the notion of reward functions. As mentioned in Chapter 4, rewards provide a feedback signal of how well the agent is performing based on the actions it took. The entire goal of the agent is to optimize its policy to receive as much reward as possible. There are two types of reward functions, the *Sparse Function*, which is easy to specify but hard to solve, and the *Non-sparse Function*, which is hard to specify but easier to solve. The reward works as the only enabler for learning, and if the reward signal is sparse and comes at the end of a sequence of actions,

it will be difficult to train in environments that are severely complex. In addition, the information received from the environment is not necessarily sparse, meaning that the reward can be modeled based on the information provided by the rich sensory outputs. Hence, it is desired to augment the sparse extrinsic reward by additional dense rewards that will aid the end-to-end learning algorithm in a much better way. In addition, policy optimization is more compatible with auxiliary objectives. One does this by constructing additional feedback signals that are very dense. They should be related to the task that the agent should solve. This dense feedback signal should be created in such way that whenever the agent succeeds in those tasks, it is probably also going to get knowledge that can be useful for the main objective. A standard sparse reward signal for navigation is usually tied to guiding the agent to a goal. The RMF will fly in a 3D environment during training with the aim of finding the waypoint. The reward function is specified such that it will guide the RMF to that waypoint by for example giving the agent a positive reward if it reaches the waypoint. One could also evaluate the distance to the waypoint from the RMF and give a reward if there is a reduction in that distance. Such sparse feedback signal would work well in an open environment with no obstacles, but when introducing objects to the environment, one would need to augment the whole training process with additional reward signals. For instance, the RMF may also receive a negative reward if it crashes or gets too close to an obstacle. In addition, the DDPG algorithm is an off-policy learning algorithm. It estimates the Q-value of being in a current state by predicting the total future reward that the agent will receive given an action, which also leverages the training process.

5.5 Planning

The ability to navigate within an environment also includes the ability to plan out a path. When doing path planning, the algorithm is effectively trying to find a collision free trajectory through the environment that connects the start state to the goal state given a set of obstacles, constraints of the robot and other environmental factors. The path generated consists of an initial configuration, a trajectory, and a desired goal configuration. The trajectory satisfies the path configurations by a function that takes time as input, and outputs a position in 3D space such that at time 0, the trajectory is at the initial configuration, and the trajectory at its end is at the goal configuration. These paths can be generated by simple *Combinatorial Methods*, but state-of-the-art methods in path and motion planning relies on sampling-based methods.

5.5.1 Sampling-based Path and Motion Planning

Sampling-based planning relies on searching for admissible paths by sampling. This can for instance be done by *Rapidly Exploring Trees*[18] or by *Rapidly Exploring Graphs*[16]. For the method to bias the exploration toward unexplored space one repeatedly samples random states starting from an initial root in the environment, and finds the nearest-neighbour. Then, a collision-free motion from the nearest node to the new node is found in the direction of the sample. Thus, creating a tree structure. The new node could be the sampled point, but not necessarily. The search tree is updated between each iteration and the process is terminated when the tree has reached the goal region. In an attempt to converge to a more optimal solution, the search tree will be continually rewired as the number of nodes in the tree goes to infinity. This is referred to as the *RRT** method[16]. These methods are well suited for exploration in complex, confined environments, but require a map to query. Even though such maps may be reliable, they are computationally expensive to derive. Hence, it is advantageous to omit this costly operation by using a method that does not need a consistent online reconstructed 3D map of the environment. In the next chapter, Chapter 6, a proposed method is given that needs very limited input data and ensures collision avoidance. It does not relay on any prior information about the environment or a constructed map. The proposed method can be combined with a coarse planner providing sparse waypoints. However, the end-to-end-learning algorithm in this thesis was tested with a sampling-based global planner *Graph-Based Planner* (GBPlanner) by Dang et al. [5]. It relies on the graph-based exploring version of the RRT*, the *Rapidly-Exploring Random Graph* (RRG)[16] algorithm.

Chapter 6

Proposed Approach

This chapter will go into the specifics of the proposed approach and what was needed so that the RMF could fly autonomously, the software used and how all the different modules that make up the autonomous system are connected and specified.

6.1 System Overview

The proposed learning-based approach for autonomous flight within confined environments was structured as seen in Figure 6.1. The system was developed in the open-source platform *Robotic Operating System* (ROS). This is a meta-operating system that allows easy modular development by organizing the different modules that make up the system into nodes. Each node communicates with the rest of the system through a message-passing process. More specifically, ROS Melodic was used with the operating system Linux, since it is the only operating system that ROS is capable of running on.

The system consists of six main components. The first one is the *Sensory Plugins Module*. This component is mainly depicting the onboard sensors of the real quadcopter, more specifically the IMU and the LiDAR sensor. The sensors will measure the state of the RMF provided and then used by the *Perception Module* and the *Rotors Wrapper Module*. The perception module filters out noisy data and extracts relevant data. Within the rotors wrapper, visual and odometry data is put together with the reference waypoint provided by the *Path Planner Module*. The path planning module uses the visual information about the environment and odometry data to plan the trajectory that should be executed. In addition, the rotors wrapper computes the reward given the information from the previous components. The state information and the reward is then fed to the *DDPG Module*. This algorithm computes actions given the state and reward provided by the rotors wrapper. The actions from the DDPG algorithm are then fed to the *Low-Level Control Module* of the pipeline. The control laws will command

the actuators given the reference provided by the DDPG module. The *RMF module* executes these commands and updates its state in the environment. These updates are sensed by the sensory plugins, and the cycle continues. The following sections will provide necessary information about each of the modules.

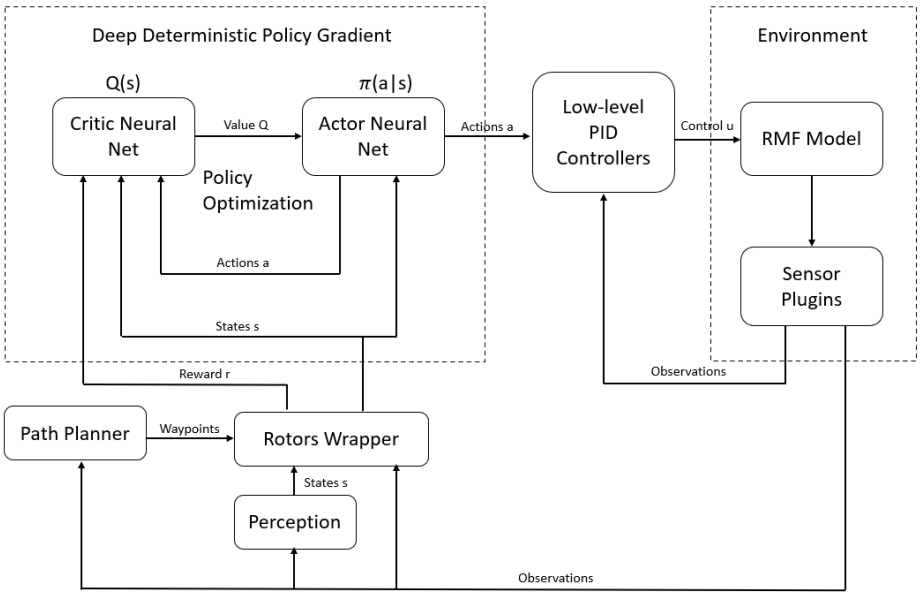


Figure 6.1: Architecture overview showing the general set up for high level control of the RMF using reinforcement learning.

6.2 Simulator

The system was developed using the *RotorS Simulator*[9] and Gazebo. The Gazebo robots simulator emulates the actuators and the physics of the environment. In addition, it offers a suite of sensors and interface for both users and programs. The RotorS Simulator was developed by the Autonomous Systems Lab of ETH Zurich and in this simulation environment, RMF was modeled. It is a framework that works with the Gazebo robot simulation tool and can be used to accurately model the dynamics and simulate the control of the RMF with reinforcement learning. In Figure 6.2 one can see how the RMF is simulated in the Gazebo environment.

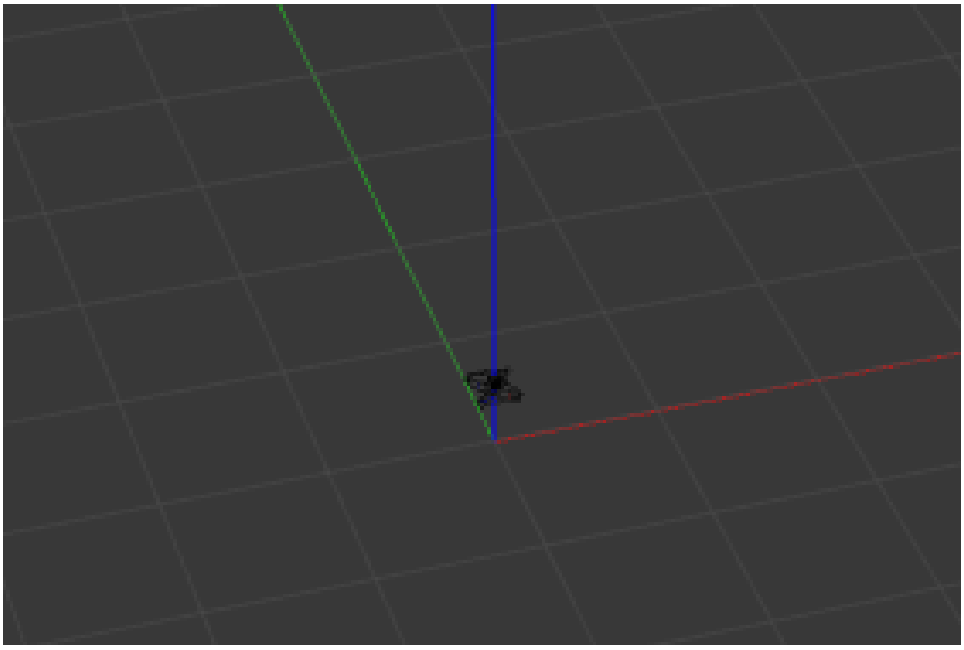


Figure 6.2: An open Gazebo environment with the RMF.

The RotorS simulator consists of several components and the most important ones is the dynamics simulator build upon the theory discussed in Section 2.3 and the parameters specified in Table 6.1, the state sensory plugins and the low-level MAV PID-controllers. There are a lot of different modules and computations that need to work cohesively for the entire system to work. In an attempt to focus on a single task in isolation, the uncertainties that are introduced by the complex pipeline are to some degree neglected in the simulation environment. In addition, using a simulator allows for simpler testing instead of using expensive physical quadcopter platforms. Even though the open-source RotorS framework supports simulation of sensors, such as IMUs and cameras, perfect state information was used to circumvent the need for state estimation algorithms. This allows for noise-free information about the position, orientation, linear and angular velocity of the RMF. This can be done by replacing the sensors with ideal odometry sensor plugins provided by the Gazebo simulator. In this way, it is possible to focus on the larger navigation problem without worrying about uncertainties in the other components. The LiDAR plugin sensor used for this project was developed and customized by the Autonomous Robots Lab (ARL) of Norwegian University of Science and Technology (NTNU) and accurately depicts the Ouster OS-1 3D LiDAR sensor. In addition, in order to detect collisions, the Gazebo contact plugin[14] was used.

RMF Specifications	
Parameters	Value
Mass (m)	0.5265
Arm length (l)	0.1
Moment of inertia about the x_b-axis (I_x)	0.01
Moment of inertia about the y_b-axis (I_y)	0.01
Moment of inertia about the z_b-axis (I_z)	0.01

Table 6.1: Specifications of the parameters used to simulate the RMF.

6.3 Waypoints

In larger environments where the goal is far away, a path planning algorithm would need to set out *Waypoints*. These waypoints essentially work as sub-goals and usually indicate a change in the direction and altitude along the desired trajectory. Hence, when the RMF reaches one waypoint it seeks out the next. It is also important to set out these waypoints quite often to minimize acute path angles. It is the path planner that generates these waypoints or nodes. In Chapter 7, the proposed method is tested by manually setting out waypoints with the GBPlanner algorithm discussed in Section 5.5. The waypoints will be specified using Cartesian coordinates (x_i^w, y_i^w, z_i^w) for $i = 0, \ldots, n$ given in the world-fixed frame {w}, where $i = n$ represents the final waypoint or the goal. Some boundaries or success regions are also specified so that the RMF needs to be within some radius δ_r to the waypoint before it should fly to a new one. The δ_r will be very small in this case, due to the volumetric constraints provided by the narrow, confined environments.

6.4 Feature Extraction

The sensory plugins produce raw LiDAR and perfect odometry data that is fed to the feature extraction pipeline, so that one can extract the useful information from the raw input information.

6.4.1 Point Cloud Features

The raw LiDAR data is generated by a LiDAR simulator accurately imitating an OS-1 Ouster LiDAR sensor. The data is sent to the perception module, where simple filtering algorithm remove redundant points and environmental noise as discussed in Section 3.4 and Section 3.5.

Statistical Outlier Removal

The conventional *Statistical Outlier Removal* (SOR) [43] method was used to filter out environmental noise. The first step in the algorithm is to calculate the mean distance $\bar{d}$ between all the distances d_i of k-neighboring points as seen in Equation (6.1).

$$\bar{d} = \frac{\sum_{i=1}^{k} d_i}{k} \tag{6.1}$$

The second step is to calculate the standard deviation σ_{sor} (Equation (6.2)) of these k-measurements.

$$\sigma_{sor} = \frac{\sum_{i=1}^{k} (\bar{d} - d_i)^2}{k} \tag{6.2}$$

Thirdly, all distances that are greater than the sum of the mean distance and the threshold n multiplied with the standard deviation of the mean distance to the query point as seen in Equation (6.3) will be removed and marked as outliers.

$$d_{out} = \bar{d} + n\sigma_{sor} \tag{6.3}$$

The lower the threshold n is, the more aggressive the filtering will be. The speed and performance of this algorithm depends on the number of neighboring k points considered.

In Figure 6.3 one can see the effect of removing outliers marked in red in an underground mine environment. One of the challenges of denoising the point cloud is leaving important environmental features.

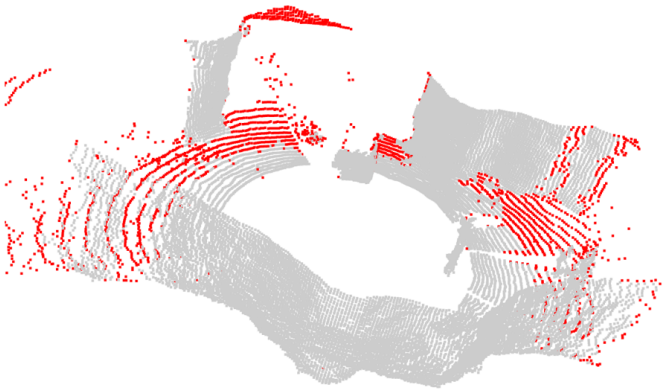


Figure 6.3: A point cloud depicting an underground environment, where the marked red points will be filtered out by the SOR algorithm.

Point Cloud Feature Extraction

After preprocessing the point cloud, the necessary features can be extracted. The features can be selected from the point cloud by first dividing the 3D point cloud into specific bins, and then sampling the closest point to the RMF in each bin. Henceforth, the proposed method for generating a sparse feature vector $\boldsymbol{s}_{pc}$ from a point cloud is to first divide it into N equal sectors in the xy-plane and then divide each sector into K stacks along the z-axis as seen in Figure 6.4. Afterwards, the closest point in each stack is sampled and the sparse distance d_i to the RMF is found. Thus, one can generate a sparse distance feature vector $s_{pc} \in \mathbb{R}^{NK \times 1}$ as seen in Equation (6.4).

$$\boldsymbol{s_{pc}} = \begin{bmatrix} d_1 \\ d_2 \\ \vdots \\ d_{NK} \end{bmatrix}. \tag{6.4}$$

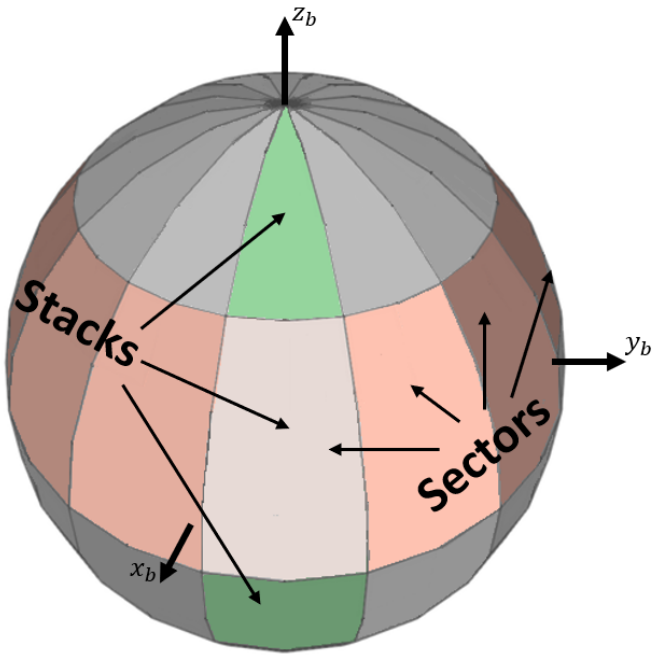


Figure 6.4: Dividing the 3D point cloud into stacks and sectors. (Figure generated by OpenGL: [1]).

These features will be continuously extracted, concurrently with the rest of the algorithms in the system. In simulation this extracted features can be visualized as red points as seen in Figure 6.5. The rainbow points depict LiDAR measurements.

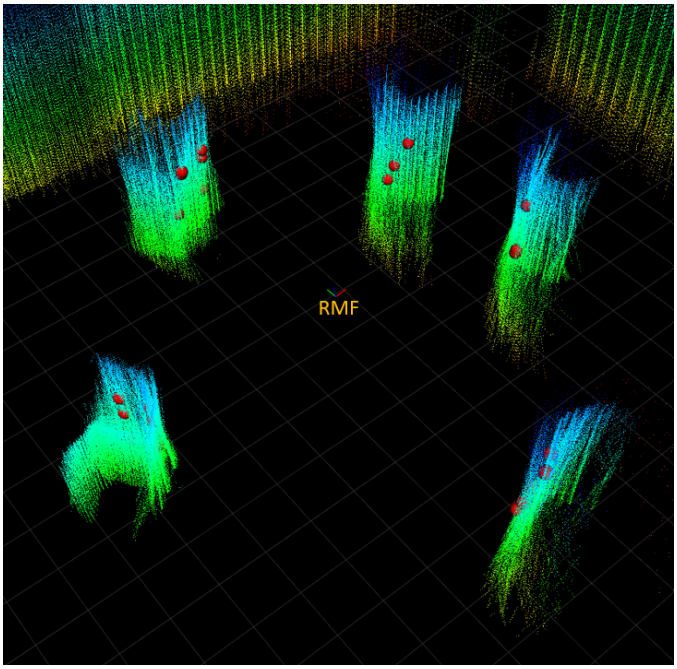


Figure 6.5: A visualization of the extracted sparse distance measurements marked in red with the RMF in the center.

The point cloud seen in Figure 6.5 consists of approximately 12 500 points but is compressed to a state vector $\boldsymbol{s}_{pc}$ (Equation (6.4)) of size NK. This means that a lot of information from the data is ignored. However, the low-cost feature extraction pipeline is not bounded by the Ouster LiDAR simulator or something analogous. If this solution would to be applied onto a real quadcopter one would not need to relay on an expensive 3D LiDAR sensor. An extremely lower-cost and lightweight system consisting of few 1D LiDAR sensors would suffice. This means that the proposed feature extraction process could be transferred to very small and cheap systems.

6.4.2 Odometry Features

Similarly, to a normal feedback control system where the output is generated with respect to the error, the states $\boldsymbol{s}_{odom}$ (Equation (6.5)) is given by the difference between the current RMF states provided by a perfect noise-free sensor and the reference value given by the waypoint. This should incentivize the robot to fly towards and stay at the commanded position.

$$\boldsymbol{s_{odom}} = \begin{bmatrix} \boldsymbol{p}^{w}_{ref} - \boldsymbol{p}^{w}_{RMF} \\ \boldsymbol{0}_3 - \boldsymbol{v}^{\boldsymbol{w}}_{\boldsymbol{wb}} \end{bmatrix} = \begin{bmatrix} x^{w}_{ref} - x^{w}_{RMF} \\ y^{w}_{ref} - y^{w}_{RMF} \\ z^{w}_{ref} - z^{w}_{RMF} \\ -u \\ -v \\ -w \end{bmatrix}, \tag{6.5}$$

where $\boldsymbol{p}^{w}_{RMF}$ is the position vector of the center of mass of the RMF and $\boldsymbol{p}^{w}_{ref}$ is the position of the waypoint, both given in the world frame. $\boldsymbol{v}^{\boldsymbol{w}}_{\boldsymbol{wb}}$ is the RMF world-fixed linear velocity.

6.4.3 Tracking Feature

The representation of the reference path is in the form of a straight-line segment that connects the different waypoints. At every point in time, it is possible to take the 3D position of the RMF and calculate the error between the reference path $\boldsymbol{p}^{w}_{ref} = [x^{w}_{r}, y^{w}_{r}, z^{w}_{r}]^{\top}$ and the position of the RMF $\boldsymbol{p}^{w}_{wb} = [x^{w},\ y^{w},\ z^{w}]^{\top}$ by applying *Trajectory-Tracking Control* as seen in Equation (6.6) [12].

$$\boldsymbol{e_{track}} = \boldsymbol{p}^{w}_{ref} - \boldsymbol{p}^{w}_{wb} = \begin{bmatrix} x^{w}_{r} - x^{w} \\ y^{w}_{r} - y^{w} \\ z^{w}_{r} - z^{w} \end{bmatrix}. \tag{6.6}$$

Hence, it is possible to introduce a single state s_{te} (Equation (6.7)) to the DDPG algorithm that represents the tracking error with the aim of reducing the divergence between the optimal and the actual path of the RMF. One could describe this tracking error with three states, vertical, along, and cross track error but only a single state s_{te} was introduced for simplification of the reward structure which will be discussed in Section 6.5.

$$s_{te} = \left|\left| \frac{\overrightarrow{W_2P} \cdot \overrightarrow{W_2W_1}}{|\overrightarrow{W_2W_1}|^2} \overrightarrow{W_2W_1} + \overrightarrow{PW_2} \right|\right|_2, \tag{6.7}$$

where W_1 and W_2 are two distinct waypoints, P is the position of the RMF. The optimal path depends on the environment and does not necessarily have to be straight. This method also assumes that there is collision checks in the planning algorithm so that the reference path is collision-free.

6.5 The Structure of the Reward Functions

In Chapter 4, it was discussed that the optimal policy maximizes the expected value of the discounted future reward. Hence, the reward function

plays a vital part in the formulation of the navigation task. The computation of the reward is done in the rotors wrapper module by using all the information provided by the sensors and the path planner. However, the design of such reward functions is not trivial.

6.5.1 Navigation Reward

It is possible to design an online, reactive *Obstacle Avoidance Controller* based on *Virtual Potential Fields* with the ability to navigate towards some goal while avoiding obstacles by coupling the odometry data with the perception data $\boldsymbol{s} = [\boldsymbol{s}_{odom}^{\top}, \boldsymbol{s}_{pc}^{\top}]^{\top}$. This can be done with respect to reinforcement learning by shaping the reward function r_t to guide the RMF towards or away from regions in the environment. This control strategy relies on deriving actions for the RMF directly from information about the environment, and it is a fast way to generate such actions that emphasizes on obstacle avoidance while navigating towards a goal, even in complicated environments. For the purpose of generating a behavior that is sufficient for the RMF to go from its origin state to a goal while avoiding obstacles, the first step is to create a reward function that attracts the robot to the goal state which may be anywhere in the environment. Such a navigation reward can seemingly be very intuitive and trivial to design. When the RMF reaches the goal, it gets a positive reward r_{goal}, and if it collides, it gets a negative reward $r_{collision}$. However, as previously discussed in Section 5.4, this makes up a rather sparse reward function and it is more preferable to add auxiliary rewards at each time step. The reward function should be rather smooth so that it can better help with guiding the RMF to the end goal region. Since it is in general preferable being close to a waypoint opposed to being far away, it is possible to encode such behavior onto the RMF using a quadratic reward structure. Similarly, if the RMF should reach the goal in an efficient way using minimum amount of energy, this logic should also be applied to the actions. Hence, we get the reward function as seen in Equation (6.8)

$$r_{quad} = -\boldsymbol{s}_{odom,t}^{T}\boldsymbol{Q}\boldsymbol{s}_{odom,t} - \boldsymbol{a}_t^{T}\boldsymbol{R}\boldsymbol{a}_t, \tag{6.8}$$

where $\boldsymbol{Q} = \boldsymbol{Q}^T \geq 0$ and $\boldsymbol{R} = \boldsymbol{R}^T \geq 0$ are design matrix parameters, $\boldsymbol{s}_{odom,t}$ is the state from Equation (6.5), and $\boldsymbol{a_t}$ is the linear acceleration in three dimensions $\boldsymbol{a}_t = [a_{x,t}, a_{y,t}, a_{z,t}]^{\top}$ both at a given time t. One should also note that this is a similar quadratic scheme as the one used in *Linear Quadratic Regulators* (LQR).

6.5.2 Obstacle Avoidance Reward

The second step is to make sure that there is a behavior that repulses the RMF from any obstacles that may exist. This is done by introducing a reward function that points the RMF away from obstacle regions and punishes the

RMF more when it approaches closer to obstacles. In order to perform safe navigation, the algorithm needs to make sure that this region is sufficiently far away from the actual obstacle. This can be represented in terms of a Gaussian reward function (Equation (6.9)) where one sums up all the rewards calculated based on each sparse distance s_{pc_i} to determine how large the reward punishment should be.

$$r_{obst} = -\sum_{i=0}^{NK} \frac{1}{\sigma_k \sqrt{2\pi}} exp(-\frac{d_i^2}{2\sigma_k^2}), \tag{6.9}$$

where N is the number of sectors, K is the number of stacks, d_i is one point cloud distance feature, σ_k is a design parameter that depends on which stack d_i lies in, where k = $\{0, \ldots, K-1\}$.

6.5.3 Combining Obstacle Avoidance with Navigation

The two reward functions are combined, the goal-oriented reward (Equation (6.8)) and the obstacle repulsive reward (Equation (6.9)), into a general highly nonlinear reward function r_t Equation (6.10) that allows the RMF to get to its goal while safely avoiding obstacles.

$$r_t = \begin{cases} -\boldsymbol{a}_t^T \boldsymbol{R} \boldsymbol{a}_t + r_{goal}, & if\ ||\boldsymbol{p}_{ref}^b - \boldsymbol{p}_{RMF}^b||_2 < \delta_r\ and \\ & \quad ||\boldsymbol{v}_{wb}^b||_2 < \delta_v, \\ -\boldsymbol{a}_t^T \boldsymbol{R} \boldsymbol{a}_t - r_{collision}, & if\ collision, \\ r_{quat} + r_{obst}, & otherwise. \end{cases} \tag{6.10}$$

δ_r is the radius of acceptance to the goal and δ_v is velocity of acceptance within the goal region.

The 2D figure, seen in Figure 6.6, has 3 red obstacles and a yellow goal region, and it illustrates the attractive and repulsive reward function (Equation (6.10)). The function for this environment has a unique global minimum in the yellow goal region. The reward function also has saddle points behind the obstacles in terms of where the goal is. Here the vector field is at a minimum in one direction and a maximum in the opposite direction.

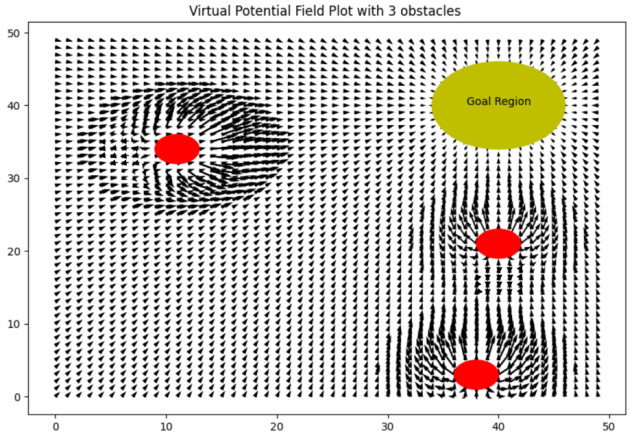


Figure 6.6: Vector field with 3 obstacles in red and a yellow goal region.

In some configurations, the global minimum will not be set perfectly at the goal region due to close obstacles having pushed the minimum away. In addition, in more complex configurations, where there are big obstacles or many obstacles that are close to each other, there can be created a local minimum. This is a problem, as the local minimum can attract the RMF if it is nearby. This problem can be omitted to some degree by making sure that the goal is put sufficiently close to the RMF and to some degree in line of sight. Another solution would be to detect if the RMF is stuck in a local minimum and override the policy by sending commands so that it moves in a safe direction and away from the minima. The advantage of using this approach is that this can be evaluated relatively fast.

6.5.4 Tracking Reward

It is possible to also design a simpler controller that assumes the path to be collision-free. This controller only uses the states $\boldsymbol{s} = [\boldsymbol{s}_{odom}, s_{te}]^\top$ when allocating actions to the RMF. This *Tracking Controller* can be designed similarly as the obstacle avoidance controller, but instead emphasizes on tracking the reference path towards the goal waypoint. Initially, the purpose of implementing and presenting the usage of the simpler tracking controller is to better understand and compare with the obstacle avoidance controller as done in Chapter 8. The tracking reward can be calculated with an exponential reward with inspiration from Craig Reynolds' path following algorithm [31]. The path that the RMF should follow has a radius δ_{track}. If the RMF is outside of this radius, it will be punished by the reward $r_{outside}$. If it is within this radius, it will only be gradually punished based on the distance to the optimal path. Hence, we can get then the following reward r_{te} (Equation (6.11)).

$$r_{te} = \begin{cases} -r_{outside}, & if\ ||\boldsymbol{p}^w_{ref\_path} - \boldsymbol{p}^w_{RMF}||_2 > \delta_{track}, \\ -exp\left(\dfrac{s^2_{te,t}}{2\sigma_{te}}\right) + 1, & otherwise. \end{cases} \tag{6.11}$$

This structure assumes that the optimal path is collision free. It is also necessary to use the navigation reward (Equation (6.8)) in order to have a full controller that incentive the RMF to the goal while also following the optimal reference path. Combining these two rewards gives Equation (6.12).

$$r_t = \begin{cases} -\boldsymbol{a}_t^T \boldsymbol{R} \boldsymbol{a}_t + r_{goal}, & if\ ||\boldsymbol{p}^b_{ref} - \boldsymbol{p}^b_{RMF}||_2 < \delta_r\ and \\ & \quad ||-\boldsymbol{v}^b_{wb}||_2 < \delta_v, \\ -\boldsymbol{a}_t^T \boldsymbol{R} \boldsymbol{a}_t - r_{collision}, & if\ collision, \\ r_{quat} + r_{te}, & otherwise. \end{cases} \tag{6.12}$$

6.6 Implementation of the DDPG Algorithm

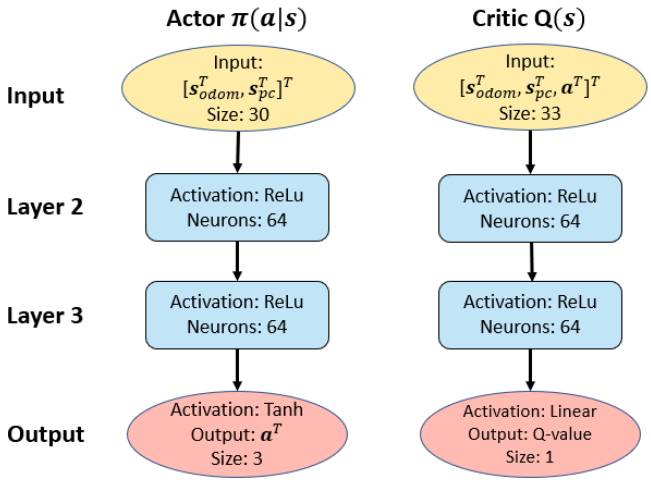
In order to apply reinforcement learning to the system, the open-source *OpenAI Gym*[25] was used together with Google's machine learning framework *Tensorflow* [36]. Tensorflow is a library for numerical computations and widely used for large scale machine learning problems. OpenAI Gym allows for easier access to existing implementations of algorithms for deep learning. Specifically, this project made use of the DDPG algorithm Appendix A.1 developed by OpenAI Gym. However, there exists several RL algorithms for continues control tasks, each with their trade-offs and benefits. DDPG is not necessarily considered as a state-of-the art algorithm despite being a relatively new algorithm from 2016. Nevertheless, it is still a well performing off-policy algorithm that compares well with many modern solutions as discussed by Henderson et al. [15], such as *Proximal Policy Optimization*[32] (PPO), *Actor-Critic with Experience Replay* (ACER) [39] and *Actor-Critic using Kronecker-Factored Trust Region* (ACKTR) [40].

6.6.1 Network Topology

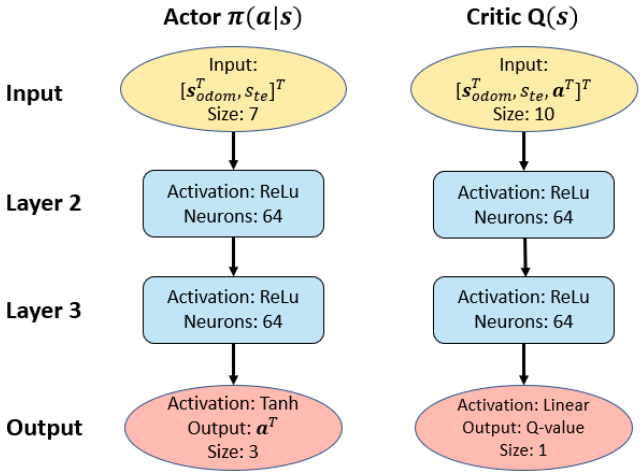
The DDPG consists of four fully-connected neural nets. These are the actor net and the critic net, and the corresponding target actor net and target critic net, respectively. The basic idea of how a neural network learns or approximates functions is that it first takes in some input data that is vectorized, feeds it through the network, and performs a series of matrix operations layer by layer. In addition, since the networks should be able to approximate any kind of functions, there are additional nonlinear activation functions embedded to the network function approximation. It is, however, not obvious how

these networks should be designed. Nevertheless, there are some obvious design choices one should make, as selecting the number of layers, the number of nodes and which activation functions to use in each layer. The Rectified Linear Unit (ReLU) returns 0 if it receives any negative input, and for any positive values it returns the value back. This activation function is widely used in the hidden layers, and is sufficient enough for this network topology. The Q-value given from the critic is equal to or larger than 0. Hence, there is no activation in the output layer of the critic, only a linear combination of the activations in the last hidden layer. Due to physical limitations of the real quadcopter the actions were constrained such that $\boldsymbol{a} \in [\boldsymbol{a}_{min}, \boldsymbol{a}_{max}]$. Thus, a hyperbolic tangent (tanh) was used in the output layer of the actor as it is symmetric around the origin and outputs values in the range of [-1, 1]. This will also further ensure stability during training.

The best way to examine which network structures that perform well is to do some cross validation. If there are too few neurons or too few layers, the function approximated will be highly linear and not be able to represent the high dimensional function that is required to map all the states to the actions such that RMF may respond well in the highly complex environment. In contrast, having too many neurons or layers may lead to the approximated function being highly nonlinear. Consequently, this can lead to overfitting, or as also experienced, not being able to converge to any solution. This implied that having too many parameters lead to unsatisfactory or unstable results. In addition, having more parameters would deem more time spent on training the excessive logic. It is therefore important to find a trade-off solution that is pseudo-optimal. This trial and error process resulted in the networks architectures described in Figure 6.7. Both controllers use the same architecture, but they vary in in the number of inputs to the actor and the critic. The actor and target actor network are structurally the same. Similarly, this also applies for the critic and target critic networks. All networks consist of 2 hidden layers each with 64 neurons. The Adam optimization algorithm was used to update the weights during training. This is a very computational efficient algorithm and works well for this problem.



(a) The fully-connected neural networks architecture used in the DDPG algorithm by the obstacle avoidance controller.



(b) The fully-connected neural networks architecture used in the DDPG algorithm by the tracking controller.

Figure 6.7: The fully-connected neural networks architectures used by the different controllers.

6.7 The Training Process

The two distinct controllers, the *Obstacle Avoidance Controller* and the *Tracking Controller* were trained in a curricular fashion as discussed in Section 5.3.1. The obstacle avoidance controller was trained by exposing the RMF to gradually more difficult environments. The main objective was always set to reaching the goal region. When the RMF got increasingly better

at this behavior, more obstacles were introduced with the aim of learning to avoid the obstacles as it navigates towards the goal region. The tracking controller was trained in only one open environment. However, in the first stage of training, the objective was set to only reach the goal region. In the second stage, the objective was also set to reach the goal region, but by strictly following the optimal straight path to the goal.

6.7.1 Environments

The available static shapes used in the Gazebo environment were the t-block (blue), pyramid (green), u-block (red) and pillar (grey) as seen in Figure 6.8.

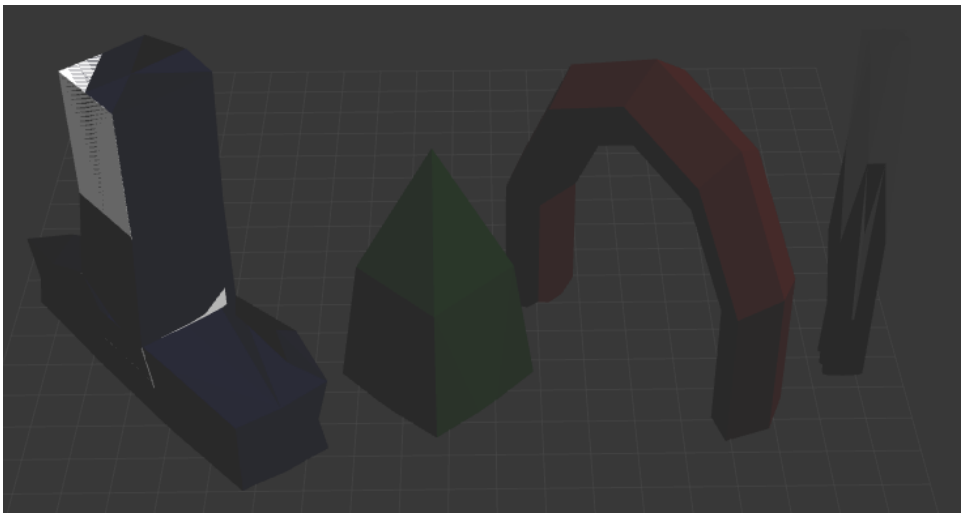
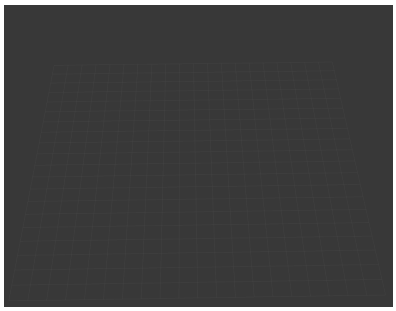


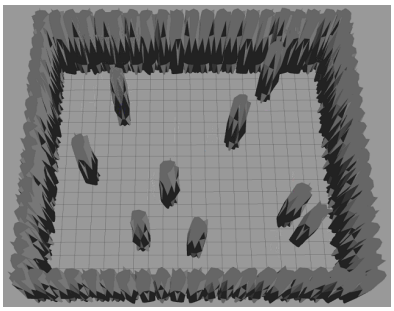
Figure 6.8: The shapes used in the simulations.

After some careful validation of different environments, it was decided to use the following four environments as specified in Figure 6.9 when training the obstacle avoidance controller. Each environment consist of a $20 \times 20\ m^2$ collidable floor. In the first environment, shown in Figure 6.9a, the objective was to learn to navigate to one goal waypoint. In the second environment, shown in Figure 6.9b, walls and 9 random pillars were added. The pillar shape was used since it varies geometrically very little in the z-direction, such that the RMF only needed to learn collision avoidance in the xy-direction. In order to diversify the data that the DDPG algorithm collects, the obstacles in the environments were randomly shuffled each epoch. In addition, the initial position of the RMF and the goal position were randomly set in the environment each time the *Terminal Conditions* were fulfilled. By using this distributed scheme, the RMF was immediately exposed to many environmental configurations, which further allows for accelerated learning. Hence, in the second environment, 9 random pillars changed position each epoch

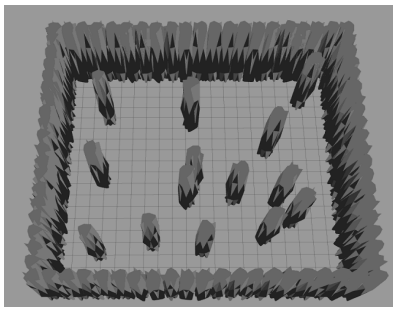
during training. The objective was to learn simple collision avoidance in the xy-direction while also navigating to a waypoint. In the third environment, shown in Figure 6.9c, more random pillar obstacles were added. The 14 pillars also changed position at each epoch. The main objective was to reach the goal waypoint and avoid obstacles in a more complex environment. In the fourth environment, shown in Figure 6.9d, more complex shapes were used, namely 2 t-blocks, 3 pillars, 2 pyramids and 1 u-block. These also change position each epoch. The objective was to learn to avoid obstacles in both in xy- and z-direction while navigating towards a waypoint.



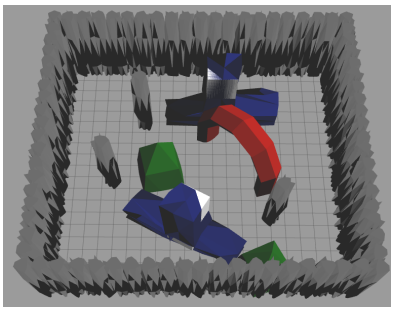
(a) An open environment.



(b) A closed environment with 9 randomly placed obstacles.



(c) A closed environment with 14 randomly placed obstacles.



(d) A closed environment with 8 randomly placed obstacles with different shapes.

Figure 6.9: The environments used for training the RMF.

6.7.2 Terminal Conditions

During training and simulation, it is important to set artificial boundaries defining when to reinitialize the environment. When the episode reaches predefined terminal conditions, the position of the RMF and the goal waypoint is reset. These are the following terminal conditions used:

- If sufficient time t has passed.
- The RMF has crashed with an obstacle or with the floor.

In addition, it is important to understand how many episodes each simulation should last. In order for the RMF to pass on to the next environment, it should easily be able to reach the goal waypoint more or less regardless of the position of the RMF and the waypoint within the said environment.

Chapter 7

Results

This chapter presents the results from training the system with the DDPG algorithm. The RMF was trained independently with the reward from Equation (6.10) and from Equation (6.12) on a computer with the specifications found in Appendix B.2. These two different approaches, the obstacle avoidance controller and the tracking controller, were then independently tested and evaluated in different simulated environments in order to analyze the robustness and the reliability of each controller. The trained RMF was also tested on a dataset, where it would navigate within a simulated underground mine environment.

7.1 The Training Setup and Results with the Obstacle Avoidance Controller

In the following sections, the results from using the obstacle avoidance controller will be presented. The action $\boldsymbol{a}$ delivered from the DDPG algorithm is defined as the linear accelerations in three dimensions $\boldsymbol{a} = [a_x,\, a_y,\, a_z]^\top$, where $\boldsymbol{a}_{max} = [1,\, 1,\, 1]^\top$ and $\boldsymbol{a}_{min} = [-1,\, -1,\, -1]^\top$. The state space vector used in the DDPG algorithm is defined as $\boldsymbol{s} = [\boldsymbol{s}_{odom}^\top,\, \boldsymbol{s}_{pc}^\top]^\top$. The RMF was trained in all four environment seen in *Figure* 6.9, starting in the simpler open environment and progressing to the most complicated confined environment with obstacles of different shapes in a curricular fashion. Throughout the training, the RMF was randomly initialized at different positions with the waypoint set out randomly with a max distance δ_{max} to the RMF. As the terminal conditions were fulfilled, the RMF and the waypoint were reinitialized at new positions. It was specified that the RMF and the waypoint were not to be initialized in contact with the obstacles or the floor. In addition, the position of the RMF and the waypoint could not be set below the floor. The topology of the networks used in the DDPG algorithm is specified in Figure 6.7. At the very beginning, the networks were initialized with random seeds. Furthermore, as the RMF progressed from one environment to

the next, the weights and biases in all the networks were transferred to the next training session. Only the hyperparameters and the parameters of the reward were changed between each environment.

7.1.1 Hyperparameters

The DDPG algorithm consists of multiple hyperparameters. They were defined in each training session as seen in Table 7.1. 1 epoch corresponds to 2000 time steps.

Parameters	Values			
	Env 1	Env 2	Env 3	Env 4
Epochs	750	500	500	500
Batch size	32	32	32	32
Actor learning rate ($\alpha_{\boldsymbol{\theta}}$)	1e-05	1e-05	1e-05	1e-05
Critic learning rate ($\alpha_{\boldsymbol{\phi}}$)	1e-05	1e-05	1e-05	1e-05
Discounting factor (γ)	0.99	0.99	0.99	0.99
Target network update rate (τ)	0.001	0.001	0.001	0.001
Max distance to waypoint (δ_{max})	4.0	5.0	6.0	6.0

Table 7.1: Specifications of the hyperparameters used in the DDPG algorithm in each environment during training.

The reward from Equation (6.10) was used throughout the four training sessions, and all of the parameters were defined in each training session as seen inTable 7.2.

Parameters	Values			
	Env 1	Env 2	Env 3	Env 4
r_{goal}	10	20	30	30
$r_{collision}$	10	20	30	30
Stack 1 (σ_0)	0.15	0.16	0.17	0.18
Stack 2 (σ_1)	0.20	0.21	0.22	0.22
Stack 3 (σ_2)	0.15	0.16	0.17	0.18
Waypoint acceptance region (δ_r)	0.25	0.30	0.30	0.35
Max velocity at waypoint (δ_v)	0.3	0.3	0.3	0.3

Table 7.2: Specifications of the reward parameters used in the reward function r_t (Equation (6.10)) in each environment during training.

There were $N = 8$ sectors and $K = 3$ stacks used, meaning that $\boldsymbol{s}_{pc}$ consisted of 24 sparse measurements. The constant matrices $\boldsymbol{Q}$ and $\boldsymbol{R}$ were

remained unchanged throughout the four environments and were defined as in Equation (7.1).

$$\boldsymbol{Q} = diag[0.7, 0.7, 1.1, 0.03, 0.03, 0.05] \tag{7.1a}$$
$$\boldsymbol{R} = diag[0.001, 0.001, 0.0009] \tag{7.1b}$$

7.1.2 Training Results

The training results from the four different environments (Figure 6.9) can be seen in the following figures. The mean reward from each epoch can be seen in Figure 7.1.

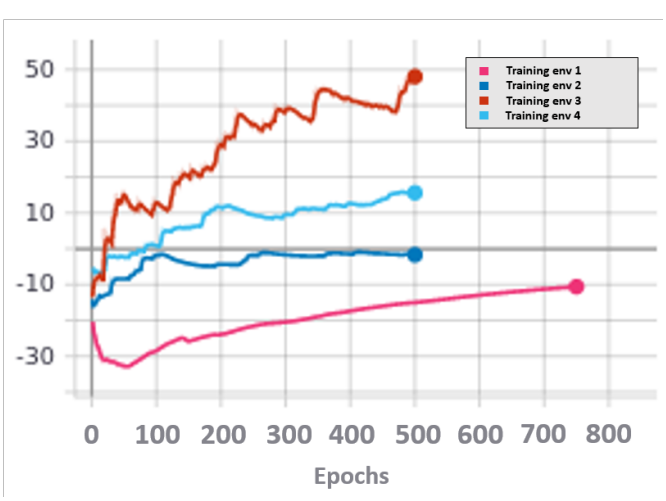
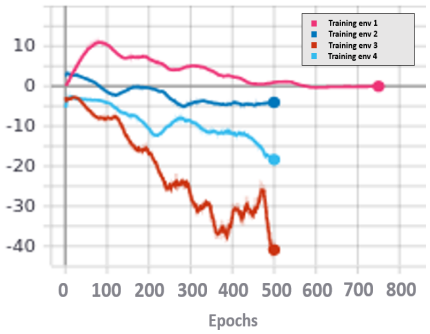
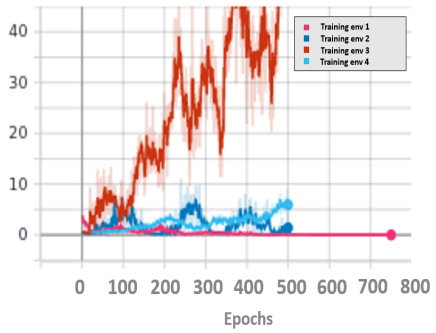


Figure 7.1: Simulation results of the mean reward return for each epoch.

The loss of the actor Equation (4.8) and the critic Equation (4.9) from all four training sessions can be seen in Figure 7.2.



(a) Simulation results of the mean loss return of actor for each epoch.



(b) Simulation results of the mean loss return of critic for each epoch.

Figure 7.2: The results from training depicting the loss of the actor and the critic.

Table 7.3 shows how long it took to train the agent in each environment. The total training time was 87 hours and 13 minutes.

	Env 1	Env 2	Env 3	Env 4
Training time	9h 49min	20h 45min	28h 8min	28h 31min

Table 7.3: Training time used in each environment.

7.2 Validating the Obstacle Avoidance Solution in Different Environments

The finalized trained RMF was evaluated on the following environments:

- Straight path environment (Figure 7.3a)
- Twisty path environment (Figure 7.4a)
- Y-path environment (Figure 7.5a)
- Large path environment with obstacles (Figure 7.6a)
- Large environment with obstacles (Figure 7.7a)
- Large environment with obstacles of different shapes (Figure 7.8a)
- Y-path environment with obstacles (Figure 7.9a)

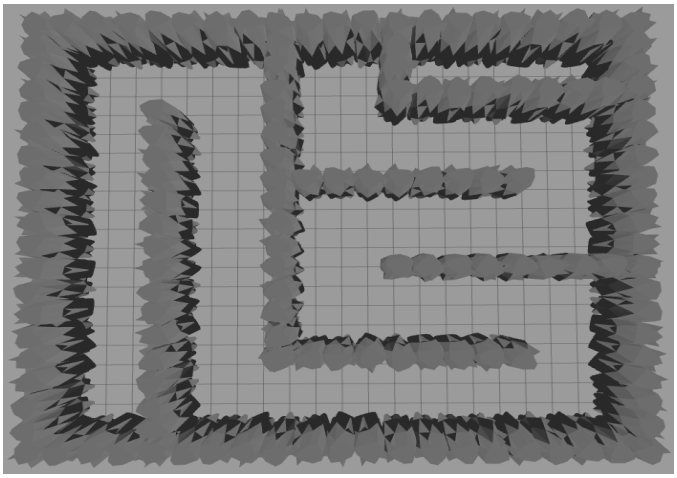
Each environment is designed such that it tests different attributes of the onboard controller. The cyan colored reference waypoints were manually

placed out without any path planner. The RMF had to be within the radius $\delta_r = 0.45$ of the waypoint in order for a new waypoint to be spawned. There were no max velocity conditions at the waypoints, except for the last one in the environment where $\delta_v = 0.3$. The trajectory is depicted in green and the rainbow colored points are depicting the point cloud used to extract the sparse distance features. The red point indicates the initial spawn of the RMF.

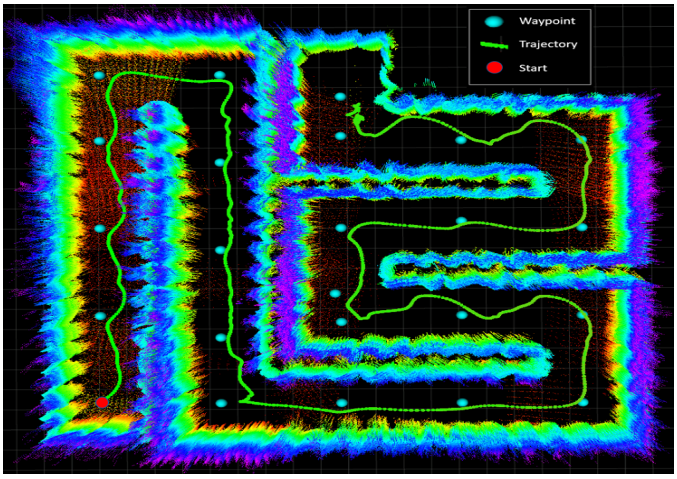
7.2.1 Collision-free Paths

In the first three environments all of the waypoints and the initial position of the RMF was at the same height $z = 3$.

The straight path environment (Figure 7.3a) mainly consists of a very long ($\approx$ 90m) and narrow ($\approx$ 2.5-2.7m width) collision-free passage. The objective is to test how capable the RMF is to traverse through a relatively straight path that is narrow with some additional 90° corners. This was done by only relying on the onboard sensors without any map construction or any prior data as seen in Figure 7.3b.



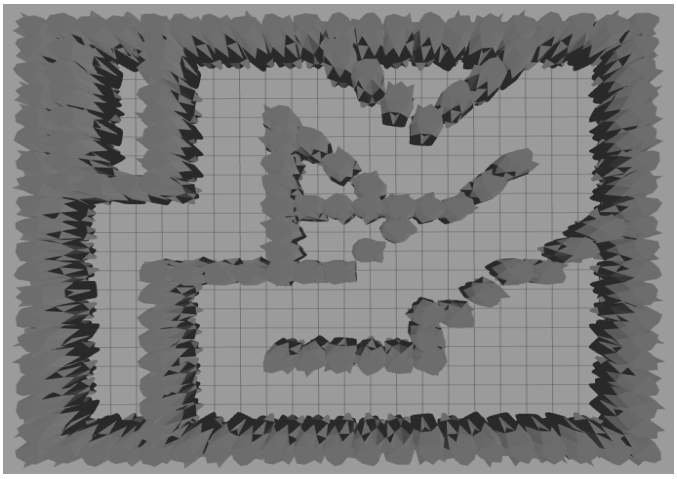
(a) Straight path environment.



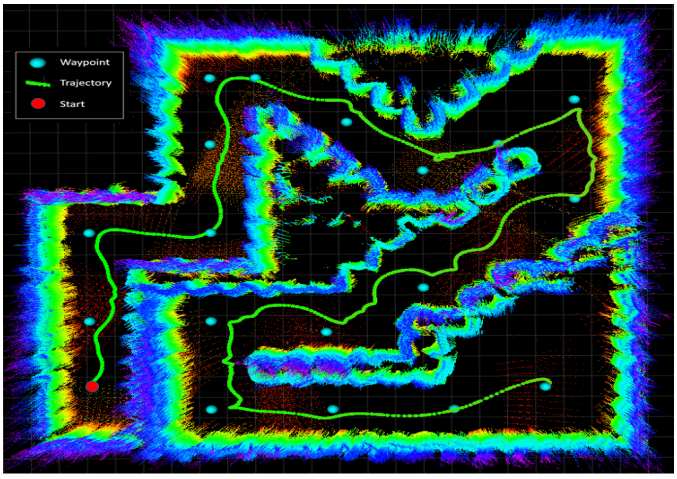
(b) Visualization of the trajectory of the RMF in the straight path environment.

Figure 7.3: Obstacle-free straight paths.

The twisty path environment (Figure 7.4a) also consists of a very long ($\approx$ 70m) and narrow ($\approx$ 2.5-2.8m width) collision-free passage. The objective is to test how capable the algorithm is to enable RMF to traverse through a winding path that is narrow. This was done by only relying on the onboard sensors without any map. The results can be seen in Figure 7.4b.



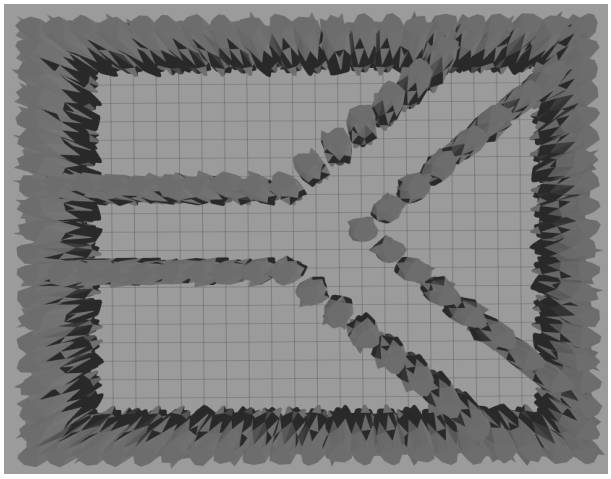
(a) Twisty path environment.



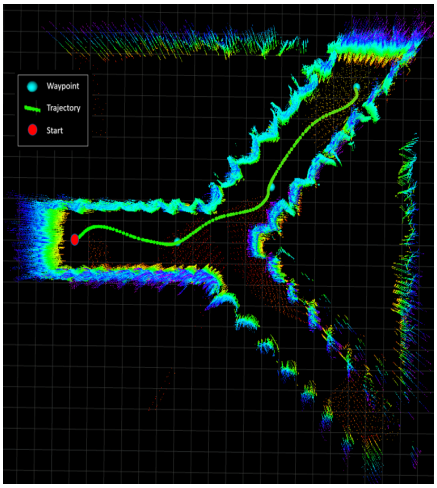
(b) Visualization of the trajectory of the RMF in the twisty environment.

Figure 7.4: Obstacle-free twisty paths.

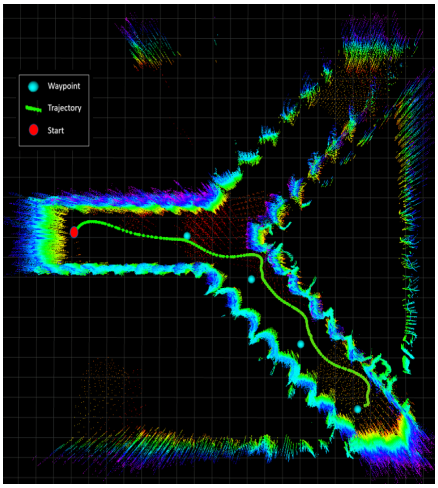
The y-path environment (Figure 7.5a) also consists of a narrow (≈ 2.5-2.8m width) collision-free passage with an intersection. The objective is to test how the RMF reacts when the waypoints are not set out in a intersection. This was done by only relying on the onboard sensors without any map. The results can be seen in Figure 7.5b and Figure 7.5c.



(a) Y-path environment.



(b) Visualization of the trajectory as the RMF goes up in the y-path environment.



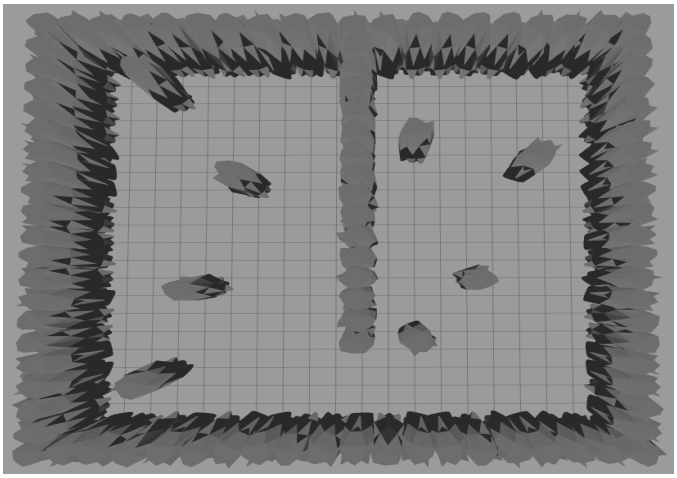
(c) Visualization of the trajectory as the RMF goes down in the y-path environment.

Figure 7.5: Obstacle-free y-path environment.

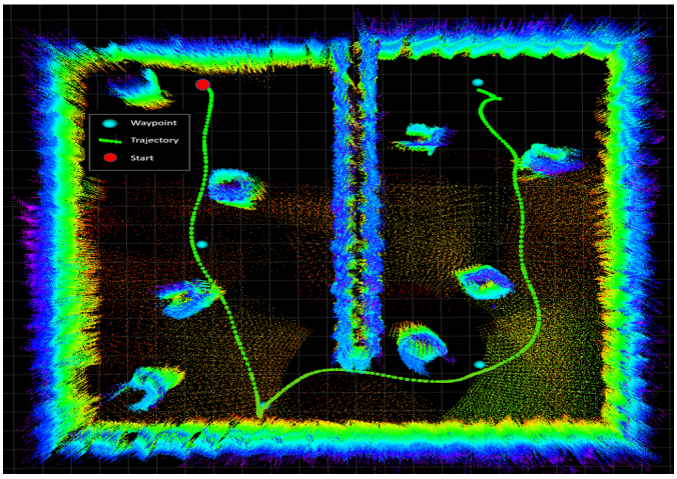
7.2.2 Paths with Obstacles

In the next environments the waypoints were set out at different heights $z = [3, 6]$.

The large path environment (Figure 7.6a) consists of a short ($\approx$ 41m) and wide ($\approx$ 3.7-8.m width) passage filled with obstacles. The objective is to test how well the RMF is able to traverse to the different waypoints while also avoiding the obstacles. This was done by only relaying on the onboard sensors without any map construction. The results can be seen in Figure 7.6b.



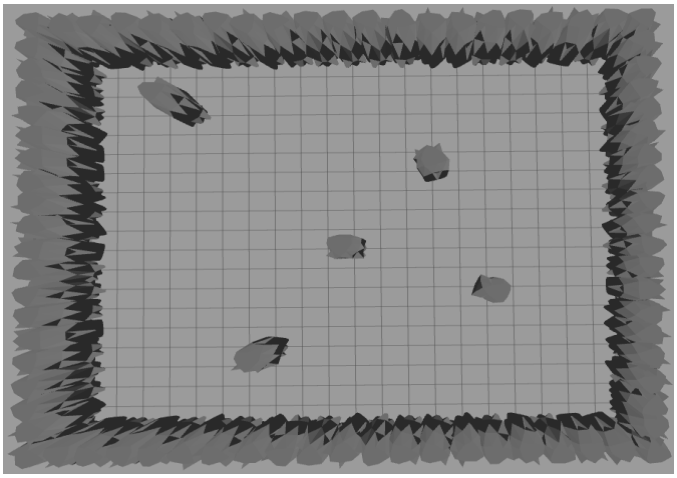
(a) Overview of the obstacle-filled path environment.



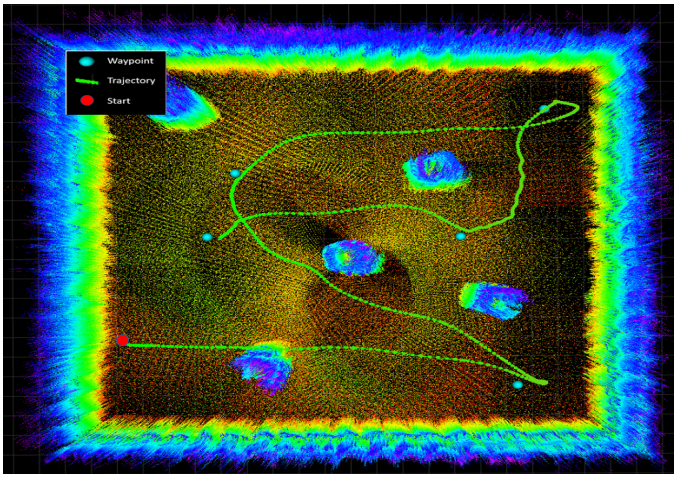
(b) Visualization of the trajectory of the RMF within the obstacle-filled environment.

Figure 7.6: Path environment with obstacles.

The large, confined environment (Figure 7.7a) is filled with random obstacles. The objective is to test how well the RMF is able to traverse to the different waypoints while avoiding the randomly placed obstacles. This was done by only relying on the onboard sensors without any online constructed map. The results can be seen in Figure 7.7b.



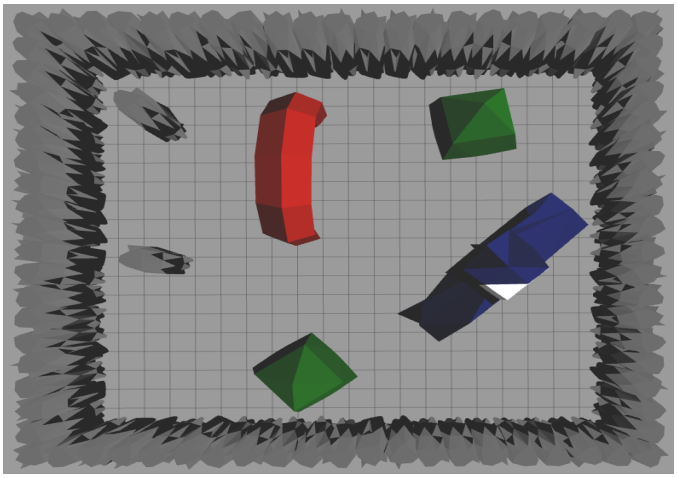
(a) Overview of the large environment with obstacles.



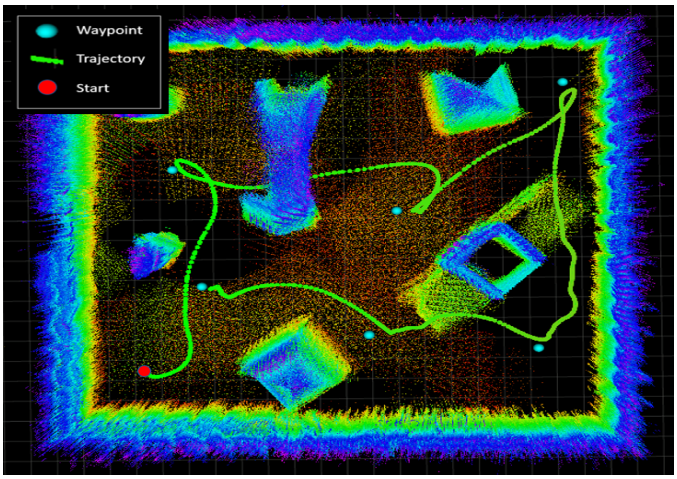
(b) Visualization of the trajectory of the RMF within a large environment with obstacles.

Figure 7.7: Large environment with obstacles.

The large, confined environment (Figure 7.8a) is filled with random obstacles in different shapes. The objective is to test how well the RMF is able to traverse to the different waypoints while also avoid the obstacles that varies in all three dimensions. This was done by only relying on the onboard sensors without any map construction. The results can be seen in Figure 7.8b.



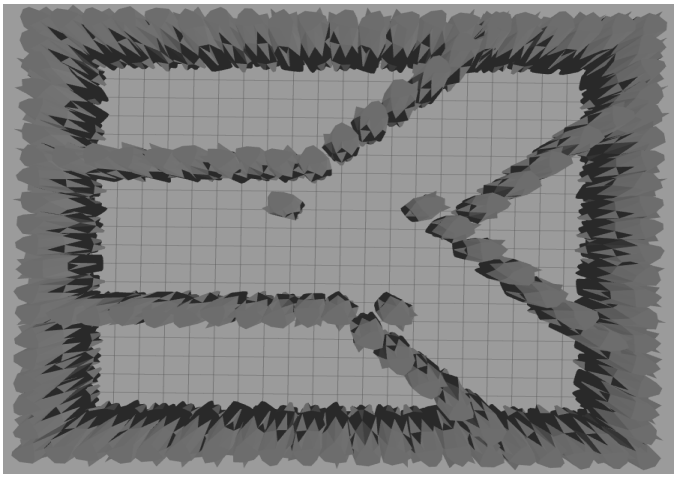
(a) Overview of the large environment with obstacles of different shapes.



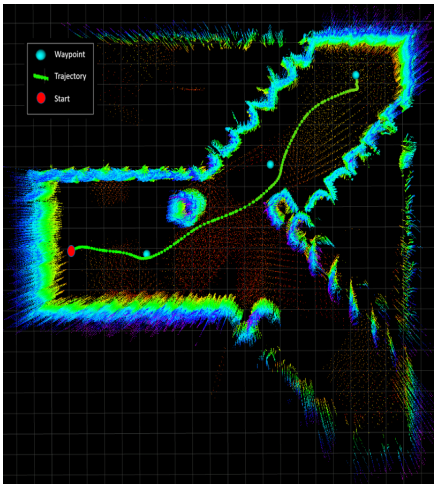
(b) Visualization of the trajectory of the RMF within a large environment with obstacles of different shapes.

Figure 7.8: Large environment with obstacles of different shapes.

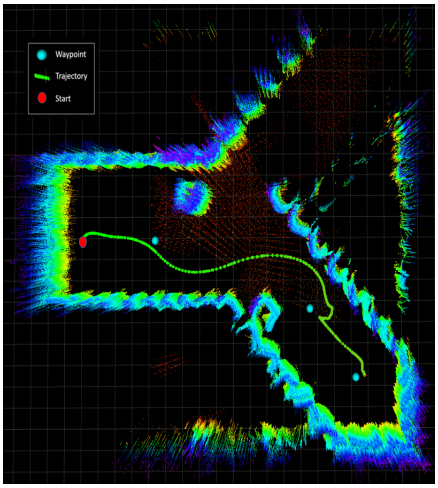
The y-path environment with obstacles (Figure 7.9a) consists of a wide passage (≈ 5.6-5.8m width) with an intersection filled with obstacles. The objective is to test how the RMF reacts when the waypoint is not set out in a intersection while also needing to avoid obstacles. This was done by only relying on the onboard sensors without any map. The results can be seen in Figure 7.9b and Figure 7.9c.



(a) An overview of the y-path environment with obstacles.



(b) Visualization of the trajectory as the RMF goes up in the obstacle-filled y-path environment.



(c) Visualization of the trajectory as the RMF goes down in the obstacle-filled y-path environment.

Figure 7.9: Y-path environment with obstacles.

7.3 The Obstacle Avoidance Controller in and Underground Mine Environment

To evaluate the proposed method in more realistic environments, the simulated RMF was further tested in the Gazebo simulator with a dataset provided by the Autonomous Robots Lab. The dataset was recorded in an underground mine environment, where the Autonomous Robots Lab deployed an "Aerial Scout" robot the specifics of which are detailed in [4]. The underground environment is one large room-and-pillar structure with multiple passages, and the objective of the deployed robot was to do robust autonomous

exploration and mapping in challenging subterranean areas. The dataset was recorded 7th of August 2019 in the Wampum Underground Facility in Pennsylvania (US).

The aerial scout was equipped with several sensors, one of which a Velodyne PuckLITE LiDAR sensor, and it conducted exploration with the onboard expert, GBPlanner. This aerial scout is analogous to the RMF in terms of sensor inputs. Thus, one can directly extract the recorded LiDAR data and a local reference path provided by the expert from the dataset. The LiDAR data had to be transformed to the body frame {b} in order for the RMF to extract the sparse distance features $\boldsymbol{s}_{pc}$. In addition, cyan colored waypoints were set out along the red local reference path. The RMF used its own odometry data and the prerecorded LiDAR data generated by the scout drone to navigate towards the waypoints. The RMF had to be within the radius $\delta_r = 0.45$ of the waypoint in order for a new waypoint to be spawned. There were no max velocity δ_v conditions.

The environment consists of a lot of environmental noise such as mist. Some of it can be seen as green clouds below (Figure 7.11). The environmental noise in the underground mine was filtered out with the SOR filter. The threshold was set $n = 0.01$ and number of neighbouring points was set $k = 40$ in the SOR algorithm. However, some noise was still present, and the results can be seen below. The RMF traversed through the underground facility starting in section 1 of the mine (Figure 7.10), then proceeded to section 2 (Figure 7.11), where it finished in section 3 (Figure 7.12). The trajectory of the RMF can be seen in green.

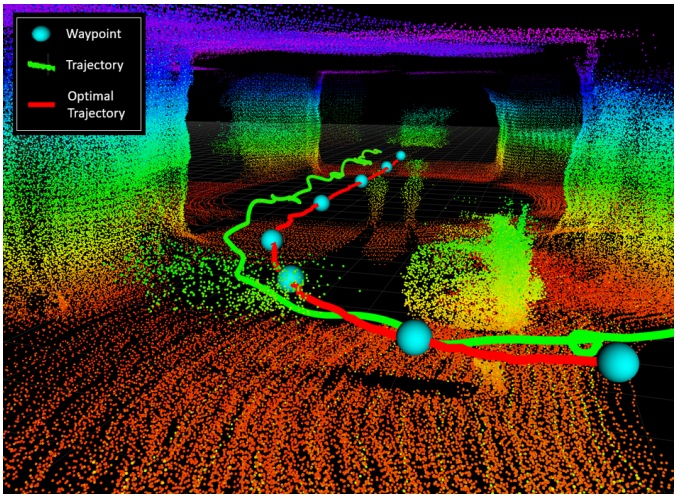


Figure 7.10: Underground mine environment section 1.

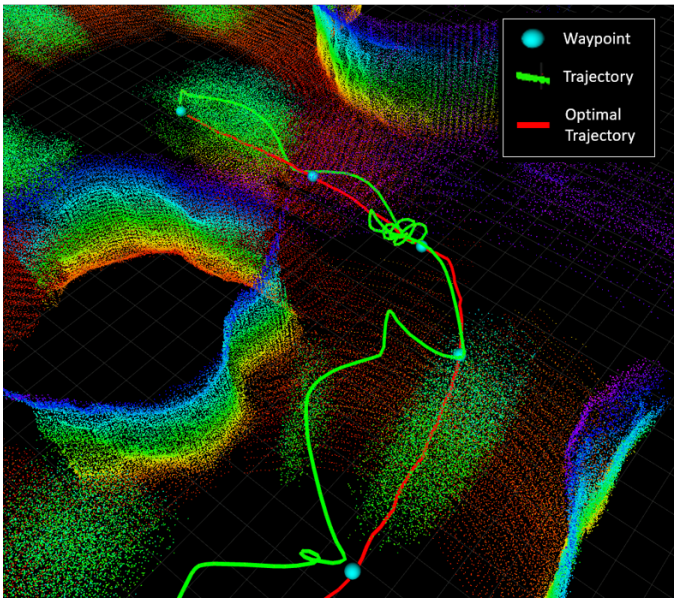


Figure 7.11: Underground mine environment section 2.

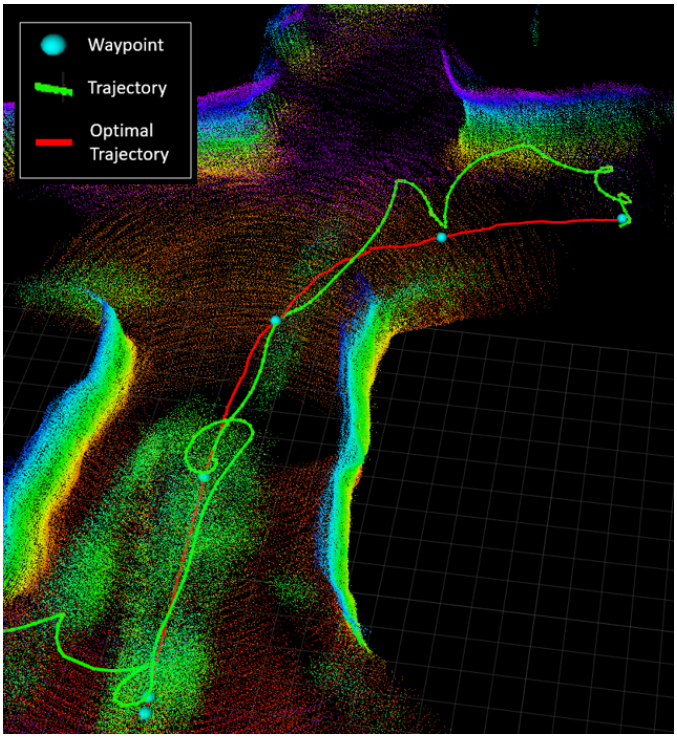


Figure 7.12: Underground mine environment section 3.

7.4 The Training Setup and Results with Tracking Solution

The following sections present the results where the RMF was trained with the reward from Equation (6.12), using the tracking reward r_{te} and the state vector $\boldsymbol{s} = [\boldsymbol{s}_{odom}^{\top}, s_{te}]^{\top}$.

The action $\boldsymbol{a}$ and the topology of the networks were the same as the one used in the obstacle avoidance solution. The RMF was trained in two sessions, both in open environments (Figure 6.9a). The tracking reward was turned off and set to $r_{te}= 0$ during the first training phase and turned on during the second. Throughout the training, the RMF was randomly initialized at different positions with the waypoint set out randomly with a max distance δ_{max} to the RMF. As the terminal conditions were fulfilled the RMF and the waypoint was reinitialized at new positions. It was specified that the RMF and the waypoint was not to be initialized in contact or below the floor.

At the very beginning, the networks were initialized with random seeds, and when training was done, the weights and biases in all the networks were transferred to the next training session. Only the hyperparameters and the parameters of the reward were changed between each environment.

7.4.1 Hyperparameters

The hyperparameters were defined as seen in Table 7.4. 1 epoch corresponds to 2000 time steps.

Parameters	Values	
	Session 1	Session 2
Epochs	750	500
Batch size	32	32
Actor learning rate ($\alpha_{\boldsymbol{\theta}}$)	1e-05	1e-05
Critic learning rate ($\alpha_{\boldsymbol{\phi}}$)	1e-05	1e-05
Discounting factor (γ)	0.99	0.99
Target network update rate (τ)	0.001	0.001
Max distance to waypoint (δ_{max})	4.0	4.0

Table 7.4: Specifications of the hyperparameters used in the DDPG algorithm during training.

The reward from Equation (6.12) was used throughout the two training sessions and the parameters of the reward were defined as seen in Table 7.5.

Parameters	Values	
	Session 1	Session 2
r_{goal}	10	10
$r_{collision}$	10	10
$r_{outside}$	0*	0.11
σ_{te}	0*	5
Radius of path δ_{track}	0*	1
Waypoint acceptance region (δ_r)	0.25	0.25
Max velocity at waypoint (δ_v)	0.3	0.3

Table 7.5: Specifications of the reward parameters used in the reward function r_t (Equation (6.12)) during training. (*r_{te} was set to 0 during the first training session. Hence, this parameter was not used)

The constant matrices $\boldsymbol{Q}$ and $\boldsymbol{R}$ were kept unchanged and were defined as in Equation (7.2).

$$\boldsymbol{Q} = diag[0.6, 0.6, 1.0, 0.03, 0.03, 0.05] \tag{7.2a}$$
$$\boldsymbol{R} = diag[0.001, 0.001, 0.001] \tag{7.2b}$$

7.4.2 Training Results

The training results from the two training sessions in the open environment (Figure 6.9a) can be seen in the following figures. The mean reward from each epoch can be seen in Figure 7.13.

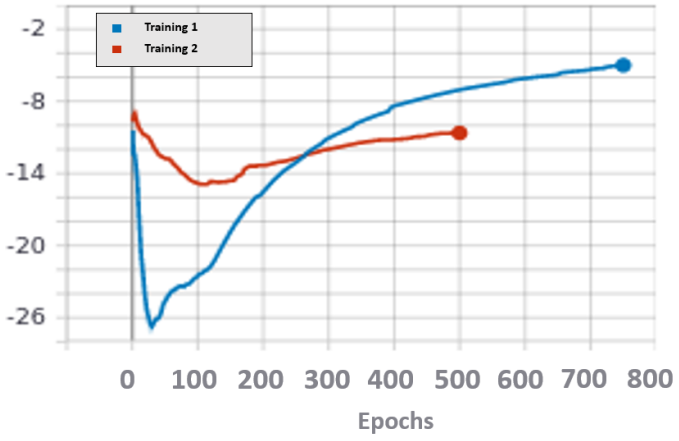
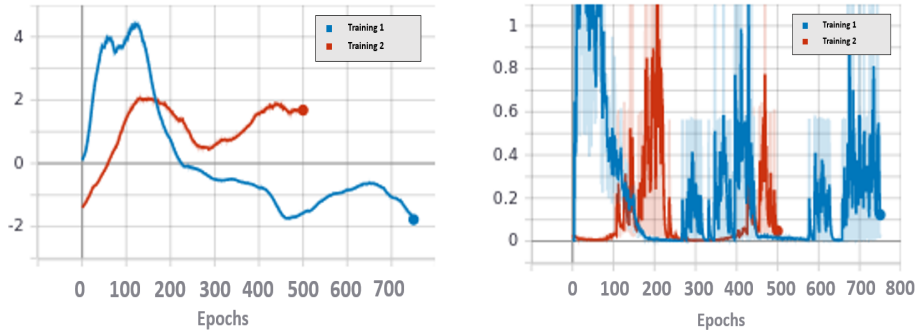


Figure 7.13: Simulation results of the mean reward return for each epoch.

The loss of the actor Equation (4.8) and the critic Equation (4.9) from the two training sessions can be seen in Figure 7.14.



(a) Simulation results of the mean loss return of actor for each epoch.

(b) Simulation results of the mean loss return of critic for each epoch.

Figure 7.14: The results from training depicting the loss of the actor and the critic.

Table 7.6 shows how long it took to train the agent in each training session. The total training time took 16 hours and 2 minutes.

	Session 1	Session 2
Training time	9h 7min	6h 55min

Table 7.6: Training time used in each training sessions.

7.5 Validating the Tracking Controller in a Simulated Environment

The finalized trained RMF was evaluated on the large path environment with obstacles (Figure 7.6a). The conditions for traversing through the environment were the same as the one used when testing the obstacle avoidance controller. All of the waypoints and the initial position of the RMF was at the same height $z = 3$. However, the tracking controller is not capable of avoiding obstacles, so the waypoints needed to be placed out, such that the trajectory was collision free. The result can be seen in Figure 7.15.

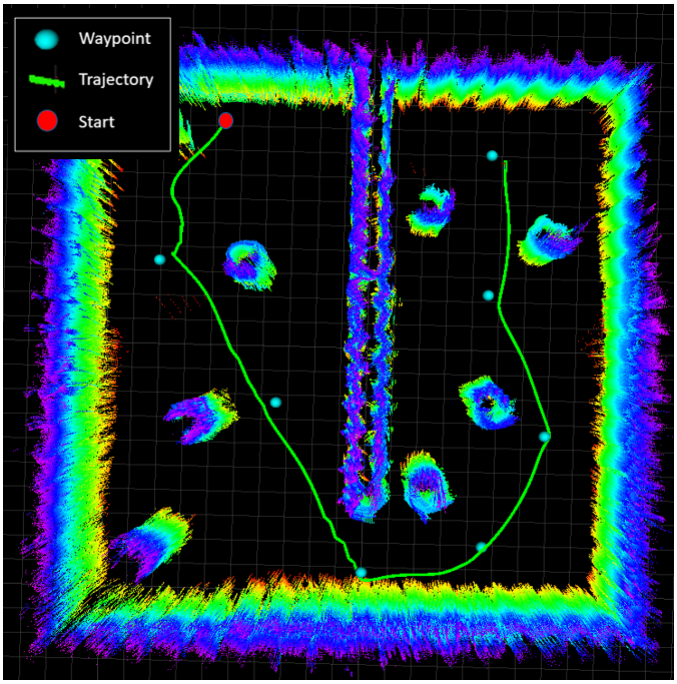


Figure 7.15: Visualization of the trajectory of the RMF within the obstacle-filled environment.

7.6 The Tracking Controller in an Underground Environment

In this section the results are presented utilizing the tracking controller by extracting the same data from the same dataset as when testing with the obstacle avoidance controller. However, the tracking controller does not use LiDAR data, and thus there is no need to filter out the environmental noise. Waypoints were set out along the extracted local reference path. The RMF used only the cyan waypoints and its own odometry data when performing navigation.

The trajectory of the RMF is seen in blue, and the results can be seen below. The RMF traversed through the mine starting from section 1 (Figure 7.16), to section 2 (Figure 7.17a), where it finished in section 3 (Figure 7.18).

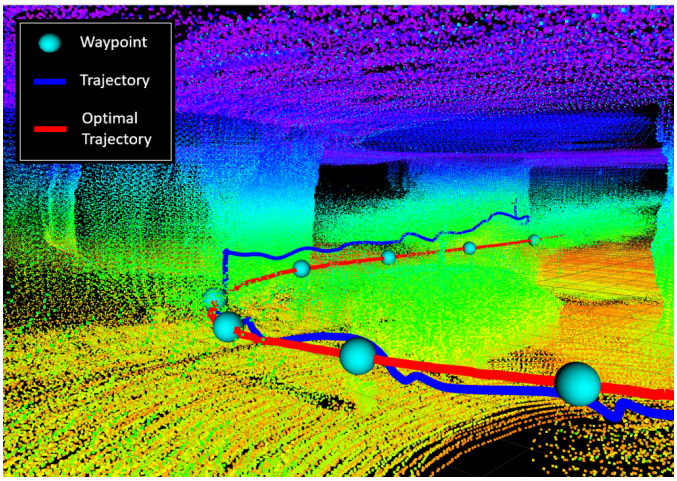
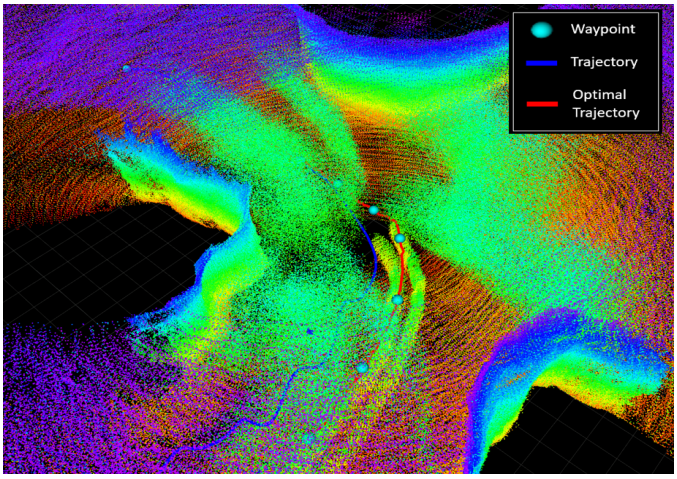
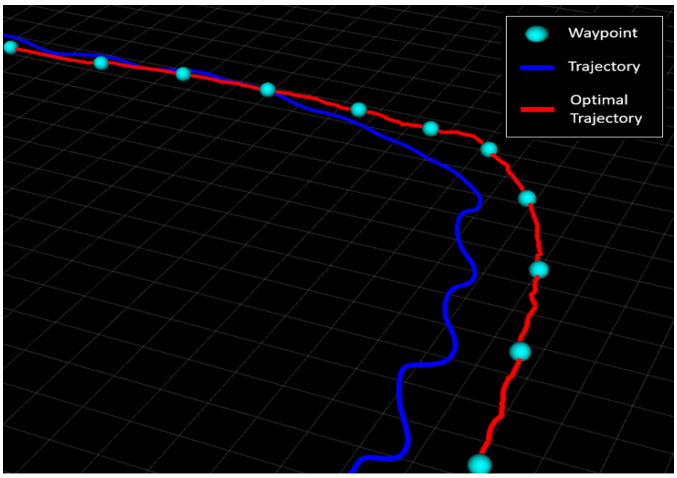


Figure 7.16: Visualization of the first section of the underground mine environment with the reference and RMF trajectory.



(a) Visualization of the second section of the underground mine environment with the reference and RMF trajectory.



(b) Visualization of the trajectory and the local reference path without any noisy LiDAR points in the second section of the underground mine environment.

Figure 7.17: Cave environment section 2.

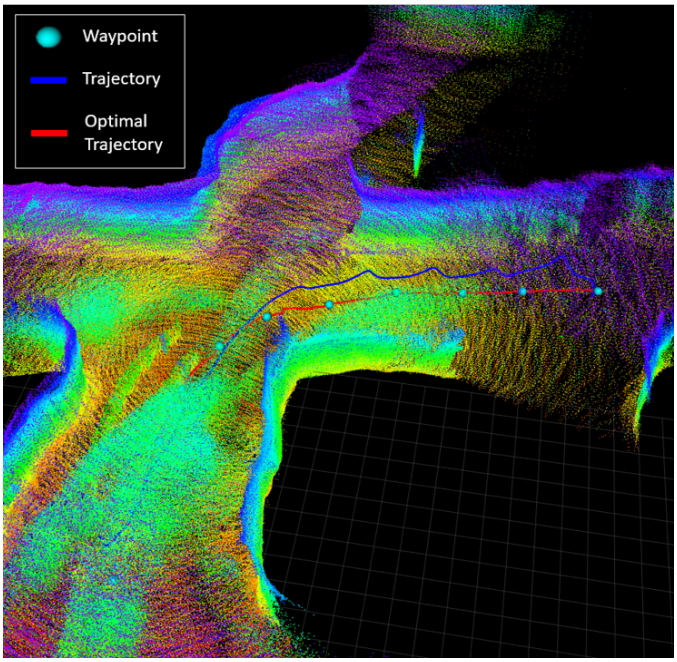


Figure 7.18: Visualization of the third section of the underground mine environment with the reference and RMF trajectory.

Chapter 8

Discussion

In this chapter the proposed approach from Chapter 6, the training results and the results from the tests of the controllers from Chapter 7 will be discussed.

8.1 The Reward Structure

One of the most influential parts of the RL system is the reward function. Having a well-shaped reward is the best way to reduce training time and maximize the probability of achieving set goals. Throughout the training process it has been observed that the reward influences the behavior of the RMF in many ways. For instance, negative rewards incentivize the RMF to get tasks done as quickly as possible. This is a desired behavior when the RMF should reach the final goal as quickly as possible. Nevertheless, the negative quadratic reward r_{quad} (Equation (6.8)) also has some limitations. When the RMF is far away from the goal, there are strong indications onto which direction it should go due to the high negativity of the reward. However, as it gets progressively closer to the waypoint, this negative reward gets quadratically smaller and almost approaches zero. Hence, the large positive r_{goal} reward was used to further encourage the RMF to reach the goal and its terminal state. Although, if this reward was set too high, the RMF would often overshoot, slightly oscillate, and then gradually stabilize at the goal region. This was also the only positive reward in the reward function. Introducing more positive rewards can yield local terminal states or other unsatisfactory behavior due to the nature of the policy selection. Waypoints that were set close to obstacles or the floor would often cause the RMF to collide as it approached the waypoints. Thus, the $r_{collision}$ reward was used to influence how conservative the RMF acted within the environment. If this reward was set very negative, the RMF would approach the goal very slowly or not at all. It was therefore important to balance the r_{goal} and $r_{collision}$.

In order to respond to situations where the RMF was approaching ob-

stacles, the r_{obst} (Equation (6.9)) was introduced with the aim of incentive a behavior that would push the RMF away from the obstacles similarly as in potential fields. Although, this introduces more nonlinearity to the overall reward function and could also lead to diminishing effects of rewards. If the reward from r_{obst} is highly negative the RMF will rather try to escape the environment instead of approaching the goal. Hence, r_{obst} was designed such that it would produce smaller negative rewards than r_{quad}. In addition, the escaping behavior was further suppressed by introducing walls around the environment, which was very beneficial during the training.

In addition, to increase the difficulty of the environments, one also increased the reward parameters as seen in Table 7.2. The σ_i were increased when more obstacles were introduced, such that the RMF learned that it should keep a greater distance to obstacles to further improve its response in more difficult environments. The region of acceptance δ_r parameter was increased to slack the navigation behavior such that RMF could easier focus on the obstacle avoidance behavior. The max distance δ_{max} to the waypoint was also increased between the environments as seen in Table 7.1, to further expose the RMF to more difficult scenarios.

The tracking controller was impelled through the reward r_{te} (Equation (6.11)) to follow the reference path. If the radius of the path δ_{track} was too small or too large, the RMF would end up slightly oscillating or not improve much when following the optimal path. Hence, it was important to find a proper radius such that the agent would improve during training. It was also important to not let r_{te} be too negative, as it would diminish all of the other rewards. From Figure 7.13 on can see that the general mean reward is reduced from the first to second training phase. This is probably due to r_{te} being introduced in the second phase, but the fact that the mean reward is not improving much during training suggests that the general reward structure is not tuned well or that one should consider another reward function.

The selection of the rewards and the reward parameters were selected on the basis of simple analysis of the finalized behavior of the agent, as well as trial and error. The reward structures were kept as simple as possible to avoid introducing more nonlinearity, unwanted minima states or other undesired behavior to the system. Nevertheless, other rewards structures could be considered, for instance one could shape r_{te} or r_{obst} in a quadratic manner. The navigation reward r_{quat} could be more adaptive and only give a reward when there was monotonic improvement when translating towards the goal. It is therefore important not to make any final conclusions.

8.2 The States

The observation space defines the information that is provided to the DDPG algorithm where states are mapped to actions, such that the RMF can safely navigate through the confined environment. The state space used in the proposed approach, the obstacle avoidance controller, can essentially be divided into two main components. The first component captures the information about the state of the RMF $\boldsymbol{s}_{odom}$, and the second one is related to the information about the environment $\boldsymbol{s}_{pc}$. The latter can genuinely be thought of as redundant during the training process in the first stage of training. This means that during the training process in the open environment the networks will learn that the information provided by these states should be associated with small weights in the networks when proposing actions. This was generally not a problem when the sparse distance feature vector was small. When this vector had more than around 15 distance measurements, it was hard to train and navigate in the second environment (Figure 6.9b), where obstacles were introduced, as collisions were imminent. In the open environment the $r_{obst} \approx 0$ almost everywhere except close to the floor. In order for the networks to not entirely neglect the distance features in the open environment, the RMF had to be exposed to situations such that the $r_{obst} < 0$. Consequently, the waypoints were purposely often set close to the floor during the training process in the open environment. The opposite effect could be seen when moving from an open environment to a confined one with too many random obstacles. If too many obstacles were introduced at the beginning of the training process, the distance states would be associated with too large weights, and the RMF would rather focus on the objective of avoiding the obstacles rather than approaching the goal region. Henceforth, it was necessary to gradually introduce more complicated environments to the RMF as it got progressively better in the simpler ones.

The state space related to the tracking controller was rather small and simple, making it easier to design the reward function, tune the parameters and faster to train. In the first phase of the training, the reward $r_{te} = 0$ was turned off. Then, after it had learned to navigate towards the goal in the open environment, the r_{te} was turned on in the second training phase. It was significantly faster to train and learn the agent to track the optimal path in this way, only focusing on one objective at a time opposed to having the reward $r_{te} \neq 0$ turned on from the beginning.

8.3 The Feature Extraction Pipeline of the Obstacle Avoidance Controller

The computation of the sparse feature vector used by the obstacle avoidance controller is a low-cost solution and gives a very explicit representation of the

RMF state relatively to its goal and the volumetric space of the point cloud. This in turn makes it easier to design the reward function. This approach also reduces the gap between the virtual and real environments as it uses non-abstracted observations. As one can see from the results in Figure 7.3b and Figure 7.4b, the RMF is able to safely traverse through the narrow, long and twisty passages only relying on the onboard sensors without any prior data or construction of an online 3D map. In addition, the RMF is capable of staying within the middle of the passage even when it goes diagonally. When the RMF is exposed to intersections where the next waypoint is set out deeper into the passage, such as the one seen in Figure 7.5, it is able to go towards the right direction relatively effectively. The general solution also works well when introducing obstacles to a confined environment as seen in Figure 7.6b, Figure 7.7b and Figure 7.9. The RMF was mostly trained in environments that only vary in the general xy-direction. It was significantly harder to train the agent in environments where obstacles varied in all three dimensions as in Figure 6.9d. This can be seen from Figure 7.1, where there is a significant improvement to the mean reward from environment 1 to environment 2 to environment 3 even though r_{goal} and $r_{collision}$ was increased with 10 between the 3 environments (Table 7.2). Nevertheless, even if there was a reduction in the mean reward in environment 4 (Figure 6.9d), the reward was still positive and it is possible to see the effect of having divided the sectors into stacks in Figure 7.8b. The RMF is able to traverse under the u-shape and over the t-shape without much struggle. This would not be possible if the point cloud would only be divided in sectors. The feature extraction pipeline works concurrently with the rest of the system which further improves the computational time, so that the inference on board of the RMF takes less than 150 milliseconds.

The proposed approach comes at the cost of disregarding a lot of sensory information. For instance, if there are objects relatively close to the RMF occupying each sector and stack, the RMF will register the entire space around it as occupied, even if there is a lot of space between each object. This can be solved by increasing the radial resolution of the sectors. However, the benefit of such minimalistic approach in terms of sensing and processing is that it is transferable and can be applied with other LiDARs. The agent was trained with the OS-1 simulator, but there was no extra work in combining the pipeline extraction process with the Velodyne PuckLITE LiDAR data from the underground mine dataset. More importantly, it is not bounded by any expensive high resolution 3D LiDAR sensors. In fact, one could even combine the proposed method with a much cheaper and lightweight system consisting of few 1D LiDAR sensors.

One does not necessarily have to increase the resolution of the sectors or the stacks to be able to move in more dense environments. The extracted sparse distances are coming from a very homogeneous divided point cloud. The approach could be more adaptive such that it can allocate more stacks

or sectors to areas with finer detail. For instance, in the collision-free test environments (Section 7.2.1), it could be more beneficial to allocate more sectors to the sides of the RMF.

8.4 Reliability

The tracking controller only relies on the odometry data. This significantly affects the reliability of the controller, as there is no coupling between the sensing and the odometry state. This can be problematic if the system does not sufficiently register all obstacles and the path planner generates paths that are obstacle-filled.

The obstacle avoidance controller can be thought of as more reliable compared to the tracking controllers as it is capable of avoiding simple obstacles. Nevertheless, the controller can struggle if the obstacles are large or if the waypoint is far out of the line-of-sight. This is generally due to the limitations within the reward function as discussed in Section 6.5. However, such challenges should be handled by a global planner and not by the local solution that the proposed method provides. In addition, the RMF does not always keep a great distance to all obstacles. This can be a problem if the environments become even more dense and can be improved upon by increasing the resolution of the sectors.

The general behavior that the agent has learned after being trained in the four environments is to simply move in the other direction when approaching an obstacle. This learned behavior can generate oscillations in the trajectory if there are a lot of obstacles nearby the RMF. The environments used during training are also wider compared to some of the test environments (Section 7.2.1), and it is possible to see some oscillations in the trajectory of the RMF in these narrow passage environments. The RMF was never exposed to such narrow situations during training, and it would not be possible to traverse through even narrower passages due to the oscillations. A solution to this would probably be to train in environments that look more similar to the ones in Section 7.2.1. Nevertheless, this also highlight one of the many strengths of using machine learning. The controller is able to generalize to many environments without any tuning, even if it has not been exposed to all the different scenarios.

8.4.1 Consistency Challenges

There were also some consistency issues with both controllers. In Figure 7.3b one can see that some extra waypoints were needed to go around one of the u-turns. In Figure 7.5 one can see that the RMF is able to better traverse through the y-path in the up-direction compared to the down-direction. In Figure 7.17b one can see some slight oscillation when the RMF is moving in one particular direction. The problems with using machine learning

compared to classical approaches is that it is very hard to analyze the final results. Such inconsistent behavior may be due to some bias during training, meaning that the RMF was exposed to certain scenarios more than others and could be solved with more training.

8.4.2 Sensory Inputs

In the simulator, odometry data came at the rate of 100 Hz and LiDAR data came at the frequency of 10 Hz. In addition, there is a difference in the time used when extracting the odometry and LiDAR measurements. This suggests that there could be small inconsistencies between the odometry and the distance states when moving through the environment. However, this inconsistency is generally thought of as very small as the feature extraction pipeline is very fast and given the rate of the inputs this issue could be neglected.

8.5 Comparing the Obstacle Avoidance Controller to the Tracking Controller

The tracking controller was simpler by design and it was assumed that the path between waypoints were collision-free. With this assumption it was not difficult to traverse through simpler environments as seen in Figure 7.15. In general, the controller mimics a simpler control strategy such as a line-of-sight guidance law [12]. The tracking controller struggled a lot in the more narrower environments, as it would most certainly crash in the steep turns. This is where the strengths of the collision avoidance controller becomes apparent. The reward from r_{obst} will naturally push the RMF to the center of the path and follow the optimal reference even in steep turns if the passage is narrow. In addition, waypoints would in practice be placed closer than 3-4 meters apart in such confined environments, which would improve the response of the tracking controller. However, with the collision avoidance controller there are less strict rules to where these waypoints need to be placed compared to most classical approaches and the tracking controller. Consequently, the obstacle avoidance controller can solve a broader set of tasks compared to the tracking controller, and it shows the real strengths of using machine learning to solve such tasks. In addition, checking if the path is collision free is a very computational expensive task. By using the obstacle avoidance controller this task can be reduced.

The proposed method also only relies on LiDAR and odometry data. In the underground mine environments, there is a lot of environmental noise such as mist, even after doing some filtering. One of the main drawbacks of the proposed methods is that it strictly relies on the raw LiDAR points, and this can affect the solution negatively if one cannot filter out all noisy

points. This does affect the trajectory of the RMF as seen in Figure 7.10, Figure 7.11 and Figure 7.12. In the first section of the mine, most of the noisy points were filtered out and the trajectory is able to follow the reference path quite well. In the second and third section of the mine more noisy points are present and the trajectory of the RMF tends to diverge a lot from the reference path.

8.5.1 The Filtering Process

One of the challenges related to point cloud filtering is removing noise without also removing other environmental features. The SOR algorithm is simple, but is limited in terms of speed and accuracy. Generally, the greater the distance from the RMF to a point was, the higher the probability of that point being removed as an outlier. This was not a problem, as the solution only cares about nearby points, and allowed for aggressive filtering by setting the threshold n quite low. However, this was still not sufficient enough, and other approaches and methods should be considered [29].

The tracking controller does not rely on any LiDAR points and there is therefore no need to filter them out. It generally performs better compared to the collision avoidance controller in following the reference as seen in Figure 7.16, Figure 7.17 and Figure 7.18. Consequently, one would need to make the collision avoidance controller more robust by for instance training it with noisy distance measurements and maybe adding additional states that can describe the noisy points, such as the intensity, to further improve the training.

8.6 Comparing the Obstacle Avoidance Controller to Sampling-based Methods

When conducting navigation and exploration in geometrically-constrained environments a state-of-the-art sampling-based method uses
high-dimensionality sensors to map the environment. Paths are found in the online constructed map by progressively expanding a graph outwards in random directions until an input query can be solved. In addition, the paths between the nodes in the graph are checked to be feasible and collision-free. Suitable control commands are found such that the aerial robot moves along the collision-free paths.

The proposed method solves the navigation task though and end-to-end approach, and omits the need to reconstruct a real-time map. It maps compressed sensor data to control actions. In addition, the method can be further combined with a coarse planner providing sparse waypoints. In general, the proposed method is extremely minimalistic in terms of sensing and processing for the task of collision avoidance. In general, reducing expen-

sive computational processes is what guided the selection of such a small number of inputs from the LiDAR. In addition, a RL navigation policy for collision-free flight require only minimalistic sensor input and computational resources. Using more data would lead to an improved result, but also make it more expensive to compute. Hence, the contributions and focus within this project lied in compressing suitable data such that a RL could learn a policy for collision-avoidance and navigation.

8.7 Challenges with Reinforcement Learning

The general problem with using machine learning for control is that it is very hard to validate, and the proof of stability is absent. This can be problematic if one aims to create a stable and reliable system that should work in critical situations. This is very much in contrast to standard control systems, where consistency and stability can be shown. Therefore, it is necessary to emphasize the importance of testing controllers that rely on machine learning as one attempted to do in Chapter 7. In addition, neural networks should be treated very much like a "black box", and one should therefore also be careful in making any conclusions about the system.

8.7.1 Challenges with the DDPG Algorithm

One of the main drawbacks of the DDPG algorithm and generally most reinforcement learning algorithms is the sensitivity to hyperparameter tuning and initialization. There are quite a few parameters that need to be set in the DDPG algorithm as seen in Table 7.1 and it can be generally difficult to configure these parameters such that the algorithm converges to something satisfactory. The DDPG algorithm is also not guaranteed to converge during training and can be generally thought of as somewhat unstable. This is because when using a replay buffer, the samples from the buffer are generally close enough to the current policy because the buffer is evolving with the policy. Hence, if the replay buffer is starting to be filled with many poor samples, the algorithm will fail. Choosing a step size is quite crucial in the algorithm, as the policy update should not move too far from the old policy. To ensure that this is the case, one should consider to include a constraint that limits the step size. PPO is an algorithm that does this. One could consider using this or something analogous, as it is also more sample efficient and easier to tune [32].

8.8 Networks

In order to make the proposed method more stable and improve the robustness in the policy selection during training the networks where slightly

overparameterized with the number of neurons in each layer. Although, one did intentionally left the networks being rather shallow to further reduce computation.

From Figure 7.2 one can deduce somewhat convergence in the networks in the first 2 environments. In the more complex environments this convergence is not present. This could imply that more time should be spent training in the more complex environments. This can also be seen from Figure 7.1 as the mean rewards from the last two environments have not converged yet. From Figure 7.14 it can be hard to conclude anything. However, the loss of the critic (Figure 7.14b) depicts oscillations in both training phases. Such oscillations may be due to not reaching optimal convergence as discussed earlier when interpreting Figure 7.13, and suggests that one should consider further tune the hyperparameters of the networks and the reward.

8.9 Other Improvements

When gradually introducing more sparse distance states, it was more difficult for the networks to learn the simple objective of navigating to one waypoint in the open environment. Larger state space implies more information has to be learned, and this can generally pose a problem if too many states are introduced. During the initial training process the DDPG algorithm was initialized with random seeds, but if one introduces more states imitation learning and similar approaches could give faster solutions that emulate expected and known behavior at the beginning. One should not need to have the system learn from scratch, and rather employ techniques and capabilities that enable the RMF to learn faster. The algorithm could require thousands of simulations before it is able to converge to an optimal policy, but by utilizing imitation learning, the algorithm could see qualitatively better performance in much shorter time.

Chapter 9

Conclusion

9.1 Overview

In this thesis a learning-based approach for fast navigation within confined environments was presented, relying only on onboard sensors without any prior knowledge about the environment nor any online map construction. The proposed method was then tested and evaluated in a diverse set of environments and in an underground mine structure. In order to develop this method a proposed end-to-end learning strategy was used relying on a reinforcement learning framework. The main contributions lay within the development of the compressed states and evaluating the RL navigation policy for collision-free flight. The reinforcement learning approach displays great robustness to confined, complex environments. It is also generalized to a large number of different environments and can be further supported by a path planner such as demonstrated in this thesis. The method presented in this thesis also utilizes low computational strategies, making it attractive to apply onto real quadcopters used for time sensitive search and rescue operations. The results also suggest that an extremely lower-cost and lightweight system consisting of only a few 1D LiDAR sensors could replace the 3D LiDAR sensor to solve the collision-avoidance problem.

9.2 Further Work

For future work it would be beneficial to further improve upon the current solution as discussed in Section 8.9.

The next step in the development process would be to apply the proposed method onto a real quadcopter. In order to migrate the solution from simulation to real-world one would need to integrate the reinforcement learning based navigation solution with the other components of the system. This means that one would need to take into account the uncertainties in the estimates provided by the other components. This can for instance be done

by training the reinforcement learning algorithm with noisy odometry and LiDAR data.

Another direction in the development process would be to also further integrate the path planner module into the reinforcement learning formulation. Hence, the path planner would also solely relay on raw LiDAR and odometry data without any construction of an online 3D map.

Appendix A

A.1 Deep Deterministic Policy Gradient Algorithm

Algorithm 1 Deep Deterministic Policy Gradient [26]

Randomly initialize actor and critic weights $\boldsymbol{\theta}$ and $\boldsymbol{\phi}$.
Set target networks with parameters $\boldsymbol{\theta}' \leftarrow \boldsymbol{\theta}$ and $\boldsymbol{\phi}' \leftarrow \boldsymbol{\phi}$.
for each epoch **do**
  Receive initial observation state s.
  Select and execute action based on current policy
  a = clip($\pi_{\boldsymbol{\theta}}$(a|s) + ϵ, a_{Low}, a_{High}), $\epsilon \sim \mathcal{N}$.
  Observe s_{next}, r, d and see if s_{next} is terminal.
  Store tuple (s, a, r, s_{next}, d) in the replay buffer.
  If s_{next} is terminal, reset environment state.
  if we should update **then**
    for each update **do**
      Sample N transitions from replay buffer.
      Compute target value

$$y = r + \gamma(1 - d)Q_{\boldsymbol{\phi}'_{next}}(s_{next}, \pi_{\boldsymbol{\theta}'_{next}}(a_{next}|s_{next})).$$

      Update critic

$$\nabla_{\boldsymbol{\phi}} \frac{1}{N} \sum_{(s,a,r,s_{next},d) \in N} (Q_{\boldsymbol{\phi}}(s, a) - y)^2.$$

      Update actor

$$\nabla_{\boldsymbol{\theta}} \frac{1}{N} \sum_{s \in N} Q_{\boldsymbol{\phi}}(s, \pi_{\boldsymbol{\theta}}(a|s)).$$

      Update target networks

$$\theta' \leftarrow (1 - \tau)\theta' + \tau\theta,$$
$$\phi' \leftarrow (1 - \tau)\phi' + \tau\phi.$$

    end for
  end if
end for

Appendix B

B.1 ARL-NTNU Computer Specifications

Type	Specifications
Operating System (OS)	Ubuntu 18.04.5 LTS
Processor	AMD Ryzen threadripper 3970x 32-core processor x 64
Graphics	NVIDIA GeForce RTX 3090/PCIe/SSE2
OS type	64-bit

Table B.1: Computer Specifications of the ARL-NTNU computer.

B.2 NTNU Computer Specifications

Type	Specifications
Operating System (OS)	Ubuntu 18.04.5 LTS
Processor	Intel Core i7-8700 CPU 3.20GHz x 12
Graphics	Intel UHD Graphics 630 (CFL GT2)
OS type	64-bit

Table B.2: Computer Specifications of the provided NTNU computer.

Bibliography

[1] Song Ho Ahn. *OpenGL Sphere Documentation*. (Accessed last: 16.05.2021). URL: http://www.songho.ca/opengl/gl_sphere.html.

[2] D.P. Bertsekas. *Reinforcement Learning and Optimal Control*. Athena Scientific optimization and computation series. Athena Scientific, 2019. URL: https://books.google.no/books?id=ZlBIyQEACAAJ.

[3] DARPA SubT Challenge. *SubT Tech Repo*. (Accessed last: 11.05.2021). URL: https://www.subtchallenge.world/openrobotics/fuel/collections/SubT%5C%20Tech%5C%20Repo.

[4] Tung Dang et al. "Field-hardened robotic autonomy for subterranean exploration." In: *Field and Service Robotics (FSR)* (2019).

[5] Tung Dang et al. "Graph-based Path Planning for Autonomous Robotic Exploration in Subterranean Environments." In: *2019 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. 2019, pp. 3105–3112.

[6] Paolo De Petris et al. "Collision-tolerant Autonomous Navigation through Manhole-sized Confined Environments." In: *2020 IEEE International Symposium on Safety, Security, and Rescue Robotics (SSRR)*. 2020, pp. 84–89.

[7] Digikey. *IMU ADIS16460*. (Accessed last: 10.06.2021. URL: https://www.digikey.com/en/products/detail/analog-devices-inc/ADIS16460AMLZ/5957823.

[8] Diy Drones. *PIXHAWK pxIMU*. (Accessed last: 10.06.2021. URL: https://diydrones.com/profiles/blogs/pixhawk-pximu-available.

[9] M. Achtelik F. Furrer M. Burri and R. Siegwart. "Robot Operating System (ROS): The Complete Reference (Volume 1)." In: (2016). Ed. by Anis Koubaa, pp. 595–625. URL: http://dx.doi.org/10.1007/978-3-319-26054-9_23.

[10] Davide Falanga et al. "PAMPC: Perception-Aware Model Predictive Control for Quadrotors." In: *CoRR* abs/1804.04811 (2018). arXiv: 1804.04811. URL: http://arxiv.org/abs/1804.04811.

[11] J. A. Fijalkowski. *Data-based Collision Resilient Navigation for Aerial Robots in Open Environments*. 2020.

[12] T.I Fossen. *Handbook of Marine Craft Hydrodynamics and Motion Control*. John Wiley & Sons, 2021.

[13] Fadri Furrer et al. "RotorS – A Modular Gazebo MAV Simulator Framework." In: vol. 625. Jan. 2016, pp. 595–625.

[14] Gazebo. *Contact Sensor*. (Accessed last: 29.05.2021). URL: http://gazebosim.org/tutorials?tut=contact_sensor&cat=sensors.

[15] Peter Henderson et al. "Deep Reinforcement Learning that Matters." In: *CoRR* abs/1709.06560 (2017). arXiv: 1709.06560. URL: http://arxiv.org/abs/1709.06560.

[16] Sertac Karaman and Emilio Frazzoli. *Incremental Sampling-based Algorithms for Optimal Motion Planning*. 2010. arXiv: 1005.0416 [cs.RO].

[17] Diederik P. Kingma and Jimmy Ba. *Adam: A Method for Stochastic Optimization*. 2014. URL: http://arxiv.org/abs/1412.6980.

[18] Steven LaValle and James Kuffner. "Randomized Kinodynamic Planning." In: *I. J. Robotic Res.* 20 (Jan. 2001), pp. 378–400.

[19] Timothy P. Lillicrap et al. "Continuous control with deep reinforcement learning." In: *ICLR*. Ed. by Yoshua Bengio and Yann LeCun. 2016. URL: http://dblp.uni-trier.de/db/conf/iclr/iclr2016.html#LillicrapHPHETS15.

[20] Antonio Loquercio et al. "DroNet: Learning to Fly by Driving." In: *IEEE Robotics and Automation Letters* 3.2 (2018), pp. 1088–1095.

[21] Hamid Reza Maei et al. "Toward Off-Policy Learning Control with Function Approximation." In: *Proceedings of the 27th International Conference on International Conference on Machine Learning*. ICML'10. Haifa, Israel: Omnipress, 2010, pp. 719–726.

[22] Volodymyr Mnih et al. "Playing Atari with Deep Reinforcement Learning." In: *CoRR* abs/1312.5602 (2013). arXiv: 1312.5602. URL: http://arxiv.org/abs/1312.5602.

[23] Society of Naval Architects, Marine Engineers (U.S.). Technical, and Research Committee. Hydrodynamics Subcommittee. *Nomenclature for Treating the Motion of a Submerged Body Through a Fluid: Report of the American Towing Tank Conference*. Technical and research bulletin. Society of Naval Architects and Marine Engineers, 1950. URL: https://books.google.no/books?id=VqNFGwAACAAJ.

[24] J. Nocedal and S. J. Wright. *Numerical Optimization*. second. Springer, 2006.

[25] OpenAI. *OpenAI Gym*. (Accessed last: 19.03.2021. URL: https://gym.openai.com/.

[26] OpenAI. *Spinning Up OpenAI Gym - Deep Deterministic Policy Gradient*. (Accessed last: 11.05.2021). URL: https://spinningup.openai.com/en/latest/algorithms/ddpg.html.

[27] Ouster. *Ouster OS1*. (Accessed last: 10.06.2021. URL: https://ouster.com/products/os1-lidar-sensor/.

[28] Ouster. *Ouster OS2*. (Accessed last: 10.06.2021. URL: https://ouster.com/products/os2-lidar-sensor/.

[29] Ji-Il Park, Jihyuk Park, and Kyung-Soo Kim. "Fast and Accurate Desnowing Algorithm for LiDAR Point Clouds." In: *IEEE Access* 8 (2020), pp. 160202–160212.

[30] Russell Reinhart et al. "Learning-based Path Planning for Autonomous Exploration of Subterranean Environments." In: *2020 IEEE International Conference on Robotics and Automation (ICRA)*. 2020, pp. 1215–1221.

[31] Craig Reynolds. "Steering Behaviors For Autonomous Characters." In: (June 2002).

[32] John Schulman et al. "Proximal Policy Optimization Algorithms." In: *CoRR* abs/1707.06347 (2017). arXiv: 1707.06347. URL: http://arxiv.org/abs/1707.06347.

[33] John Schulman et al. "Trust Region Policy Optimization." In: *CoRR* abs/1502.05477 (2015). arXiv: 1502.05477. URL: http://arxiv.org/abs/1502.05477.

[34] Seeed Studio. *NVIDIA Jetson TX2-modul*. (Accessed last: 10.06.2021. URL: https://www.elfadistrelec.no/no/nvidia-jetson-tx2-modul-seeed-studio-102110402/p/30176873.

[35] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. Cambridge, MA, USA: A Bradford Book, 2018.

[36] Google Tensorflow. *Tensorflow*. (Accessed last: 19.11.2020). URL: https://www.tensorflow.org/.

[37] Velodyne. *Velodyne LiDAR*. (Accessed last: 10.06.2021. URL: https://velodynelidar.com/products/puck-lite/.

[38] Matrix Vision. *USB 2.0 board-level camera - mvBlueFOX-MLC*. (Accessed last: 10.06.2021. URL: https://www.matrix-vision.com/USB2.0-single-board-camera-mvbluefox-mlc.html.

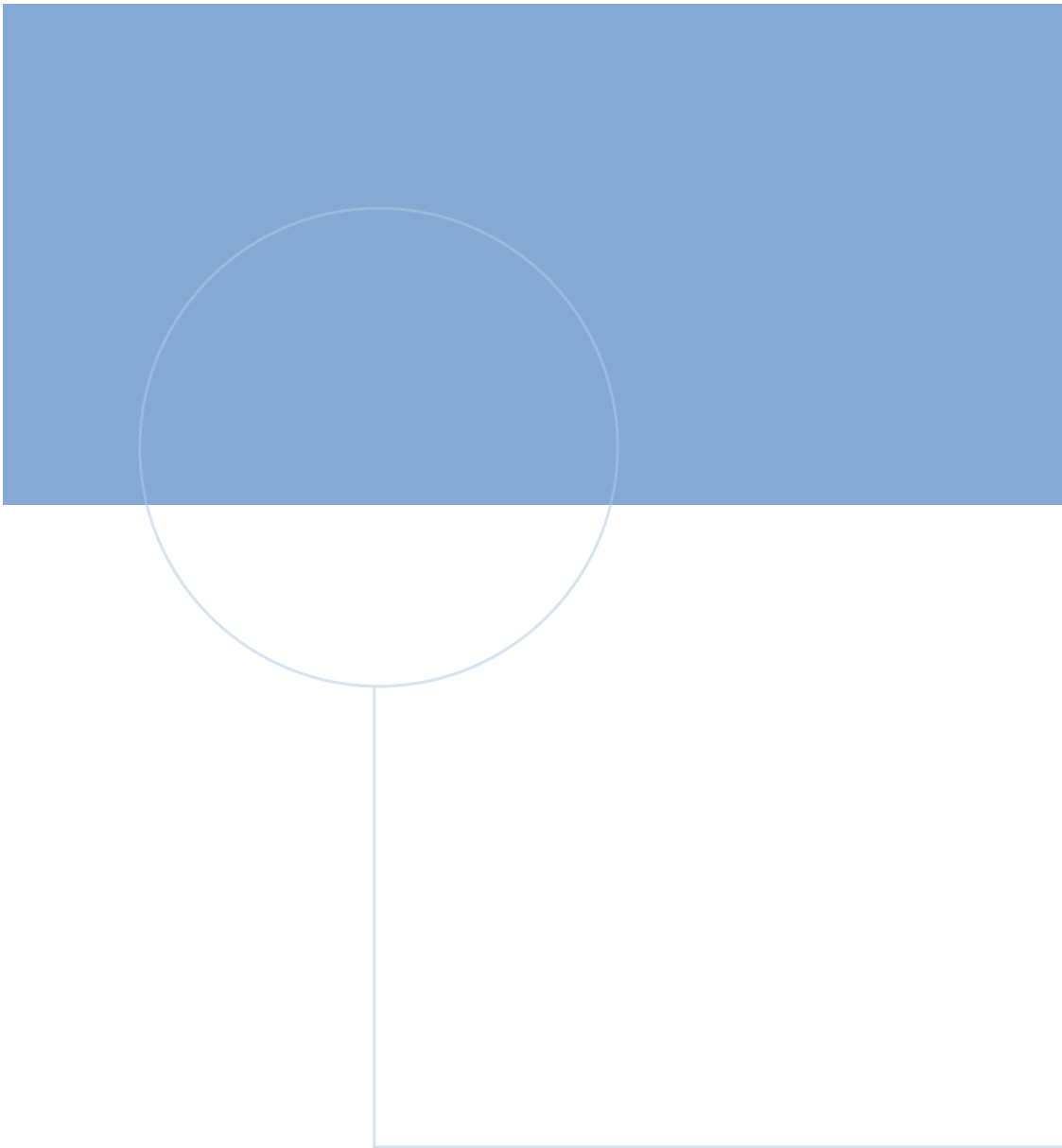
[39] Ziyu Wang et al. "Sample Efficient Actor-Critic with Experience Replay." In: *CoRR* abs/1611.01224 (2016). arXiv: 1611.01224. URL: http://arxiv.org/abs/1611.01224.

[40] Yuhuai Wu et al. “Scalable trust-region method for deep reinforcement learning using Kronecker-factored approximation.” In: *CoRR* abs/1708.05144 (2017). arXiv: 1708.05144. URL: http://arxiv.org/abs/1708.05144.

[41] Nguyen Xuan-Mung and Sung Kyung Hong. “Robust Backstepping Trajectory Tracking Control of a Quadrotor with Input Saturation via Extended State Observer.” In: *Applied Sciences* 9.23 (2019). URL: https://www.mdpi.com/2076-3417/9/23/5184.

[42] Brady Zhou, Philipp Krähenbühl, and Vladlen Koltun. “Does computer vision matter for action?” In: *CoRR* abs/1905.12887 (2019). arXiv: 1905.12887. URL: http://arxiv.org/abs/1905.12887.

[43] Qian-Yi Zhou, Jaesik Park, and Vladlen Koltun. “Open3D: A Modern Library for 3D Data Processing.” In: *arXiv:1801.09847* (2018).



NTNU

Norwegian University of
Science and Technology