Master's thesis

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

Henrik Fjellheim

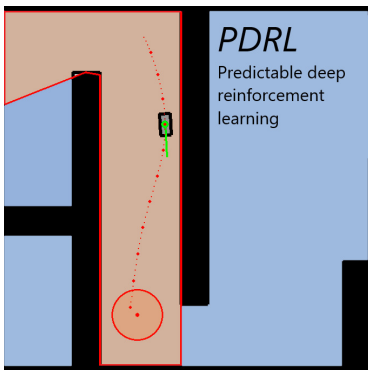
Short-Term Trajectory Planning for a Non-Holonomic Robot Car: Utilizing Reinforcement Learning in conjunction with a Predefined Vehicle Model

Master's thesis in Computer science
Supervisor: Rudolf Mester
June 2023

NTNU
Norwegian University of
Science and Technology

Henrik Fjellheim

Short-Term Trajectory Planning for a Non-Holonomic Robot Car: Utilizing Reinforcement Learning in conjunction with a Predefined Vehicle Model



Master's thesis in Computer science
Supervisor: Rudolf Mester
June 2023

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

NTNU
Norwegian University of
Science and Technology

Abstract

The use of deep reinforcement learning (DRL) for autonomous vehicles is a hot topic in the autonomous driving industry. Many autonomous vehicular systems rely in part or entirely on DRL to solve their tasks. These tasks range from warehouse work inside locked containers to driving on public roads and interacting with humans. A limitation of many of the most promising DRL methods is that they are structured in such a way that they will only give a single action/output for a single situation/state, making them hard to predict and potentially hazardous. DRL relies on deep neural networks, both to understand the dynamics of the environment as well as to make optimal decisions. These networks are sub-symbolic and not very transparent, so understanding why an action was taken and how to avoid it happening again is not trivial.

In this thesis, a proposed architecture aims to make any implicit DRL method predictable and capable of planning. The architecture, called Predictable-DRL (PDRL), combines ideas from DRL with the architecture of a model-predictive controller (MPC) from control theory to create a model-based reinforcement learner (MBRL). A DRL method will be used to take optimal actions, that the architecture will construct into a trajectory. Similar to other MBRLs, PDRL explicitly maintains and updates a transition model of the system. This model is used for projecting and predicting future input data and for the planning. The PDRL is developed with a specific non-holonomic robotic vehicle in mind, the **LIMO** by AgileX, so the first step is to create a model of this system, which will be provided to PDRL as a starting point.

The new architecture was tested and compared to its straightforward DRL counterpart, demonstrating a reduction in collisions during both training and navigation tasks. Similar to other MBRL methods, the score achieved by the new architecture is relatively lower compared to model-free counterparts, and it is also more susceptible to system disturbances.

A key advantage of the proposed architecture, similar to other MBRLs, lies in its ability to anticipate and "hallucinate" future states, enabling the agent to learn from potential collisions without actually experiencing them. This feature greatly facilitates online learning. Furthermore, the architecture exhibits potential for effective communication of intentions in multi-agent environments. With further development, the agent could be extended to adhere to traffic rules, such as the right-hand rule, when interacting with other agents during driving scenarios.

Take a look at the youtube-channel connected to this thesis, to better understand what the environment and agent looks like, before reading the thesis: https://www.youtube.com/@PDRLMasterThesis.

Sammendrag

Bruken av dyp forsterkende læring (DRL) for autonome / selvkjørende kjøretøy er et høyaktuelt tema i bilindustrien. Mange autonome kjøretøysystemer støtter seg delvis eller helt på DRL for å løse oppgavene sine. Disse oppgavene spenner fra lagerarbeid inne i låste containere til kjøring på offentlige veier og interaksjon og sammarbeid med mennesker. En begrensning ved mange av de mest lovende DRL-metodene er at de er strukturert på en slik måte at de bare gir én enkelt handling for en enkelt situasjon, noe som gjør dem vanskelige å forutsi og potensielt farlige. DRL baserer seg også på dype nevrale nettverk, både for å forstå dynamikken i omgivelsene og for å ta optimale beslutninger. Disse nettverkene er sub-symbolske og ikke veldig transparente, så å forstå hvorfor en handling ble tatt og hvordan man kan unngå at det skjer igjen, er ikke trivielt.

I denne masteravhandlingen blir det foreslått en arkitektur som tar sikte på å gjøre enhver (implisitt) DRL-metode forutsigbar og i stand til å planlegge. Arkitekturen, kalt Predictable-DRL (PDRL), kombinerer ideer fra DRL med arkitekturen til en modellprediktiv regulator (MPC) fra kontrollteori for å skape en modellbasert forsterkningslærer (MBRL). En DRL-metode vil bli brukt til å ta optimale handlinger, som arkitekturen vil konstruere til en planlagt bane for roboten. På samme måte som andre MBRL-er, vedlikeholder PDRL en matematisk modell av dynamikken til omgivelsene sine, som den oppdaterer jevnlig. Denne modellen brukes til å projisere og forutsi fremtidige situasjoner, som dermed muligjør planlegging. PDRL er utviklet med en bestemt ikke-holonomt robotisk kjøretøy i tankene, nemlig **LIMO** av AgileX, så det første steget i utviklingen av PDRL er å opprette en matematisk modell av dette systemet, som vil bli gitt til PDRL som et utgangspunkt.

Den nye arkitekturen ble testet og sammenlignet med sin direkte DRL-motpart, og viste en reduksjon i kollisjonsrate både under trening og gjennom navigasjonsoppgaver. På samme måte som andre MBRL-metoder, er den oppnådde scoren til den nye arkitekturen relativt lavere sammenlignet med modellfrie motparter, og den er også mer sårbar for systemforstyrrelser.

En nøkkelfordel med den foreslåtte arkitekturen, på linje med andre MBRL-er, ligger i dens evne til å forutse og "hallusinere" fremtidige tilstander, noe som gjør det mulig for agenten å lære fra potensielle kollisjoner uten å faktisk oppleve dem. Denne funksjonen legger til rette for online læring, altså læring etter utplassering i agentens operative miljø. Videre har arkitekturen potensial for effektiv kommunikasjon av intensjoner i fleragentmiljøer. Med videreutvikling kan agenten utvides for å følge trafikkregler, som for eksempel høyre-håndsregelen, når den samhandler med andre agenter i kjørescenarioer.

Det ble opprettet en YouTube-kanal med videoer for å gi leseren en forståelse av omgivelsene der agentene skal operere, samt gi en visuell representasjon av PDRL før de begynner å lese oppgaven: https://www.youtube.com/@PDRLMasterThesis.

Preface

This thesis, written in spring 2023, is an individual semester project conducted at the Institute of Computer Science (IDI) at the Norwegian University of Science and Technology (NTNU) under the supervision of a professor from the institute. My academic focus as a student has primarily been on the fields of robotics and artificial intelligence, and this thesis builds upon my passion for these areas. Considering the ongoing advancements in these technologies, I firmly believe they will play a significant role in shaping our collective future. It is my sincere hope that we can harness their potential to enhance our societies. In order to achieve this, I strongly advocate for a diligent approach to the development of new systems. Rather than rushing to release products to outpace competitors, we should prioritize the process of verification, testing, and comprehensive understanding of these systems.

The project I have undertaken is highly exploratory, allowing me to pursue my interests while also generating valuable insights into the realms of robotics and AI for the institute. The primary objective is to conduct experiments with state-of-the-art reinforcement learning algorithms that can enable safe navigation for a small mobile robot. Safety is of paramount importance when integrating AI into robotics, especially in the context of self-driving cars, and other systems where humans are in harms way. Therefore, I feel that my work in this project is relevant to the ongoing discourse on AI, that we hear about in the media.

Engaging in a robotics system for my Master's thesis presents a unique opportunity, and I am sincerely grateful to my supervisor, Rudolf Mester, for granting me this chance. Throughout my studies at NTNU, I have delved into both robotics and computer science, and I often find that students are not afforded sufficient opportunities to explore the fascinating intersection between these two fields. Rudolf's extensive experience and expertise in the automotive industry and computer vision have proven to be invaluable assets throughout the course of this project.

Author H. Fjellheim, supervisor R. Mester

Trondheim, June 8, 2023

Acknowledgements

Throughout the course of this project, I received substantial assistance from my peers, professors, and PhD candidates. Foremost, I would like to acknowledge my supervisor, Rudolf Mester, whose extensive expertise in the field of autonomous driving proved invaluable. It was under his guidance that I became aware of the problem this thesis wants to address, namely the issue of unpredictability associated with end-to-end machine learning approaches in autonomous driving tasks. Our lengthy meetings in his office consistently evolved into captivating discussions, often veering into various theoretical domains pertinent to autonomous driving, even if not always directly related to the thesis at hand.

I would like to acknowledge the contributions of Alexander M. Aas, a fellow master's student at NTNU, with whom I collaborated in preparation for this master's thesis. Aas made significant contributions to the procurement process of the robot platform by conducting extensive research on various platforms. Furthermore, he provided valuable assistance in the development of the GAZEBO simulation environment, specifically by creating a version compatible with Ubuntu 22.04. Unfortunately, Aas had to withdraw from the project, and as a result, the software for this project never got to the hardware itself.

Acknowledgment should also be extended to N. Barboni, another master student under Mester's supervision. N. Barboni proved to be an exceptional discussion partner during the implementation of the environment, especially regarding the circograms. Additionally, N. Barboni made significant contributions by writing code that was later utilized in this thesis.

Furthermore, I would like to express my sincere gratitude to my fellow master students, Solveig Jørgensen Mohr and Andrea Stette Jessen, for their invaluable assistance and support throughout this semester. Having colleagues who were facing similar challenges and with whom I could discuss everything from technical implementations in LaTeX to robotics theory has been incredibly beneficial. Their contributions and collaboration have greatly enhanced the quality and depth of my work, and I am truly thankful for their guidance and camaraderie.

During the testing phase of the LIMO for driving parameters, I received valuable assistance from Henrik Münchhausen, a master student at TU Braunschweig (TUB), Germany. Professor Ruoyu Peng of TUB, whom I had reached out to when searching for a suitable robot platform, recommended the LIMO for our project. Under Professor Peng's guidance, Münchhausen was involved in constructing a test course for the LIMO at TUB. Together, we discovered errors in the LIMO's data sheet, particularly regarding its turn radius. I would like to express my gratitude to both Professor Peng and Münchhausen for their invaluable support throughout the acquisition and testing process of the LIMO.

Although this is turning into a triumphant acceptance speech at the Oscars, I would also like to express my sincere appreciation to my fiancée, Ine, for her invaluable support throughout a challenging semester. Ine provided assistance during stressful times and consistently went above and beyond, even getting me iced coffee when my caffeine levels were alarmingly low.

Abbreviations

In table 1 are some of the abbreviations that will be used throughout the paper:

Abbreviation	Description
ANN	Artificial Neural Network – an umbrella term for various neural network architectures.
APF	Artificial Potential Field – a method for representing the environment using a gradient of desirability.
CCF	Center Coordinate Frame – with the origin at the center of the vehicle.
CNN	Convolutional Neural Network.
CTM	Coordinated turn motion model – used to simulate car-like motion.
DRL	Deep Reinforcement Learning – utilizing deep networks to encode high-dimensional spaces.
IDI	Institute for Computer Science at NTNU.
LIDAR	Light Detection & Ranging sensor – used for measuring distances.
LQR	Linear Quadratic Regulator – for optimal control.
LMPC	Learning Model Predictive Controller – proposed by Rosolia and Borrelli (2017).
MBRL	Model-Based Reinforcement Learning – also known as explicit RL.
MPC	Model Predictive Controller – a method from control theory.
NTNU	Norwegian University of Science and Technology.
PDRL	Predictable Deep Reinforcement Learning – the architecture proposed in this thesis.
RL	Reinforcement Learning.
RNN	Recurrent Neural Network.
ROS	Robot Operating System – a widely used communication framework in robotics.
TD	Temporal Difference – a tool used by many reinforcement learning methods.
VCF	Vehicle Coordinate Frame – with the origin at the vehicle's rear axle.
WCF	World Coordinate Frame – with a pre-defined and stationary origin.

Table 1.: A selection of abbreviations Used in the Paper

Contents

1. Introduction

In the introduction, the motivations for undertaking this research project are presented, and the objectives of the project are defined. The paper provides an overview of its structure and outlines the plan for the master's thesis. Additionally, the **LIMO** robotics platform is briefly discussed as it has had a significant influence on the work presented in this thesis.

1.1. Background and Motivation for The project

Autonomous vehicles continue to be a prominent subject in the fields of AI and robotics, attracting significant scientific attention and funding from industries such as automotive. The Society of Automotive Engineers (SAE) has established a taxonomy international (2021) that classifies vehicles into six categories based on increasing levels of autonomy:

1. Level 0 - No Automation: The driver has complete control over the vehicle at all times.
2. Level 1 - Driver Assistance: The vehicle is equipped with systems that provide some assistance to the driver, such as cruise control or lane departure warning.
3. Level 2 - Partial Automation: The vehicle has some level of automation, such as steering and acceleration, but the driver is still responsible for monitoring the environment and taking over when needed.
4. Level 3 - Conditional Automation: The vehicle can handle some aspects of driving in certain conditions, such as on highways, but the driver must be ready to take control at any time.
5. Level 4 - High Automation: The vehicle can operate autonomously in certain conditions and environments without any human intervention, but there may be limitations.
6. Level 5 - Full Automation: The vehicle is capable of operating autonomously in any condition or environment without any human intervention.

Despite decades of research and development, there are only a few fully autonomous systems currently operational, and none that operate at large scale on public road networks. In recent years, several level 3 ("eyes off") and level 4 ("mind off") cars have been deployed primarily in the southwest region of the USA, including systems like Google's *WAYMO*, *NURO*, and *Cruise*. The regulations for deploying autonomous vehicles in non-public road settings are generally less stringent, allowing for deeper levels of autonomy in warehouse robots. One notable example is Amazon's fully autonomous warehouse robot, *Proteus*, which stands out from previous warehouse robots as it has been deemed safe enough to operate alongside humans and is not confined to a restricted area of the warehouse CLARK (2022). However, one challenge with privately owned robotics, such as *Proteus* by Amazon, is that the specific implementation details are not publicly available.

This thesis aims to implement an algorithm that enables a car-like agent to exhibit predictability when navigating through an unknown environment while solving way-point missions. Predictability, in this context, refers to the ability of the robot's actions and trajectories to be anticipated

and communicated before execution, allowing for interventions to prevent any harmful impact on its surroundings. The pursuit of predictability is an important aspect of developing an explainable AI-system (XAI), being a challenge for all sub-symbolic AI methods Arrieta et al. (2020). Sub-symbolic AI-systems are inherently difficult for human observers to comprehend, as their decisions lack structured or symbolic accessibility. Unfortunately, these are also the methods that have achieved the greatest success in the last decade of research.

Deep reinforcement learning (DRL) is a set of promising sub-symbolic algorithms capable of addressing navigation challenges in autonomous vehicles Kiran et al. (2021). However, a challenge with DRL is that most state-of-the-art methods generate a single action outputs for a given situation, resulting in unpredictability and potential hazards. While an agent may perform flawlessly within a simulated environment, its behavior becomes subject to disturbances and unfamiliar situations once deployed, leading to unpredictable and harmful actions. Compounding this issue is the inherent sub-symbolic nature of DRL agents, which complicates the retrospective understanding of their decision-making processes.

To mitigate these challenges, this thesis proposes a fusion of DRL with concepts from control theory and a predefined vehicle model. By integrating these components, it becomes possible to enable any DRL agent to telegraph its intended actions, preemptively averting potentially harmful behaviors. The architecture introduced in Chapter 5, coined as predictable DRL (P-DRL), can be categorized as a model based reinforcement learning (MBRL). From testing it was found that it can both enhances the model's capacity to operate safely within unknown environments and facilitates secure online training post-deployment.

The selection of autonomous vehicles as the focus area of this thesis is driven by the availability of the **LIMO** robot platform developed by **AgileX** robotics. The robot was recently acquired for the Department of Computer Science (IDI) at the Norwegian University of Science and Technology (NTNU), and the aim is to implement algorithms specifically for this platform. The driving dynamics and sensory equipment of the LIMO have significantly influenced the development in this thesis. Therefore, while the presented results are generally applicable to any autonomous vehicle, all tests are conducted with the LIMO robot platform in mind.

1.2. Research Method and objectives

This project involves both development and research aspects. To facilitate reinforcement learning and testing of the proposed deep RL architecture, a simulator was developed. The simulator incorporates a physical vehicle model and provides a state and action space suitable for testing RL agents with a simple way-point mission. The vehicle model will be adjusted to emulate the characteristics of the LIMO robotics platform by fine-tuning its parameters. The following are the objectives and research question of the thesis:

Objective 1 [O1] *Develop a simulation environment for testing autonomous vehicles. The environment should include a dynamic vehicle model with adjustable parameters to match the desired vehicle dynamics. It should also incorporate obstacles, simulated sensor data, and predefined goal states for the vehicles to reach.*

Objective 2 [O2] *Develop the PDRL architecture by combining concepts from model predictive control (MPC) and deep reinforcement learning (DRL), enabling any implicit DRL algorithm to incorporate planning. This architecture should contain, and maintain, an explicit vehicle model to predict future states.*

Research question 1 [RQ1] *How does the incorporation of the PDRL architecture, with its added*

computational overhead, and reliance on an explicit vehicle model, impact the scoring performance of the agent compared to using DRL directly?

Research question 2 [RQ2] *To what extent does the incorporation of the PDRL architecture, with its enhanced predictability, enable online training for the agent and mitigate collisions during the training process?*

1.3. Structure of the paper

In Chapter 2, the relevant background theory will be presented, mainly including ideas from control theory and reinforcement learning that will be relevant and useful for understanding the later composition of the P-DRL architecture. Additionally, other concepts such as LIDAR sensor systems and artificial neural networks will be discussed.

Then, in Chapters 3 and 4, the development of the environment will be presented. Chapter 3 focuses on the dynamic vehicle model, which is a coordinated turn motion (CTM) with Ackermann steering geometry. In Chapter 4, the surrounding environment and its representation will be developed. This includes adding obstacles, determining coordinate axes, and establishing the representation of the environment for the DRL agents. Circograms and a local momentary map, known as the "vision box," will be utilized to provide the agents with sufficient data to solve their missions.

Chapter 5 presents the P-DRL architecture and 6 covers the experimental setup necessary to test the system. The system will be compared to plain DRL counterparts, and the experiments and results will be presented in Chapter 7. Finally, the results will be discussed, and the research questions will be answered in Chapter 8.

1.4. The LIMO robotics platform

The simulated environment, mathematical dynamic vehicle models, and the final system architecture presented in this thesis are all designed with the LIMO robotic platform in mind. The selection and procurement of a mobile robotic platform were crucial in the pre-project phase leading up to this thesis. The chosen platform needed to be versatile enough to serve multiple student projects and master theses in the future. After considering various available platforms, the LIMO was selected for its on-board GPU and its similarity to a real car in terms of steering geometry, specifically Ackermann steering geometry (see Chapter 3). The LIMO, as depicted in Figure 1.1 after unpacking, is equipped with LIDAR, IMU, wheel odometry, and stereo cameras with built-in depth perception. The robot platform provides numerous useful packages and interfaces, designed in both ROS-1 and ROS-2. The focus of this thesis is not to implement hardware-related aspects but rather to develop a simulator and design an agent architecture for safe and predictable navigation, that can later be used in the LIMO. The only direct use of the LIMO in this project, was to extract physical parameters needed to tune the simulator and agent (see section 7).

1.5. Related works

Chapter 4 explores the concept of "circograms," which are distance measurement schemes based on distance that can be used as control inputs. The use of distance measurement as a control input is not a new idea, and the term "circogram" for this application is borrowed from another

Figure 1.1.: The LIMO after unpacking and test-driving.

thesis. In 2019, P. Klose and R. Mester Klose and Mester (2018) introduced and investigated the use of circograms to determine modes for a discrete finite state machine controlling a vehicle navigating a virtual city. In this paper, the same circograms are employed as control inputs for the short-term planner algorithm. Chapter 4 also explores the potential combination of circograms with the dynamic vehicle model, later called dynamic circograms.

A side-effect of implementing, and testing agents in a simulated environment was a lot of 2D pose-files. Poses are sets of (position, heading), and throughout this project hundreds of thousands of hours of simulated vehicle driving was produced and stored in 2D pose files. This came in handy for four other master students at IDI, NTNU. The master students Mohr S. and Jessen are aiming to contribute to the research of verification of situational awareness systems on autonomous ships through their master thesis. The objective is to develop, implement and evaluate a simulator for camera-based object detectors that can generate test data to assess a tracker within the external situational awareness system. It was a great help for them to get pose files, to test their simulator for different conditions. The master students Kanstad S. and Hjelle E. are working on a ship detection system, using cameras and deep learning. For them it was useful to get pose files to then simulate boats navigating a marina, that they could then work to detect. A screenshot of their work can be seen in figure 1.2. Boats might not be the same as cars, but the difference was not enough to cause any issue for them.

An unintended outcome of implementing and testing agents in a simulated environment was the generation of numerous 2D pose files. Poses consist of position and heading information, and throughout this project, hundreds of thousands of hours of simulated vehicle driving data were produced and stored in these pose files. This turned out to be valuable for four other master students at IDI, NTNU. Two of the master students, Solveig J. Mohr and Andrea S. Jessen, are focusing on contributing to the research of verification of situational awareness systems on autonomous ships through their master's thesis. Their objective is to develop, implement, and evaluate a simulator for camera-based object detectors that can generate test data to assess a tracker within the external situational awareness system. The availability of pose files proved to be helpful for them to test their simulator under different conditions. The two other master students, Sondre M. Kanstad and Emil Hjelle, are working on a ship detection system using cameras and deep learning. For their project, having access to pose files enabled them to simulate boats

navigating a marina, which they could then use for detection purposes. Figure 1.2 showcases a screenshot of their work. Although boats differ from the cars that were simulated in the PDRL project, the disparity did not pose any significant issues for them.

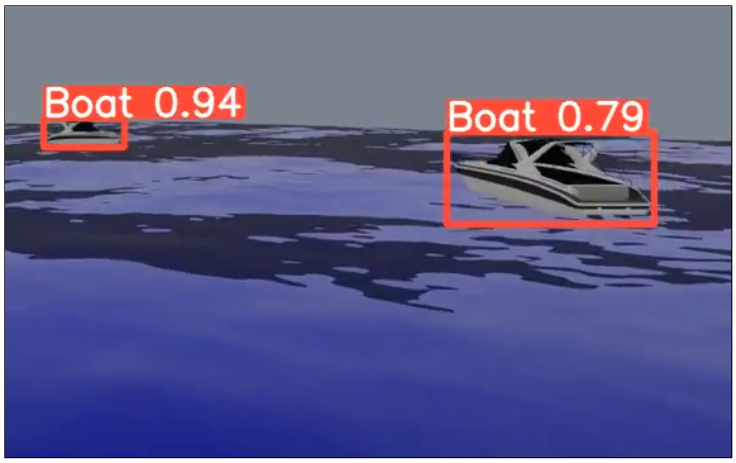


Figure 1.2.: An example of the work done by other master students, while using pose files produced by this project. This particular screenshot is from the work of Kanstad S. and Hjelle E., using the pose-files to simulate boats navigating in a marina.

2. Theory

This chapter presents the key background theory necessary to comprehend the master thesis. The LIDAR sensors on the **LIMO** play a crucial role in determining how the environment is represented for the subsequent agents. The proposed architecture in this thesis incorporates elements from control theory and deep reinforcement learning (DRL), thus relevant concepts from these domains are introduced. Finally, a brief overview of deep neural networks is also provided at the end of this chapter, as they are employed in both the DRL algorithms, as well as for the vehicle model approximation module.

2.1. LIDAR sensory system

The LIDAR sensory system is a widely utilized and powerful sensor in autonomous vehicles. LIDARs emit laser beams in various directions and measure the time of flight (TOF) to calculate the distance to reflecting surfaces, leveraging the speed of light. These beams can be emitted simultaneously in multiple directions or through a rotating sensor to create a point cloud (PC) of distance measurements. Depending on the sensor's configuration, the point cloud can represent a 3D view or a 2D view along a plane where the sensor is mounted Campbell et al. (2018). In the case of the LIMO, the LIDAR rotates along a single axis, producing a distance map along the plane where the sensor is mounted. Figure 2.1 illustrates how LIDAR provides comprehensive information about surrounding obstacles, facilitating the agent's understanding of its environment.

The point cloud generated by LIDAR can be used to generate an environment map and detect obstacles. An essential characteristic of LIDAR sensors is the angle between emitted beams, as excessively large angles can result in significant gaps in the field of perception. Nonetheless, even 2D versions of LIDAR can be employed in advanced driver assistance systems to detect and avoid obstacles, thanks to their accuracy Catapang and Ramos (2016). By retaining the point cloud data as the agent moves and captures additional points, it is possible to create dense global maps of the environment by incorporating simultaneous localization and mapping (SLAM) algorithms. The combination of LIDAR sensors and SLAM is adequate for generating consistent maps during exploration and is increasingly adopted as more autonomous vehicles are equipped with LIDAR systems Khan et al. (2021).

2.2. Path and trajectory planning

In both this thesis and the preceding pre-project, the terms "path" and "trajectory" are used interchangeably to describe the output of the planner component. However, there exists a subtle distinction between these two terms. A path refers to a sequence of points in the configuration/state space of a robot, spanning from the initial state to the final state. Essentially, it is a list of states that the robot must traverse, starting from the current state and ultimately reaching its goal state. These points are often referred to as "via-points" or "waypoints" and can encompass the entire vector of robot state variables or a subset thereof.

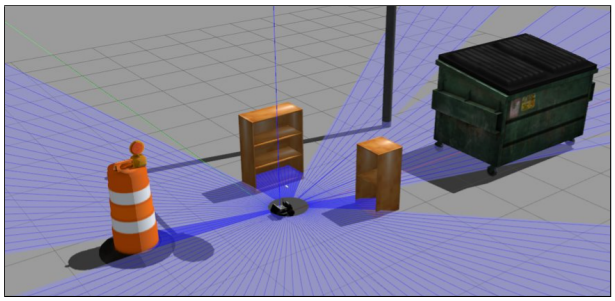


Figure 2.1.: The LIMO comes with a 2D lidar system; meaning a LIDAR sensor that scans about one rotation axis to produce measurements in the plane. This figure was simulated in GAZEBO.

On the other hand, a trajectory represents the realization of the path as a function of time, extending from the initial time t_0 to the final time t_f. A trajectory connects the via-points through actual vehicle movement (Spong et al. 2020, p. 252). The reason for the frequent interchangeability of these two terms is that a path serves as a generalization of a trajectory and can give rise to multiple distinct trajectories. Some planning algorithms generate a path that requires an additional component for execution, while others directly produce the complete trajectory as their output.

2.3. Ideas from control theory

Control theory is a mathematical field that utilizes statistics and dynamical system models to design feedback loops for regulating and controlling systems. It predates the field of AI and is widely applied in various domains such as process engineering (e.g., petroleum and food industries), flight engineering, automotive industry, animal population control, and more. For any mathematically modelled process, whether linear or nonlinear, a control input can be designed to achieve the desired output, provided that the system is observable and controllable (Chen 2013, p. 183-222). In this section, two prominent methods from control theory will be briefly discussed: linear quadratic regulators (LQR) as an optimal control algorithm and model predictive control (MPC) as a planning architecture. These ideas will later be combined with reinforcement learning to design the predictable DRL (PDRL) agent of this thesis.

2.3.1. Linear quadratic regulators

LQR, or linear quadratic regulator, utilizes a quadratic cost function to determine the optimal control input for a linear system. For a linear system represented by the **ABCD** matrices (see section 3.1), the optimal control signal can be obtained by minimizing the quadratic cost function, given the transition and input matrices **A** and **B**, through the solution of the **Riccati** equation (Anderson and Moore 2007, p. 26). In the context of LQR, the optimal control signal aims to drive the process towards a desired output value while considering the desired use of input. In control theory, the control signal or input at time t is often denoted as u_t, while in reinforcement learning, a is used to represent an action taken in an environment. In the application of this thesis, both notations refer to the same system input. To achieve optimal control using LQR, both the desired value and the desired use of input need to be tuned through the **Q** and **R** matrices in the quadratic cost function. The definition of optimal control heavily depends on the specific use case and system tuning. If the system is nonlinear, it needs to be linearized, typically using the system Jacobian, before applying the LQR controller. Although LQR is not a planner

algorithm, it can be utilized with a pre-calculated path to generate a trajectory by producing optimal control signals to reach each waypoint along the path (see section 2.2).

2.3.2. Model predictive controller (MPC)

The concept of a control algorithm that can learn the environment and provide optimal control signals to a process has been in existence since the 1950s. Model predictive control (MPC) has gradually evolved and found successful applications in various industrial domains. In general, an MPC algorithm uses experiences from the target environment to update an internal understanding of the surrounding environment, which is stored in an explicit model. It then uses this model, iteratively improved, to select optimal actions that minimize a specified cost function Lee (2011).

A simple and generic MPC can be implemented by storing state-action-new state triplets in a buffer, performing linear regression over the state space to obtain a linear state transition model, and using LQR with this linear model to generate optimal control signals iteratively. Algorithm 1 presents such a simple MPC. In this algorithm, a local buffer is utilized to produce local linear transition models since a linear transition model is required for LQR. In the proposed algorithm's **"SOLVE"** function, control inputs ($\mathbf{u}_k$) are planned for each step along the control horizon. However, only the first action in the control sequence ($\mathbf{u}_0$) is used before re-planning in the next time-step. For the LQR solver in this algorithm, which focuses on minimizing cost using current data rather than path planning, this represents a waste of computational power, and the control horizon N could be set to one. However, for a path planner solver that aims to reduce cost over the entire path, re-planning remains a valuable tool as a plan generated in the next time-step is more informed and likely to perform better than the previous one. In the case of replacing the **SOLVE** function in algorithm 1 with an optimal path planner instead of LQR, the linear regression algorithm could also be substituted with another model approximator since a linear model may no longer be necessary.

In recent years, research in MPC has focused on improving algorithm efficiency Lee (2011). Generating an optimal trajectory in a high-dimensional, nonlinear environment is a computationally intensive process. In 2017, the learning-MPC (LMPC) algorithm was introduced, which combines multiple ideas to enhance basic MPC. LMPC employs a direct solver that relies on the predicted model to generate optimal paths. Additionally, it defines a set of safe states that are the only permissible terminal states in a proposed trajectory generated by the direct solver. This speeds up the process and improves stability during early exploration. The safe set comprises previously explored states, but it can be relaxed to also allow linear combinations of these, to further reduce the computational burden Rosolia and Borrelli (2017).

Another approach introduced in 2020 involved using the optimal control outputs from an MPC framework to train an artificial neural network (ANN) imitation learner. The MPC is first employed to generate optimized training data for the ANN, which is then trained based on this data. ANN predictions can be executed much faster than an MPC, requiring only a single matrix multiplication for each layer of the network. If the ANN can accurately encode the MPC output, it can significantly speed up the overall system Tătulea-Codrean et al. (2020).

2.4. Reinforcement learning

The main goal of this thesis is to develop a reinforcement learning agent to operate on a vehicle-like robot. This section will therefore discuss the existing modern reinforcement learning methods

Algorithm 1 Generic MPC

Input: Control horizon N, total run-time T
Output: Control signals $\leftarrow \{u_0, u_1, ..., u_T\}$

```
for t = 0, 1, ..., T do
    X_t ← Current environment state
    if Buffer is too small then
        u_t ← 'random' action
    else
        for k = 0, ..., N do
            Compute local neighbourhood from Buffer.
            A_k, B_k ← Linear regression using local Neighbourhood.
        end for
        u_k ← SOLVE((A_k, B_k)_{k=0}^N, X_t)
        Take action u_0, observe next state X_{t+1}
        Append (X_t, u_0, X_{t+1}) to Buffer
    end if
end for

function SOLVE((A_k, B_k)_{k=0}^N, X_t)
    X_k ← X_t
    for k = 0, ..., N do
        u_k ← LQR(A_k, B_k, X_k)
        X_{k+1} ← A_k X_k + B_k u_k
        X_k ← X_{k+1}
    end for
    return (X_k, u_k)_{k=0}^N
end function
```

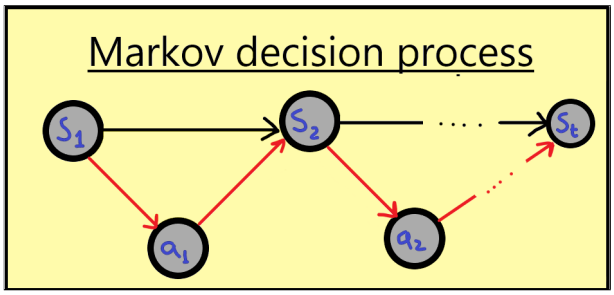


Figure 2.2.: The Markov Decision Process (MDP), introduced by Richard Bellman in the 1950s, has been influential in the development of dynamic programming and reinforcement learning as a useful representation of dynamical environments. Transitions between states can be stochastic or deterministic based on the selected action.

and frameworks, present state-of-the-art RL results, as well as modern applications of RL. Chapter 5 will then discuss how the ideas from this section and Section 2.3 will be combined to create a unique and predictable framework to use with existing DRL methods.

2.4.1. Basics of reinforcement learning

Regulator design using control theory was already a well-established field of engineering in the 1950s when Richard Bellman (and others) developed a class of practicable methods for solving optimal control problems in dynamical systems. This class, known as "dynamic programming" (DP), proved impractical for solving optimality problems due to the curse of dimensionality. DP incurs exponentially growing computational costs with problem complexity (Sutton and Barto 2020, p. 14). Bellman introduced Markov Decision Processes (MDPs) as a discrete and stochastic version of the optimal control problem. Any dynamic system that can be transformed into an MDP is suitable for solving using dynamic programming and reinforcement learning (Sutton and Barto 2020, p. 2). Figure 2.2 illustrates the basic structure of an MDP, with states (S_n), actions (a_n), and goal/terminal states (S_t). Transitions can be stochastic or deterministic, and the state and action spaces can be discrete or continuous. Each transition is associated with a reward or cost (negative reward), and the goal of optimal control algorithms is to optimize the expected accumulated reward from the initial state (S_1) to the terminal state (S_t). It is important to note that this optimization focuses on the "expected" accumulated reward, as stochastic state transitions still allow for optimizing expected outcomes.

Reinforcement learning (RL) is a distinct branch of machine learning that sets itself apart from both supervised and unsupervised learning approaches. Unlike these other methods that rely on stored data and offline learning, RL engages in sampling and exploring an environment, often in an online fashion, to enable learning of unknown environments. In this context, online learning refers to the process of learning while actively sampling the environment, rather than after-the-fact analysis. The primary objective of RL is to address optimal control problems in dynamical systems, commonly represented as Markov Decision Processes (MDPs) (Sutton and Barto 2020, p. 2). The term "reinforcement learning" encompasses various methodologies aimed at solving dynamic programming problems by utilizing iterative and recursive algorithms, efficient environmental sampling, and techniques such as bootstrapping to attain practical solutions to DP problems.

The Bellman equations provide a set of functional equations used to solve the dynamic programming problem. They involve the state of the dynamical system and a value function to recursively or iteratively determine the optimal value or action policy for controlling the system. Equation

2.1 represents one form of the Bellman equation. To comprehend the Bellman equation, it is first necessary to understand certain concepts.

In the equations, π denotes an action policy, which represents a rule for selecting actions in a given state: $\pi(s) = a$. The value of a state, denoted as $v_\pi(s)$, is the sum of all expected rewards obtained by following the policy π. In most cases, immediate rewards are considered more important than rewards far into the uncertain future. Hence, each future reward is discounted using a factor $\gamma \in [0, 1]$. The quantity $G_t = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \dots$ represents the sum of all future discounted rewards, where t indicates the current discrete time-step and $t + n$ refers to a time-step n steps into the future.

The key aspect of the Bellman equation is its recursive nature, allowing for iterative solution methods instead of requiring a single extensive operation, as in "pure" dynamic programming. In the final step of Equation 2.1, G_{t+1} is replaced by $v_\pi(s')$ due to the properties of the expected value $\mathbb{E}$. This substitution is valid because $v_\pi(s')$ represents the expected future reward of the next state s', while G_{t+1} corresponds to the actual future reward in state s'. Therefore, instead of sampling the entire G_t, which would necessitate a complete episode from S_1 to S_t, the value of a state can be estimated based on the current understanding of the value of the next state $v_\pi(s')$ (Sutton and Barto 2020, p. 60). This approach is known as **bootstrapping** and is one of the techniques that makes the Bellman equations more practical than performing the full dynamic programming algorithm.

$$\begin{aligned} v_\pi(s) &= \mathbb{E}_\pi[G_t | S_t = s] \\ &= \mathbb{E}_\pi[R_{t+1} + \gamma G_{t+1} | S_t = s] \\ &= \mathbb{E}_\pi[R_{t+1} + \gamma v_\pi(s') | S_t = s, S_{t+1} = s'] \end{aligned} \tag{2.1}$$

Methods that utilize G_t for learning are known as *Monte Carlo* (MC) methods, while methods that employ the approach shown in Equation 2.1 are referred to as *temporal difference* (TD) methods. Equation 2.2 presents the update rules that gradually allow an agent to learn about an environment by exploring states and receiving rewards. The learning rate, denoted as $\alpha \in [0, 1]$, is used to prevent the complete replacement of previous knowledge with new discoveries, favoring a gradual estimation process. High values of α generally leads to a lot of variance in training, while low values lead to the method getting stuck in local optimal solutions.

At the top of Equation 2.2 are the MC methods, which necessitate waiting until all of G is discovered before learning can occur. In contrast, the TD alternative below utilizes bootstrapping to learn incrementally for each state along the way. The term $(R_{t+1} + \gamma V(s_{t+1}) - V(s_t))$ represents the TD error, which corresponds to the difference between the expected value and the actual value of a state (Sutton and Barto 2020, p. 91-138).

$$\begin{aligned} V(s_t) &= V(s_t) + \alpha(G_t - V(s_t)) \\ V(s_t) &= V(s_t) + \alpha(R_{t+1} + \gamma V(s_{t+1}) - V(s_t)) \end{aligned} \tag{2.2}$$

Temporal difference learners have the capability to learn while exploring, enabling online learning, as opposed to waiting until the end of an episode for offline learning. However, while G_t serves as an unbiased estimator of the value, the bootstrapping approach introduces bias towards the previously learned value function. So, MC learning exhibits higher variance during training since significant updates occur at large intervals, TD learning demonstrates lower variance with smaller and quicker updates. The bias introduced by TD methods increases the likelihood of getting stuck

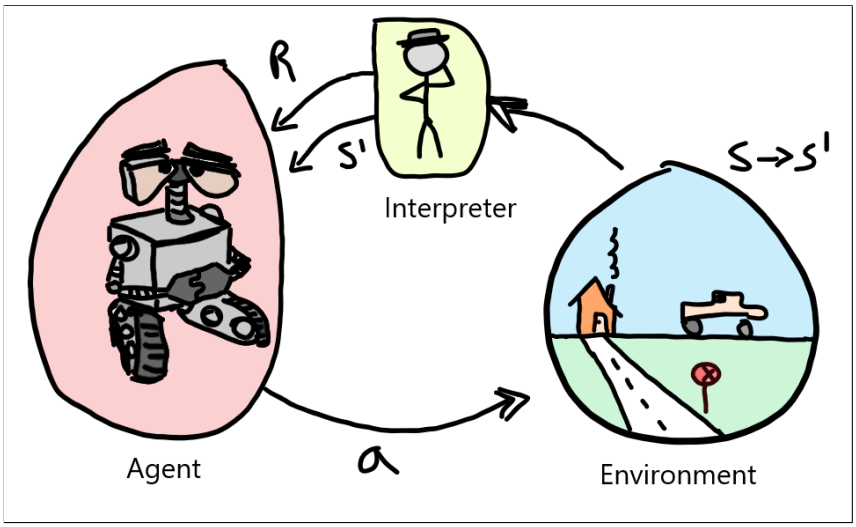


Figure 2.3.: The reinforcement learning cycle.

in local optima, and TD methods are sensitive to the initial values of $V(s)$, as these values can determine the optima in which they become stuck.

Equation 2.2 illustrates TD(0), where no steps are taken before learning. TD(n) is also a possibility, wherein n steps are taken before performing an update, making it approach the MC method. However, a more efficient approach to interpolate between these two methods is through the utilization of *eligibility traces* (E). The fundamental concept is that all recently visited states become eligible for an update once a new TD error is discovered during training. Eligibility starts with a value of 1 immediately after being visited and decays at a rate of $\lambda \in [0, 1]$, ensuring that only recently visited states receive an update. Empirical evidence demonstrates that this method operates as an "online" MC method, allowing for rapid online learning with low variance, akin to TD(0), while mitigating some of the bias inherent in TD(0) by employing more than one data point to update the eligible states (Sutton and Barto 2020, p. 287-317).

MDPs, as previously mentioned, serve as a simple and effective framework for representing a wide range of problems in the context of dynamic programming (DP) and reinforcement learning (RL). From the perspective of the agent or learner, MDPs can be understood as the RL learning cycle, depicted in Figure 2.3. A fundamental assumption underlying MDPs is the Markovian property, which states that past states should not impact the current state and the decisions made at present. This property assumes that the current state alone contains all the necessary information for achieving optimal control (Sutton and Barto 2020, p. 48). An agent can learn optimal control within an MDP environment by engaging in the RL cycle, which involves interactions of individual state-action-state-reward sequences. This RL cycle serves as the foundation for modern RL methods to sample and learn from their environments. In the RL cycle, an interpreter (often the RL system designer) plays a crucial role in assigning rewards or costs to states or state transitions. Subsequently, the RL algorithm samples the environment and gradually learns to optimize the accumulated rewards over time.

2.4.2. Policy and Value

In addition to formulating Bellman's equation as shown in equation 2.1, it can also be expressed in terms of an optimal policy π^* that optimizes future expected (and discounted) rewards. The optimal value function associated with this policy, denoted as $v^(s)$, is presented in equation 2.3.

In this equation, the action $a \in A$ that maximizes the future expected value is selected, and bootstrapping is employed once again to replace the future discounted reward G_{t+1} with the expected future discounted reward $v(S_{t+1})$. This formulation, known as the *Bellman optimality equation*, captures the essence of optimizing the policy to maximize rewards (Sutton and Barto 2020, p. 63). It is worth noting that other formulations of Bellman's equation exist, particularly those that do not utilize the $\max_a$ operator, which will be explored later in this section.

$$\begin{aligned} v^*(s) &= \max_{a \in A} q_{\pi^*}(s,a) \\ &= \max_a \mathbb{E}_{\pi^*}[G_t | S_t = s, A_t = a] \\ &= \max_a \mathbb{E}_{\pi^*}[R_{t+1} + \gamma G_{t+1} | S_t = s, A_t = a] \\ &= \max_a \mathbb{E}_{\pi^*}[R_{t+1} + \gamma G_{t+1} | S_t = s, A_t = a] \\ &= \max_a \mathbb{E}_{\pi^*}[R_{t+1} + \gamma v^*(S_{t+1}) | S_t = s, A_t = a] \end{aligned} \tag{2.3}$$

When discussing the values of states, it can be beneficial to consider the Q-value of an action-state pair. The $q_\pi(s,a)$ represents the value of choosing action a in state s, assuming that policy π will be followed for the remainder of the episode. Unlike using the value function $v_\pi(s)$, the Q-value directly informs the selection of the optimal action in a given state, by picking the highest value state-action pair. Modern RL methods make use of both Q-values and value functions (V-values), as both approaches are viable for representing value. Equation 2.3 can be rearranged to incorporate Q-values instead of values, yielding equation 2.4. This equation, in conjunction with equation 2.3, illustrates the similarity between the optimal Q-value of an action-state pair and the optimal value of a state, provided that the action is chosen to maximize the future expected (and discounted) value.

$$q^*(s,a) = \mathbb{E}[R_{t+1} + \gamma \max_{a'} q^*(S_{t+1}, a') | S_t = s, A_t = a, A_{t+1} = a'] \tag{2.4}$$

$$Q(s,a) = Q(s,a) + \alpha[R + \gamma \max_{a'} Q(s',a') - Q(s,a)] \tag{2.5}$$

Equation 2.4 forms the core of the Q-learning method, which involves sampling the environment by selecting actions that are believed to optimize the Q-value. The iterative update rule used in Q-learning is derived from equation 2.4 and is expressed in equation 2.5. In this equation, s' and a' denote the next state and action, respectively, often represented as s_{t+1} and a_{t+1}. The learning rate $\alpha \in [0,1]$ is used to control the impact of each update and limit overestimation bias during environment sampling. This iterative update process can be applied while sampling the environment and is analogous to a gradient descent method, gradually adjusting the Q-value towards the correct value based on sampled experiences (Sutton and Barto 2020, p. 200).

As observed in equation 2.5, the current optimal action a' in the next state s' is used to estimate the Q-value of the current state-action pair. This approach is referred to as an off-policy method because it uses the maximum Q-value to estimate the future expected reward, irrespective of the current policy being followed. This can be advantageous as the policy used during training is often exploratory, incorporating random noise to prevent convergence to suboptimal solutions.

Learning the optimal actions with respect to a random and noisy policy may cause difficulties (Sutton and Barto 2020, p. 131).

In contrast, SARSA is an on-policy method that estimates the Q-value using $Q(s,a) = Q(s,a) + \alpha[R + \gamma Q(s',a') - Q(s,a)]$, as opposed to employing the $\max_a$ operator. Instead of choosing the action that maximizes the next Q-value in the next state, SARSA selects the next action a' in the next state s' based on the current policy. Therefore, the SARSA algorithm estimates the Q-value based on the Q-value associated with the current policy being followed (Sutton and Barto 2020, p. 129). The policy often incorporates an epsilon-random selection between a random move and the action corresponding to the highest Q-value. This allows SARSA to potentially discover the optimal policy as long as the epsilon-random selection is gradually reduced during training. If not, SARSA may converge to a policy that still includes random moves with an epsilon chance. An example of a problem where this can be an issue is the "cliff-walk" example, where the objective is to navigate along a cliff. The SARSA policy would tend to avoid paths close to the cliff's edge due to the risk of randomly stepping off, while Q-learning does not face this problem and can walk on the very edge.

Both SARSA and Q-learning are both examples of so-called *value methods*, as they both focus on finding the Q-value function rather than the policy π directly. There also exists methods, called *policy gradient methods*, which instead of using gradient decent to estimate value functions, use similar tools to estimate the policy function $\pi(s)$ directly. The advantage of the policy-gradient approach, is that in the case of a continuous action space A, it is difficult to approximate the Q-values as it takes discrete state-action pairs as input. The policy gradient methods on the other hand, naturally handle continuous action spaces, as the policy function outputs a (vector of) continuous number(s) for a state input (Sutton and Barto 2020, p. 322-337). It is always possible to discretize a continuous space by using a coarse coder, like the *tile coding* as proposed by Sutton and Barto (Sutton and Barto 2020, p. 217), but this is just an approximation of the real space.

SARSA and Q-learning are both examples of value-based methods in reinforcement learning, as they focus on finding the Q-value function rather than directly estimating the policy π. However, there are also policy gradient methods that aim to estimate the policy function $\pi(s)$ directly, using tools similar to gradient descent.

One advantage of policy gradient methods is their ability to handle continuous action spaces A. In the case of value-based methods, approximating Q-values can be challenging when dealing with continuous state-action pairs as inputs. Policy gradient methods, on the other hand, naturally handle continuous action spaces since the policy function directly outputs a continuous value (or vector of them) for a given state input (Sutton and Barto 2020, p. 322-337). Discretizing a continuous space using techniques like tile coding, as proposed by Sutton and Barto, is possible but it only provides an approximation of the true continuous space (Sutton and Barto 2020, p. 217).

2.4.3. Actor-Critic: best of both worlds

In reinforcement learning, policy-based methods are well-suited for dealing with continuous action spaces. However, as it is not possible to use boot-strapping for policy based methods, they will, similarly to Monte Carlo (MC) methods, tend to have higher variance and slower learning than TD methods. On the other hand, value-based methods can estimate expected returns and perform bootstrapping efficiently but struggle with continuous action spaces.

To combine the benefits of both approaches, the Actor-Critic (AC) architecture is commonly

used in reinforcement learning Grondman et al. (2012). In this architecture, the policy gradient method, often referred to as the "actor," is responsible for selecting actions from the continuous action space. The actor then update its gradients using the knowledge provided by the "critic," which estimates the expected returns of the actions chosen by the actor, using bootstrapping.

By incorporating the critic, the actor can benefit from low-variance updates and utilize the estimated value function to guide its learning. The actor receives feedback from the critic, enabling it to improve its policy and make more informed decisions. It is worth noting that AC methods, like other TD or bootstrapping methods, are biased toward the current value function of the critic. To overcome local optima and reach to reach global optimal solutions, random exploration is typically incorporated into the learning process.

2.4.4. Function approximation: RL for real world applications

For relatively small state and action spaces, it is possible to estimate the exact Q-values for each possible state-action pair by exploring the entire state space and storing the estimated values in a table. This approach, known as *tabular* reinforcement learning, becomes infeasible for larger spaces. The sheer amount of exploration required and the size of the value tables would be impossibly large (Sutton and Barto 2020, p. 195). To address this, function approximators are used to estimate values and policies for different states. The function approximators try to estimate a global state-value or state-policy function, eliminating the need to visit each state individually.

When employing function approximators, it is crucial to ensure that they possess sufficient dimensionality to accurately represent the state and action space of the environment. Also, not all functions can be approximated with a linear function, so non-linear function approximators is sometimes necessary. Deep artificial neural networks, capable of approximating non-linear functions, can be scaled to any dimensionality (Goodfellow et al. 2016, p. 168). The RL methods that use deep artificial neural networks are coined as deep RL (DRL), and conceptually are very similar to the tabular counterparts. While using bootstrapping for example, the temporal difference (TD) error indicates the error or loss in the current value function, and can be used to train the artificial neural network while exploring the environment.

DRL methods and other RL methods employing function approximators can encounter divergence issues when estimating value and policy functions. This phenomenon, known as *The Deadly Triad*, arises from the combination of function approximators, off-policy learners, and bootstrapping (such as TD), which can lead to training divergence Van Hasselt et al. (2018). One method to reduce the deadliness of the triad, is to employ target networks while training.

2.4.5. Target networks: dealing with maximization bias

RL methods, such as Q-learning, that employ bootstrapping by maximizing the previously estimated value function for further updates, can encounter issues of overestimation. Q-learning is derived from the Bellman optimality equation, as shown in Equation 2.4, and the problem of overestimation arises from the use of the *max* function (Sutton and Barto 2020, p. 134). Previously visited states that happened to yield high rewards in specific instances can lead to an overestimation of the values for nearby states in the explored state space. This problem, referred to as *maximization bias*, results in slower convergence during the learning process and a tendency to converge to local optima Van Hasselt et al. (2016).

To address this issue, this paper proposed the use of a *target* function in the update process Mnih et al. (2015). The target function maintains a separate version of the estimated value function, which is utilized for bootstrapping. Instead of estimating the value function for a state by taking the maximum value from the same value function for the next state, it uses the target function instead. The use of a target function is demonstrated in Equation 2.6, where the target function is denoted as Q_2. The next action is still selected as the best action according to Q_1, but the value of choosing action a' in the next state s' is determined by Q_2. After updating Q_1 using this update rule, Q_2 can be updated either with a similar update rule or directly from Q_1 using a linear combination: $Q_2 = (1-\tau)Q_2 + \tau Q_2$. The former approach is employed by *Double Deep Q-learning*, as proposed in Mnih et al. (2015), while the latter is utilized in the **DDPG** DRL method Lillicrap et al. (2015).

$$\begin{aligned} Q(s,a) &= Q(s,a) + \alpha[R + \gamma \max_{a'} Q(s',a') - Q(s,a)] \\ Q_1(s,a) &= Q_1(s,a) + \alpha[R + \gamma Q_2(s', \arg\max_{a'} Q_1(s',a')) - Q_1(s,a)] \end{aligned} \tag{2.6}$$

Since the target functions are often approximated using neural networks, they are commonly referred to as *target networks*. In Section 2.5, the "deadly triad" that causes divergence in DRL methods was mentioned. Despite this triad, many modern DRL methods, such as DDPG Lillicrap et al. (2015), successfully combine these three components to great success. This is likely due to the use of *target networks* in DDPG, as they introduce stability and damping to the training process to remove divergence Van Hasselt et al. (2018).

In summary, target functions and target networks provide solutions for mitigating maximization bias, as well as introducing stability and damping during the training process. They can be incorporated into any RL method, including actor-critic methods, where separate targets can be utilized for the actor and the critic.

2.4.6. Hierarchical RL and the curse of dimensionality

When searching for optimal solutions in MDPs, the challenge of *the curse of dimensionality* arises. The number of state and action variables in the environment causes and exponential growth in computations needed to find the optimal solutions, as more and more possible combinations needs to be combined and explored (Sutton and Barto 2020, p.87). Modern RL methods try to overcome this challenge, for example by using function approximators as discussed in section 2.4.4.

Another approach is to avoid relying solely on the simple MDPs described in Section 2.4.1 when seeking optimal solutions. Instead, real-life problems can often be decomposed into components of varying complexity, taking advantage of their hierarchical nature. By decomposing problems and solving them at different levels of complexity, the number of variables for each search is significantly reduced Hengst (2010). This is accomplished through the use of semi-MDPs, where parent tasks select actions that persist for several time-steps, while child tasks handle tasks of lower complexity. In simple MDPs, each action is chosen for a single state and remains effective for a single time-step only.

2.4.7. Model based vs model free reinforcement learning

In the context of RL, a model refers to a transition function that takes states and actions as inputs and produces subsequent states as outputs. It can be considered the RL agent's perception

of reality and plays a crucial role in determining the agent's learned policy for optimal decision-making in the environment (Sutton and Barto 2020, p. 159). The RL algorithms discussed thus far in this section are examples of *model-free* or *implicit* reinforcement learning algorithms. This means that the mathematical transition function of the environment is not explicitly learned, but is implicitly embedded within the agent's policy or value function. The agent learns which states or state-action pairs yield favorable rewards, but does not acquire knowledge of how to reach specific states. Without an explicit model of the environment, these agents cannot predict future states or plan ahead. While their actions may appear strategic as they navigate toward global goals and avoid local optima, they do not possess the ability to communicate their planned actions. This is because they do not do any planning, and only chose current actiosn that they believe will maximize future expected reward. Without a plan, they cannot communicate their future intentions either, making them unpredictable to other agents.

In contrast, *model-based* or *explicit* RL algorithms (MBRL) is a different branch of RL methods. These approaches aim to learn an explicit model of the environment while simultaneously discovering the optimal policy. By incorporating an explicit transition model, often represented using function approximators, MBRL enables planning and predictable actions, distinguishing it from model-free RL methods Moerland et al. (2020).

A comprehensive survey conducted by Moerland et al. (2020) explores and compares model-based reinforcement learning (MBRL) with implicit or model-free approaches. MBRL integrates reinforcement learning with planning, resulting in higher computational requirements for training and execution compared to standard RL algorithms. The production of the explicit model, often utilizing function approximation, introduces additional parameters that need to be tuned during training. Computational speed remains a primary limiting factor in training RL agents, particularly in large-scale real-life continuous state spaces where simulations can be computationally expensive Moerland et al. (2020). Consequently, introducing more complexity carries significant implications. Furthermore, the utilization of a model introduces additional uncertainty and approximation errors, increasing the likelihood of unstable agents and variance in training, slowing down training (Moerland et al. 2020, p. 43).

Safety is a major concern when dealing with implicit learners as predicting their actions while navigating the environment can be challenging due to the absence of an explicit plan. In contrast, model-based learners offer improved safety capabilities as their plans can be explicitly observed during agent movement, and simple safety constraints can be enforced to prevent the execution of risky actions (Moerland et al. 2020, p.40-44).

MBRL has recently gained considerable attention, prompting researchers to conduct benchmarking and comparative analyses of various MBRL algorithms to evaluate their effectiveness and limitations. The benchmarking efforts in Wang et al. (2019) revealed that no single MBRL algorithm consistently outperformed others across all test scenarios. Instead, different algorithms excelled in different scenarios, suggesting the potential for combining components from multiple models to create a more robust approach. Within this study, three significant challenges, or dilemmas, were identified, which posed difficulties for all candidate MBRL models.

The first dilemma is the dynamic bottleneck, where MBRL models reached performance plateaus well below their implicit model counterparts. The second is the planning horizon dilemma, where most MBRL models achieved optimal results with shorter planning horizons due to the exponential expansion of search spaces (the curse of dimensionality in the state space). However, too short a planning horizon can also lead to a drop in performance. Lastly, the early termination dilemma emerged, where the standard technique of early termination, commonly used in RL to prevent excessive harm to the agent, significantly decreased the performance of MBRL agents

Wang et al. (2019). These findings present challenges for MBRL agents and emphasize the areas that require further investigation and improvement.

Despite the aforementioned challenges, MBRL models have found increasing applications in various domains, extending beyond video games and traditional research fields where RL methods have predominantly been employed in the past decades. In a study by Doll et al. (2012), it is suggested that our own brains utilize both approaches, implying that a combination of explicit and implicit methods may also hold relevance in robotics applications. This perspective encourages the exploration and consideration of potential benefits that can arise from integrating both MBRL and other approaches in the field of robotics.

2.5. Artificial neural networks: universal function approximators

In subsection 2.4.5, the utilization of artificial neural networks (ANNs) as potent function approximators in reinforcement learning (RL) was emphasized. Due to their significant influence on modern RL, often referred to as deep RL, this section will briefly cover the fundamental concepts of ANNs without delving into excessive detail. ANNs are employed in this project, both as part of the DRL algorithms and to account for disturbances in the proposed dynamic vehicle model.

Artificial Neural Networks (ANNs) are computational models inspired by the structure and functioning of biological neural networks in the human brain. They consist of interconnected nodes, or "neurons," organized into layers. Information flows through these layers, with each neuron performing a simple computation on its inputs and passing the result to the next layer (Russel and Norvig 2016, p.727-736). The computation of each neuron is shown in equation 2.7, which describes the process from input signals a_i to output signal y. The neuron's computation involves a linear combination of adjustable weights w_i and inputs, followed by the application of a non-linear activation function. Without the non-linear activation function, the neuron would only be able to approximate linear functions. Neurons learn by calculating the error (referred to as the loss L) of the output signal compared to the correct output. The gradient of the loss, $\frac{\delta L}{\delta w_i}$, is then computed by backtracking the operations performed within the neuron. Tuning the weights to minimize the loss is achieved using the *delta* rule, which adjusts the weights in the direction of minimizing the error. This approach, known as gradient descent, involves computing the gradient and descending towards the minimum value of the loss (Russel and Norvig 2016, p.727-736).

$$y = f\left(\sum_{i=1}^{n} w_i \cdot a_i + b\right) \tag{2.7}$$

A single neuron can generally only solve linearly separable problems, and to enhance its power, neurons are organized into networks. Figure 2.4 illustrates a common network configuration where layers of neurons are arranged in depth, and each neuron is connected to all neurons in the previous and next layer. This configuration, known as a dense feed-forward network, enables data flow in one direction while learning occurs in the opposite direction, after error is computed at the end (Russel and Norvig 2016, p.727-736). The network consists of an input layer on the left side, where input data is inserted, and an output layer on the right side. Each layer between the input and output layers is called a "hidden layer." In theory, a single hidden layer with a height of n can solve an n-dimensional problem. However, organizing the layers in depth, rather than

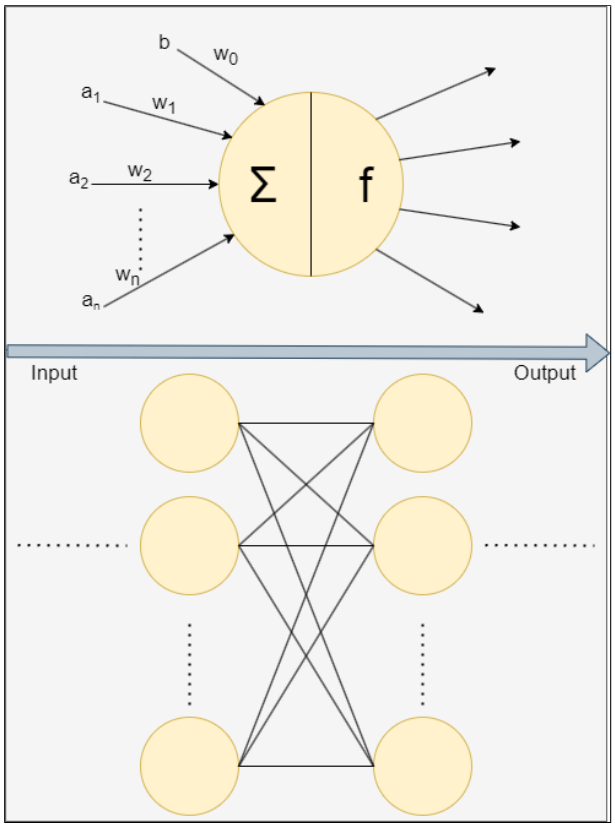


Figure 2.4.: A simple schematic of an artificial neural network (ANN). In this configuration, each neuron is densely connected to all nodes in the previous and the next layer of the network. This type of configuration is called a densely connected, feed-forward network.

just stacking neurons in height, provides greater computational power as it allows one layer to abstract a problem for the subsequent layers to process (Sutton and Barto 2020, p. 225).

Explicitly calculating the gradient for updating each weight in the network becomes exponentially more computationally expensive when adding more dense layers of nodes. To address this, stochastic gradient descent is used, which involves updating weights after computing the gradient for a single data point, without waiting for the full gradient to be computed. However, this introduces a lot of variance during training as the gradient that solves one instance of the data may not be the best gradient for all the data. Variance can to some degree be beneficial as it allows the system to avoid local optimal solutions, which would occur if the full gradient were computed and the system became stuck. Variance can also cause a slow training process, as one update negates the training of the previous update. Instead of computing gradients for every single instance, a mini-batch of data can be used. A mini-batch is smaller than the entire dataset, and using a smaller or bigger batch is a trade-off between variance and speed (Goodfellow et al. 2016, p. 288-300).

Using ANNs with several layers is referred to as deep learning and is a powerful tool that can solve tasks ranging from image recognition to drone piloting. However, ANNs require significant time and data for training. Different variations of the dense feed-forward network, such as convolutional neural networks for multi-dimensional data and recurrent neural networks for time-dependent patterns, are optimized for different tasks (Goodfellow et al. 2016, p. 326-415).

3. Dynamic vehicle model

This chapter presents the dynamic vehicle model that will be used for simulating and controlling the vehicle. The model chosen is a coordinated turn motion model (CTM), which is suitable for representing non-holonomic vehicles limited to circular motions in the plane, and is showcased in figure 3.1. The model plays a crucial role in the agent discussed in Chapter 5, serving as the basis for control schemes, simulations, testing, and algorithm training. Towards the end of the chapter, limitations of the simple CTM, such as error accumulation and instability, will be examined. The initial version of the CTM outlined in the preceding pre-project thesis serves as the basis for the content presented in this chapter, resulting in some overlap with the corresponding chapter in the thesis.

3.1. Mathematical models

Mathematical models are essential tools for representing complex physical systems using mathematical language. These models enable the simulation of real systems, providing predictions of system outputs for given inputs without the need for extensive testing on the physical system itself. By developing a model, it becomes possible to achieve predictive and planning behavior, as well as train and test agents within a simulated environment. The process of creating a model involves defining a set of state variables that represent the system, based on the required or interesting states for the simulation. This can be accomplished using an Nx1 state vector $\vec{x}$, where N represents the number of states needed to describe the system. The next step is to understand the underlying physics of the system, determining how the different states interact with each other and how they are influenced by various inputs (Chen 2013, p.6-40).

A very common way of representing a discrete mathematical model of a linear system is the "ABCD" method, shown in equations 3.1 . The A matrix describes how the state affect each other from one time step to the next, while the B matrix maps from input to states. In this model

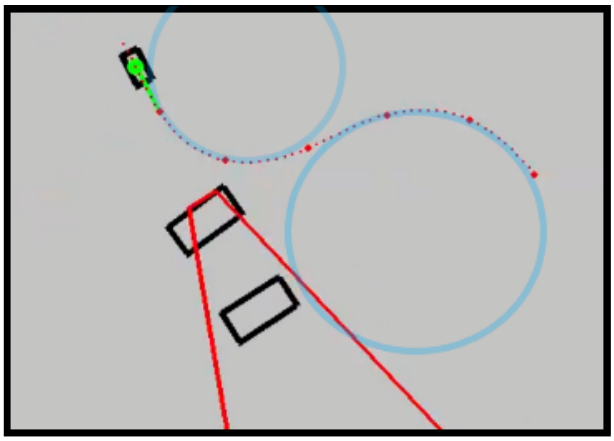


Figure 3.1.: In this figure, the circular motions are clearly shown for a vehicle driving with the CTM dynamics.

you also have the measurements, y, and C matrix mapping what states are actually observed or not. Lastly the D matrix is mapping how the inputs can directly affect our measurements, bypassing the system itself. As mentioned, this model is only intended for linear models, where all state variables can be expressed as a linear combination of the other state variables (Chen 2013, p.6-40). All mathematical model are simplification of the real counterpart, and linearization is one very common simplification step to apply to a non-linear system. Having a linearized system would again allow the "ABCD" structure, as well as many powerful control theory approaches which will be discussed further in chapter 5.

A widely used approach for representing discrete mathematical models of linear systems is the "ABCD" method, as shown in equations 3.1. The A matrix describes how the state affect each other from one time step to the next, while the matrix B maps inputs to states. The measured states y are determined by the matrix C, which indicates what states are observed. Finally, the matrix D describes how inputs can directly affect the measurements, bypassing the system itself. It is important to note that this model is specifically designed for linear systems, where all state variables can be expressed as linear combinations of other state variables (Chen 2013, p.6-40). Mathematical models are simplifications of their real counterparts, and one common simplification technique is linearization, which is often applied to nonlinear systems. Linearizing a system allows for the use of the "ABCD" structure and facilitates the application of various powerful control theory approaches, which will be further discussed in Chapter 5.

$$\vec{x}_{k+1} = \mathbf{A}_k\vec{x}_k + \mathbf{B}_k\vec{u}_k \tag{3.1a}$$
$$\vec{y}_k = \mathbf{C}_k\vec{x}_k + \mathbf{D}_k\vec{u}_k \tag{3.1b}$$

3.1.1. Non holonomic systems

A model, such as the CTM model described in this chapter, in which the possible states are constrained to paths within the state space, is referred to as a nonholonomic model. Unlike holonomic models, which are solely restricted by geometric constraints between the state variables, nonholonomic systems impose additional constraints that limit the position of the modeled object to specific paths in the state space Soltakhanov et al. (2009). Modern cars, employing variations of Ackermann steering geometry, serve as examples of nonholonomic systems, as their movement is constrained to circular paths where the vehicle's heading direction aligns with the tangent of the circle. Conversely, a human walking represents a holonomic system, capable of changing direction on the spot and not being restricted to specific paths.

3.2. Ackermann drive

This section explores the Ackermann vehicle model. The LIMO robot platform used in this project (see Section 1.4) can be configured as an Ackermann-driven vehicle, making the following description relevant for later discussions. The Ackermann steering geometry is named after Rudolph Ackermann, who patented this design for horse-drawn carriages in 1818 (N. 1906, p. 63), as shown in figure 3.2. However, the principles of Ackermann steering are still employed in modern-day cars. The aim of this method was to prevent the wheels of the carriage from slipping while turning, which would occur with a fixed-wheel design.

The key principle is to arrange all wheels in such a way that their axles converge toward a common center point, as depicted in Figure 3.2. Only the front wheels can be steered, and the center point must align with the rear-wheel axle. To achieve this steering mechanism, the inner wheel must

turn at a sharper angle than the outer wheel, causing the axles to intersect at a common center point that lies along the back wheels' axle (N. 1906, p. 63). Without this differential steering, using a fixed angle between the front wheels, the wheels would slip during turns.

The pure form of Ackermann steering described above is not commonly used in modern cars; instead, various extended or modified versions are employed. For instance, race cars adopt a scheme known as "reverse Ackermann geometry" (Milliken et al. 1995, p. 715). In the case of the LIMO, the steering geometry is simplified by utilizing a fixed angle between the front wheels, resulting in some slipping during turns. The consequences of this simplification are further explored in chapter 6.

Another consequence of Ackermann steering geometry is that the vehicle can only move in the direction indicated by the front wheels. In other words, the heading vector of the vehicle will always be tangential to the circular motion path of the vehicle, as illustrated in Figure 3.4.

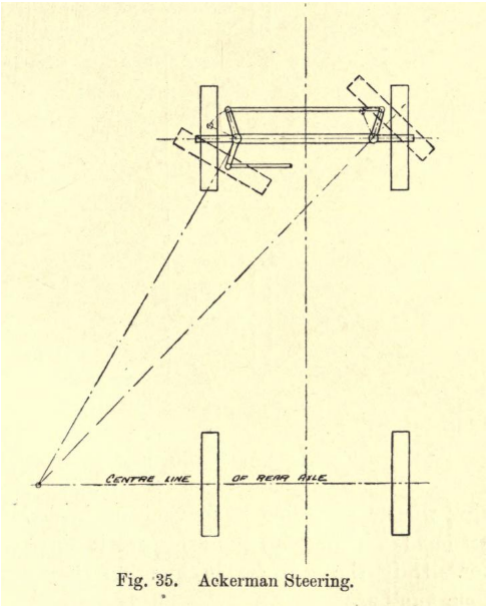


Figure 3.2.: The principle of Ackermann steering, as presented in N. (1906)

Figure 3.3.: The LIMO, developed by **AgileX**

3.3. Simple CTM

For a vehicle with Ackermann steering geometry, all possible motions can be described as circular paths of varying radii on the plane. One straightforward approach to representing such

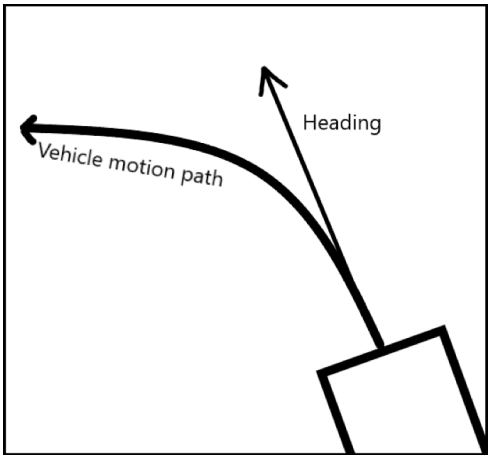


Figure 3.4.: The heading vector is always tangential to the motion path, while using Ackermann steering geometry

circular motion is through the *Coordinated Turn Motion* (CTM) model, as depicted in Figure 3.1. Interestingly, CTM can be applied not only to cars and other wheeled vehicles but also to ships and boats steered by a single rudder. While this thesis focuses on cars, it should be noted that for boats, the driving direction of the vehicle would need to be reversed, as the steering axle is opposite. The model presented in this section takes steering signals from the wheel and the throttle as input and simulates the circular motions of a vehicle. The model requires a set of parameters, and by appropriately tuning these, it can simulate any non-holonomic vehicle constrained to circular paths on the plane. Since this model primarily serves computer simulations, it is discretized in this thesis. A similar continuous description of CTM, which inspired the formulation in this thesis, can be found in Yuan et al. (2014).

3.3.1. Coordinate frames and abbreviations

First it is needed to establish coordinate frames important parameters to be used throughout this section. The coordinate frames used for this model is shown in figure 3.5, showing a world coordinate frame (WCF), denoted with W, a vehicle coordinate frame (VCF), denoted V, and a center coordinate frame (CCF) denoted C. The world coordinate frame is a static environment frame, and all static and dynamic objects will have coordinates in this frame. The CTM motion model will also be implemented mainly in this frame. The vehicle coordinate frame is centered about the back axle of the vehicle, as the vehicle is in this case assumed to be an Ackermann driven vehicle that turns about the back axle (see section 3.2). for a ship or a boat that turns with a rudder, the vehicle coordinate frame needs to be flipped about its y-axis. The center coordinate frame will be used later, to create circograms (see chapter 4), and is a coordinate centered around the center of the vehicle, but with the same rotation as the vehicle coordinate frame. When addressing a coordinate in a specific coordinate frame, the relevant letter will be added like this: (x^v, y^v) (example for vehicle coordinate frame). If no specific coordinate frame is given, it should be assumed that it exists in the WCF.

An overview of parameters to be used can be seen in figure 3.6, and they are briefly explained in table 3.1. In this report, ω and α will be used extensively to represent turn rate and steering angle respectively. The turn rate ω is the rate at which the heading ψ of the vehicle changes, while the the steering angle α is the angle of the wheels/rudder of the vehicle. Instead of using α, a equivalent model could also use $c = \frac{1}{R}$; being the inverse curve radius of the turning motion, and the mapping between the two is shown in equation 3.2. The equation simply shows the

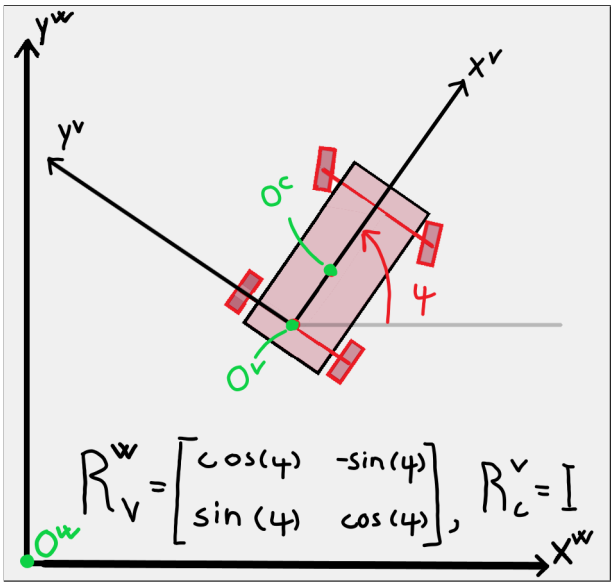


Figure 3.5.: A sketch of all relevant coordinate frames for the motion model. ψ is the heading of the vehicle, given in world coordinates.

relation between turn radius R, tangential speed $\vec{v}_k$ and turning rate ω_k. In the CTM, the speed of the vehicle is always tangential to the circular motion, while linear motions occur when the turn radius approached infinity. Further, the equations shows how steering angle α relates to the turn radius, given the distance between front and back axle of the vehicle d, which is derived from figure 3.6, which shows how these parameters are related.

$$\omega_k = |\vec{v}_k| \cdot c_k \tag{3.2a}$$

$$\tan(\alpha_k) = \frac{d}{R_k} = c_k d \tag{3.2b}$$

$$\implies c = \frac{\tan(\alpha)}{d} \tag{3.2c}$$

3.3.2. Linear transition dynamics

First, the linear transition dynamics, that exists for a subset of the states will be discussed. These variables are position of the vehicle in WCF and velocity in both directions of the WCF, which is a 4x1 vector, as shown in equation 3.3. These are the most important states for simulating the movement of the vehicle, but other, non-linearly dependent states will be discussed further in section 3.3.4.

$$\mathbf{X_k} = [x_k, y_k, \dot{x}_k, \dot{y}_k]^T \tag{3.3}$$

Given the coordinate frames shown in figure 3.5, the transitions/update rules of the position and velocities of the vehicle can be described as seen in equation 3.4. This is the CTM model described by Yuan et al. (2014), and uses the fact that in the circular motion ω_k, the turn rate at time k, is required to find the affects of the speed vector on the position. As the vehicle is non-holonomic, moving along circular paths, the $\dot{x}_k$ alone cannot be used to predict the next

Relevant parameters	
θ	The turn angle of the vehicle, given in VCF
ω	The turn rate of the vehicle, given in VCF
r	Throttle input signal
z	Steering input signal
d	Distance between front and back axel of vehicle
α	The current steering angle, imposed by the front wheels
ψ	The heading of the vehicle, given in WCF
v	The current speed of the vehicle
α_{ref}	The steering angle reference, resulting from z
v_{ref}	The speed reference, resulting from r
c	The inverse curve radius, $c = \frac{1}{R}$
K_v	The velocity proportional gain
K_α	The steering angle proportional gain

Table 3.1.: Symbols used in the motion model.

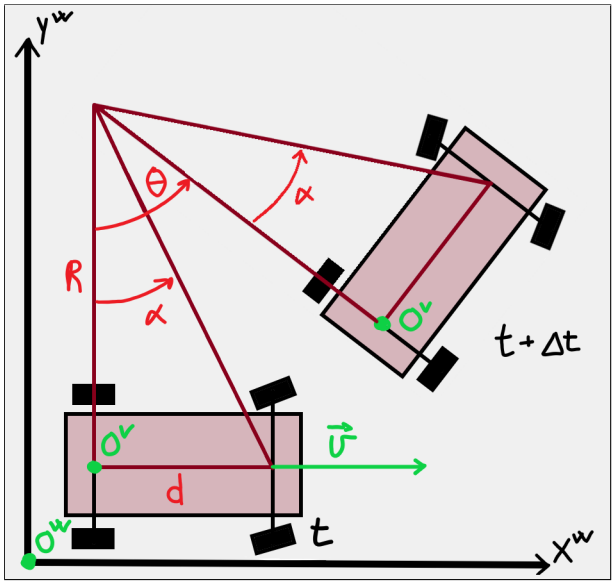


Figure 3.6.: A sketch of the CTM system, and the involved parameters needed to model the system mathematically.

x-position. If the vehicle was holonomic, $x_{k+1} = x_k + \dot{x}_k \Delta t$ would suffice for a discrete update rule. The velocity update rules can be seen as a rotation, or projection of the vehicles velocity vector of the vehicle, onto the two coordinate axis.

$$x_{k+1} = x_k + \frac{\sin(\omega_k \Delta t)}{\omega_k}\dot{x}_k - \frac{1-\cos(\omega_k \Delta t)}{\omega_k}\dot{y}_k \tag{3.4a}$$

$$y_{k+1} = y_k + \frac{1-\cos(\omega_k \Delta t)}{\omega_k}\dot{x}_k + \frac{\sin(\omega_k \Delta t)}{\omega_k}\dot{y}_k \tag{3.4b}$$

$$\dot{x}_{k+1} = \cos(\omega_k \Delta t)\dot{x}_k - \sin(\omega_k \Delta t)\dot{y}_k \tag{3.4c}$$

$$\dot{y}_{k+1} = \sin(\omega_k \Delta t)\dot{x}_k + \cos(\omega_k \Delta t)\dot{y}_k \tag{3.4d}$$

Then, seeing that all these variables are linear combinations of each other, the state transition matrix $\mathbf{A}$ is constructed as seen in matrix 3.5, and the update rule becomes $\mathbf{X}_{k+1} = \mathbf{A}_k \mathbf{X}_k$. The transition matrix stays constant, as long as the turn rate of the system is kept constant. An interesting observation, is that the matrix seems to have singularities, for linear motion when the turn rate is zero. This is not the case as shown in equation 3.7 by using L'Hopitals rule, and the matrix holds for the linear motion case as well. However, since this transition model will be implemented in code later, a seperate A-matrix is created for linear cases to avoid code crashes, and is shown in matrix 3.6.

$$\mathbf{A}(\omega) = \begin{bmatrix} 1 & 0 & \frac{\sin(\omega_k \Delta t)}{\omega_k} & -\frac{1-\cos(\omega_k \Delta t)}{\omega_k} \\ 0 & 1 & \frac{1-\cos(\omega_k \Delta t)}{\omega_k} & \frac{\sin(\omega_k \Delta t)}{\omega_k} \\ 0 & 0 & \cos(\omega_k \Delta t) & -\sin(\omega_k \Delta t) \\ 0 & 0 & \sin(\omega_k \Delta t) & \cos(\omega_k \Delta t) \end{bmatrix} \tag{3.5}$$

$$\mathbf{A}(\omega)|_{\omega \to 0} = \begin{bmatrix} 1 & 0 & \Delta t & 0 \\ 0 & 1 & 0 & \Delta t \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{3.6}$$

$$\lim_{\omega \to 0} \frac{\sin(\omega \Delta t)}{\omega} = \lim_{\omega \to 0} \frac{\frac{\delta}{\delta \omega}\sin(\omega \Delta t)}{\frac{\delta}{\delta \omega}\omega} = \lim_{\omega \to 0} \frac{\cos(\omega \Delta t)\Delta t}{1} = \Delta t \tag{3.7a}$$

$$\lim_{\omega \to 0} \frac{1-\cos(\omega \Delta t)}{\omega} = \lim_{\omega \to 0} \frac{\frac{\delta}{\delta \omega}(1-\cos(\omega \Delta t))}{\frac{\delta}{\delta \omega}\omega} = \lim_{\omega \to 0} \frac{\sin(\omega \Delta t)\Delta t}{1} = 0 \tag{3.7b}$$

3.3.3. Input signals

In this section, the input signals will be modelled into the system described in section 3.3.2. For this application, the possible actions are to turn the steering wheel, and to adjust the throttle. Instead of implementing separate breaking dynamics, the behaviour of a modern electric car will be copied - where letting go off the throttle will reduce the speed - as this is how breaking is implemented on the LIMO. Instead of assuming the driver / agent of the system sets the steering angle (α_{ref}) and reference velocity (v_{ref}) directly, an abstraction layer is added between the two

called signals. The steering wheel signal will be given as $z \in [-1, 1]$, where -1 corresponds to maximum steering to the left, and $+1$ to the right. For throttle the signal is named $r \in [-1, 1]$ where all values above 1 corresponds to forward driving, and all below 0 corresponds to reversing. If the current throttle signal is below the previous signal, the vehicle will break its speed, and if not it will accelerate. The use of signals is done to abstracts away the mechanical / electrical effects of controlling a vehicle and presenting it as a simple choice of numbers for the driver instead - much like the use of steering wheel and throttle is an abstraction for human drivers in normal cars are.

$$v_{ref} = \begin{cases} k_{max} \cdot r & \text{if r} \geq 0 \\ k_{min} \cdot r & \text{otherwise} \end{cases}$$
$$\alpha_{ref} = c_{max} \cdot z \tag{3.8}$$

Equation 3.8 shows the linear relations added between signals and reference values for α and v. This is a simplification, and im reality the relation is not linear, and can depend on parameters like the electrical motor time constant, mechanical motor friction, gears and gear shifts and so on. The K_{max} and K_{min} are used as a vehicle might have different maximum speeds in the forward and backwards direction, while it is assumed that a vehicle has the same maximum steering angle in both directions.

$$|\vec{v}_{k+1}| = |\vec{v_k}| + K_v(v_{ref} - |\vec{v}_k|) = |\vec{v}_k| + K_v e_{v,k} \tag{3.9a}$$
$$\alpha_{k+1} = \alpha_k + K_\alpha(\alpha_{ref} - \alpha_k) = \alpha_k + K_\alpha e_{\alpha,k} \tag{3.9b}$$

$$K = \frac{\Delta t}{\tau + \Delta t} \tag{3.10}$$

After the reference values are set, their effect on the vehicle is modeled as a first order system response, to a step input from the driver / agent, meaning the input references values map to the actual values as a first order differential equation. This type of response can be observed in figure 3.7, and which shows the first order response to a series of step inputs. In this system, the first order dynamics are shown in equation 3.9. This equation also holds a gain, K, for each of the two inputs. The gain is directly proportional to the time constant of the system, as shown in equation 3.10. The time constant indicates how long it takes for a system to adapt to a new step input, where 5τ is the time it takes for a first order system to reach 99.3% of the final output. This value needs to be tuned, to closely match the response of the real system. Hidden behind this gain or time constant is all the mechanical and electrical properties of the vehicle, which is heavily simplified in this use case.

Inputs directly affecting the state variables are modeled using the B_k matrix, when using the "ABCD" model Chen (2013), as described in section 3.1. By adding the B-matrix to the linear model presented in section 3.3.2, the resulting model is shown in equation 3.11. First, notice that an extra state variable is added to $\mathbf{X}$, being the steering angle α, which does not directly affect any of the other state variables in the linear model. The update rule of α, shown in equation 3.9 is simply added to the B-matrix, while the speed update rule is a bit more convoluted. This is because the normed velocity vector is not directly used as a state variable, but the speed vector

$\vec{v} = [\dot{x}, \dot{y}]^T$ itself. To go from speed norm to the vector itself, the current heading ψ_k of the vehicle has to be taken into account, as shown. This relationship is derived from figure 3.5, where it can be observed that the heading is the angle from the $\mathbf{X}^W$ axis to the $\mathbf{X}^V$ one.

$$\mathbf{X_{k+1}} = \mathbf{A_k X_k} + \mathbf{B_k} u_k \tag{3.11a}$$

$$\begin{bmatrix} x \\ y \\ \dot{x} \\ \dot{y} \\ \alpha \end{bmatrix}_{k+1} = \begin{bmatrix} 1 & 0 & \frac{\sin(\omega_k \Delta t)}{\omega_k} & -\frac{1-\cos(\omega_k \Delta t)}{\omega_k} & 0 \\ 0 & 1 & \frac{1-\cos(\omega_k \Delta t)}{\omega_k} & \frac{\sin(\omega_k \Delta t)}{\omega_k} & 0 \\ 0 & 0 & \cos(\omega_k \Delta t) & -\sin(\omega_k \Delta t) & 0 \\ 0 & 0 & \sin(\omega_k \Delta t) & \cos(\omega_k \Delta t) & 0 \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix}_k \begin{bmatrix} x \\ y \\ \dot{x} \\ \dot{y} \\ \alpha \end{bmatrix}_k + \begin{bmatrix} 0 & 0 \\ 0 & 0 \\ \cos(\psi_k) K_v & 0 \\ \sin(\psi_k) K_v & 0 \\ 0 & K_\alpha \end{bmatrix}_k \begin{bmatrix} e_v \\ e_\alpha \end{bmatrix}_k \tag{3.11b}$$

When it comes to the **C** and **D** matrices of the **"ABCD"** representation shown in equation 3.1, those are not relevant currently. The **C** matrix represents which of the state variables that are being observed or measured, using predictors or sensors. Currently the entire state vector is assumed available, so that matrix would be the identity matrix. For the **D** matrix, no direct effects from input to output is currently modeled, so it would be set to the zero matrix.

3.3.4. Non-linear dynamics

In the model presented above in section 3.3.3, there are two important pieces missing - the turn rate ω_k and the heading of the vehicle ψ_k at time step k is not yet derived. The first step is to use figure 3.6 to derive the turning angle θ_k, which will then be used to find both ω and ψ. From the figure it is observed that $\tan(\alpha_k) = \frac{d}{R_k}$ which is the same relation as expressed in equation 3.2, and is used in equation 3.3.4 to make an expression for θ_{k+1}. From the derivations, and expression for ω_{k+1} is also produced. Both of these expressions, as well as the one for heading is shown in equation 3.13. Heading is simply updated, by adding the new turning angle, which can be a positive or negative angle, depending on the direction of the turn. As seen in the model presented in section 3.3.3, the other state variables depend on these new variables in a non-linear fashion, and cannot be easily added to the linear transition matrix **A** and is kept separate for simplicity sake.

$$\theta_{k+1} = \omega_k \Delta t = \frac{|\vec{v}_k|}{R} \Delta t = |\vec{v}_k| \frac{d}{dR} \Delta t = \frac{|\vec{v}_k| \tan(\alpha_k)}{d} \Delta t \tag{3.12}$$

$$\theta_{\mathbf{k+1}} = \frac{|\vec{v}_k| \Delta t \tan(\alpha_k)}{d} \tag{3.13a}$$

$$\omega_{\mathbf{k+1}} = \frac{|\vec{v}_k| \tan(\alpha_k)}{d} \tag{3.13b}$$

$$\psi_{\mathbf{k+1}} = \psi_k + \theta_{k+1} \tag{3.13c}$$

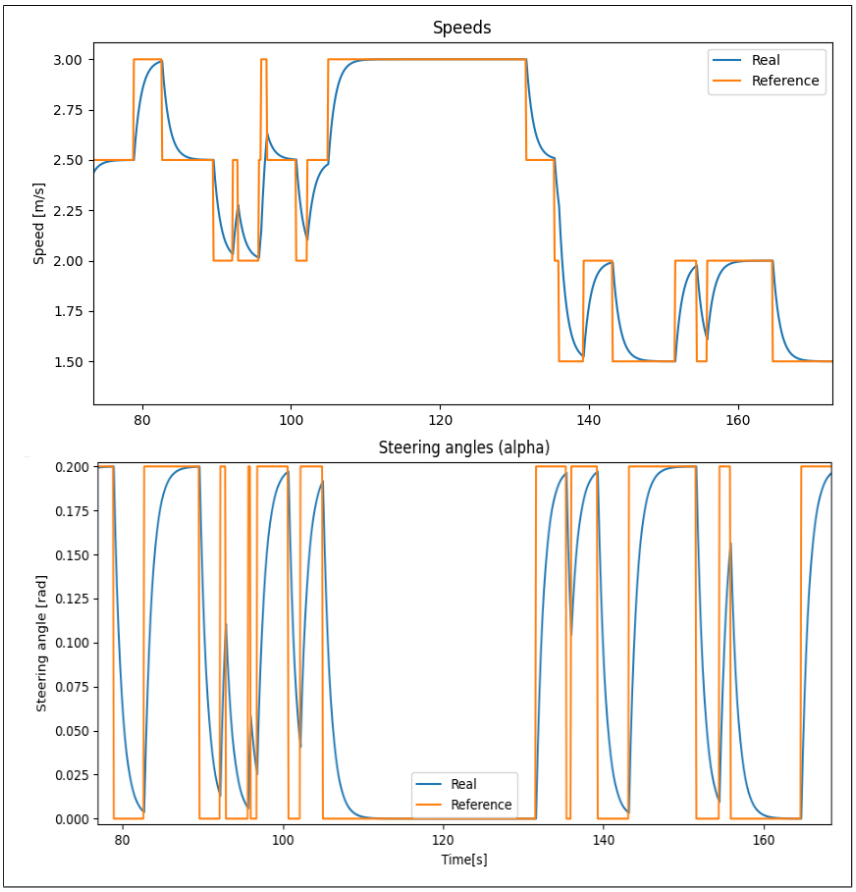


Figure 3.7.: Showcase of how the steering inputs / references relate to the real values as first order responses to the step inputs. The vehicle has no breaking, similar to the LIMO robot platform, so breaking is done by setting a lower speed signal than the current speed signal.

Again, it is important to point out that this is a discrete model, which for simplicity assumes that values for turn rate or velocity is kept constant while updating for the next step. The same is done in the non-linear part of the model, where it is assumed that turn rate and heading is constant during an update. This is a limitation of discretization, that becomes a problem if the value of Δt is set too large. From the Nyquist-Shannon sampling theorem, Δt should be at least half that of the fastest / smallest time constant of the system itself, to avoid problems with the discrete sampling of the continuous system Por et al. (2019). However, in addition to the discrete sampling of a continuous process, simulating it can also accumulate error and cause instability depending on what order of approximation is used. This will be discussed further in section 3.5.

3.3.5. Full model overview

Finally, the discoveries of the last three sections can be combined to a full model. The overview can be seen in figure 3.8, with some small alterations. In the figure, it is not shown how the matrices $\mathbf{A}_{k+1}$ and $\mathbf{B}_{k+1}$ are calculated, using the outputs of the previous simulation. Also, some new $\mathbf{C}$-matrices are added, to sample the relevant variables from the full state space vector $\mathbf{X}_k$. These matrices are shown in equation 3.14, and are simply multiplied by the state space vector to get the relevant variable(s). Such a vector is often called an observation vector, and can be used to indicate what measurements / predictions are available when calculating the input vector $\mathbf{u}_k$, through sensors and predictors Chen (2013).

$$\mathbf{C}_1 = \begin{bmatrix} 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \end{bmatrix} \mathbf{C}_2 = \begin{bmatrix} 0 & 0 & 0 & 0 & 1 \end{bmatrix} \mathbf{C}_3 = \begin{bmatrix} 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix} \tag{3.14}$$

3.4. System disturbance

As the vehicle model will be used to simulate real-world dynamics and predict future trajectories, it is valuable to incorporate errors into the model. This allows for testing the impact of having an imperfect model on system predictions. Since this project focuses on the digital/simulated environment rather than the physical robot, introducing errors is necessary to prevent a perfectly accurate predictive model during testing.

The term "disturbances" refers to all effects that are not accounted for in the simplified system model. Together with noise from sensory systems, disturbances contribute to errors in any control system. Disturbances can manifest as mechanical effects such as frictions, drag, collisions, servo slack, or electrical effects. Many of these disturbances are unbiased, meaning that, on average, they do not decrease the average accuracy of the model since they equally affect model predictions in all directions around the expected values. However, some disturbances, such as friction or servo slack, are biased and introduce predictable systematic errors that can be measured and compensated for (Chen 2013, p. 294, 334, 336). In this project, biased disturbances are of particular interest, as the subsequent system architecture will include components to account for these biases, which need to be tested.

Introducing disturbances in this project is achieved simply by adding or subtracting values from certain model parameters. These parameters are the time constants τ_α or τ_v, the input signal coefficients $k_{\max}$, $k_{\min}$, and $c_{\max}$, or the wheelbase distance d. Adding errors to these parameters is much simpler than directly modifying the transition matrix $\mathbf{A}$. The matrix $\mathbf{A}$ represents the

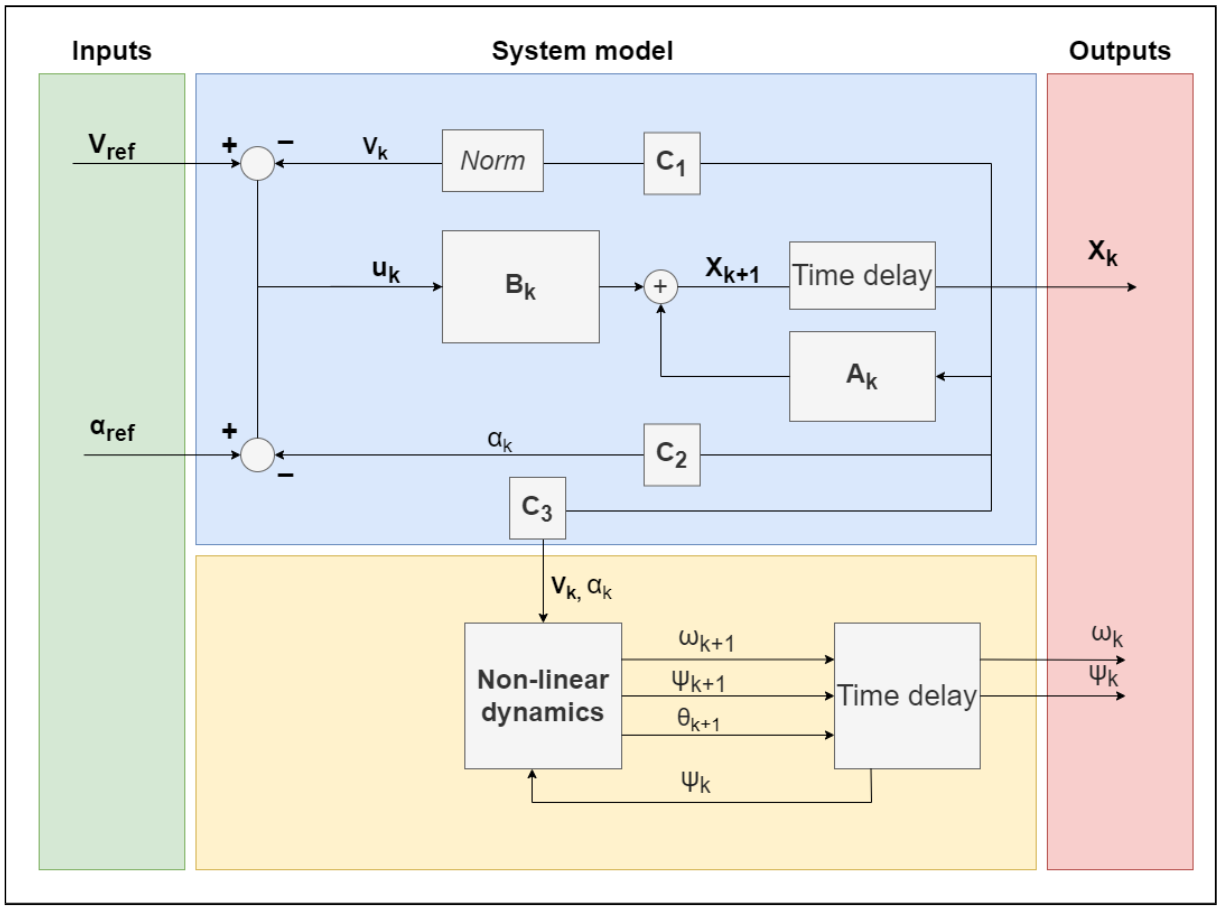


Figure 3.8.: The complete simple model system schematic. The modules dealing with linear and non linear system dynamics are separated by the color scheme of the figure. The blue box contains all linear dynamics from section 3.3.2, and the yellow box contains the non linear dynamics of section 3.3.4. **C**-matrices are used to sample relevant variables from the full state vector $\mathbf{X}_k$.

kinematic model of the system, and introducing random numbers into its entries would likely result in an unphysical system that lacks real-world coherence. Additionally, as mentioned in Section 3.3.2, **A** contains a rotation matrix for the velocity vector, and adding random numbers to a rotation matrix can disrupt its properties as a rotation matrix. Rotation matrices belong to the special orthogonal (SO(n)) group, which is a Lie group for each n. Introducing disturbances without compromising the properties of the group would require applying *Lie algebra* Sola et al. (2018).

3.5. Notes on numerical error and instability

The system described in this chapter, being a vehicle with Ackermann steering geometry, is partly described by a collections of differential equations (**DE**) as seen in equation 3.11. Such collections of DEs generally has no direct solution. This means that given a set of inputs to the system, being α_{ref} and v_{ref}, it is generally not possible to calculate the outcome state of the system explicitly (Sauer 2006, p. 283). In the modelling of the system done in this thesis, Euler's method is being used, both to discrete and simulate the system. Euler's method, shown for a single variable is shown in equation 3.15, and is the simplest formulation of an explicit Runge-Kutta solver (Sauer 2006, p. 314), being a first order approximation of the actual system. The "h" in the equation is some discrete time-step (Δt), while the y represents some state or variable that varies over time, as described by **f**. The function **f**, depends on both the previous value for y, as well (in some time-variant cases), the current time to determine the next value of y.

$$y_{n+1} = y_n + hf(t_n, y_n) \tag{3.15}$$

Being a first order approximation means that in the Taylor Series, that can be used to approximate the differentiation of a function **g** according to the Taylor Theorem, only the first term of the series is kept, and the rest discarded. Equation 3.16, shows the Taylor series for an arbitrary, but twice differentiable function g. In the equation, $g(x+h) = g(x)+hg'(x)$, is the first order approximation, as it only keeps the first differentiation term while approximating the differentiation of the system. An example of a first order approximation is ignoring acceleration and assuming constant speed to determine the next position of a car. It can be observed that this first term, is in fact (with some re-naming), the Euler's method, where the function $f(t_n, y_n) = \frac{d}{dt}y_n$. The second term, $\frac{h^2}{2}g''(c)$, which is discarded, represents the cumulative error of using this approximation. This error is called the "truncation error", as the taylor series has been truncated (Sauer 2006, p. 294) and its easy to see that the smaller the time-step **h**, the smaller the accumulated error of the approximation will be (Sauer 2006, p. 244).

$$g(x + h) = g(x) + hg'(x) + \frac{h^2}{2}g''(c) \tag{3.16}$$

So, where in the system model is Euler's method being used? It is being used in equation 3.4a and 3.4b, to estimate the next position. In this case, the function **f** from equation 3.15 is given by $f = \frac{\sin(\omega_k \Delta t)}{\omega_k}\dot{x}_k - \frac{1-\cos(\omega_k \Delta t)}{\omega_k}\dot{y}_k$, and depends on the current speed as well as turn-rate to estimate the next value. This is no exact solution, as it assumes constant velocity throughout the time-step, and as discussed, an error will accumulate depending on the size of the time-step Δt. The update to the speeds ($\dot{x}_k and \dot{y}_k$) and are then computed explicitly, with no approximations, and then used to compute the turn-rate of the next time-step (see equation 3.13). The immediate

consequence of using the Eulers method / first order approximation, is shown in figure 3.9. The error is shown to accumulate over time, especially when using large time-steps. One interesting thing to note, is that the error will sometimes drop back down. This is due to the effect of changing directions; so if the right turn was under-estimated, then the left turn will be under-estimated as well, and the position will get closer to "reality".

Generally, decreasing the time-step will reduce the truncation error, but doing so is also expensive as it requires more calculations by the simulation. The eigenvalues of the state-transition matrix $\mathbf{A}$ can be used to estimate the dynamics of a system, and therefore to predict how small the step size needs to be in order to capture the fastest dynamics (Sauer 2006, p. 284). In some extreme cases, with sufficiently high time-steps, the Euler method can also become unstable, as the numerical approximation is so slow to react to fast dynamics that the numerical method keeps overshooting more and more, and the error grows exponentially. To avoid this, step size (Δt) is chosen so that for each eigenvalue λ, $\Delta t\lambda$ is within a center of radii 1, centered around -1 on the complex plane. In this thesis, this is not done, but Δt is chosen experimentally to avoid instability. Figure 3.10 shows what can happen to the cumulative error, if the numerical method becomes unstable.

Another thing to note, is that error in figure 3.9 is not always changing, sometimes it is stationary. This happens when the vehicle does not accelerate as much, and the constant speed approximation becomes correct for a short duration. Some more advanced numerical methods, like Runge-Kutta 4, 5 (RK45), takes advantage of this, by using a more accurate / higher order approximation (RK5), to determine what step-size is needed in the lower order RK4 simulation. This method is called "embedded pairs", and it improves performance of the simulation, by matching low step-sizes to high activity levels in the system (high acceleration), and increasing the step-sizes when the activity level is lower (constant velocity / low acceleration). For this, the error between the more accurate fifth-order approximation RK5 method and the less accurate forth-order approximation RK4 method is used to detect where RK4 starts to drift, and the step size needs to be adjusted (Sauer 2006, p. 326). In the test conducted in figure 3.9, the simulation is not compared to the real system, but rather to another simulated system running at a lower time-step, which is going to be closer to reality as the error term is smaller. This is similar to what is done with embedded pairs, and is a lot simpler than comparing to the real system, which would require driving the real vehicle and recording its movement accurately. For the applications of this thesis, Eulers methods will suffice, as outside of testing and training agents, the simulation will only be used to simulate short 3-10 second trajectories, as will be discussed in chapter 7. Setting a decently low step size is therefor all that is needed, to achieve sufficient accuracy without costing too much performance wise. In addition to the explicit Euler's method that is employed in this study, an implicit (also known as backwards) version of Euler's method also exists. Although the implicit method possesses superior stability properties, it necessitates additional computational steps for each time-step, which were deemed unnecessary for the current application.

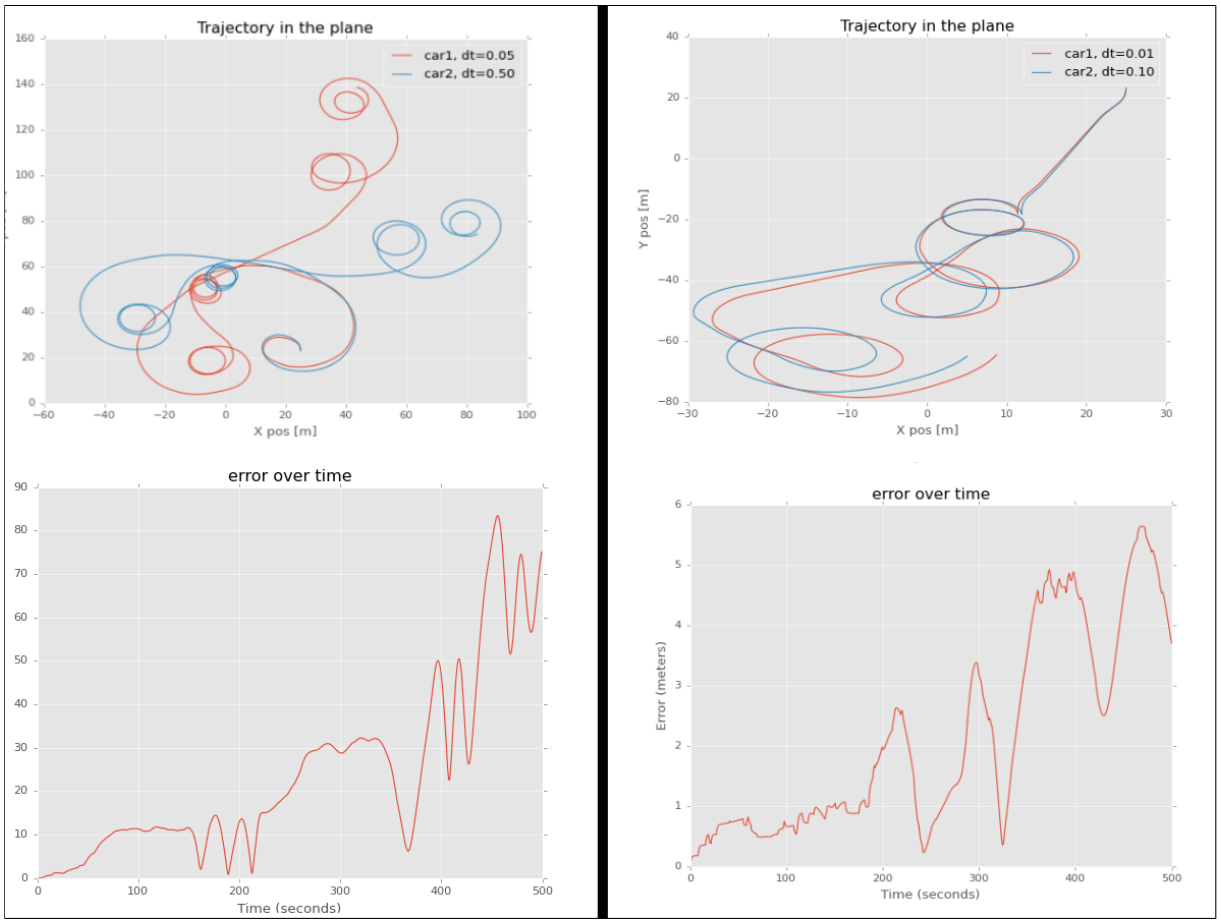


Figure 3.9.: This figure illustrates the numerical issues using a lower order approximation method, like Euler's method, to simulate real system dynamics. The error generated by such an approximation, is dependent on the time-step used in the simulation, but will grow gradually over time. The experiments are done by running two cars with the same simulated physics over a duration of 500 seconds. On the left, time-steps of 0.05 seconds and 0.5 seconds are used, and the vehicles end up far away from each other, as the simulation error grows to almost 100 meters. In the left-most scenario, where the time-steps are both lowered, the errors are a lot lower.

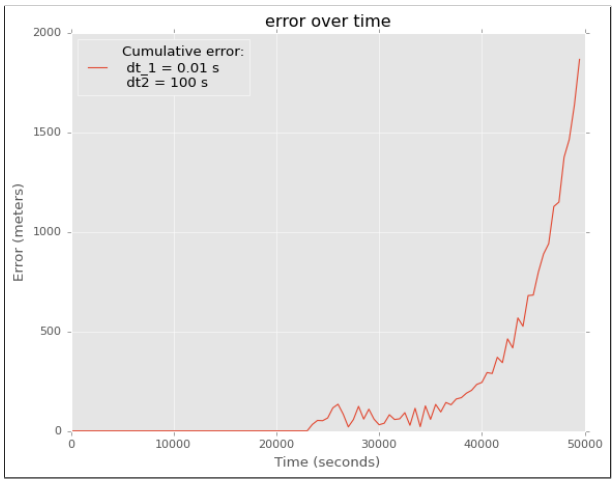


Figure 3.10.: This figures illustrates what can happen when using Euler's method as numerical approximation, where the time-step is a lot larger than the fastest transients in the system. Instability causes the cumulative error to grow exponentially over time, as the approximation keeps over / under-shooting more and more with each time-step. This is known as numerical instability, and renders the approximation or simulation utterly useless.

4. Environment and representation

Apart from the vehicle model discussed in Chapter 3, an essential consideration is the design of the environment surrounding the agent and how it is perceived. This chapter presents the environment and explores various options for representing it to the agent. The project's focus is on short-term planning, which does not necessitate a comprehensive global understanding of the entire environment but rather a localized snapshot. Consequently, this project does not involve the exploration, utilization, or creation of consistent global maps. A showcase of the difference between the two approaches is shown in figure 4.1. It is important to note that the chosen environment and representation are specifically tailored to the **LIMO** robotic platform, taking into account its available sensory systems, potential missions, and driving dynamics.

4.1. The environment

When creating a simulated environment for this task, it was crucial to develop something that could resemble the environments in which the **LIMO** would learn and eventually be deployed. Despite the **LIMO**'s sensors, such as depth cameras, providing a 3D understanding of its surroundings, a decision was made to keep the environment simple by using a 2D, top down simulation. The design of the environment draws inspiration from top-down, 2D **grid-worlds**, which serve as the foundation for many reinforcement learning (RL) testing environments Sutton and Barto (2020). The rationale behind this approach is that if the agent can learn to plan within this 2D environment, it can then utilize its available sensor data to perceive and translate its real-world surroundings into the more familiar 2D environment. For instance, LIDAR data can be projected onto ground level to quickly generate a 2D map directly from the sensor data. Other sensor data, such as stereo vision, could also be employed as long as it provides a perception of distance or depth.

To mimic the dynamics of the **LIMO** (or at least as closely as possible in a computer simulation), the environment will be continuous, while also incorporating stochastic elements with adjustable parameters for noise and disturbances (see section 3.4). These characteristics make the environment more alike the real-world environments that the robot could be deployed to later (Russel and Norvig 2016, p. 45). There will be various implementations of the environment, as depicted in figure 4.2, some consisting solely of static walls, while others include semi-static elements with a few moving parts. These dynamic obstacles, however, will not be other agents but rather reactive dummy agents (see section 4.2.2.1), thus maintaining a single-agent environment for the time being. All dynamic objects in the environment, including the agent itself, will be described using the dynamic vehicle model developed in chapter 3.

4.1.1. Local momentary maps

The objective of this project is to explore short-term trajectory planning, which requires some form of map or environment understanding. Without the ability to look ahead or predict future states on a map, planning becomes impossible. Providing the agent with a map of the entire simulated environment would enable long-term planning, but it is also unrealistic for the agent's

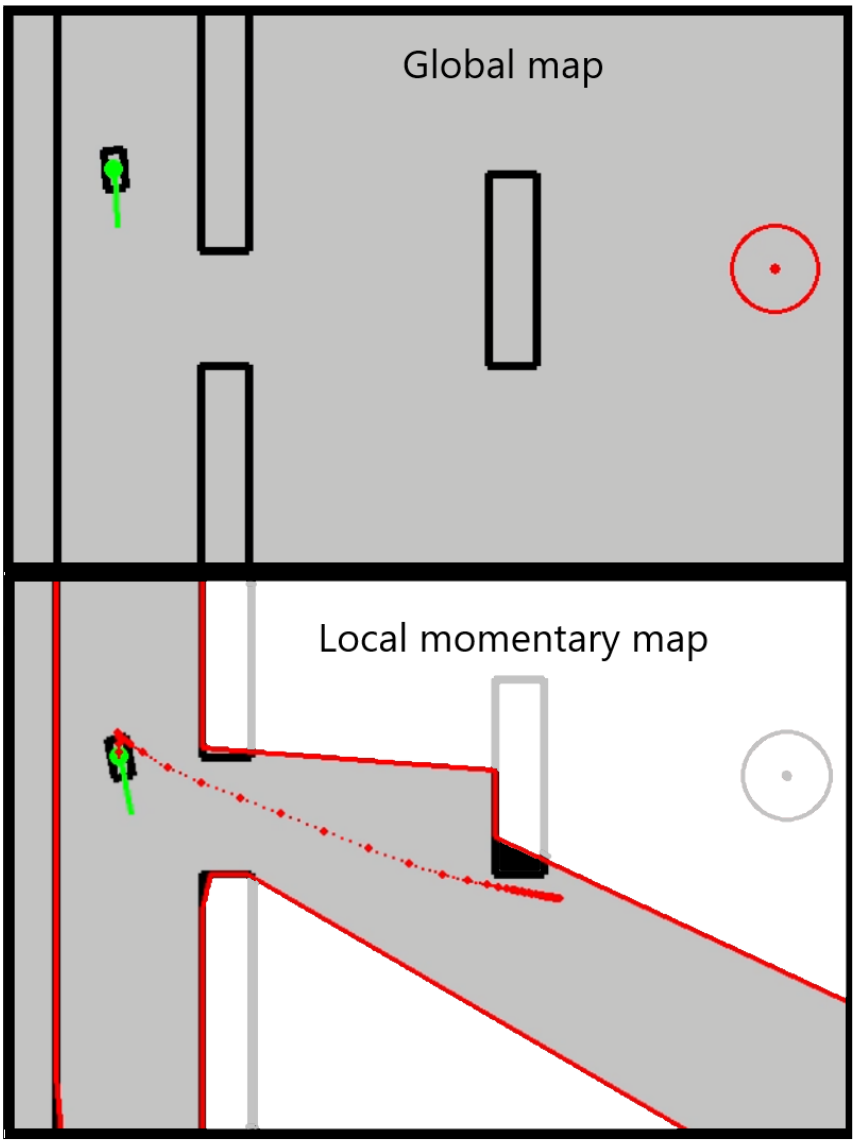


Figure 4.1.: The difference between having a global, or a local map for planning. The lower half of the figure is a preview of how the PDRL algorithm uses the vision box (red outline) to plan trajectories in the local map.

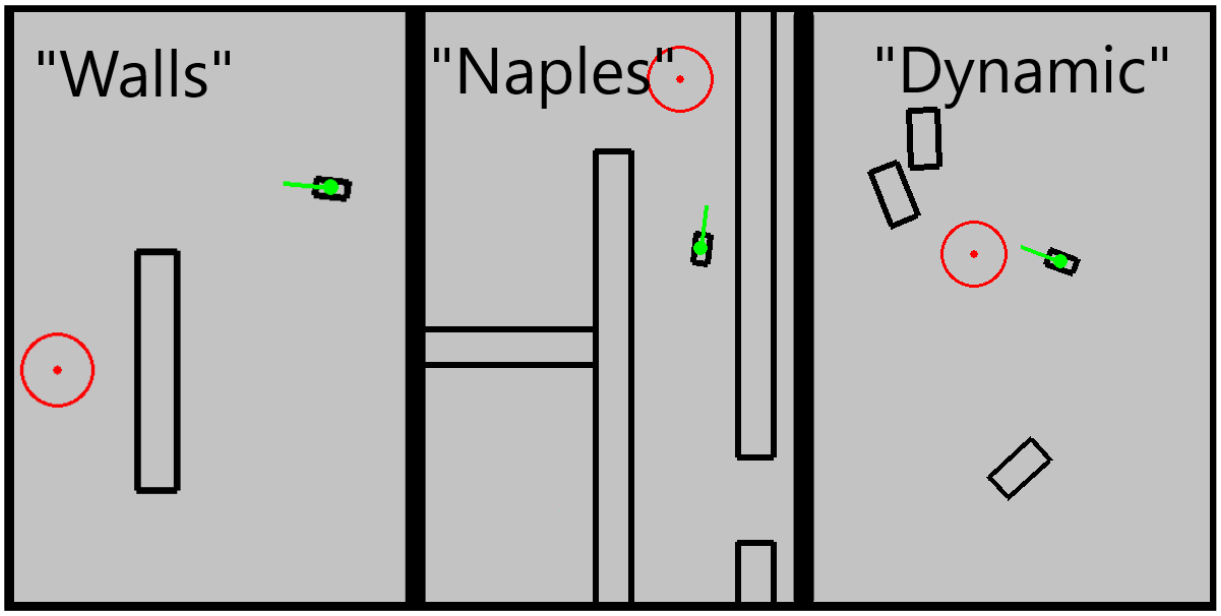


Figure 4.2.: The different configurations later used for testing. The green arrow indicates where the agent is located and heading, and red are goal states. "Walls" is an open area, with an outer perimeter, as well as walls that block of the goals. "Naples" is a lot more constricted, with narrow passages and tight gaps. "Dynamic" on the other hand contains both a static outer perimeter, as well as dynamic obstacles that move around to obstruct the agent. The dynamic obstacles are based on the reactive agents developed in appendix C.

later deployment. Due to the stochastic, semi-static, and partially observable nature of the environment, with vision-impairing obstacles, the agent must rely on **local momentary** maps for planning. These maps are generated in real-time based on the current available information. The dynamic and stochastic nature of the environment necessitates frequent map updates, and the maps are only applicable for short-term planning. The creation and updating of local maps will be discussed in detail in section 4.2.3.

While it is indeed possible to generate and maintain a globally consistent map from sensor data, this task is complex and deserves its own dedicated thesis. One branch of methods aimed at solving this problem are called simultaneous localization and mapping (SLAM) algorithms. They involve combining different sensory data with noise models, vehicle models, and non-linear optimization techniques to create global and consistent maps for the exploring agent. However, for this thesis, the agent will only have access to local data when making decisions.

4.1.2. States and actions

When determining the state space of the agent, it is important to provide it with enough information to solve its tasks and with enough possible actions to reach desired states. A system is considered *controllable* if there exists a sequence of actions that can take the agent from the current state to a desired goal state. As the system is a non-holonomic car-like vehicle, the reachability of goal states may vary regardless of the chosen actions. Checking controllability for linear systems is relatively straightforward given a state and action space (Chen 2013, p. 184), but the system used in this case contains non-linear components. However, since the agent is given the same available actions as a human driver (see section 3.3.3), which involve steering the wheel and manipulating the throttle, it can be assumed that the agent should be able to reach similar

target states as a human driver. Braking is handled similarly to the braking on the LIMO itself, where fully releasing the throttle results in maximum deceleration.

Observability is a property of a system that allows retracing the initial state given the sequence of actions taken and visited states. The observability of a system depends, among other factors, on how many of the system's states are observed by the agent, as relevant states must be visible or derivable from the observed states (Chen 2013, p. 193). Once again, for the system used in this thesis, it is sufficient to consider the information available to a human driver while driving and provide the same information to the agents. They should then be able to generate short-term plans in the local environment. The *sufficient* state is shown in equation 4.1 and includes the current speed $|\vec{v}|$, current wheel input z, current throttle position r, and the target location specified in the vehicle coordinate frame $(x, y)^V$. Please refer to chapter 3, and specifically figure 3.5, for the coordinate systems used in this project.

$$S_k = \begin{bmatrix} x_k^V \\ y_k^V \\ |\vec{v_k}| \\ z_k \\ r_k \end{bmatrix} \tag{4.1}$$

4.1.3. Goal states

For whatever mission the agents are given to solve, they need be given target or goal states to reach. These goals states can be final goal states, that solve the mission, or intermediate waypoints that needs to be reached in order to reach a final goal. As mentioned, this project will focus on local maps and short term planning, so the goals states selected for the agent will be in close proximity to the current position of the agent. If an agent is able to solve these short term and local goal tasks, it can later be combined with a global planner that breaks down a large task into several local tasks. That concept is explored in Leung et al. (2021) where a deep neural network is trained to select goals for a lower level action selection neural net. This hierarchical approach is beyond the current project, which will focus on short term goals only. What missions are to be solved will be discussed further in section 5.1.

4.2. Obstacle representation

Since both static and dynamic obstacles will be present in the environment, the sufficient state space reveled in equation 4.1 needs to be expanded to include obstacle representation. The objective is for the agent to utilize its onboard sensory systems to detect and map the physical environment into a 2D local momentary map of its surroundings. Therefore, the focus is on how obstacles are represented to the agent. In this case, the chosen representation is static circograms, which can accurately convey the distances to obstacles. These circograms can be further developed to enable planning in the perceived environment. Two approaches was explored for achieving this: dynamic circograms and the vision box.

4.2.1. Circograms

Circograms, or static circograms (SC), are distance-based 2D environment representation tools that consider the actual distance from the vehicle's hull to obstacles in its surroundings. The circogram represents similar information as the raw LiDAR sensor data, consisting of an array

of distances at fixed angles from the ego-vehicle. However, unlike LiDAR, which measures the distance from the sensor to obstacles, the circogram projects these distances to the center of the vehicle and incorporates information about the shape of the ego-vehicle's surface. An example of such a representation is shown in Fig. 4.3. From the circogram, it is possible to determine the real distance from each side of the vehicle to potential obstacles, not just from the sensor to the obstacle.

Circograms can be constructed directly from LiDAR data if the relative location of the sensor on the vehicle and the shape of the vehicle's hull are known. They can also be constructed from other distance information sources such as radar, ultrasound, stereo vision, and more. The sensory data is first used to create a 2D representation, known as the local momentary map, of the surrounding obstacles. The circograms are then generated within this local map of the environment. Circograms are not only used to represent the current situation of the agent but also simulated for future situations, enabling the vehicle to plan its actions using the simulated circograms. Different techniques for simulating future circograms, such as dynamic circograms and the vision box, will be discussed later in this chapter. For further implementation details of static circograms, refer to Appendix B.

While the concept of a circogram is not new and the term "circogram" is used for various concepts in different fields, the specific application and name "circogram" for this context were coined by R. Mester and P. Klose in Klose and Mester (2018). In the implementation presented in this paper, circograms are assumed to be provided as a set of azimuthal directions, offering a momentary abstraction of the current driving situation without considering the kinematic aspects or changes in the environment.

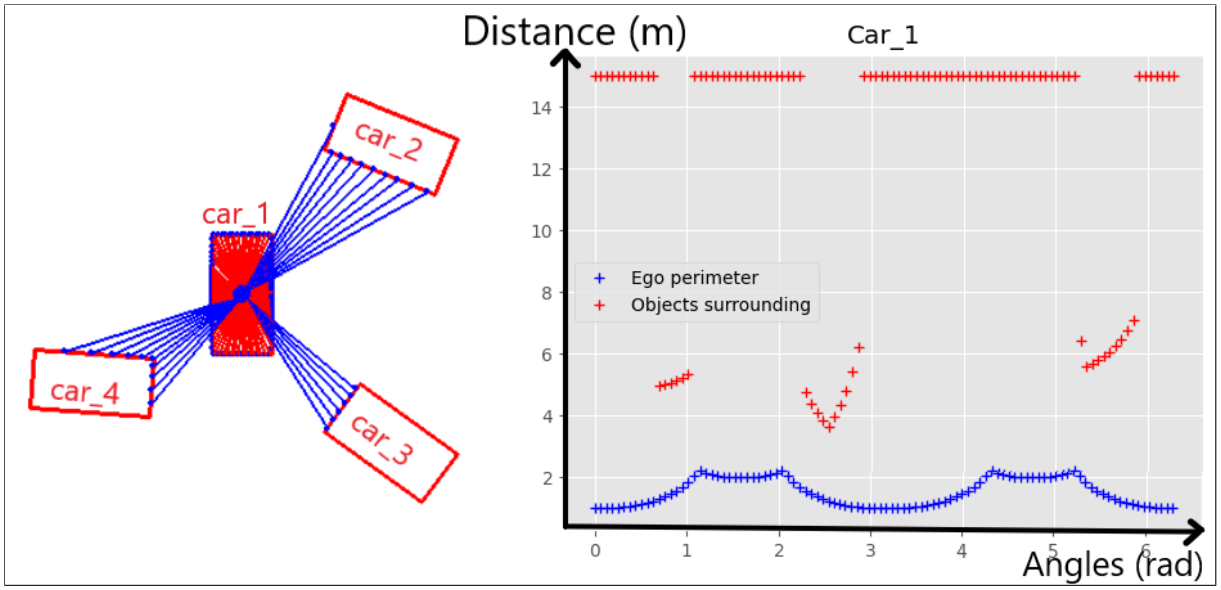


Figure 4.3.: The static circogram. The distances to all distant objects are measured or computed and compared to the size of the vehicle's hull to obtain the real distance. The graph in the figure shows the angle at which the measurement is taken and the measured/computed distance.

4.2.2. Dynamic circograms

This is the first approach for simulating future inputs to the agent, that can be used for planning, that will be discussed. As distance alone is not a reliable indicator of collision risk, the distance-based circograms can be expanded. For example, if the agent is driving down a narrow alley, it

should not pose a problem as long as it maintains a straight path. In the case of a non-holonomic vehicle like a car or a boat, it is not possible to move in all directions, so those directions should not be considered risky for the driver. Consider a car driving on a road: even if the distance to the sidewalk is only half a meter, it is the objects several meters ahead that pose a greater risk. A dynamic circogram includes not only distance information but also the rate of change of distances to obstacles to calculate the risk of collision.

There are various ways to implement a dynamic circogram (DC). Some possible approaches include combining a series of static circograms, tracking objects in the environment to determine relative speeds (as done in Pfeiffer and Franke (2010)), or using the motion of the ego-vehicle in conjunction with the current circogram. In this thesis, the focus will be on the last approach, as the dynamic vehicle model is already implemented as a motion model (see Chapter 3).

Fig. 4.4 illustrates the implementation of dynamic circograms. At a discrete time-step t_n, the vehicle's motion model is used to predict the future position of the vehicle's center, given the current steering inputs and a predetermined Δt. The size of Δt is a tunable parameter that determines how far into the future the DC should look. A vector from the current center position to the new center position, denoted as $\vec{a}$, is then constructed. Next, given a list of circogram rays (containing the real distance to surrounding objects), normalized vectors $\hat{r}$ along each ray are constructed, and the risk is calculated as shown in (4.2) using dot products. Here, l represents the predicted distance the vehicle will move along a given ray, while d_{real} represents the "real" distance obtained from the static circogram. In this context, θ represents the angle between the two vectors, and since $\hat{r}$ is normalized, l corresponds to the length of the projection of the vehicle's motion onto a given circogram ray.

$$l = \vec{a} \cdot \hat{r} \tag{4.2a}$$

$$\implies l = |\vec{a}||\hat{r}| \cos\theta \tag{4.2b}$$

$$\implies l = |\vec{a}| \cos\theta \tag{4.2c}$$

$$risk = \frac{l}{d_{real}} \tag{4.2d}$$

One could imagine that having a $risk \geq 1$ would imply a guaranteed collision within the next Δt seconds, as the projected motion along a ray would be equal to the distance along that ray. However, as illustrated in Fig. 4.4, when the obstacle moves relative to the vehicle, it could potentially move out of the way. Therefore, the "risk" value does not guarantee collision but is rather an indication.This is the drawback of hallucinating future risks and distances based on current information alone. Fig. 4.5 demonstrates one of the strengths of the representation, where walls to the left and right of the ego-vehicle are determined to be non-risky since the car is driving straight and cannot collide with these walls.

The next step is to propagate the circograms and risk to the next time-step t_{n+1} so that the agent can plan using perceived future risks. The turn angle θ_n of the vehicle is obtained when using the motion model to create $\vec{a}$. This angle is then used to rotate all the risk vectors, $\text{risk} \cdot \hat{r}$, to align with the new orientation of the vehicle. This rotation is performed vector by vector using the 2D rotation matrix shown in Equation 4.3. Subsequently, these rotated risks are provided to the agent, which hallucinates the action it would take in such a state. This new action is used to determine a new $\vec{a}$, which generates new risk vectors based on the original distances. These new risk vectors are then rotated for the subsequent step. Hence, while the agent is stationary and has only generated a single static circogram to assess distances from its current

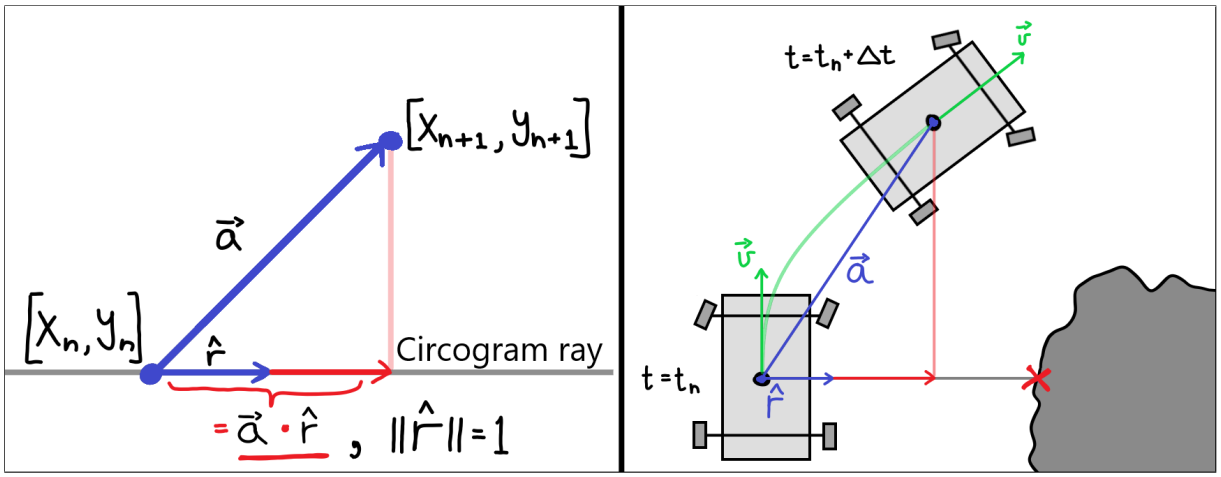


Figure 4.4.: The implementation of the dynamic circogram. The vehicles future position is predicted using the vehicle motion model, and then used to calculate risk of collision.

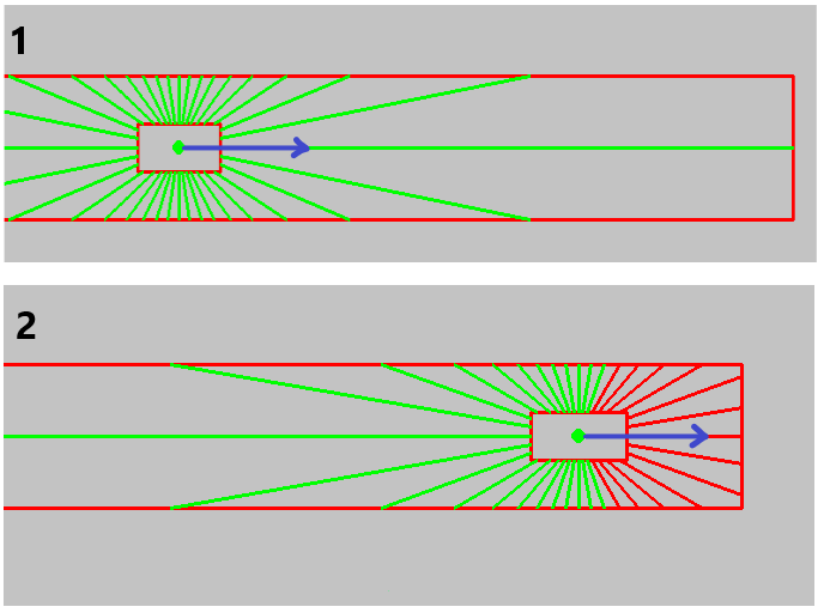


Figure 4.5.: An example use case for the dynamic circogram. The green lines indicate the directions in which distances are measured, while the red lines indicate risks that exceed the pre-determined risk threshold. The blue arrow represents the velocity vector of the vehicle. It is worth noting that risks are determined not only by distance but also by the expected motion of the vehicle.

position, it hallucinates multiple future risks using the motion model, dot products, and rotation matrices.

$$\begin{bmatrix} r_1 \\ r_2 \end{bmatrix}_{n+1} = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix} \begin{bmatrix} r_1 \\ r_2 \end{bmatrix}_n \tag{4.3}$$

4.2.2.1. Dynamic obstacles and reactive driver

A simple test/use-case for the DC environment representation, is to use it directly as input to a reactive agent. In a simulated environment, the agents uses dynamic and static circograms as its sole input to randomly drive around in an environment with multiple other similar drivers. This was developed both to test vehicle dynamics, and circograms, as well as to be used as " dynamic obstacles" in the later environments. More details are to be found in appendix C.

4.2.3. Vision box

The vision box offers a much simpler approach for generating future sensor data for decision making and planning. Figure 4.6 illustrates the concept of the vision box. It starts by generating a single circogram from the vehicle's current position, which is used to create a box around the agent. This initial circogram can be derived either from the local momentary map or directly from LIDAR data. The resulting box serves as a basis for simulating future circograms by selecting actions, hallucinating the vehicle's movement within the box using the motion model, and generating new circograms for the new positions. The agent can iterate this process to hallucinate a sequence of actions and positions within the vision box, while remaining stationary in the original position.

Compared to dynamic circograms (DC), this method is much simpler as it does not involve rotation matrices or dot products. However, it requires generating a new circogram after each time step. Similar to DCs, the generated sensory data is limited to the local boundary of the vehicle's vision range during planning. The distances from the circograms are incorporated into the sufficient state representation, as shown in equation 4.1, enabling the agent to learn to navigate using these distances.

Since the plan is constructed iteratively within the vision box, the accuracy of the generated circograms decreases over time, as the vision box represents only a momentary snapshot of the environment. Furthermore, additional information may be revealed as the agent executes the plan, such as the presence of a hole in a wall. To address this issue, re-planning is triggered using a vision-update rule, which will be discussed in chapter 5.

4.2.4. Feature engineered features

There are primarily two approaches to incorporating sensory data into a machine learning agent. One approach is to directly input the sensory data into a function approximator, enabling it to discover the relevant features within the data itself. This approach is closely associated with end-to-end machine learning methods, which will be further discussed in section 5.3.1. In end-to-end learning, a function approximator solves the entire task from input to output as a single module. Deep neural network architectures are commonly used for this purpose, as demonstrated successfully in Pfeiffer et al. (2017), where raw 2D LIDAR data alone was sufficient for a deep neural network agent to navigate a complex environment. The depth of the network allows different layers to extract features that subsequent layers can utilize for decision-making, without

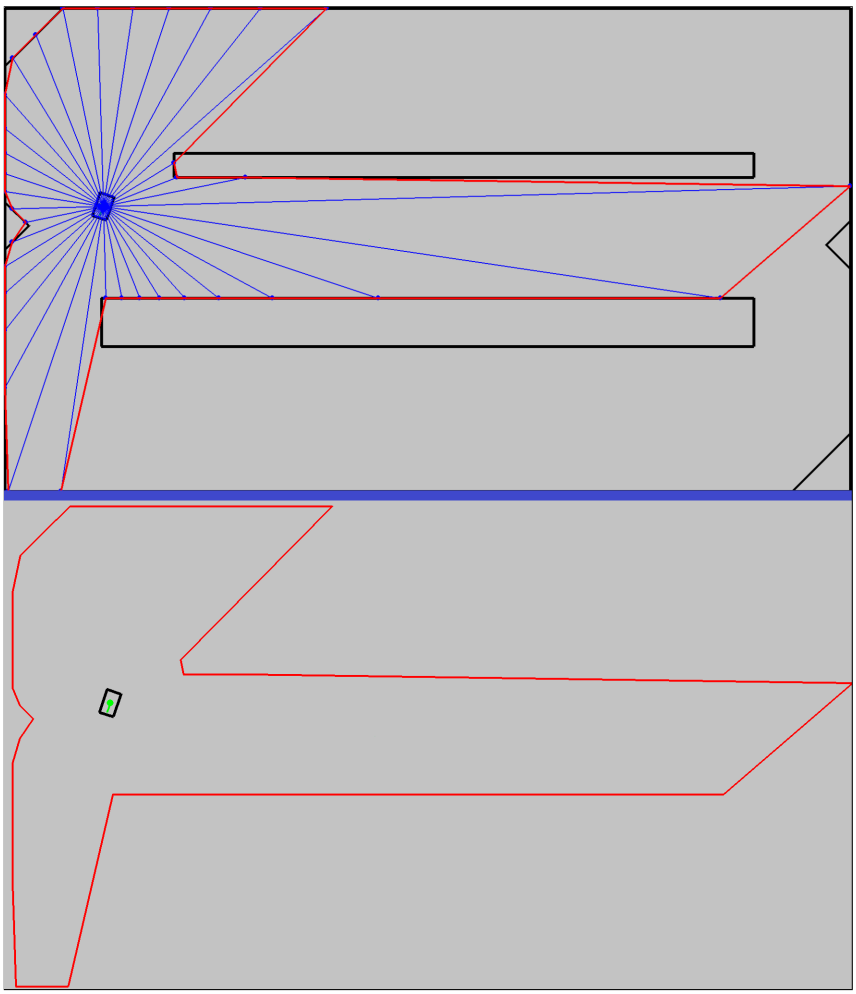


Figure 4.6.: The concept of the vision box. For this project, the vision box is a local momentary map used for planning sequences. The vision box is created from an initial circogram in the agents current position. This initial circogram can be derived either from the local momentary map or directly from sensor data.

requiring any explicit feature engineering by the human designer. Therefore, the deep network, when provided with both distance data and information about the vehicle's speed and rotation, can autonomously assess which distances pose a risk.

Alternatively, features can be engineered and extracted based on a human understanding of the environment and what might be important to the agent. Dynamic circograms, in their various forms, exemplify feature engineering, as the calculated risk is based on the designer's comprehension of what constitutes a risk. It abstracts the notions of distance and motion into the concept of risk, which is then fed to the agent. On the other hand, the vision box approach allows the agent itself to extract and learn the crucial features directly from the simulated distance data.

5. Methodology

This chapter covers the primary research efforts of the master thesis, in addition to the development of the environment as explained in previous chapters. It begins by establishing the goals and missions that the agent aims to accomplish, followed by an examination of the agent's components, and culminating in the presentation of the complete architecture. The architecture, referred to as the Predictable Deep Reinforcement Learning (P-DRL) framework, builds upon the principles of a Model Predictive Control (MPC) controller discussed in section 2.3, while replacing the Linear Quadratic Regulator (LQR) and linear regression models with deep reinforcement learning and artificial neural networks, respectively.

5.1. Mission planning and selection

In this project, an agent is assigned the task of moving from its current state to a local goal state by creating and executing short-term plans. Such a goal-oriented agent offers great flexibility, as the goal states can be modified to create a wide range of different missions for the agent. By explicitly representing goals as states to the agent, they can be easily changed by a human designer or another algorithm. For instance, a visual tracking system can be employed to detect and track objects within the agent's field of view, allowing the tracked object to become the agent's goal state, thus enabling the agent to follow the target. If an agent is capable of solving an environment by achieving a specified goal, the goal can be altered to enable the same agent to tackle a variety of different missions and tasks within the same environment, without requiring any modifications other than the goal itself. In contrast, designing a reflexive agent that solely reacts to current inputs would necessitate significant effort to change the agent's goal, as goal states are not explicitly represented in such a design (Russel and Norvig 2016, p. 50).

The primary mission of this project is the "timed joyride" mission, where random goal states are selected from the local environment. The current goal state is included as part of the agent's full state space, as discussed in section 4.1.2. The minimal goal state consists of the goal location given in the vehicle coordinate frame (VCF) $(x, y)^V$, but it can be extended to include goal velocity $|\vec{v}|goal$ and goal heading $\alpha goal$. The agent also needs to know its current velocity and heading, as shown in section 4.1.2, in order to determine necessary corrections.

The selection process for new goal states in the timed joyride mission is illustrated in figure 5.1, which depicts how local goal states are chosen to be reached within a specified time interval. If a goal state cannot be reached within the given time, a new goal state is selected. The goal states will primarily be randomly generated within a radius of the vehicle, but they can also be predetermined if there is a specific challenge or driving pattern that needs to be tested.

The mission planner employs a straightforward check to determine if the agent has reached its goal. It examines the agent's current state, which includes its location (in VCF), heading, and speed. Pre-defined thresholds are utilized to evaluate whether the goal has been achieved or not. This approach is effective for final goal states where precise heading and speed in a specific position are crucial for accomplishing specific tasks, such as parallel parking. However, for intermittent goal states with lower precision requirements, the binary thresholding can result

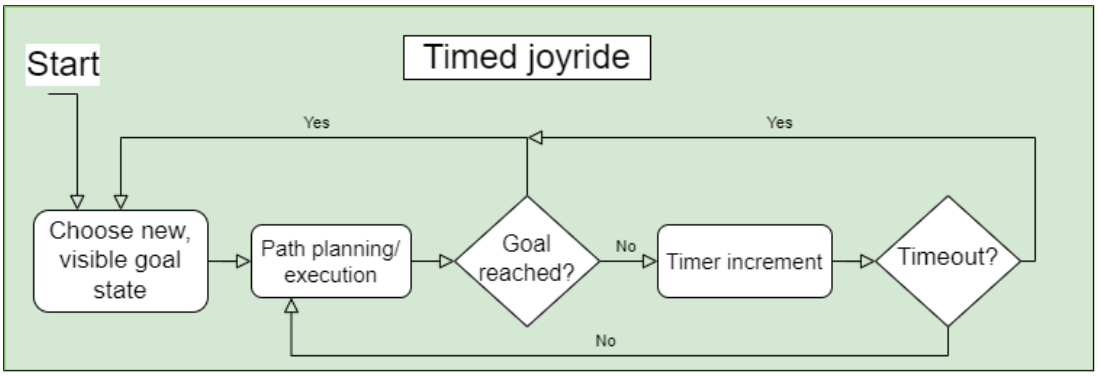


Figure 5.1.: This figure illustrates the command flow for the timed joyride mission. This mission will be used to benchmark the agent's performance. However, with a general goal-based agent design, the resulting agents are not limited to solving only this mission, as more sophisticated methods for choosing goal states can be employed to solve a wider range of missions.

in less smooth driving. Occasionally, the agent may miss its target and need to readjust its angle and speed. In the case of the timed-joyride mission, strict adherence to the goal states is unnecessary since there are no specific tasks to fulfill. Therefore, the thresholds were intentionally set to be generous.

5.2. The DRL component

The core component of the P-DRL architecture is the DRL algorithm. This component is responsible for selecting actions that will be used to construct a sequence of actions, or a plan. While there are numerous state-of-the-art DRL methods available, this project will focus on deep deterministic policy gradient (DDPG), which was developed in 2016 at *Google Deepmind*. Prior to reading this section, it is recommended to review Chapter 2, specifically section 2.4.

5.2.1. DDPG

Deep DPG is an extension of deterministic policy gradients (DPG) that incorporates deep neural networks to better handle large and high-dimensional action and state spaces. DPG is an off-policy implicit RL algorithm that employs an actor policy and a critic value function, enabling continuous action spaces and reducing training variance through bootstrapping Kiran et al. (2021). (Refer to 2.4 for more details.)

This method draws inspiration from successful deep Q-learning (DQL) algorithms by utilizing an experience replay buffer, which stores previous state-action-state-reward tuples for training the deep networks. Instead of solely relying on the most recent experience, the buffer allows for a broader range of data points to be averaged, leading to reduced variance in training Lillicrap et al. (2015). Stable training is typically faster, as the agent avoids the need to re-learn information frequently. However, including excessive data in the buffer can slow down training due to the time required for neural network training. In DRL, newer data is typically more relevant as the policy improves and approaches the task solution.

The ability of DDPG to learn from past experiences stems from its usage of the Bellman equation of optimality, similar to Q-learning. Being an off-policy learner, DDPG updates the critic's Q-value with respect to the current understanding of optimal value, regardless of the action cho-

sen in the previous experience. This differs from an on-policy learner like SARSA (see section 2.4.2).

Another concept borrowed from DQL is the employment of target networks, as discussed in section 2.4.5. During bootstrapping in RL training, overestimation of the value function can occur. This issue, known as maximization bias, arises from using the max function while updating the Q-value, as shown in equation 2.5. The solution involves maintaining two separate networks: one for action selection and another for value estimation. Equation 2.6 demonstrates this approach, where network Q_1 is used to train the target network Q_2. To minimize training variance, it is common to train the target network with a lower learning rate than the primary network. In DDPG, this is achieved by employing the update function depicted in equation 5.1, which gradually updates the weights (ϕ) of the target network by incorporating a small portion of the weights from the primary network. The parameter $\tau \in [0, 1)$ ensures that the target network approximates the primary network slowly over time Lillicrap et al. (2015). Both the actor network and the critic network has a target network, resulting in four total networks to train.

$$\phi_{target} \leftarrow \tau\phi_{target} + (1 - \tau)\phi \tag{5.1}$$

Finally, due to the deterministic nature of the policy (action selection function), it is necessary to introduce noise to the output actions during training. The original paper utilizes an Ornstein-Uhlenbeck (OU) process to generate noise for the actor. This process is particularly suited for physical environments with momentum, as it models the velocity of a Brownian motion particle experiencing friction, producing temporally correlated yet unbiased values for the process Lillicrap et al. (2015).

5.2.2. Choosing DRL algorithm

DDPG is not the latest and not necessarily the best performing DRL algorithm available. However, due to its extensive testing and proven performance in autonomous vehicles, it remains a strong candidate. In a 2020 study Yu et al. (2020), several DRL algorithms were compared for a navigation task, and DDPG was found to generate more optimal paths, converge faster during training, and exhibit good generalization to other tasks. In a real-world autonomous driving scenario, DDPG was selected in a study by Wang et al. (2018) due to its ability to handle the complex state space and continuous action space, resulting in a functional agent capable of fast and safe driving. Furthermore, in a 2021 survey by Kiran et al. (2021) comparing different (D)RL methods for autonomous driving, DDPG was highlighted as one of the most influential algorithms available.

Other notable contenders to DDPG are TD3 and SAC. TD3, or "twin delayed DDPG," was introduced in 2018 to address the sensitivity of DDPG to hyperparameter tuning. With improper hyperparameters, overestimation of the value function can occur, leading to a breakdown of the policy as it attempts to exploit these overestimated states. TD3 mitigates this issue by training two separate Q networks and consistently using the smallest/most pessimistic Q-value as the target network for value estimation during bootstrapping Fujimoto et al. (2018). SAC, or "Soft actor-critic," also published in 2018, incorporates stochastic policy optimization into DDPG-like algorithms. Unlike deterministic policy training, SAC trains a policy that maximizes the trade-off between expected return and randomness Haarnoja et al. (2018). Both TD3 and SAC claim superior performance compared to DDPG in various tests. TD3 asserts faster training and better performance in continuous control tasks, while SAC boasts significantly higher sample efficiency than DDPG, leading to faster training Fujimoto et al. (2018) Haarnoja et al. (2018).

Despite the promising results of TD3 and SAC, DDPG was chosen for this project. DDPG has been in use for a longer period than the two newer methods, resulting in a larger number of papers and studies focused on DDPG compared to the other two combined. This is evident in the number of citations on the DDPG paper, which exceeds the combined citations of the other two algorithms on platforms such as *Google Scholar*. The additional time has also allowed for numerous studies that specifically apply DDPG to autonomous vehicles. While TD3 and SAC papers primarily compare themselves to DDPG using toy problems as benchmarks, papers such as Kendall et al. (2019) demonstrate how DDPG can learn to drive a car in a physical environment, and how it can be taught to fly a drone in 3D space, as shown in Bouhamed et al. (2020). Another advantage of DDPG is the abundance of available code to facilitate project initiation.

5.3. The P-DRL Agent

This section covers the implementation of the P-DRL agent, which combines the MPC algorithm, a DDPG agent, and the dynamic vehicle model to create a short-term planner.

5.3.1. Structured Vs end-to-end architectures

In the field of autonomous vehicle design, a distinction arises between two system architectures: structured and end-to-end. Firstly, the end-to-end approach is examined. Within this framework, sensor data is directly inputted into a function approximator (typically a neural network), which generates control signals, effectively solving the entire problem in a single step. This approach has demonstrated significant success in autonomous navigation, as evidenced by research in Pfeiffer et al. (2017). In this paper, a deep neural network is directly fed 2D LIDAR data as well as the target location. From this input, the agent can safely navigate an obstacle course without collision, even transitioning from simulated to real-world environments. However, this approach does possess a notable limitation: using sub-symbolic function approximators like deep neural networks makes it challenging to predict the agent's intentions and understand the reasoning behind its actions. This limitation hinders both the ability to prevent future accidents and to analyze the causes of any that may occur.

On the other hand, the structured approach offers an alternative. It involves decomposing the decision-making process into multiple modules, allowing for a closer examination of each step. This modular design facilitates a deeper understanding of the system's operations, as the inter-modular communication can be observed and recorded. In this project, a combination of architectures is employed. The overall architecture is structured, but when decomposing the modules, there are end-to-end systems operating at lower levels.

5.3.2. Combining control theory and RL

The combination of reinforcement learning (RL) and control theory has already been explored in various applications, and is not a novel concept. RL and control theory methods share common ground, as both seek to find an optimal solutions to state spaces given a reward/cost function. More specifically, DDPG and MPC try to solve the problem for a continuous state space with unknown system dynamics.

A comparative study of model predictive control (MPC) and deep RL (DRL) was conducted in Lin et al. (2021) for the adaptive cruise control of a car. They utilized a first-order model to predict the system dynamics, and the RL learner was implemented using the DDPG algorithm. The results revealed that DRL outperformed MPC in the presence of disturbances and noise

while performing similarly to MPC in their absence. In Lubars et al. (2021), a unified control scheme that combines both methods in the context of on-ramp merging for cars was proposed. The authors found that DRL agents are prone to safety issues and poor robustness in scenarios beyond their training data. However, compared to MPC, DRL produced more efficient and comfortable rides in terms of jerk and fuel consumption. Therefore, they proposed a mixing algorithm that integrates both algorithms to operate where they perform the best. It is worth noting that these results do not necessarily generalize to all implementations of MPC and DRL, as there are many tunable parameters and different implementations of both algorithms.

In this paper the methods will be combined directly. Algorithm 1 tries to sketch a general MPC algorithm, using linear regression to model system dynamics from a replay buffer, and LQR to produce optimal control signals. In the P-DRL architecture, shown in algorithm 2, the linear regression will be replaced by an ANN trying to learn the model dynamics. As the system dynamics is already hand-crafted in chapter 3, the ANN will mostly try to correct for noise and disturbances that causes the proposed model to deviate from reality.

The **SOLVE()** function from algorithm 1, responsible for producing actions, will use DDPG instead of LQR. LQR requires the estimated model dynamics as input to find optimal control signals. However, as mentioned in section 5.2.1, DDPG is an implicit algorithm. The DRL algorithm does not keep or rely on an explicit representation of the model dynamics to choose optimal actions. Instead, the policy is learned directly from the estimated expected future rewards. However, as DDPG is only capable of outputting a single action for each inputted state, the vehicle model will be used to estimate future states, which can then be fed back to the DDPG agent for further action outputs. This is shown in the **SOLVE()** function in algorithm 2. The result is a sequence of actions from the DDPG agent, as well as a trajectory generated by the dynamic vehicle model, which can then be executed in the environment.

5.3.3. Generating future states

In algorithm 2, the model M is utilized to generate the subsequent hallucinated state H_{k+1} based on the current state and action. As explained in section 4.2.3, the input state of the agent encompasses both the *Sufficient state* defined in equation 4.1 and the distance data from the current circogram. Therefore, relying solely on the vehicle model discussed in chapter 3 is insufficient for acquiring the next state. It is also necessary to generate the circogram within the vision box, considering the current hallucinated position of the vehicle. The process entails generating the circogram within the vision box, obtaining the complete state vector, feeding it to DDPG to obtain a new action, simulating the next state within the vision box, and repeating the process. The end result is a trajectory of simulate states, and a sequence of actions that can be used to navigate the real environment.

5.3.4. Re-planning

In algorithm 2, a *sequence maintainer* (SM) is used to determine whether or not a new action sequence needs to be generated. Re-planning should be done every time-step. Not only since it requires computation, but also as that takes away the value of a predictable planning agent if it changes its mind all the time. However the plans should not always be allowed to run its full course, as the plan is made from the vision box, being a local and momentary understanding of the environment. Not only could dynamic environments change with time, but new and better paths might be discovered around corners.

The SM takes the current actual state X_t, the hallucinated equivalent state h and the remaining

Algorithm 2 P-DRL

Input: Control horizon N, total run-time T, A pre-defined vehicle model M
Output: Generates and executes optimal control signals $\{a_0, a_1, ..., a_T\}$

```
Buffer ← {}                                   ▷ State, action, state, reward - replay buffer
A ← {}                                        ▷ Action queue
REPLAN ← TRUE
for t = 0, 1, ..., T do
    X_t ← Current environment state
    if Buffer ≥ sample-size then
        Sample the buffer.
        M_ANN ← UPDATE                        ▷ The ANN based vehicle model
    end if
    if REPLAN then
        if M_ANN more accurate than M then    ▷ This is the simple approach. A better
approach would be to use M_ANN to correct the residual errors of M
            A, H ← SOLVE(X_t, M_ANN)          ▷ H is the hallucinated state trajectory
        else
            A, H ← SOLVE(X_t, M)
        end if
        REPLAN ← FALSE
    end if
    h ← POP(H)                                ▷ The hallucinated state h
    REPLAN ← Sequence-Maintainer(X_t, h, A)
    a ← POP(A)
    Take action a, observe next state X_{t+1}
    Append (X_t, a, X_{t+1}) to Buffer
end for

function SOLVE(X, M) ▷ Hallucinates a plan, using current state and a given vehicle Model
    DDPG-Buffer                               ▷ Is a maintained replay buffer for the DDPG
    Generate Vision Box from X
    H_k ← X                                   ▷ Rename the hallucinated states to H
    for k = 0, ..., N do                      ▷ Construct a plan
        a_k ← DDPG(H_k)                       ▷ Deep deterministic policy gradient
        H_{k+1} ← M(H_k, a_k)
        R ← Interpreter(H_k, a_k, H_{k+1})
        Append H_k, a_k, H_{k+1}, R to DDPG-Buffer
    end for
    if DDPG-Buffer ≥ sample-size then
        Sample the buffer and train DDPG
    end if
    return (a_k, H_k)_{k=0}^N
end function
```

action queue A as inputs. The simple first check is to see if there are any actions left in the queue. Next, each circogram distance measurement is compared between the two states to look for large deviations. The hallucinated circogram distances of h was generated from the vision box as the plan was made, and deviations can mean one of three things.

1. A dynamic obstacle has moved into the vision box, causing the distance to be shorter than expected.
2. A wall in the vision box has been found to be an open space, causing distance to be longer than expected.
3. The dynamic vehicle model M contains residuals compared to the real dynamics of the vehicle. This will cause the hallucinated trajectory, and therefore the hallucinated distance rays to be distorted.

The more serious of the cases is when distances are measured to be shorter than expected, as that means unexpected obstacles has appeared. If distances are longer than expected, that only means there might be an opening for the agent to exploit. However as small deviations are expected in any system that contains measurement noise and model disturbances, re-planning doesn't need to be triggered for every slight deviation. A threshold is set, and the SM checks each pair of rays, real and hallucinated, to see if a re-planning is required.

Figure 5.2 shows a common case that appears when using the vision box. At a distance, details of obstacles will be captured accurately, and the PDRL can plan dangerous routes. However, as the P-DRL approaches the obstacle, the difference between expected / hallucinated circograms and the actual measured ones will trigger a re-planning with a more accurate vision box representation.

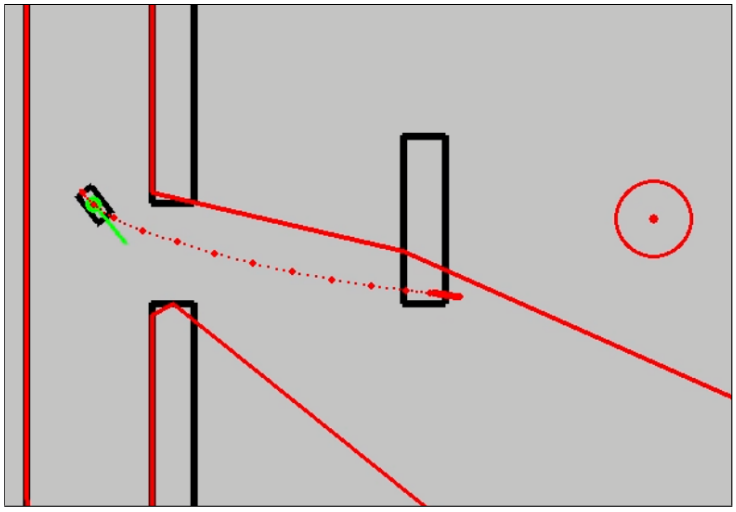


Figure 5.2.: This figure illustrates a common situation while using the vision box. The red outline illustrates where the vision box was located while planning. The planned trajectory is marked as red dots, where the big red dots marks every position where a re-planning is considered by the agent. As the vision box is constructed from circogram rays, beamed out at different angles, uneven objects at a distance will often not be captured reliably.

5.3.5. The Interpreter

The interpreter is part of any RL learning cycle, as shown in figure 2.3. The job of the interpreter is to shape the measured state to a usable state for the agent, as well as to provide it with rewards

for its actions. The sufficient state for training the DDPG presented in chapter 4, and contains information about goal-state, speed, angle, steering inputs as well as circogram distances.

These state variables are easy to get directly when working in a simulated environment, as done in this thesis but for a physical system some adaptations are required. Speed and angle can be read directly from the on-board sensors of the LIMO, the relative position of the goal state can be found by using the wheel odometry of the LIMO and circogram rays needs to be constructed from the local momentary map. In the case of the LIMO the local momentary map can be constructed directly from LIDAR distance measurements and information from the depth camera. The non-linear LIMO vehicle model can then be used in combination with an *Extended Kalman filter* to filter out sensor noise and model disturbances. This does however require that noise and disturbance is measured before applying the filter Ribeiro (2004).

Rewards are what determine what is an optimal decision or not for the agent, as it only optimizes the future expected reward of each action. In this project, a simple sparse reward function is used. Most states provides the agent with a small negative reward (cost) for the distance it has to the goal state. This is to ensure the agent tries to reach the goal as fast as possible, while taking the risk that the agent will prefer a local optimal state behind an obstacle instead of looking for a path around to get to the goal. Next the agent is rewarded heavily for reaching the goal and punished slightly for colliding. As the agent is planning and telegraphing moves before executing them, the agent can be prohibited from planning a colliding action sequence, so collisions do not need the heavy penalty that a freely exploring agent might require to avoid collisions. By itself, having a large or small reward / cost for a specific outcome only determines how quickly an RL agent will learn a specific behaviour. When using multiple different rewards it is also important to size them according to the other rewards, to make sure specific outcomes are preferred over others.

5.3.6. Collision free planning

As the agent always keeps the next N actions it plans to execute, after each planning the trajectory is tested to see if it collides with the edges of the vision box. There are two ways used to look for collisions in these trajectories.

First, Each consecutive pair of points in the trajectory is tested to see if a direct line between them crosses any of the walls of the vision box. However, unlike the intermediate value theorem, which can guarantee an intersection of an axis between two sample of a continuous function, the obstacles in this environment is a bit more complex. One example of when this test fails is when the vehicle is rounding a sharp corner, where a straight line would between consecutive points could cut the corner, and this test would produce a false positive collision. The sampling in the environment is however quite frequent, and the consequence of abandoning a plan to do re-planning is very small. The second method uses the hallucinated circograms that are produced while planning. If at any point the distance from the hull to any walls in the vision box becomes less than zero that means there is a collision. The first test looks at the car a single point particle, while this test captures the shape of the vehicle as well.

Both the collision tests look for collisions in the vision box rather than the real world. This is because, as can be seen from algorithm 2, all the work of the DDPG is being done in the hallucinated vision box environment. It uses hallucinated states and rewards to train, so the inside of the vision box is the only reality the DDPG agent knows. Compared to a normal DDPG training environment, where a new measurement / vision box is captured for each decision, this DDPG agent needs to take several actions from one. The consequences of this is seen in figure 5.3. The agent in case A (left side) is trained using normal DDPG, and from training expects

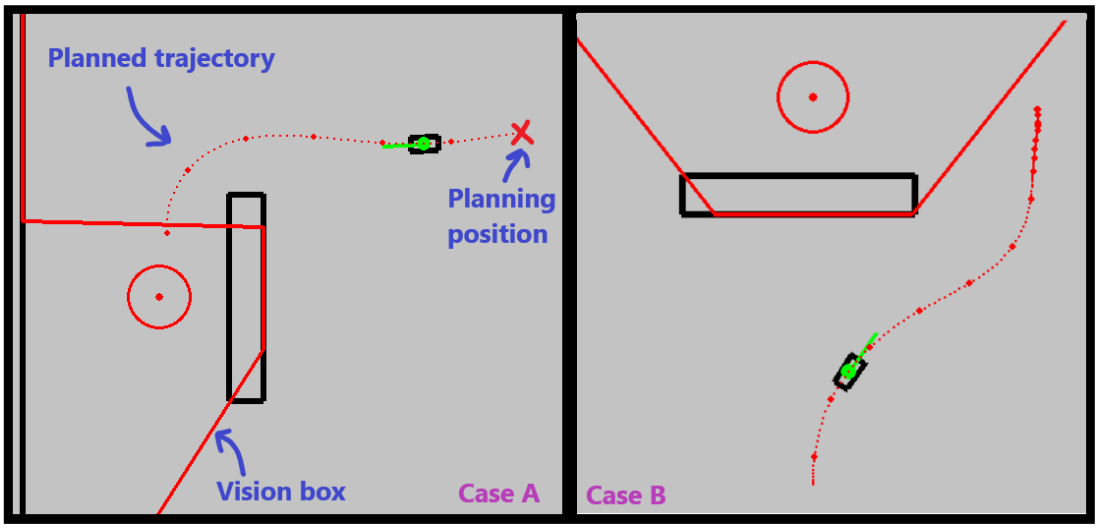


Figure 5.3.: Two cases illustrating the effect of collision free planning. Case A (left) illustrates an agent that has been allowed to plan for collision, and from training it expects an opening to the goal state around the corner. The agent in case B has not been allowed to plan for collision, and plans a more passive approach. How the agent has been taught to avoid planning for collision is discussed further in section 5.3.7

there to be an opening around the corner. The P-DRL agent in case B is not allowed to plan a collision path, and is a lot more passive with its training. It is trained to first look if there is an opening, and then re-plan if there is. The consequence of only operating in the simulated vision box environment is that the agent is given less information, and learns to move to positions where more information can be gathered.

5.3.7. Learning modules

Learning happens at two levels in algorithm 2; the ANN model trying to learn the vehicle dynamics, and the DDPG agent learning to choose optimal actions inside the vision box. The ANN model is quite straight forward, as it simply records state transitions and actions, and trains on predicting the next state, given current state and action. It keeps a buffer of the sufficient states, so without the circogram distance data, which it can sample to train on the vehicle dynamics. The original idea was for the model to only learn the residuals of the built in model instead of learning the entire model itself. It could then have been maintained and used to correct the pre-defined model on the go. The challenge of this approach is that a small deviation will compound over time, making it very difficult for a network to predict the current deviation without having access to the entire trajectory. For such a time-dependent and multi-dimensional problems, where different inputs affect each other, the use of long-short term or convectional architectures would be required (Goodfellow et al. 2016, p. 326-415). This project only used a simple feed-forward deep neural network, and the residual approach was not used for now.

The other learning happens within the **SOLVE()** function, which probably will be implemented as a class object in the code, to maintain its internal states. Here a separate buffer is kept of the extended state-space, involving circogram distance rays, actions and rewards. This is what is required to train the DDPG agent, which uses such a replay buffer to reduce variance in training (see section 5.2.1). As DDPG is off-policy it doesn't actually matter what actions was previously taken, they can still be used to estimate future expected rewards. The learning takes place inside the simulated vision box environment after each planned move is taken, and reward

is given.

To learn in a collision free environment, each time the agent plans a move that causes either of the two collision cases discussed in section 5.3.6, a negative reward (cost) is given, the agent learns without actually doing the actions, and then has to do the planning all over again. In addition to just having learned something new, the agents actions are also infused with OU noise (see section 5.2.1) while training, so the new proposal is most likely different from the first.

It is worth noting, that all learning for the DRL part of the agent happens within the hallucinated reality of the vision box. The DRL module never gets to use the "real" states directly, and relies on the hallucinated counterparts for taking actions, getting rewards, and learning state transitions. It it therefor going to be difficult for the agent to learn efficiently, if the vehicle model M is inaccurate.

5.3.8. Vehicle model approximation

In algorithm 2, the pre-defined vehicle model M or the approximated model M_{ANN} is seen being passed to the **SOLVE()** algorithm. As discussed in section 5.3.7, the M_{ANN} was originally going to be a residual model, slowly learning and correcting the residual errors of the pre-defined model M. In the current implementation, the accuracy of both models is calculated and used to switch.

5.3.9. Online learning

Online learning means learning after the agent is deployed into its operating environment, while offline learning is learning from a simulation. To train any DRL method, a simulated environment is needed initially. Training RL involves a lot of trial and error with a lot of variance in performance. Introducing a deep network, making it DRL, increases the variance in training even further Van Hasselt et al. (2018). DRL methods such as DDPG introduce target networks to reduce variance, and speeding up training, but training is still a slow process that requires rewards to be experience before being able to incorporate them in the value and policy functions.

A good example of how slow DRL training can be is shown in the DDPG paper Lillicrap et al. (2015), the TD3 paper Fujimoto et al. (2018) and the SAC paper Haarnoja et al. (2018) - where each of the methods use millions of steps before fully learning the HOPPER-v1 benchmarking environment. The HOPPER-v1 is one of the common 2D toy problem that can be used to benchmark RL methods, and seeing how many attempts these methods need before solving the environment, by making the three joint foot jump. As the agents are often required to do random actions to explore and avoid local optimal solutions, even after hundreds of thousands of episodes sudden drops in performance also appear in the graphs of the mentioned papers. In the search for the optimal solution, the agents not only gets close to, but also has to experience crashing and failing completely.

In this paper the agent is given the pre-defined model to predict and prevent its own collisions - even during training. However, this means that the agents simulates its own collisions - meaning that a simulated version of the agents does collide. This is exactly how all DRL is trained normally; by allowing simulated agents to collide, to give the agent experience of the environment. Many DRL agents are then never deployed into a physical environment, as they are made to perform in the simulated world. Agents that are deployed into the real physical world relies on the simulations before-hand to have been as close to reality as possible, so that they can still perform.

In this thesis, real world deployment will be simulated by adding disturbance to the simulated environment, so that the internal model the agent is given to plan its future actions will differ from the environment where it performs the planned actions. It will then be the job of other modules to adjust for these disturbances over time. As the P-DRL agent is capable of predicting collisions and do re-planning to avoid them, it should be possible for it do do online learning, as long as the model disturbances are not too substantial. Directly using an implicit DRL method, like DDPG, would not allow for safe online learning, as it will only learn and update its policy and value functions after it has experienced collisions and failures.

5.3.10. System architecture

Figure 5.4 shows the full overview of the P-DRL system. Most of the components are discussed in the previous subsections of section 5.3, or from section 5.1. The system architecture tries to split the different components into three different categories.

The mission planner and the (real) environment are outside the agent. The environment can be simulated, or the real thing. In case of a simulated environment, as used in this thesis, disturbances are added to simulate the imperfections of the built in model given to the agent. The mission controller generates goals, or stacks of goals to the agent. For the simple joyride mission described in section 5.1, goals are mostly randomly picked in close proximity to the agent - as the objective is only to test its ability to do short term planning.

The goal-stack, the interpreter and the sequence maintainer are part of the pre and post-processing of states, actions, goals and rewards. This is where plans are translated to action inputs and executed, action sequence update signals are generated and the current state needed to start planning is constructed.

Lastly, the action generator itself is the components that create a plan within the vision box. By using the vehicle new states can be hallucinated from the original (real) state, and actions corresponding to these hallucinated future states can be generated by the DRL module. For this project DDPG was chosen as the DRL method, but in theory any implicit DRL method could serve the same purpose.

5.4. What about MBRL?

As mentioned briefly back in section 2.4.7 model based reinforcement learning (MBRL) is an alternative to the implicit / model free algorithms when working with DRL. The difference between the two approaches is whether or not the model of the environment is represented explicitly as it is with MBRL or not. MBRL could sound like a reasonable choice for a task where a pre-defined model will be used for planning, but in the end the model free model DDPG was chosen.

Many of the modern MBRL algorithms recently developed and published aim to learn the action policy or value function, while simultaneously learning an explicit representation of the model. Methods like dreamer Hafner et al. (2019), I2A Racanière et al. (2017) and MBMF Nagabandi et al. (2018) take this approach to MBRL. The developers behind AlphaZero took a different approach, by providing the explicit model beforehand Silver et al. (2017). The method was created to solve problems like chess and Shogi, which are environments that are simple to model, compared to continuous, stochastic real world environments. By utilizing Monte Carlo tree search (MCTS), the algorithm evaluates and plans trajectories through the state-action space, leveraging the explicit model provided.

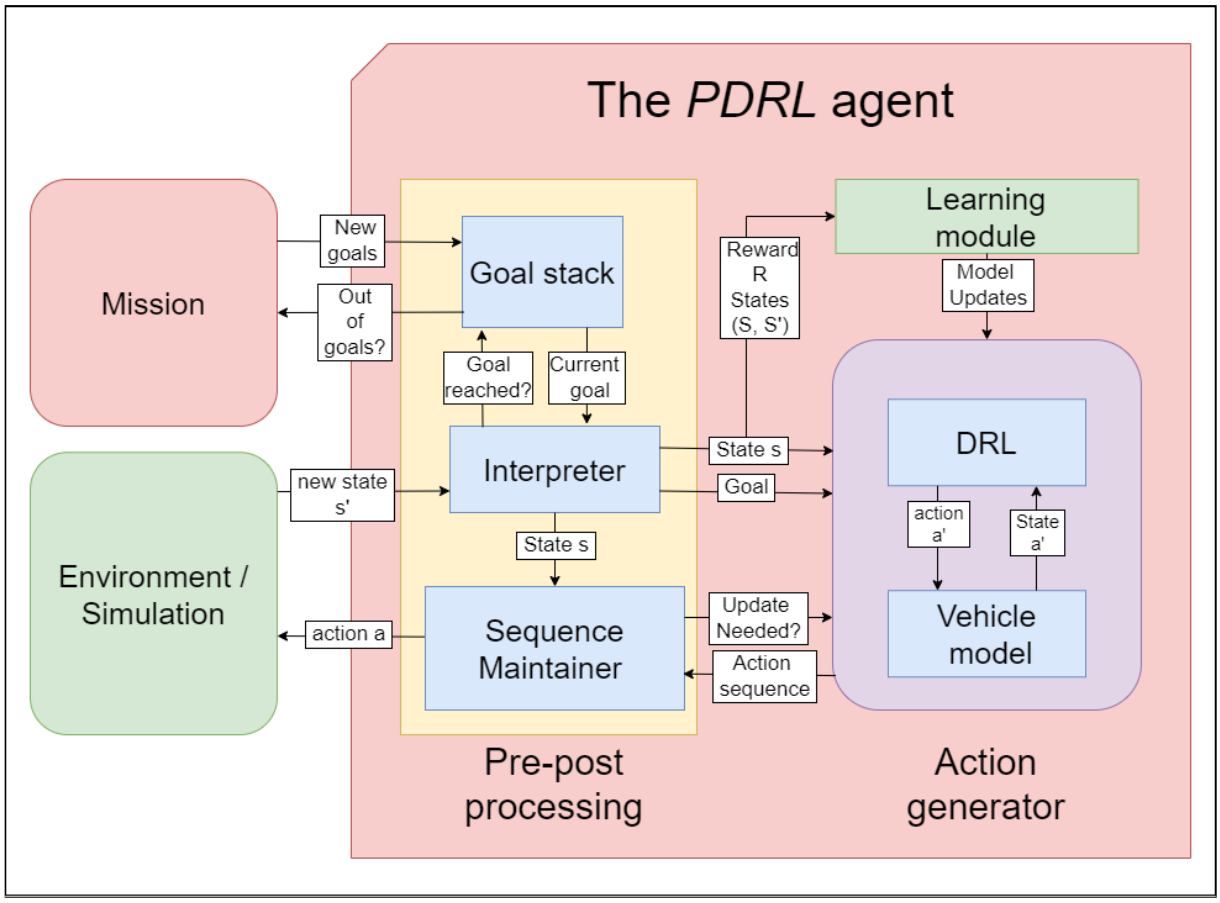


Figure 5.4.: The full overview of the PDRL system architecture.

Given that a pre-defined model has already been established for this project, a comparable approach to AlphaZero will be devised for the P-DRL algorithm. However, instead of employing Monte Carlo tree search (MCTS), the DDPG algorithm will be utilized to plan trajectories, as it aligns better with the state and action space utilized in this project. Unlike AlphaZero, the pre-defined model in this context may not perfectly replicate the environment in which P-DRL will eventually be deployed. Consequently, an additional model will be developed with the objective of enhancing the explicit representation of the environment. In practice the P-DRL method itself should therefore be considered a model based reinforcement learning algorithm, that uses a model free reinforcement learning algorithm as a tool for doing trajectory planning.

6. Experimental setup

This chapter provides brief information about the experimental setup. While most of the work was conducted in software and simulation, the **LIMO** was also tested to obtain relevant parameters for the simulator.

6.1. The LIMO parameters

The process of selecting and ordering the LIMO was part of the preliminary pre-project thesis. A summary of this process and some information about the LIMO can be found in section 1.4. After the LIMO arrived this semester, tests were performed to assess the accuracy of the data sheet provided by AgileX. Key parameters of interest included maximum speed, acceleration time constant, maximum turning angle, and turning time constant. The time constant was estimated by measuring the time it took for the LIMO to reach about 99% of its final output for a given input, which should correspond to about 5τ. The maximum speed was determined by allowing the LIMO to accelerate to its maximum speed and measuring the time it took to travel 10 meters. Finally, the maximum steering angle was tested by executing the sharpest possible turn with the LIMO and measuring the turn radius. The relationship between turn radius and steering angle can be observed in equation 3.2.

The most significant deviation from the LIMO documentation was the theoretical turning radius. The documentation stated that a turning radius of 0.4 m should be achievable. However, testing revealed that a more accurate value was 0.63 m (corresponding to 0.31 radians of steering angle, with a wheelbase of 0.2 m). It was also observed that both wheels of the LIMO seemed to turn at the same angle, indicating that the wheels would experience slipping throughout the turn, as the inner wheels should have a steeper angle in an Ackermann system. All experimental parameters are presented in table 6.1. Please note that the measurements were not conducted with high precision, so further tuning may be required. The model prediction artificial neural network (ANN) will be also be used to compensate for these imprecisions.

6.2. Implementing P-DRL

The project is primarily written in the programming language **Python**, and all the code can be found in the open-source repository on GitHub. Python was chosen due to its extensive range of tools and libraries that are beneficial for machine learning, particularly deep learning. The main

Experimental parameter	Estimated values
α_{max}	0.31 [radians]
v_{max}	1 $[\frac{m}{s}]$
τ_v	0.4 [s]
τ_α	0.4 [s]

Table 6.1.: Experimental parameters from testing the LIMO

libraries utilized in this thesis are *TensorFlow* and *PyTorch*, developed by Google and Facebook, respectively. Further details on accessing and running the code are provided in Appendix D.

https://github.com/henrfj/master_thesis_2023

In addition to the GitHub repository, there is also a dedicated YouTube channel created to showcase some of the project's results. The visualizations are generated using the Python library *PyGame*.

https://www.youtube.com/@PDRLMasterThesis

One of the crucial parameters in P-DRL is the planning horizon, which determines the number of planning steps performed before executing the plan. In all the experiments conducted in this project, the plan consisted of six consecutive actions, each spanning 0.5 seconds, resulting in a 3-second trajectory. The model-free DDPG agent would update its action every 0.5 seconds.

6.3. Implementing environment

The project utilized the Python GYM package to implement all the environments, following the standardized GYM environment framework. GYM is a widely adopted API for training RL agents in various environments. It is common for different RL researchers to use the same gym-environments such as "Hopper," "Moon Lander," or "Walker" to facilitate result comparison and comprehension across different papers. However, since there were no pre-existing environments that met the specific requirements of this project, a new environment was developed. Nonetheless, this new environment was integrated into the same API framework, enabling straightforward integration and training of RL methods beyond those employed in this study.

6.3.1. Implementing dynamic obstacles

Dynamic obstacles were implemented as an addition to the static environments. These obstacles are implemented as reactive agents, drifting mindlessly throughout the environment at low speeds. They follow the same dynamics as the P-DRL agent but differ in terms of shape, size, speeds, and behavior. The implementation of these agents is discussed in more detail in Appendix C. Dynamic circograms are used as the sole input for the reactive agents, which was also the initial plan for P-DRL. However, in the final iteration, P-DRL eliminated the use of dynamic circograms and currently utilizes the vision box for environment representation.

6.4. DDPG: Learning and exploration rates

This section will cover some learning parameters that were kept constant across all tests conducted in this project. These parameters include the learning rates for the actor and critic (α and β) in DDPG, the update rate of the target network (τ), the exploration rate, and the discount factor (γ).

Consistent with the original DDPG paper, a lower learning rate was set for the actor compared to the critic. Using $\alpha = 0.000025$ and $\beta = 0.00025$ falls within the range used by the Google team when benchmarking DDPG. These values allow for a quick training phase without significant issues with variance. The parameter τ, as described in Equation 5.1, which is used to update the target networks, was set to $\tau = 0.1$. The original paper used a lower τ but slightly higher learning rates. The effects of these changes seem to have counteracted each other as high variance was not a significant issue Lillicrap et al. (2015).

During the learning and exploration phase, the Ornstein-Uhlenbeck process with $\theta = 0.15$ and $\sigma = 0.2$, as mentioned in the original DDPG paper, was employed. Given that this environment, like the ones tested in the original paper, involves momentum, using this process may result in better exploration compared to a simple Gaussian process, which is commonly used Lillicrap et al. (2015). However, unlike the original paper, once the performance and score plateaued, the noise was completely turned off. In theory, this should have no effect since DDPG is an off-policy learner that learns the optimal policy regardless of noise in the current exploration policy. However, for environments that require high precision over time, removing the noise was necessary to explore the entire environment. "Naples" is an example of such an environment (see Figure 4.2), where narrow streets and tight corners would not allow a noisy policy to navigate the maze-like environment without collision. Turning off the randomness towards the end is one approach to achieving the Greedy in the Limit of Infinite Exploration (GLIE) training situation for an RL agent. GLIE implies that for an infinite training process, each state-action pair is visited infinitely often, but at the "end" of training, the policy becomes fully greedy, which helps train an RL agent while avoiding pitfalls of local optimal solutions Singh et al. (2000). Additionally, a relatively low discount factor of $\gamma = 0.99$ was adopted from the original DDPG paper Lillicrap et al. (2015). This causes the agent to place more emphasis on the expected (discounted) future rewards compared to a what a lower discount factor would entail.

6.5. The vehicle model approximator

The vehicle model approximation, denoted as M_{ANN} in algorithm 2, is a simple feed-forward deep neural network. In this project, a sequential model was created using the *Keras* module of the *TensorFlow* library in Python. The network consists of two hidden layers with a width of 64 units each. The input layer has a width of six, which accommodates a sufficient state representation of four states and a selected action vector of size two. The output layer has a width of four, representing the next state. All hidden layers use rectified linear units as activation functions, while the output layer has no activation function, allowing for positive and negative values in the final linear combination. The model was compiled with the *Adam* optimizer and used mean squared error loss to compute gradients. The Adam optimizer is a well-known, computationally efficient optimization algorithm that performs well with noisy and sparse data. It requires minimal parameter tuning and was left untuned for this project Kingma and Ba (2014). Adam was also used to optimize all four networks in DDPG, similar to the approach taken in the original DDPG paper. The model was trained using mini-batches of 512 states, with training performed on the vehicle model approximator each time DDPG was trained.

7. Experiments and Results

This chapter presents the experiments conducted and their results. Three different experiments were performed to test various aspects and scenarios for the P-DRL agents. The main focus of the experiments is to compare DDPG to P-DRL in terms of score and collision rates. The experimental setup, which is similar for all experiments, is described in Chapter 6.

In addition to the graphs presented in this chapter, a youtube channel was created to provide a visual demonstration of the final product. The channel showcases DDPG and P-DRL operating in the three different environment configurations.

https://www.youtube.com/@PDRLMasterThesis.

7.1. Experiments

This section covers the experiments conducted, their purpose, hypotheses, and the parameters used in each experiment.

7.1.1. Experiment: Training comparison

As discussed in sections 5.3.6 and 5.3.9, no DRL method can train without initially using a simulated environment. The P-DRL is unique in that it can predict, avoid, and learn from collisions without experiencing them. However, even the P-DRL can start off like any other DRL method and use its internal model of the environment for training solely in a simulated environment. This would be similar to first training a DDPG algorithm in the simulated environment and then incorporating it into the P-DRL for deployment. In this context, "deployment" refers to putting the agents into another simulated environment where disturbances are introduced to the tunable parameters of the environment model. Otherwise it would refer to putting the agents into the real operating environment.

The first experiment aims to compare the training of the P-DRL agent using collision-free planning and online learning with the conventional training of DDPG through trial and error. Additionally, it will examine the performance of a pre-trained DDPG agent when deployed within the P-DRL architecture. The experiment will initially be conducted without any model disturbance, followed by the introduction of small disturbances to observe their effects. The average scores and collision rates will be analyzed.

The hypothesis is that the DDPG agent will not be affected by disturbances during training since it explicitly learns the environment model regardless of its characteristics. On the other hand, the P-DRL relies on the predefined internal model even during training, as it generates state-action-state-reward transitions within the vision box. It is assumed that the P-DRL will eventually be able to adapt to the disturbances by utilizing the supervised learning module to learn the new environment model more accurately.

7.1.2. Experiment: Added disturbance

The next experiment aims to introduce disturbance after deploying the models to the environment. This can simulate the agent being relocated to a new location with surfaces that are more slippery or have better grip, resulting in different acceleration and turning rates due to wheel slippage or improved traction.

In this experiment, no training will be performed, and the performance of the P-DRL agent will be compared to that of a DDPG agent as disturbance is added. The hypothesis is that the P-DRL, relying on its internal model, will struggle as disturbance is accumulating along its proposed trajectories. The DDPG agent is likely also experience collisions and a drop in rewards.

7.1.3. Experiment: Online learning

The final experiment aims to test and compare the agents' ability to learn after deployment into an new environment. It follows the same scenario as the previous experiment, where disturbance is introduced to the system after initially training without disturbance. However, in this case, online learning will be allowed to correct for the disturbances.

The hypothesis for this experiment is that both the DDPG and the P-DRL will be able to learn the new environment, but while P-DRL mostly avoids collision, DDPG will have a higher rate of collisions during the online-learning phase.

7.2. Experimental Results

This section presents the results obtained from conducting the three different experiments. The results are primarily shown in the form of score or collision rate graphs. The experiments were repeated multiple times, and the results are presented as multi-run averages (MRA). Additionally, a floating average for a single run, known as one-run average (ORA), is also provided. It is important to note that due to the variability in training, no two training runs will be identical. DDPG is known to have a sensitive convergence, where selecting appropriate hyperparameters is crucial to avoid instabilities and failures during training Haarnoja et al. (2018). However, since that was not the focus of this thesis, runs that did not complete the training process were excluded from the results.

7.2.1. Results: Training Comparison

Figure 7.1 presents a comparison between DDPG and P-DRL in scenarios with and without disturbance. It is important to note that in this context, disturbance refers to slight differences in the tunable parameters of the internal model used by P-DRL compared to the parameters used for simulating the vehicle's movement in the environment. P-DRL utilizes its internal model to predict future states and make decisions, while DDPG does not have an explicit representation of the internal model. The number of episodes varies across the training scenarios as different scenarios required varying numbers of episodes to reach a stable state.

In addition to training DDPG and P-DRL separately, it was also experimented with adding pre-trained DDPG networks to P-DRL. This is shown in figure 7.2, which clearly shows where the DDPG networks are added to the P-DRL training scheme. The figure also shows the training progression of the approximated vehicle model M_{ANN}, showing how mean-square-error loss approaches zero during its training. Each time the agent needed re-planning, the feed-forward network was trained on sampled mini-batches from the buffer.

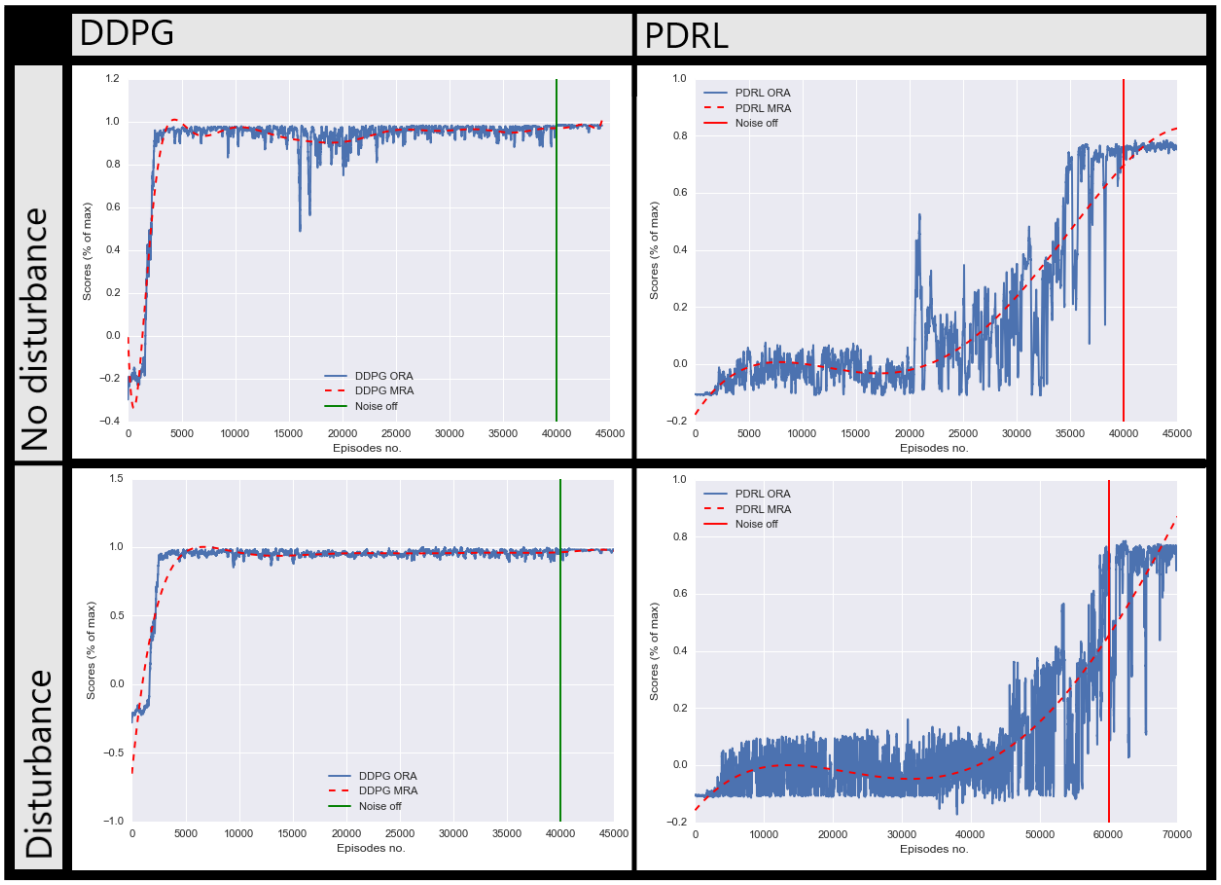


Figure 7.1.: This figure compares the training scores for different algorithms with and without disturbance in the simulated environment. The left column represents the DDPG algorithm, while the right column represents P-DRL. ORA stands for one-run average, indicating the average performance of a selected run. MRA stands for multi-run average, representing the interpolation of performances from multiple similar runs. The "Noise off" line indicates the removal of Ornstein-Uhlenbeck noise from action selection.

Finally, Figure 7.3 presents the collision probabilities for DDPG and P-DRL during training in an environment with disturbance. The impact of switching from the pre-defined model to the approximated model can be clearly observed in these collision-rate graphs.

7.2.2. Results: Added disturbance

In this experiment, disturbance was introduced after training to observe its impact on the agents when deployed in a slightly different environment from their training environment. No further training was conducted for the agents in this experiment. Figure 7.4 illustrates the effect of introducing the same disturbance to both the DDPG and P-DRL agents. The ORA reveals that the DDPG agent exhibits less variance in performance compared to the P-DRL agent. Both methods experience a decrease in average performance, but DDPG maintains a higher average score.

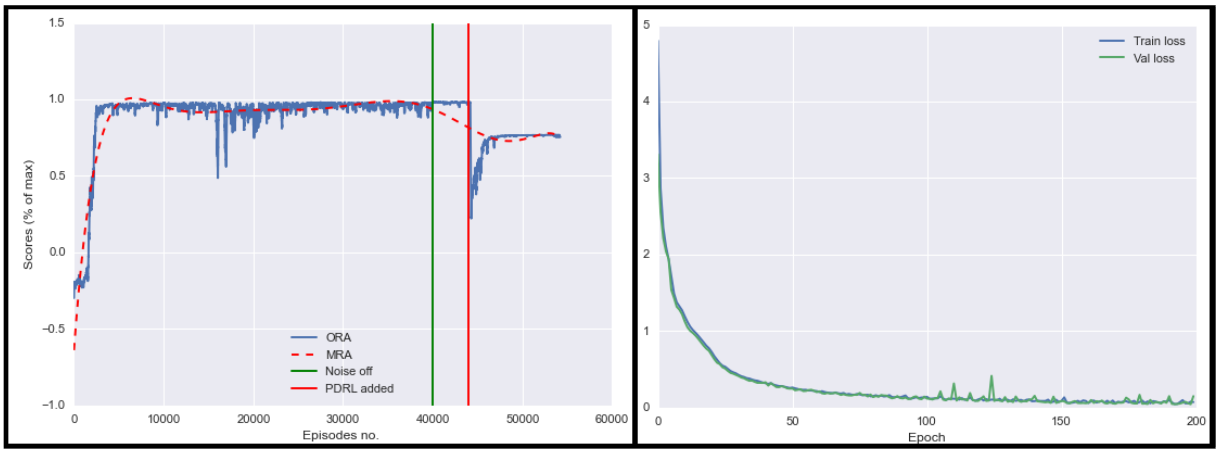


Figure 7.2.: The left graph shows the effect of using pre-trained DDPG networks as a starting point in P-DRL training. The "Noise off" line represents the removal of Ornstein-Uhlenbeck noise from action selection, and the "PDRL added" line indicates the start of P-DRL training and the end of DDPG training. The right graph displays the training progression of the approximated vehicle model M_{ANN}, which happens while P-DRL is training.

7.2.3. Results: online learning

In this experiment, a slight disturbance was introduced to an environment where pre-trained agents were operating. Unlike the previous experiment, the agents were allowed to learn and improve. The results align with the findings in the lower row of Figure 7.1, where DDPG demonstrates faster learning and achieves a higher score. On the other hand, P-DRL requires the internal model to update its approximation before returning to previous score levels. Once the approximation reaches a higher level of accuracy than the internal model, the collision rate (and consequently the score) experiences a leap, as observed in Figure 7.3. Until then, the collision rates for PDRL was similar to what is seen in figure 7.3, before switching models.

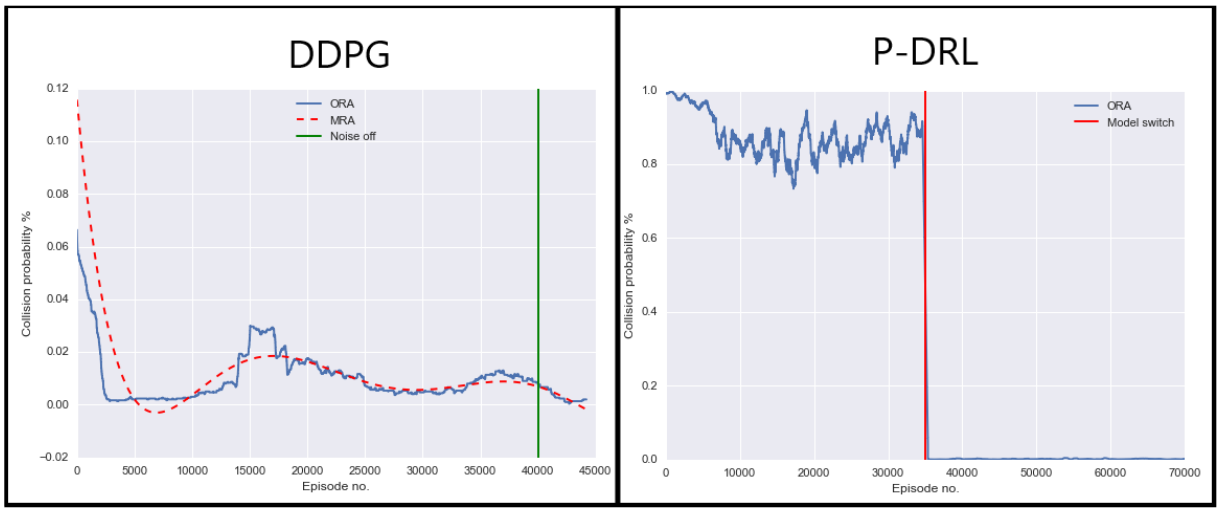


Figure 7.3.: The collision probabilities of DDPG and P-DRL during training. In both cases, disturbance was introduced to the simulated environment. Without disturbance, P-DRL showed no collisions due to its ability to anticipate future states. The red "Model switch" line for P-DRL indicates the threshold at which M_{ANN} replaced M in the **SOLVE()** algorithm of P-DRL. This occurs when M_{ANN} is considered to be more accurate than the pre-defined model.

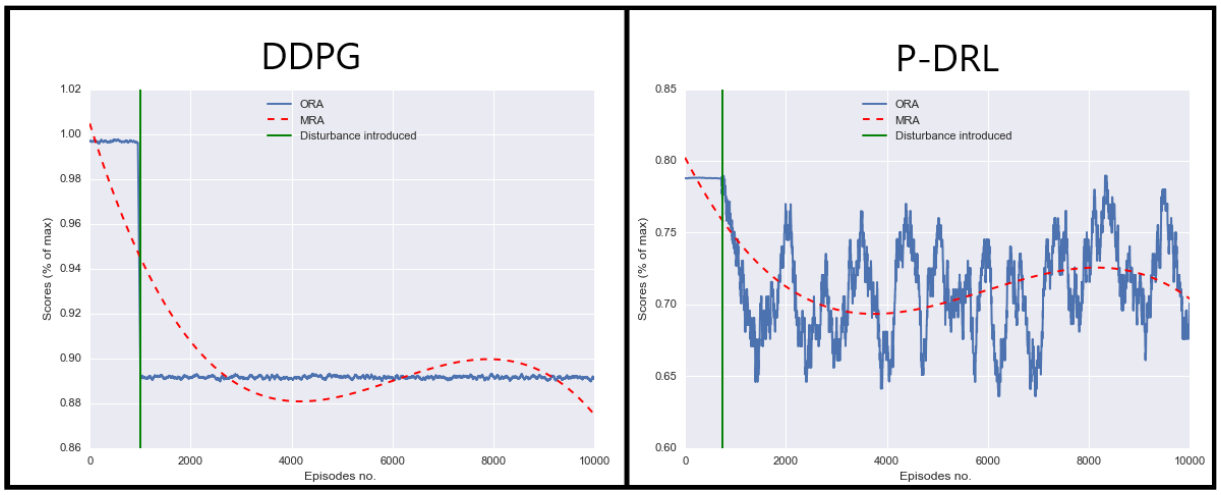


Figure 7.4.: In this experiment, a small disturbance was added to an environment where two pre-trained agents were operating. Neither of the agents underwent further learning, resulting in a performance drop, but no recovery. The ORA indicates that the P-DRL agent experiences greater variance in performance after the introduction of disturbance compared to DDPG.

8. Evaluation and Conclusion

In this chapter, the discoveries and results from chapter 7 will be discussed. Then, an evaluation of the goals and objectives of this project will be conducted, leading to a conclusion. Towards the end of the chapter, potential areas for future research will be explored.

8.1. Discussion

This section will discuss the results obtained from chapter 7. The objective of the experimentation was to assess the performance of the PDRL architecture. This was achieved by comparing the scores and collision rates of the PDRL agent with those of another agent running the DDPG algorithm.

8.1.1. Scoring comparison

In the first experiment, both a DDPG agent and a PDRL agent were trained in the same environment. As depicted in figure 7.1, it is evident that DDPG converges faster and achieves a higher score compared to PDRL. This disparity in scores can be attributed to the more informed decision-making process of DDPG. Unlike PDRL, DDPG receives new sensor input for each action, enabling it to move closer to obstacles and optimize its trajectory. Only the initial action proposed by PDRL in each plan is as informed as every action taken by DDPG. Figure 8.1 illustrates that DDPG follows more aggressive paths in the same environment, benefiting from up-to-date environment representation at each action decision. Consequently, DDPG reaches the goal state faster, resulting in a higher score. This trend is consistent with the observations in figure 5.3, where PDRL, restricted from planning for collisions with the vision box, exhibits more cautious driving behavior.

The decline in score of PDRL is evident in Figure 7.2, where a pre-trained DDPG agent is incorporated into the PDRL architecture. As the DDPG module is constrained to operate within the hallucinated vision box, it experiences a decrease in performance due to the need to re-learn certain actions. Despite subsequent re-learning and stabilization in score, the agent never achieves the same level as before the integration of PDRL.

Figure 7.2 does however show an interesting possibility with PDRL. Instead of training PDRL from scratch, a pre-trained DDPG agent can be added, to speed up the training process. The DDPG agent can be train using the internal model of PDRL.

8.1.2. Collision avoidance

Collisions have a significant impact on an agent's scoring ability in this environment, as they result in substantial negative rewards. The DDPG agent initially experiences frequent collisions but quickly learns to avoid obstacles, as depicted in Figure 7.3. After several thousand episodes,

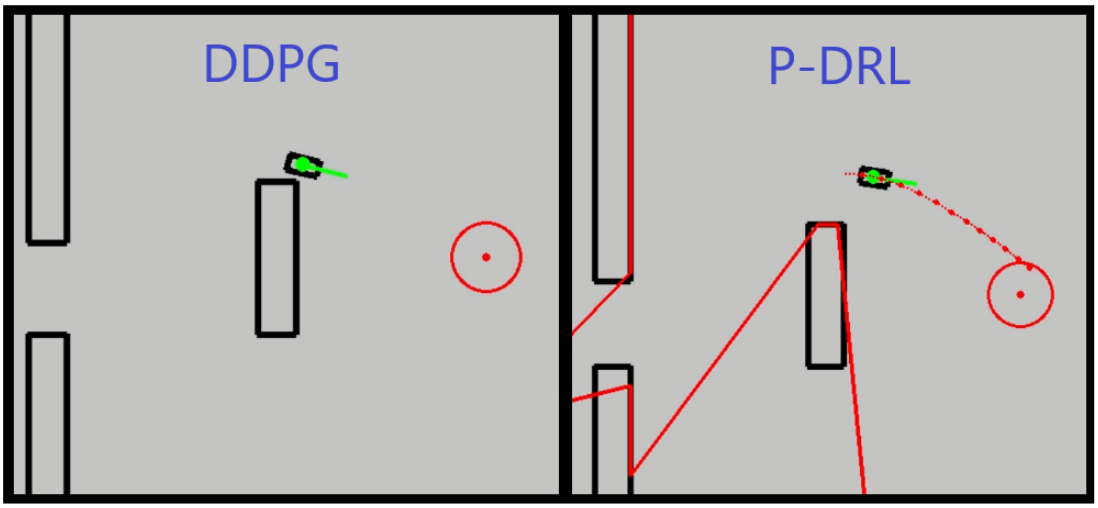


Figure 8.1.: This figure illustrates some of the distinctions between direct DDPG and DDPG used in PDRL. It can be observed that DDPG generally learns more aggressive and shorter trajectories, leading to higher scores. It may take some collisions to learn this behaviour, but the end result performs better.

collision rates increase again as the agent attempts to optimize its trajectories by taking shortcuts to save time. Eventually, the agent stabilizes and achieves optimal solutions, resulting in a decrease in collision rates to zero once noise is turned off during training.

In the case of PDRL, collision rates are zero when a perfect internal model is utilized. PDRL can accurately predict future collisions and learn from them without actually experiencing them. However, when disturbance is introduced to the environment, PDRL exhibits significantly higher collision rates, as shown in Figure 7.3. This continues until the internal model is replaced with the learned approximation model, indicated by the red vertical line. This observation is also supported by Figure 7.1, where PDRL's learning progress in a disturbed environment only begins around episode 45000, which likely coincides with the replacement of the internal model.

The higher collision rates observed in PDRL can be attributed to the propagation and accumulation of errors in the internal system model along the planned trajectory. With each hallucinated action, the hallucinated state deviates further from the actual trajectory. PDRL employs the **Sequence maintainer** module (see Section 5.3.4) to trigger re-planning if the experienced states diverge significantly from the hallucinated states. However, due to the vehicle's momentum, re-planning may not always occur in time to prevent a collision. Additionally, the newly proposed plan after re-planning retains the same flaws as the previous one.

Interestingly, after switching the model, PDRL's collision rates remain almost zero for the remainder of the training session, as depicted in Figure 7.3. In contrast to DDPG, which experiences an increase in collision rates during optimization, PDRL is not allowed to plan for collisions, resulting in minimal collision rates throughout the training. Similar to DDPG, PDRL employs noise during training to facilitate exploration. However, unlike DDPG, PDRL can use its internal model to assess the effect of noisy moves without executing them.

8.1.3. Dealing with disturbances

When DDPG is trained from scratch in an environment, the presence or absence of disturbance beforehand does not significantly affect its performance. DDPG is an implicit DRL method that

learns the dynamics of the environment through its policy and value functions. As long as the dynamics remain consistent over time, DDPG's training remains unaffected. This is evident in Figure 7.1, where DDPG performs similarly in both the disturbed and undisturbed cases.

On the other hand, PDRL is highly susceptible to the presence of disturbance. As discussed in Section 8.1.2, PDRL heavily relies on an accurate internal model, and the accumulation of errors along a trajectory leads to frequent collisions during training. These collision rates adversely affect the score and persist until the internal model is replaced with a more accurate approximation. In Figure 7.1, this switch appears to occur around episode 45000. For the undisturbed case, PDRL had completed learning around this same episode.

8.1.4. Deployment and online learning

The final two experiments involved introducing a small disturbance after the completion of learning. This was done to simulate the scenario where the agents are deployed in an environment that slightly differs from the simulated training environment. Figure 7.4 demonstrates the significant impact of the added disturbance on PDRL compared to DDPG, for the same reasons discussed in Sections 8.1.2 and 8.1.3. While DDPG experiences a minor decrease in score, which can be recovered through a few hundred episodes of training, PDRL requires the replacement of its internal model with a more accurate approximation, before being able to even start learning a new policy.

Once the internal model is replaced, PDRL exhibits almost zero collision rates in the new environment, as shown in Figure 7.3. Conversely, Figure 7.3 reveals that DDPG experiences a slight increase in collision rates while attempting to optimize the immediate solution. This pattern is likely to repeat during online learning, where a few collisions may occur as the agent attempts to find the global optimum by cutting corners more and more closely.

8.1.5. The model approximator

For this project, a simple, un-tuned feed-forward neural network was employed to approximate the vehicle model. However, in this configuration, the network exhibited slow learning and required a substantial amount of data to prevent overfitting. The training data was divided into training and validation sets, and only when the validation data reached a certain accuracy threshold was it utilized to replace the pre-defined vehicle model. This process became a bottleneck for the PDRL agent since all the learning it performed before the introduction of the new model had to be discarded, and the learning process had to start anew.

Using an artificial neural network may not be the most optimal choice for approximating the model. An alternative approach, proposed by Rosolia et al. (2018), involves employing a K-nearest neighbors model to select the most closely related states from a buffer, and subsequently training a linear model using efficient and robust linear regression. This linear approximation may be less accurate as the vehicle deviates from the current state, but it can be quickly updated upon reaching a new state.

Another approach could be to focus on learning the residuals of the model rather than the complete vehicle model. The pre-defined model can be regarded as an accurate initial estimation, and an approximator could be employed to learn only the residuals of the model. Instead of switching between the pre-defined and approximated models, the residuals could be gradually corrected as the agent undergoes training. This was the original plan for the project; however, time constraints prevented its implementation.

A final approach, inspired by the LMPC agent proposed by Rosolia and Borrelli (2017), involves utilizing a safe agent (possibly a human driver) to collect sufficient data for the model approximator to be effective before deploying the LMPC. This approach could be employed when deploying PDRL in a new environment.

8.1.6. Compared to other MBRL methods

In Chapter 5, the PDRL algorithm was categorized as a model-based reinforcement learning (MBRL) algorithm, primarily due to its utilization of an explicit environment model for planning. However, PDRL differs slightly from other MBRL approaches as it relies on the model-free DRL algorithm DDPG to utilize the model for planning. Nonetheless, its performance remains comparable to other MBRL algorithms. In the benchmarking study conducted by Wang et al. (2019), it was observed that the optimal solutions obtained by MBRL algorithms tended to plateau below those achieved by their model-free counterparts. This trend persisted no matter the amount of training and data available to the MBRL algorithms. This aligns with the findings of this project, which can be attributed to the challenge of predicting actions in hallucinated future states during planning, as opposed to directly utilizing observed states.

8.2. Evaluation of goals and objectives

In the introduction, two objectives and two research questions were outlined for this project, as discussed in section 1.2. The aim if these objectives and questions were to summarize the purpose and goals of the entire project. In this section, an evaluation of each objective and research question will be conducted to assess the extent to which they have been addressed and answered throughout the project.

Objective 1 [O1] One of the objectives of this project was to develop a simulation environment capable of training and testing autonomous agents. The simulation environment aimed to replicate realistic car-like motion using the CTM model, simulate obstacles and collisions, and generate simulated sensor data through circograms. Throughout the development process, it was observed that certain parameter configurations could lead to errors and instability in the system, primarily due to a first-order Euler approximation employed within the system. The development of this simulation environment required a significant amount of time, with a substantial portion of the project's duration dedicated to its implementation, and is not recommended for others facing a similar time-frame.

Objective 2 [O2] Another objective of this project was to develop the PDRL architecture, which enables any DRL algorithm to plan and communicate its upcoming actions. This was achieved by employing the "vision box" technique as a local momentary map and using the internal vehicle model to simulate motion and predict future states. The PDRL architecture also includes the maintenance of the internal model, ensuring its alignment with the actual operating environment of the agent. To accomplish this, a basic feed-forward network was used to learn the environment, and a simple model-switching mechanism was triggered once the model achieved a certain level of accuracy.

Research question 1 [RQ1] One research question addressed in this project was the impact of the PDRL approach on the performance of the agent compared to traditional model-free DRL methods. PDRL extends the implicit, model-free algorithm by incorporating an explicit model, transforming it into a model-based DRL (MBRL) method. It was observed that, similar to other MBRL methods, there was a significant decrease in performance when

compared to the model-free counterparts. This issue was particularly pronounced when introducing model disturbances to evaluate the ability of PDRL to adapt to deviations in the real operating environment. It was discovered that the internal model maintenance strategy employed in PDRL did not correct errors quickly enough, leading to a further lag in performance compared to the model-free counterpart.

Research question 2 [RQ2] Another research question addressed in this project focused on the impact of the PDRL architecture on online training and collision avoidance. In the absence of model disturbances in the operating environment, PDRL demonstrated the ability to utilize its internal model for accurate prediction of future actions, enabling collision-free training and exploration. However, despite this capability, PDRL still achieved lower scores compared to the model-free counterparts, similar to other MBRL methods in the same situation. The scenario changed when model disturbances were introduced, as longer planning trajectories in PDRL resulted in error accumulation, leading to higher collision rates during training compared to the model-free counterparts. Once again, the internal model maintenance strategy employed in PDRL was identified as insufficiently responsive, as it failed to promptly correct model errors.

8.3. Conclusion

In direct comparison to the model-free counterparts, PDRL exhibited higher variance and a lower score threshold. Regardless of the amount of training provided, PDRL consistently reached its performance ceiling earlier than the DDPG algorithm tested in the same environments. This trend aligns with observations from other model-based reinforcement learning methods and can be partly attributed to the inherent difficulty of predicting actions in hallucinated future states during planning, as opposed to using observed states directly.

PDRL also struggles in dealing with disturbances compared to DDPG. Whether it was online learning or immediate response to disturbances, PDRL consistently underperformed in these scenarios. This drawback stems from the reliance on an internal model for planning trajectories instead of selecting actions in each new state. Errors accumulated rapidly along the planned trajectory, highlighting PDRL's strong dependence on an accurate internal model. The infernal model was also not maintained rapidly enough to change this trend.

The approach used to approximate the vehicle model and replace the pre-defined model upon reaching a certain accuracy threshold proved to be unsuccessful. PDRL was often left waiting for the internal model to be replaced before progressing with its learning. Interestingly, after the approximation process, collision rates of PDRL dropped to nearly zero and remained low throughout the rest of the training. This observation indicates the potential of PDRL as an online learner even after deployment. However, it is evident that a new method for approximating the vehicle model is required.

Based on the testing results, it appears that the most effective use of the PDRL in its current state would involve initially training a DDPG agent in a simulated environment and subsequently integrating it into PDRL. DDPG demonstrates faster convergence to a superior solution and readily adapts to PDRL. Prior to deployment, a safer agent can also be employed to collect data in the operating environment, enabling the update of the internal model approximation before allowing PDRL to operate in the environment. This approach leverages the strengths of both DDPG and PDRL, combining the efficiency of DDPG's training with the adaptability and safety-enhancing potential of PDRL in real-world scenarios.

8.4. Future Work

This section covers the areas of future work required to comprehensively test and further develop the PDRL architecture.

8.4.1. Improved Model Approximator

Having a robust model approximator to account for the errors in the pre-determined model is crucial for deploying a PDRL agent in real-world scenarios. The current model approximation method has two main limitations. Firstly, it aims to learn the entire system model instead of focusing on learning the residuals of the pre-determined model. Focusing on learning residuals would provide a better starting point and eliminate the need to discard all learning before applying the new model. Secondly, using a feed-forward deep neural network (ANN) may not be the most suitable choice. Although feed-forward nets are powerful universal function approximators, they can be slow and require substantial amounts of data to learn effectively. Exploring alternative approximators, such as linear approximators based on the current state or RNN architectures that consider temporal dependencies between consecutive states, could be more efficient and effective.

8.4.2. Planning horizon

As discussed in section 6.2, the planning horizon used in this project was kept constant at 3 seconds, with trajectories consisting of 6 actions spanning 0.5 seconds each. It would be interesting to vary the length of the planning horizon and the duration of each action to investigate their impact on the agent's performance. The planning horizon was identified as one of the influential parameters for MBRL agents in the study conducted by Wang et al. (2019). Therefore, exploring different planning horizons would be valuable if PDRL is further developed.

8.4.3. Early termination

The issue of early termination in MBRL was also addressed in the benchmarking study conducted by Wang et al. (2019). Early termination refers to methods, such as not allowing PDRL to plan for collisions, that aim to stop the agent before causing damage to itself or the environment during training. Although this master project did not explicitly investigate the impact of early termination in PDRL, it would be interesting to explore its effects by toggling the early termination on/off or exploring alternative approaches to early termination. This would provide insights into the trade-offs and potential performance implications of employing early termination techniques in MBRL.

8.4.4. Multi agent interactions

The PDRL agent was tested in both static environments ("Walls" and "Naples") and dynamic environments with reactive agents. Across all tests, PDRL underperformed compared to the DDPG counterparts, as discussed in Section 8.1. However, one of the strengths of PDRL and other MBRL agents is their ability to plan ahead and communicate their plans with other agents. This aspect was not explored in the conducted tests. A future experiment could involve placing multiple PDRL agents in an environment where they can communicate their plans with each other. Communication could be direct between the agents or facilitated by a central traffic management system. Figure 8.2 illustrates an example of implementing communication by checking for overlapping trajectories and using traffic rules to determine which agents need to

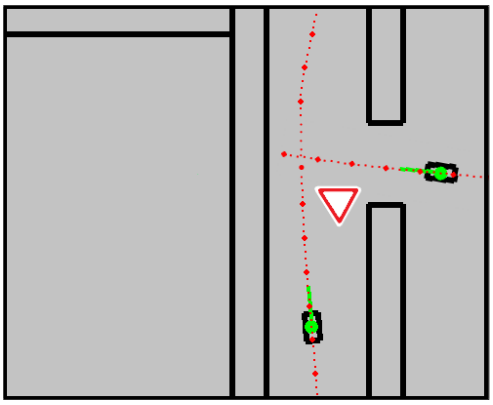


Figure 8.2.: This figure illustrates a scenario where two planning agents communicate their plans, in order to avoid collision.

yield to others. Investigating multi-agent interactions would provide insights into the collaborative capabilities of PDRL and its potential for more sophisticated decision-making in complex environments.

8.4.5. Long-Term Planning

The PDRL architecture developed in this thesis has primarily focused on short-term planning. The next logical step is to integrate it with a long-term goal selector to tackle more complex tasks. The mission planner shown in Figure 5.4 can be replaced by a higher-level algorithm responsible for choosing goals for the short-term planner in order to achieve global or long-term objectives. This approach would resemble the operation of Hierarchical Reinforcement Learning (HRL) algorithms, as discussed in Section 2.4.6. To address the long-term planning aspect, a different DRL method could be employed to solve this higher-level task and coordinate with the short-term planner.

8.4.6. Test PDRL on the LIMO

Throughout the project, the development and testing of the PDRL architecture have been influenced by the AgileX LIMO robotic platform. The environments were designed and tuned to mimic the behavior of the LIMO, and the sensory systems and environment representation were specifically tailored for it. The motivation behind testing PDRL in environments with added system disturbances was to evaluate its performance when deployed on the physical LIMO. Therefore, the natural next step would be to implement and test the PDRL architecture on the LIMO robotic platform. This would provide valuable insights into the real-world applicability and effectiveness of PDRL in a physical robot scenario.

A. Coordinate frame transforms

The general transform function is shown in Equation A.1, transforming a point from frame B to frame A. The transform consists of a rotation between the frames, using the rotation matrix R_B^A, and then adding the translation O_B^A, which represents the origin of frame B given in A's reference frame.

When transforming a point from the vehicle coordinate frame (V) or the center coordinate frame (C) to the world coordinate frame (W), the rotation matrix shown in Equation A.2 is used. When transforming between (V) and (C), there is no rotation, and the identity matrix can be used as shown in Equation A.3. The origins of (C) or (V) given in (W) then need to be added to obtain the full transform.

$$p^A = R_B^A p^B + O_B^A \tag{A.1}$$

$$R_V^W = \begin{bmatrix} \cos\psi & -\sin\psi \\ \sin\psi & \cos\psi \end{bmatrix} = R_C^W \tag{A.2}$$

$$R_C^V = I \tag{A.3}$$

For the inverse transform, going from (W) to (V) or (C), the inverse transform of Equation A.4 is used. Since rotation matrices are symmetrical, transposing them is equivalent to inverting.

$$p^B = R^T p^A - R^T O_B^A \tag{A.4}$$

B. Circogram implementation

This appendix will cover in more detail how the circograms were implemented, which were discussed in chapter 4.

B.1. Sides, lines and vertices

The vehicle structure in the code consists of the vehicle class, which is a child of the Object class. The Object class represents an object containing N vertices given in world coordinates. Sides are defined as lines connecting two consecutive vertices. Therefore, all objects include N vertices, sides, and lines.

The vehicle inherits these characteristics but is defined by a center point, width, length, and a heading. The vertices, sides, and lines are computed based on this information. The vehicle also includes the motion model (see chapter 3). For each movement of the center point, the vertices, sides, and lines are recomputed. As vehicles and objects are considered rigid objects, vertices only need to be calculated once in the vehicle coordinate frame (VCF). However, for each motion of the vehicle, the vertices in the world coordinate frame (WCF) need to be recomputed.

B.2. Side and line intersections

The first step in the circogram algorithm is to compute ego-intersections, which are intersections with the vehicle's own hull. Since circogram rays always start along the x-axis in the center coordinate frame (CCF) and then rotate about the z-axis of the CCF until they return to the start, this calculation is performed only once for each vehicle. These intersections are initially obtained in the CCF. However, for each time-step, they need to be transformed into the WCF, considering the current position and heading of the vehicle.

Once the center of the vehicle and all the ego-intersections in the WCF are available, circogram rays are created as lines crossing these points. The circogram rays are defined by the general line equation shown in figure B.1. To detect intersecting lines, a 2-step process is followed. First, all other objects are iterated, and their vertices in the WCF are inserted into the line equation. The sign of the result indicates whether the vertex is above or below the line. If an object has vertices on both sides, it indicates an intersection.

The final step is to find the exact point of intersection. Based on the previous step, the sides that are intersected can be determined. These sides correspond to the sides between vertices with different signs (see figure B.1). The lines corresponding to these sides can be derived from the two corner points, and the intersection with the ray line can be computed using a simple linear system.

Finally, the intersection that corresponds to the shortest distance is chosen for each ray, and the process moves on to the next ray. However, there is a potential issue with using lines: they are not directed like vectors, and intersections on the other side of the object may be detected. To address this, a simple check is implemented: the distance from the ego-intersection to the

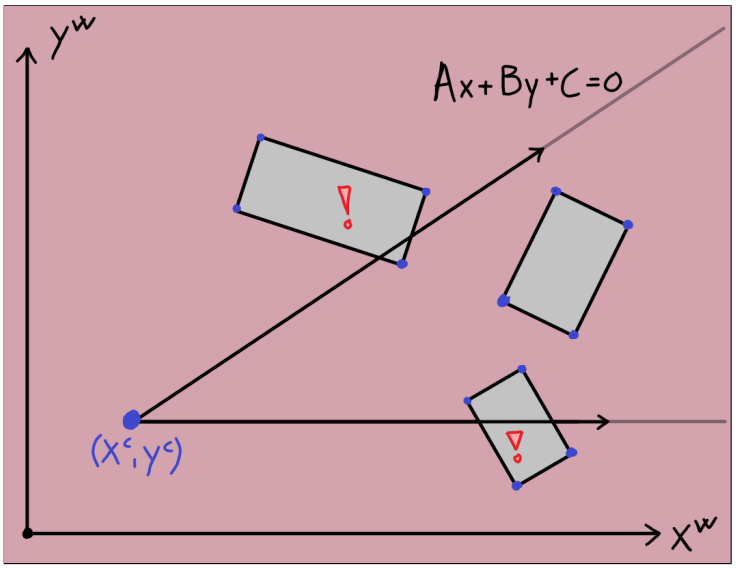


Figure B.1.: The implementation of static circograms

object-intersection should be shorter than the distance from the center to the ego-intersection. This check is illustrated in figure B.2.

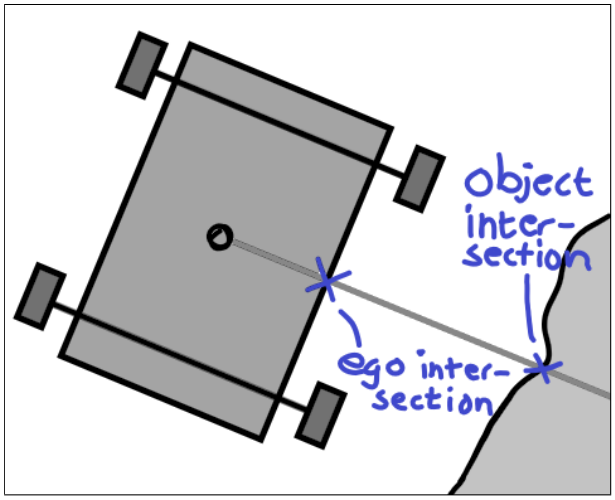


Figure B.2.: The implementation of static circograms: the real distance is found as the difference between ego-intersection and object intersection.

B.3. Circogram code

```
def static_circogram_2(self, N: int, list_objects_simul, d_horizon: float,
↪  gaussian_noise=None):
        """
        Params:
            - N: number of circogram rays
            - list_objects_simul: other objects
            - d_horizon: maximum range of rays
            - Gaussian_noise: list of noise parameters:[mu, sigma]
        Returns:
            - a static circogram
```

```
    """
    # Static arrays >> dynamic arrays
    dist_center_P1 = np.zeros(N)
    dist_center_P2 = np.zeros(N)
    P1 = np.zeros((N, 2))
    P2 = np.zeros((N,2))
    angles = np.zeros(N)
    if gaussian_noise:
        mu, sigma = gaussian_noise
        noise = np.random.normal(mu, sigma, N)
    for n in range(N):
        """
        The 1.st ray of the circogram should always be in the heading direction,
        no matter the orientation of the ego-vehicle.
        """
        angle = 2*np.pi/N * n #
        #angle = 2*np.pi/N * n + np.pi/2 # Now it will go from the the heading (yC),
        ↪  instead of xC.
        angles[n] = angle
        # Find P1 and its distance from the center
        P1[n]=self.find_intersection_line_ego(angle, self.length, self.width)
        dist_center_P1[n]=self.dist_point_point(P1[n], self.position_center)
        # Find P2 and its distance from the center
        ray_line = self.eval_line_point_point(self.position_center, P1[n])
        all_intersections = []
        all_distances = []
        for obj in list_objects_simul:
            for i in range(len(obj.lines)):
                if self.is_line_intersect_entity(ray_line, obj.sides[i]):
                    loc_point = self.find_intersection_line_line(ray_line, obj.lines[i])
                    distance = self.dist_point_point(loc_point, self.position_center)
                    if self.dist_point_point(loc_point, P1[n]) < distance: # Should be
                    ↪  on correct side of object!
                        all_intersections.append(loc_point)
                        all_distances.append(self.dist_point_point(loc_point,
                        ↪  self.position_center))
        # Choose shortest distance intersection
        if all_intersections:  # if it not empty
            min_index = all_distances.index(min(all_distances))  # find position of the
            ↪  minimum distance in the list
            P2[n]=all_intersections[min_index]
            dist_center_P2[n]=all_distances[min_index]
            if gaussian_noise:
                #
                dist_center_P2[n] += dist_center_P2[n]*noise[n]
                # Alsp move P2
                vec_wcf = P2[n] - self.position_center
                vec_ccf = self.WCF_rotate_CCF(vec_wcf)
                # IN CCF
                vec_ccf = vec_ccf + vec_ccf*noise[n]
                P2[n] = self.CCFtoWCF(vec_ccf)
        else: # No intersections
            P2[n]=None
            dist_center_P2[n]=d_horizon

    circogram = [dist_center_P1, dist_center_P2, P1, P2, angles]
    return circogram
```

C. The reactive agent

The reactive agent is illustrated in fig. C.1 and follows a three-state finite state machine (FSM) using thresholds to transition between the states. The states are denoted as D (drive), A (avoidance), and S&R (stop and reverse). The driver is depicted with a "spring" attached to the throttle, indicating that when the driver is out of risk, it defaults to the D-state, which involves a predefined throttle level. The pseudo-code for the agent's implementation is presented in algorithm 3, where states 0, 1, and 2 correspond to states D, A, and S&R, respectively. In the algorithm, t_n and s_n represent the throttle and steering signals from the agent, which control the motion model of the vehicle. Both t_n and s_n fall within the range of [-1, 1], representing maximum forward and backward speeds, as well as steering angles.

The following list provides links to some demos showcasing the agent's performance in different scenarios:

1. **DEMO 1 - driving in a "city".**
2. **DEMO 2 - Drivers in an open setting.**

For the final project, this agent served primarily to test the vehicle motion model, the environment, and as dynamic obstacles for evaluating the final agents.

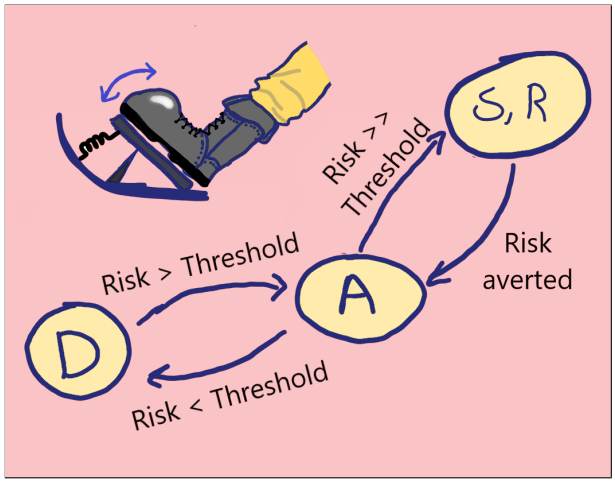


Figure C.1.: The reactive test-driver is a three-state finite state machine. The states are: D (drive), A (avoidance), S&R (stop and reverse). The throttle is illustrated with a spring, indicating how the driver defaults back to a given throttle setting when not performing collision avoidance.

Algorithm 3 The reactive agent

Input: DC, t_n, s_n, state
Output: t_{n+1}, s_{n+1}, new-state

```
risks, distances, angles = DC
max-risk-index = max(risks)
R = risks[max-risk-index]                          ▷ R : max risk
A = angles[max-risk-index]                         ▷ A : angle of R

if state == 0 then                                 ▷ D : Default drive
    t_{n+1}, s_{n+1} = t_max, 0
    if R > threshold then
        new-state = 1
    end if
else if state == 1 then                            ▷ A : Avoidance action
    t_{n+1}, s_{n+1} = Avoid-collision(R, A)
                                                   ▷ Simple PD controller
    if R < threshold then
        new-state = 0
    else if R >> threshold then
        new-state = 2
        risky-distance = distances[max-risk-index]
        risky-angle = A
                                                   ▷ Save these two to be used in state 2
    end if
else if state == 2 then                            ▷ S,R : Stop, Reverse
    t_{n+1} = t_min                                ▷ Reversing
    if distances[risky-angle] >> risky-distance then
        self.state = 1                             ▷ Risky angle was cleared
    else if R >> threshold then
        self.state = 1                             ▷ Reversing is too dangerous
    end if
end if
```

D. The code base

This is a brief introduction to the codebase available on in GitHub

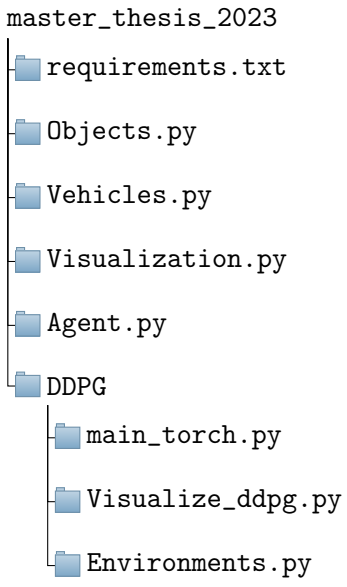
https://github.com/henrfj/master_thesis_2023.

To get started, it is recommended to download the required Python libraries by running the following command in your terminal:

```
$ pip install -r requirements.txt
```

In the root directory of the project, you will find the Python files Objects.py and Vehicle.py, which implement the properties of all static and dynamic objects in the project. Visualization.py contains the visualization class based on Pygame, which is used throughout the project. Agent.py contains the code for the reactive, dynamic circogram-based agents used as dynamic obstacles in the project.

In the sub-folder DDPG, you will find the core components of the thesis. The file main_torch.py includes the code for training the different agents, both P-DRL and DDPG. Multiple versions of both agents were tested and compared, with the most successful DDPG agent being called v22, while v40 is the most successful P-DRL version. The file visualize_ddpg.py uses the trained models stored in the "checkpoints" directory to visualize episodes for the agents. This file heavily relies on the visualization.py file in the root directory. Finally, the file environments.py implements the different training environments for DDPG and P-DRL. It's worth noting that all environments are implemented as GYM environments, to make them more compatible with other RL methods.



Bibliography

Anderson, B. D. and Moore, J. B.: 2007, *Optimal control: linear quadratic methods*, Courier Corporation.

Arrieta, A. B., Díaz-Rodríguez, N., Del Ser, J., Bennetot, A., Tabik, S., Barbado, A., García, S., Gil-López, S., Molina, D., Benjamins, R. et al.: 2020, Explainable artificial intelligence (xai): Concepts, taxonomies, opportunities and challenges toward responsible ai, *Information fusion* **58**, 82–115.

Bouhamed, O., Ghazzai, H., Besbes, H. and Massoud, Y.: 2020, Autonomous uav navigation: A ddpg-based deep reinforcement learning approach, *2020 IEEE International Symposium on Circuits and Systems (ISCAS)*.

Campbell, S., O'Mahony, N., Krpalcova, L., Riordan, D., Walsh, J., Murphy, A. and Ryan, C.: 2018, Sensor technology in autonomous vehicles: A review, *2018 29th Irish Signals and Systems Conference (ISSC)*, IEEE, pp. 1–4.

Catapang, A. N. and Ramos, M.: 2016, Obstacle detection using a 2d lidar system for an autonomous vehicle, *2016 6th IEEE International conference on control system, computing and engineering (ICCSCE)*, IEEE, pp. 441–445.

Chen, C.-T.: 2013, *Linear System Theory and Design*, 4th edn, Oxford University Press, Inc., USA.

CLARK, M.: 2022, Amazon announces its first fully autonomous mobile warehouse robot.
URL: *https://www.theverge.com/2022/6/21/23177756/amazon-warehouse-robots-proteus-autonomous-cart-delivery*

Doll, B. B., Simon, D. A. and Daw, N. D.: 2012, The ubiquity of model-based reinforcement learning, *Current opinion in neurobiology* **22**(6), 1075–1081.

Fujimoto, S., Hoof, H. and Meger, D.: 2018, Addressing function approximation error in actor-critic methods, *International conference on machine learning*, PMLR, pp. 1587–1596.

Goodfellow, I., Bengio, Y. and Courville, A.: 2016, *Deep Learning*, MIT Press. `http://www.deeplearningbook.org`.

Grondman, I., Busoniu, L., Lopes, G. A. and Babuska, R.: 2012, A survey of actor-critic reinforcement learning: Standard and natural policy gradients, *IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews)* **42**(6), 1291–1307.

Haarnoja, T., Zhou, A., Abbeel, P. and Levine, S.: 2018, Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor, *International conference on machine learning*, PMLR, pp. 1861–1870.

Hafner, D., Lillicrap, T., Ba, J. and Norouzi, M.: 2019, Dream to control: Learning behaviors by latent imagination, *arXiv preprint arXiv:1912.01603* .

Hengst, B.: 2010, *Hierarchical Reinforcement Learning*, Springer US, Boston, MA, pp. 495–502.
URL: *https://doi.org/10.1007/978-0-387-30164-8*$_3$63

international, S.: 2021, Taxonomy and definitions for terms related to driving automation systems for on-road motor vehicles.
URL: *https://web.archive.org/web/20211220101755/https://www.sae.org/standards/content/j3016_202104/*

Kendall, A., Hawke, J., Janz, D., Mazur, P., Reda, D., Allen, J.-M., Lam, V.-D., Bewley, A. and Shah, A.: 2019, Learning to drive in a day, *2019 International Conference on Robotics and Automation (ICRA)*, IEEE, pp. 8248–8254.

Khan, M. U., Zaidi, S. A. A., Ishtiaq, A., Bukhari, S. U. R., Samer, S. and Farman, A.: 2021, A comparative survey of lidar-slam and lidar based sensor technologies, *2021 Mohammad Ali Jinnah University International Conference on Computing (MAJICC)*, IEEE, pp. 1–8.

Kingma, D. P. and Ba, J.: 2014, Adam: A method for stochastic optimization, *arXiv preprint arXiv:1412.6980* .

Kiran, B. R., Sobh, I., Talpaert, V., Mannion, P., Al Sallab, A. A., Yogamani, S. and Pérez, P.: 2021, Deep reinforcement learning for autonomous driving: A survey, *IEEE Transactions on Intelligent Transportation Systems* **23**(6), 4909–4926.

Klose, P. and Mester, R.: 2018, Simulated autonomous driving in a realistic driving environment using deep reinforcement learning and a deterministic finite state machine.

Lee, J. H.: 2011, Model predictive control: Review of the three decades of development, *International Journal of Control, Automation and Systems* **9**(3), 415–424.

Leung, J., Shen, Z., Zeng, Z. and Miao, C.: 2021, Goal modelling for deep reinforcement learning agents, *Machine Learning and Knowledge Discovery in Databases. Research Track: European Conference, ECML PKDD 2021, Bilbao, Spain, September 13–17, 2021, Proceedings, Part I 21*, Springer, pp. 271–286.

Lillicrap, T. P., Hunt, J. J., Pritzel, A., Heess, N., Erez, T., Tassa, Y., Silver, D. and Wierstra, D.: 2015, Continuous control with deep reinforcement learning.
URL: *https://arxiv.org/abs/1509.02971*

Lin, Y., McPhee, J. and Azad, N. L.: 2021, Comparison of deep reinforcement learning and model predictive control for adaptive cruise control, *IEEE Transactions on Intelligent Vehicles* pp. 221–231.

Lubars, J., Gupta, H., Chinchali, S., Li, L., Raja, A., Srikant, R. and Wu, X.: 2021, Combining reinforcement learning with model predictive control for on-ramp merging, *2021 IEEE International Intelligent Transportation Systems Conference (ITSC)*, IEEE, pp. 942–947.

Milliken, W. F., Milliken, D. L. and Metz, L. D.: 1995, *Race car vehicle dynamics*, SAE international Warrendale.

Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A. A., Veness, J., Bellemare, M. G., Graves, A., Riedmiller, M., Fidjeland, A. K., Ostrovski, G. et al.: 2015, Human-level control through deep reinforcement learning, *nature* **518**(7540), 529–533.

Moerland, T. M., Broekens, J. and Jonker, C. M.: 2020, Model-based reinforcement learning: A survey, *arXiv preprint arXiv:2006.16712* .

N., W.: 1906, *Modern steam road wagons*, London, New York, Bombay, Longmans, Green, and co., London.

Nagabandi, A., Kahn, G., Fearing, R. S. and Levine, S.: 2018, Neural network dynamics for

model-based deep reinforcement learning with model-free fine-tuning, *2018 IEEE international conference on robotics and automation (ICRA)*, IEEE, pp. 7559–7566.

Pfeiffer, D. and Franke, U.: 2010, Efficient representation of traffic scenes by means of dynamic stixels, *2010 IEEE Intelligent Vehicles Symposium*, IEEE, pp. 217–224.

Pfeiffer, M., Schaeuble, M., Nieto, J., Siegwart, R. and Cadena, C.: 2017, From perception to decision: A data-driven approach to end-to-end motion planning for autonomous ground robots, *2017 IEEE International Conference on Robotics and Automation (ICRA)*, pp. 1527–1533.

Por, E., van Kooten, M. and Sarkovic, V.: 2019, Nyquist–shannon sampling theorem, *Leiden University* **1**(1).

Racanière, S., Weber, T., Reichert, D., Buesing, L., Guez, A., Jimenez Rezende, D., Puigdomènech Badia, A., Vinyals, O., Heess, N., Li, Y. et al.: 2017, Imagination-augmented agents for deep reinforcement learning, *Advances in neural information processing systems* **30**.

Ribeiro, M. I.: 2004, Kalman and extended kalman filters: Concept, derivation and properties, *Institute for Systems and Robotics* **43**(46), 3736–3741.

Rosolia, U. and Borrelli, F.: 2017, Learning model predictive control for iterative tasks. a data-driven control framework.

Rosolia, U., Zhang, X. and Borrelli, F.: 2018, Data-driven predictive control for autonomous systems, *Annual Review of Control, Robotics, and Autonomous Systems* **1**, 259–286.

Russel, S. and Norvig, P.: 2016, *Artificial intelligence, A modern approach*, third edn, Pearson, Harlow, England.

Sauer, T.: 2006, *Numerical analysis*, second edn, Pearson educational, Boston.

Silver, D., Hubert, T., Schrittwieser, J., Antonoglou, I., Lai, M., Guez, A., Lanctot, M., Sifre, L., Kumaran, D., Graepel, T. et al.: 2017, Mastering chess and shogi by self-play with a general reinforcement learning algorithm, *arXiv preprint arXiv:1712.01815* .

Singh, S., Jaakkola, T., Littman, M. L. and Szepesvári, C.: 2000, Convergence results for single-step on-policy reinforcement-learning algorithms, *Machine learning* **38**, 287–308.

Sola, J., Deray, J. and Atchuthan, D.: 2018, A micro lie theory for state estimation in robotics, *arXiv preprint arXiv:1812.01537* .

Soltakhanov, S., Yushkov, M. and Zegzhda, S.: 2009, *Mechanics of non-holonomic systems: A New Class of control systems*, Foundations of Engineering Mechanics, Springer Berlin Heidelberg.
URL: *https://books.google.no/books?id=lm7MFRzYtOsC*

Spong, M. W., Hutchinson, S. and Vidyasagar, M.: 2020, *Robot modeling and control*, second edn, John Wiley & Sons, Ltd, Hoboken, USA.

Sutton, R. S. and Barto, A. G.: 2020, *Reinforcement Learning, an introduction*, second edn, MIT Press, London.

Tătulea-Codrean, A., Mariani, T. and Engell, S.: 2020, Design and simulation of a machine-learning and model predictive control approach to autonomous race driving for the f1/10 platform, *IFAC-PapersOnLine* **53**(2), 6031–6036.

Van Hasselt, H., Doron, Y., Strub, F., Hessel, M., Sonnerat, N. and Modayil, J.: 2018, Deep reinforcement learning and the deadly triad, *arXiv preprint arXiv:1812.02648* .

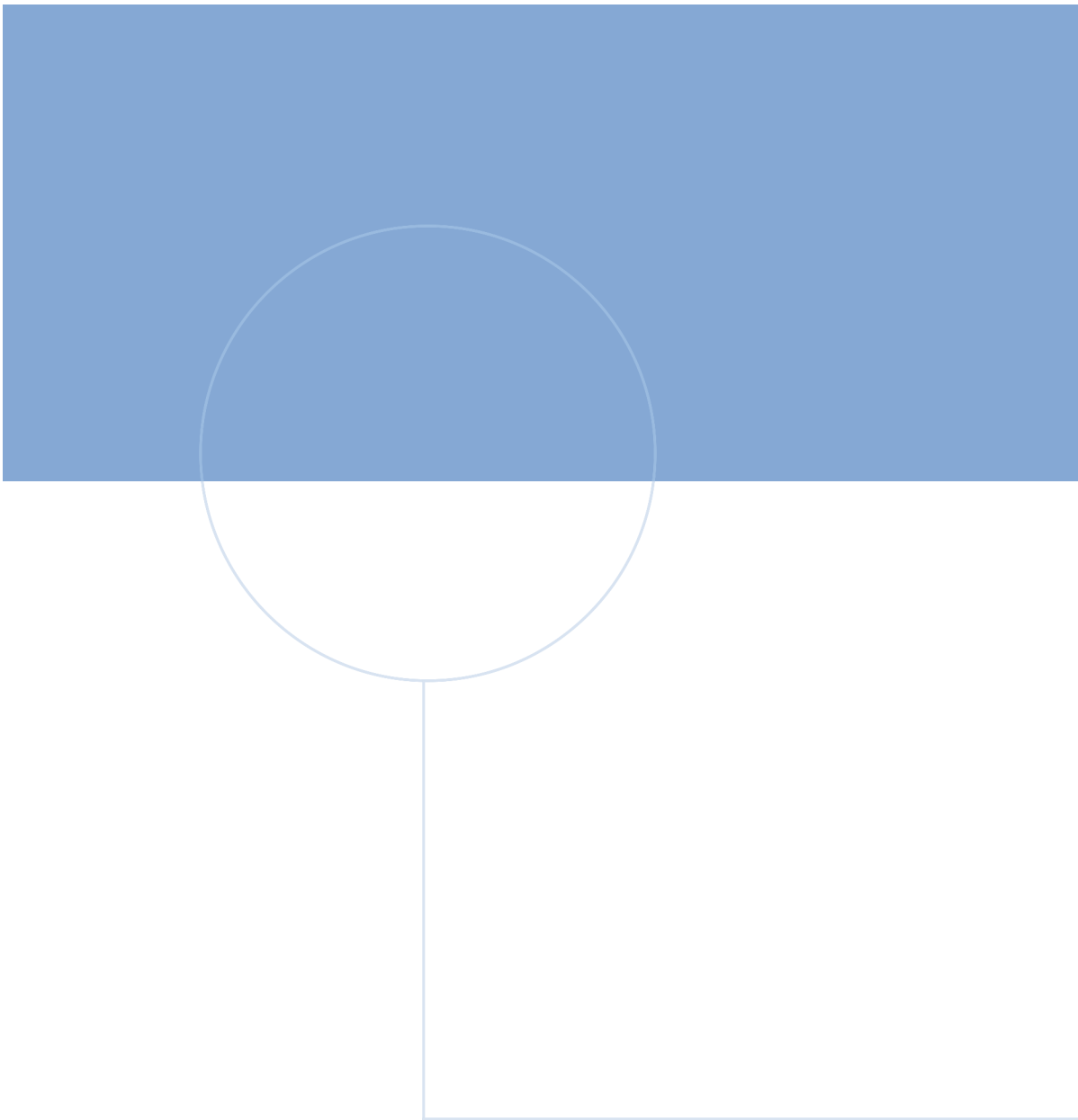
Van Hasselt, H., Guez, A. and Silver, D.: 2016, Deep reinforcement learning with double q-learning, *Proceedings of the AAAI conference on artificial intelligence*, Vol. 30.

Wang, S., Jia, D. and Weng, X.: 2018, Deep reinforcement learning for autonomous driving, *arXiv preprint arXiv:1811.11329* .

Wang, T., Bao, X., Clavera, I., Hoang, J., Wen, Y., Langlois, E., Zhang, S., Zhang, G., Abbeel, P. and Ba, J.: 2019, Benchmarking model-based reinforcement learning.
URL: *https://arxiv.org/abs/1907.02057*

Yu, J., Su, Y. and Liao, Y.: 2020, The path planning of mobile robot by neural networks and hierarchical reinforcement learning, *Frontiers in Neurorobotics* .
URL: *https://www.frontiersin.org/articles/10.3389/fnbot.2020.00063/full#B17*

Yuan, X., Lian, F. and Han, C.: 2014, Models and algorithms for tracking target with coordinated turn motion, *Mathematical Problems in Engineering* .



NTNU

Norwegian University of
Science and Technology