FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO

Teaching Robot Learning in ROS2

Filipe Reis Almeida

Mestrado em Engenharia Informática e Computação

Supervisor: Armando Jorge Miranda de Sousa

Co-Supervisor: Gonçalo da Mota Laranjeira Torres Leão

July 27, 2023

Teaching Robot Learning in ROS2

Filipe Reis Almeida

Mestrado em Engenharia Informática e Computação

July 27, 2023

Abstract

Robot Learning is one of the most important areas in robotics and its relevance has only increased in more recent years. The Robot Operating System (ROS) has been one of the most used architectures as it allows to either program and control physical robots or to create and run equivalent simulations. Using ROS is not a simple task and is especially complicated for new developers. In addition to the high entry level, the first version of ROS (ROS 1) is reaching its end-of-life and a lot of users are yet to make the transition to the new version (ROS 2).

When combined with Robotics, Machine Learning (ML) originates the field of Robot Learning with Reinforcement Learning (RL) being the most popular strategy used to train robots to perform tasks. Although RL is presented in a lot of teaching scenarios, it is rarely taught in conjunction with robotics. Applying RL to Robotics is not necessarily an intuitive process and there is value in learning the two topics together. All these factors result in a large demand for tools to aid users when trying to have all these technologies working concurrently.

This dissertation aims to develop a learning kit that can be used to teach Robot Learning in different educational scenarios and for students with different levels of education and expertise in topics related to Robotics. As physical robots are usually too expensive to be used for educational purposes, the kit created works entirely in simulated scenarios and only uses open-source free software. All examples provided use Flatland, a lightweight two-dimensional simulator, and are compatible with ROS 2. The kit also contains several tutorials that start by introducing the user to the simulation environment and teaching basic robot controls. From that, they move to more complex concepts, such as using RL to train the robot to perform more complex tasks. The kit will have a large relevance in this field since it teaches the user how to apply RL in Robotic applications while also focusing on ROS 2, to which there is a lack of similar offers available on the market.

Since the examples shown in the tutorials involve mostly path planning tasks, some initial work was done to study the viability of using RL for this problem. This work not only showed that it was a viable option, but it also contributed to many decisions made when creating the kit's framework. This dissertation will also present the methodology and results of this work.

To evaluate how the kit performs in real scenarios, user tests were conducted to better understand its applicability. Once the kit was ready to be tested, several subjects with different experience levels in Robotics were asked to follow the tutorials. After finishing them, they were asked to fill out a form with their experience, providing both quantitative and qualitative feedback, and also to answer a short quiz about the topics covered. The results gathered show very positive feedback, with almost all the tested subjects agreeing that the kit provided a productive learning experience.

Keywords: ROS, Robot Learning, Educational Robotics

Resumo

O Robot Learning é uma das áreas mais importantes da robótica e sua relevância tem aumentado nos últimos anos. O Robot Operating System (ROS) tem sido uma das arquiteturas mais utilizadas, pois permite programar e controlar robôs físicos ou criar e executar simulações equivalentes. Usar ROS não é uma tarefa simples e é especialmente complicado para novos utilizadores. Além do alto nível de entrada, a primeira versão do ROS (ROS 1) está a chegar ao seu fim de vida e muitos utilizadores ainda não fizeram a transição para a nova versão (ROS 2).

Quando combinado com Robótica, Machine Learning(ML) origina a área de Robot Learning sendo Reinforcement Learning (RL) a estratégia mais popular para treinar robôs. Embora RL esteja inserido em muitos cenários de ensino, raramente é ensinado em conjunto com a robótica. Aplicar RL à Robótica não é necessariamente um processo intuitivo e é vantajoso aprender os dois tópicos juntos. Todos esses fatores resultam numa grande procura por ferramentas para auxiliar utilizadores a tentar colocar todas estas tecnologias a funcionar simultaneamente.

Esta dissertação tem como objetivo desenvolver um kit de aprendizagem que possa ser utilizado para ensinar Robot Learning em diferentes cenários educacionais e a alunos com diferentes níveis de escolaridade e especialização em temas relacionados com Robótica. Como robôs físicos costumam ser demasiado caros para serem usados para fins educacionais, o kit criado funciona inteiramente em cenários simulados e utiliza apenas software gratuito e *open-souce*. Todos os exemplos fornecidos usam Flatland, um simulador bidimensional eficiente, e são compatíveis com ROS 2. O kit também contém vários tutoriais que começam por introduzir o utilizador ao ambiente de simulação e aos controlos básicos do robô. A partir daí, passam para conceitos mais complexos, como usar RL para ensinar o robô a executar tarefas mais complexas. O kit terá uma grande relevância nesta área, pois ensina utilisadores a aplicar RL em aplicações Robóticas, além de se focar em utilizar ROS 2, para o qual faltam ofertas semelhantes disponíveis no mercado.

Como os exemplos mostrados nos tutoriais envolvem principalmente tarefas de planear trajetórias (*path planing*), algum trabalho inicial foi feito para estudar a viabilidade de usar RL para este problema. Esse trabalho não só mostrou que era uma opção viável, como também contribuiu para muitas das decisões tomadas na hora de criar a estrutura do kit. Esta dissertação também apresentará a metodologia e os resultados deste trabalho. Para avaliar o desempenho do kit em cenários reais, foram realizados testes com utilizadores para entender melhor sua aplicabilidade. Com o kit pronto para ser testado, vários voluntários com diferentes níveis de experiência em Robótica foram convidados a seguir os tutoriais. Após a finalização, foi-lhes pedido a preenchessem um formulário sobre sua experiência, fornecendo feedback quantitativo e qualitativo, e também que respondessem a um pequeno questionário sobre os tópicos abordados. Os resultados obtidos mostram um feedback muito positivo, com quase todos os voluntários testados a concordarem que o kit proporcionou uma experiência de aprendizagem produtiva.

Keywords: ROS, Aprendizagem Robótica, Robótica Educacional

Acknowledgements

Firstly, I would like to thank my supervisor, Professor Armando Sousa, and my second supervisor, Gonçalo Leão, for all the guidance throughout this project. Without their help, the success found in this project would not be possible.

I would also like to acknowledge all the family and friends who have helped me at some point in my academic life. I will always be grateful for all the people that, through all the good and bad times, were there to support me when needed.

Filipe Reis Almeida

"You don't have to see the whole staircase,
Just take the first step."

Martin Luther King, Jr.

Contents

List of Figures

List of Tables

Abbreviations

AI	Artificial Intelligence
APF	Artificial Potential Fields
DDPG	Deep Deterministic Policy Gradient
DDS	Data Distribution Service
DQN	Deep Q-Network
DRL	Deep Reinforcement Learning
eREPS	Episodic Relative Entropy Policy Search
FEUP	Faculty of Engineering of the University of Porto
IMU	Inertial Measurement Unit
IR	Intelligent Robotics
IW	Intelligent Wheelchairs
LiDAR	Light Detection and Ranging
M	Mean
ML	Machine Learning
MLP	Multilayer Perceptron
NN	Neural Networks
OS	Operating System
PPO	Proximal Policy Optimization
RL	Reinforcement Learning
ROS	Robot Operating System
SAC	Soft Actor Critic
SARSA	State–action–reward–state–action
SDK	Software Development Kit
SL	Supervised Learning
SLAM	Simultaneous Localization and Mapping
SD	Standard Deviation
TRPO	Trust Region Policy Optimization
UL	Unsupervised Learning
VM	Virtual Machine
XML	Extensible Markup Language

Chapter 1

Introduction

This first chapter will introduce the work developed in this dissertation. It starts by providing some context in the field of Educational Robotics which explains the motivations behind this project. It will also briefly present the work developed as well as propose relevant research questions.

1.1 Context

As technology around it develops, the field of Robotics is becoming an ever-growing part of the daily life of human beings. There is also an increasing need for robots to become as autonomous as possible, which means this field is often paired with Artificial Intelligence (AI), creating the field of Intelligent Robotics (IR). IR already has a lot of applications in modern society, ranging from simple robots used as children's toys or learning tools to machines with very practical purposes, such as domestic appliances or even industrial robots. With the growing number of applications for it, a larger amount of qualified developers in this field is required, increasing the demand for tools to aid its learning.

Throughout the years of the existence of the field of Robotics, several frameworks were created to develop applications but, in the past decades, the most popular has been the Robot Operating System (ROS). ROS is a Software Development Kit (SDK) originally idealized in 2007 at the Stanford Artificial Intelligence Lab that was later acquired by Google [46]. Despite presenting a lot of advantages, it also suffers from a major issue: it presents a hard learning curve for new users. The first version of ROS (ROS 1) was released in 2010 and its latest and last distribution (ROS Noetic) will have its end-of-life by 2025. The replacement is ROS 2 which had its first non-Beta release in 2017 and offers improvements in multiple areas, among them being compatibility and security [12].

1.2 Motivation

ROS is currently going through a transitional period as most ROS 1 users are yet to start using ROS 2 and migrate their current projects. Learning ROS 1 is not a prerequisite to learning ROS

2, meaning that new users should also start with the most recent version since the transition is inevitable. One of the problems that are delaying this change is the lack of tutorials and courses available for ROS 2.

In Robotics, simulations are often used to be able to perform experiments without the cost of producing the actual robot. However, the ultimate goal in Robotics is usually to create physical machines that are capable of performing tasks in the real world. This means that there is a lot of value in using simulations and real robots in parallel when teaching Robotics. The issue is that the high costs of physical robots make their usage almost impossible in most teaching scenarios.

AI can be applied to robots using several different strategies. One of the most popular ones is Reinforcement Learning (RL). There is a limited amount of tutorials to help new learners use RL for Robotics, which becomes even more scarce when looking for information when using it paired with ROS. All these shortcomings in the current market paired with the increasing demand for tools to teach IR, originated an interest in creating a teaching tool that encapsulates all the mentioned technologies. It is intended to focus on using ROS 2 to apply RL in a simulated environment providing a solid introduction to IR with many concepts that can also be applied in real robots

1.3 Objectives

This dissertation aims to develop and evaluate a kit that can be used to teach RL in conjunction with Robotics using ROS as the base. To achieve this goal, it provides a framework that combines several components and as well as relevant examples in a simulated environment. The kit will contain:

- A simulation framework with a simple differential drive robot model and several maps and scenarios. As the robot is only able to move across a horizontal plane, it is possible to use a 2D simulation. By removing one of the axes, the simulation becomes more lightweight and a lot simpler to understand. The simulator chosen for this kit is the Flatland Simulator [1].
- A set of tutorials on how to control the robot using ROS 2. The first tutorial will serve as an introduction to the framework and the following tutorials show the user how to apply RL algorithms to teach the robot how to perform a task.

Since the simulator used is 2D, there are some limitations on what type of tasks the robots can perform. This is relevant as it was necessary to decide what the RL agent would try to learn in the tutorials. The final decision was to use path planning as the main task to be explored in the kit. Seeing that a two axes configuration can represent grounded robots moving in a horizontal plane, using a 2D simulation does not present many issues. In addition to that, path planning is visually intuitive while also being very relevant in modern-day Robotics with many applications [44].

[1] `https://github.com/avidbots/flatland`

To help ensure the success of the RL algorithms in the tutorials, some initial work was done to test the viability of using it to tackle path planning in a Flatland and ROS 2 simulation. This work consisted of placing robots in several different scenarios and having them attempt to learn how to navigate to a target area. The first part targeted using just one robot with the second evolving into using two traveling side by side. To maintain an approach that considers real-life scenarios, the second part of this work re-envisioned the problem from the perspective of using Intelligent Wheelchairs (IW) instead of abstract robots. Although the setup is very similar since an IW is very similar to a differential drive robot in the context of a 2D simulation, this approach requires several other considerations that relate to the real-life application. In some scenarios during this work, the Artificial Potential Fields (APF) [65] algorithm was used as a baseline of comparison for RL.

The goal for the kit developed is to provide simple and replicable tutorials that can be followed by users with different levels of expertise in topics related to Robotics and RL. To validate the kit created, user tests were performed and, after collecting relevant metrics, an evaluation was made.

This kit has a similar approach as the one developed by Ventuzelos *et al.*[62]. It shares a lot of the framework since it also uses Flatland and ROS 2 to teach Robot Learning but also focuses on ROS 1 and using a real robot. This meant there was a broad focus and the tutorials did not go into much detail. The main difference between the kit proposed by Ventuzelos *et al.* from the one being presented in this paper is that by focusing only on Robot Learning with ROS 2 and Flatland, the tutorials in this kit go into much more depth in each of the components of its framework. This work also uses different RL frameworks and resources as well as being able to use Flatland directly with ROS 2, without needing the ROS 1 bridge [2].

1.4 Research questions

The research questions that this dissertation will attempt to address are:

[2] https://github.com/ros2/ros1_bridge

RQ1 *Is it possible to create tutorials about current day IR, namely ROS 2 and RL, that can be followed by users with different levels of previous knowledge on the topics?* As the kit is meant to be used in different stages of education, the tutorials should be simple enough to be used by students with less knowledge of the explored topics while still having valuable content for more experienced users.

RQ2 *Can users learn about IR, namely ROS 2 and RL, autonomously?* The kit aims to have its users learn on their own with the support of tutorials and other online resources. The problem is that there could be a lot of value gained in teaching IR with in-person lessons so it is important to determine what are the advantages and disadvantages of the autonomous approach.

RQ3 *Can a learning kit that combines Robotics, ROS 2 and RL be effective in teaching all the topics at once?* Despite being strongly connected, all these topics can be taught individually. There could be advantages in teaching everything in parallel but it could also be the case that the amount of information is too excessive for users to learn simultaneously.

1.5 Document structure

The rest of the document will start by exploring the literature relative to all the different tools that will be used in the kit in Chapter 2. Chapters 3 and 4 will present the initial work done to study the viability of using RL in path planning inside a simulation. Chapter 5 will demonstrate how the learning kit was developed and tested. Finally, Chapter 6 will present the conclusions and possible future work for this dissertation.

All the tutorials and examples developed for the kit are available in Appendices A.1, A.2, A.3, A.4 and A.5, as well as a slideshow created to explain a method to manage Flatland layers in Appendix A.6. The form used in the user tests is also available in Appendix B.1 followed by a quiz that all subjects answered in Appendix B.2. All the answers to both the form and the quiz by each subject are in Appendix B.3. To help better understand the results provided by this dissertation, a repository [3] was setup with several videos of experiments done in the various parts of this work. Three articles originated from this dissertation. The first one corresponds to the work presented in Chapter 3 and was already published in the IEEE International Conference on Autonomous Robot Systems and Competitions (ICARSC) 2023 [27] [4]. The second is a journal article that corresponds to the work presented in Chapter 4 and is ready to be submitted to a journal yet to be decided. The final article is based on the work presented in Chapter 5 and was already submitted to the Robot 2023 conference [5].

[3] https://github.com/FilipeAlmeidaFEUP/dissertation_videos/
[4] https://www.icarsc.pt/
[5] https://robot2023.isr.uc.pt/

Chapter 2

Teaching Robot Learning

Before the development of the learning kit, a thorough literature review was conducted to better understand how IR is taught in the present day. This chapter presents that review and is divided into several sections, each one tackling a different subject relevant to this dissertation.

2.1 Robot Learning

Robot Learning is a field that combines Machine Learning (ML), a subset of AI, with Robotics. ML is used to allow computers to learn how to perform a task without human intervention and be able to adapt to their environment [2]. The difference between ML to Robot Learning is applying the concept specifically to robots instead of generalizing it to computers.

ML has three main paradigms: Supervised Learning (SL), Unsupervised Learning (UL) and Reinforcement Learning (RL). SL requires a data set that indicates what the correct outputs are for certain inputs. This allows an agent to learn by example and figure out the patterns that allow the mapping between the inputs to their correct output. In UL, despite also relying on a pre-existing data set, there are no labels in it, meaning that the computer is tasked with identifying the best outputs by itself by recognizing patterns in the existing data. RL has the particularity of not requiring any prior knowledge by learning while performing the desired task. The agent learns by experimenting, trying all the solutions, and, over time, starts learning what outputs produce the best results for each input [2]. Each of these paradigms has its advantages and its effectiveness depends mainly on the type of task to be learned.

In Robot Learning, the inputs and outputs require a more specific definition. Since the agent represents the robot itself, the input is usually the perception that the robot has of the environment it is inserted in, given by the sensors it contains. The output is an action the robot can perform by using its actuators, with a well-defined beginning and end. Obtaining a trustworthy data set for the inputs a certain robot is going to encounter before it starts performing the task is usually not feasible. In Robotics there are a lot of factors that influence the environment, making it hard to replicate scenarios. This means that data sets produced by other experiments are usually not reusable. The only reliable data comes from inserting the robot in the specific scenario it is going to train. These reasons make both SL and UL less adequate to be used with Robot Learning. In comparison, RL does not require any prior data and the inputs appear while performing the task. This creates a scenario where there is an autonomous, adaptable and continuous acquisition of knowledge as the different states of the environment are presented to the agent. For that reason, Robot learning is very closely related to RL which is why the kit to be developed will use this paradigm of ML [1].

2.2 Problems of using Reinforcement Learning in Robotics

RL presents several advantages when applied in Robotics. The fact that no prior data is necessary, its adaptability to the environment and the ability to learn without human intervention allows the agent to discover solutions for very complex tasks. Despite being one of the best paradigms to use in Robot Learning, it also has some disadvantages that become very evident in certain scenarios.

One of the main issues in RL is the time and resources it usually requires to reach an acceptable solution. Although it also contributes to the ability to eventually perform intricate tasks, the large number of repetitions required by RL algorithms to train the agent can be an obstacle to its usage as projects typically have limited time and computational resources. Robotics simulations can frequently accelerate time to help circumvent this issue but physical robots do not offer a lot of possible options to speed up the process.

When creating a RL model, the first version is very unlikely to immediately produce the desired result. There is never a guarantee that the agent will be able to converge to a solution at all as it is possible that it does not find any patterns that allow it to do so [41]. This means that the model will need to be experimented with several versions and, due to its unpredictability, RL normally requires a lot of trial and error. The fact that, even for minor changes in the model, the process of learning has to usually be restarted, only exacerbates the concerns about the excessive time.

A problem related to ML in general is over-fitting the model, happening when the model learns to specifically for the data used in training and is not effective for scenarios that present a slightly different problem. This issue has also been detected in RL [72]. Another problem that seems to occur when over-training in RL is Catastrophic Forgetting which happens when the algorithm starts using a solution and small changes to incorporate new data end up affecting the entire network and disrupt the patterns already learned [7]. The over-training risks create the necessity to define good stopping criteria for the training process, which is often not easy to accomplish.

2.3 Reinforcement Learning Algorithms

RL is a paradigm of ML and, by itself, it can not be used in a practical scenario. To create a model that can actually learn, a well-defined way to map the effectiveness of each output for the different inputs and how to update them over training is required. This idea is known as a policy.

[1]`https://en.wikipedia.org/wiki/Robot_learning`

Throughout the years, several algorithms with different policies were developed and each one presents its own concrete definition of RL. In the past years, most new algorithms being presented make use of Neural Networks (NN) and are part of a subset of RL called Deep RL (DRL) [28].

Despite all having distinct approaches, all RL algorithms share the same paradigm, meaning that there is a lot in common between them [47]. Some fundamental concepts in all RL algorithms are:

- State - A set of numerical values that represent the environment at a given point in time. The state serves as the input for the algorithm.
- Action - A set of numerical values that represent an action that the agent can perform. It serves as the output and is decided based on the previous state. Performing an action typically changes the state of the environment.
- Reward - For each pair state-action the agent uses, a reward can be calculated. The reward can be based on any information available, including the states immediately before and after performing the action and even on the chosen action itself. The reward is how the algorithm labels pair state-action as a positive or negative decision. The larger the reward, the better the action was for the state.
- Step - Is the name given to the process of choosing an action to the given state, performing the action, reading the new state and attributing a reward for the pair state-action. The step needs to have a well-defined beginning and end but different actions may take different times to complete so the time each step takes can be variable.
- Episode - Is defined by all the steps taken between an initial state and an end state. The initial and end states are defined by the model and depend on the environment and the agent and the task to be performed. When an end state is reached, no actions can be executed, so the agent needs to be reset to an initial state.

When applying a RL algorithm to a robot, some adaptations need to be considered. The input can only be read by the robot's sensors, which means that the state has to be described by its readings instead of having an omnipresent knowledge of the environment. RL algorithms require a discrete model to fit the state-action pattern but the environment that robots exist in is a continuous one. In theory, it should be necessary to translate the continuous environment to a discrete one. In reality, this is already done because, in the current computation architecture, it is not possible to truly represent a continuous system. Every component is dependent on update rates, from the processor to the robot sensors and actuators, in either simulation or reality. Pseudo-continuous systems are created with very high update rates. The only adaptation that needs to be made is to coordinate the updates of all components to read or write values at the correct timings.

The rest of this Section will briefly present several of the most relevant RL algorithms and describe some advantages and disadvantages in their usage.

2.3.1 Q-Learning

The Q-Learning algorithm is one of the simplest forms to implement RL. For each pair of a state and a possible next action, a Q-value is recorded (Q(S,A)), which determines how good the action is for that state. All the Q-values are stored in a data structure that is known as the Q-table [66]. As the agent learns by executing actions, the Q-values are updated based on the Bellman Equation (2.1). In the equation, α is the learning rate, R(S,A) is the reward for the pair state-action, γ is the discount rate and MaxQ'(S',A') is the maximum expected future reward.

$$NewQ(S,A) = Q(S,A) + \alpha \left[R(S,A) + \gamma MaxQ'(S',A') - Q(S,A) \right] \tag{2.1}$$

Q-Learning is an algorithm very simple to understand and implement that is capable of, in the right circumstances, reaching an optimal solution. It is model-free, meaning that it does not require a model of the environment to learn. Its simplicity is also a disadvantage, as one of the major limitations it has is that it can only work with discrete and finite state and action spaces.

2.3.2 Deep Q-Network

Deep Q-Networks (DQN) are essentially an improvement on Q-Learning by taking advantage of Neural Networks NN. Instead of creating the Q-table, DQN use a NN to store the data, where states are mapped to pairs of action and Q-value. Depending on the environment, the NN has to be shaped in a way that allows the state to be used as the input and each output node represents an action. Similar to Q-Learning, the Q-values on the NN are updated in training by using the Bellman Equation (2.1) [35].

Unlike Q-Learning, DQNs allow for the input to contain continuous values. In simpler scenarios, training a NN might not be more efficient than using a Q-table but, the more complex the problem is, the more using continuous values can simplify the environment state, allowing DQNs to converge faster. Despite this improvement, the output space still only allows for discrete values. Another problem for DQNs is their difficulty to train for planning for temporally extended goals [4], only focusing on going from an initial state to a desired final state.

2.3.3 State–action–reward–state–action

When Q-learning calculates a new Q-value, it takes into consideration the action taken, the reward, the initial and final states, and the best possible action that can be taken in the new state. The State–action–reward–state–action (SARSA) is different because it considers all the possible actions that can be performed in the new state by modifying the Bellman Equation (2.2). By using a random instead of the best Q-value of the new state, the algorithm will become more conservative while training, by taking paths that are less prone to cause negative rewards [49].

$$NewQ(S,A) = Q(S,A) + \alpha \left[R(S,A) + \gamma Q'(S',A') - Q(S,A) \right] \tag{2.2}$$

When there is a large negative reward near the best possible path, the fact that it takes more risks means that Q-Learning (optimal policy) will reach the optimal solution faster. On the other hand, SARSA (near-optimal policy) is more capable of avoiding large penalties, making it a better alternative if those mistakes can cause issues (ex: damaging a real robot). Another advantage with SARSA is that it is usually more stable than Q-Learning and often has an easier time converging.

2.3.4 Proximal Policy Optimization

The Proximal Policy Optimization (PPO) is based on Policy Gradient methods. Instead of recording Q-Values, Policy Gradient methods try to optimize parameterized policies related to the expected return by gradient descent. The issue with this method is its high sensitivity to hyper-parameter tuning. PPO attempts to resolve this problem by striking a balance between ease of implementation, sample complexity, and ease of tuning, trying to compute an update at each step that minimizes the cost function while ensuring the deviation from the previous policy is relatively small [2]. The general idea is to avoid the current policy changing too much compared to the old one [52].

The main advantage it has compared to previous Policy Gradient methods is the improvement in the stability of the policy during training. PPO is one of the most effective RL algorithms to date but several other Policy Gradient algorithms have been developed such as the Deep Deterministic Policy Gradient (DDPG) [29] or the Trust Region Policy Optimization (TRPO) [51].

2.3.5 Reinforcement Learning Frameworks

When it comes to tools for AI, Python is the programming language that offers more options. For RL in particular, the Gym[3] package offers a simple interface capable of representing environments to be used by RL agents. This package can be used in combination with other packages that provide the algorithms. Some of those packages include the Keras-RL[4] or the Stable-Baselines3 [5] that provide several of the most popular RL algorithms such as DQN, PPO or DDPG.

2.4 Path Planning using Reinforcement Learning

There is already a considerable body of research on the use of RL for mobile robot navigation, in order to get the agent to move from a source to a target position in an environment with obstacles [58, 68, 71, 40, 8, 48, 62, 63, 69, 75]. The most common RL algorithms used are DRL techniques, including DQN [68, 71, 48, 63, 69, 75], SAC [8, 69], PPO [40] and DDPG [58].

For the RL algorithms' state space, many approaches use the readings from a Light Detection and Ranging (LiDAR) sensor [58, 48, 62, 63]. These approaches do not use all the rays from the LiDAR to avoid overly large state spaces (which slow down the training process), but instead

[2]`https://openai.com/blog/openai-baselines-ppo/`
[3]`https://www.gymlibrary.dev/`
[4]`https://keras-rl.readthedocs.io/en/latest/`
[5]`https://stable-baselines3.readthedocs.io/en/master/`

use only a subset of the rays (typically 9 or 10 rays), equally sampled over the robot's frontal 180º angular range. Examples of other data used for the state space include images captured by a visual sensor [68, 71] and the relative distance and/or angle between the robot to the target position [58, 48, 63]. Rather than using raw sensor values for the input, Pan *et al.* [40] first convert the robot's laser data to a local grid map.

The reward function in the related work rewards the robot reaching the destination and penalizes colliding with obstacles [58, 68, 71, 40, 8, 48, 62, 69, 75]. Many functions also take into account the robot's distance and/or angle to the destination [58, 71, 48, 63, 69], and some penalize being close to obstacles [8, 75] or taking too much time to reach the destination [40].

For training, most of the approaches use a simulator, including Gazebo [63, 48, 71] and CoppeliaSim (and its predecessor, V-REP) [58, 8]. Most approaches either use a 3D environment [58, 68, 71, 8, 63, 48] or a 2D grid [69, 75]. The former simulation approaches have the advantage of more closely emulating a physical robot's environment, while the latter ones are more computationally efficient.

2.5 Path Planning with Intelligent Wheelchairs

As was already mentioned in Section 1.3, despite the kit not containing a real robot, one of the concerns was to retain some considerations for real-life scenarios. One of the scenarios that were explored during some of the initial work is the concept of IW. It is interesting to have some proximity to the Intellwheels2 funded project. As solving accessibility issues becomes more prevalent in modern society, the development of IW has seen a lot of recent improvements [15].

Since IW and path planning using RL are usually connected, this topic presents some research in common with Section 2.4. Similarly, RL [67, 30, 14, 48], namely DRL [14, 48] are very common approaches. In contrast, there are also approaches without Machine Learning that are based on path planning optimization algorithms, namely A* [50, 24] and Ant Colony Optimization [53].

The navigation algorithms for IW in the literature rely on various types of input. One typical source of information is visual sensors, namely LiDAR [30, 53, 48, 20], Infrared proximity sensors [42] and 3D scanners [54]. Some approaches use an Inertial Measurement Unit (IMU) sensor to measure the chair's acceleration and velocity [36, 30, 54]. Maekawa *et al.* use eye-tracking glasses to acquire eye-gaze data from the wheelchair. Some approaches also use pre-planning algorithms, namely Simultaneous Localization and Mapping (SLAM), to describe nearby obstacles [36, 20].

Simulations are also a common tool used in this problem, either for helping to train a RL agent [14, 48] or to assess a path planning algorithm [50, 53]. Several 3D simulators have been used, including Gazebo [53, 48], CoppeliaSim [50] and CARLA [14].

2.6 Teaching robotics with ROS

ROS is the most prominent SDK used for robotics applications. Its success is the result of multiple factors, among them being open source, working very similarly for both real robots and simulations, having a large community of users to provide help and already being used in a lot of the robotic industry [6] [38]. The reason for the name ROS is the fact that it is based on a concept of an Operating System (OS). It works with several processes called nodes that operate mostly independently of each other. Nodes communicate with each other via different types of messages, such as topics or services, through the publisher/subscriber model. The independence given to each one allows nodes that rely on completely distinct strategies or are written in different programming languages to coexist. Currently ROS offers the most support for Python and C++. This architecture allows for a framework that encourages adaptable and reusable code across all Robotics areas [46, 38].

2.6.1 Transition from ROS 1 to ROS 2

In 2007, ROS was first idealized by the Stanford Artificial Intelligence Lab. It was adopted by Google and was supported by the Willow Garage robotics research lab between 2008 and 2013. After that, the project was taken over by Google's OpenSource Robotics Foundation [38] and became the most prominent robotics framework to date.

The first version of ROS, also known as ROS 1, has released several distributions to the public over the years, with the first one being released in 2010. The latest distribution is ROS Noetic, released in 2020, and is going to have its support terminated in 2025. No other new distributions for ROS1 will be released which means it is approaching its End-of-Life [7].

Despite many advantages, ROS 1 lacked some functionalities that needed to be addressed. To solve these issues, ROS 2 was developed and its first non-Beta distribution was released in 2017 [8]. One of the core ideas when creating ROS 2 was that no functionalities were lost in the process. The plan was to only add features that would widen the range of possibilities and compatibility. One of the most important changes was the added support for most common OS, like Windows or MacOS, since ROS 1 was conceived for Linux-based applications. ROS 2 is built with a Data Distribution Service (DDS) as the base for the communication between its nodes. This means that it is now suitable for Real-Time distributed embedded systems thanks to its various transport configurations and scalability [31]. Security features in ROS are also becoming more of a concern as Robotics becomes a more mainstream field. The usage of a DDS also contributes to more secure message protocols between nodes [12]. Along with the reformulated base architecture, a multitude of other changes have also been included, among them new dependencies for C++ and Python, a new threading model and a more complete form to develop launch files [9].

[6] https://www.ros.org/blog/why-ros/
[7] http://wiki.ros.org/Distributions
[8] https://docs.ros.org/en/foxy/Releases.html
[9] http://design.ros2.org/articles/changes.html

2.6.2 Learning ROS

Due to its relevance, over the years, ROS has had a lot of courses, books and tutorials that target its teaching. For ROS 1, the official documentation [10] offers an introduction to all the components offered by the framework as well as a lot of tutorials that mainly target its usage paired with the Gazebo Simulator. Several books have been written about ROS 1 [38, 39]. Although the books end up being written for a distribution that eventually loses support, the core concepts of ROS remain the same, which means that the materials provided maintain their relevance.

ROS 2 on the other hand, partly due to how recent it still is, suffers from a shortage of materials to aid its learning. A dedicated documentation is available [11] and, much like the one for ROS 1, provides both descriptions of its tools and practical tutorials. Some available material also offers a comparison between ROS 1 and 2, providing a valuable tool for users to transition [56].

2.7 Teaching Robotics with a Simulation

Developing and testing real robots requires a lot of resources, making simulations is a key part of Robotics. A lot of different scenarios have emerged where simulators are a necessary stepping stone to develop robots. Even in a situation where the goal is to build a real robot, starting with a simulation provides a safe, low cost and controllable environment for testing and verification purposes. The ability to be accelerated and easily replicated allows for several setups to be experimented with [10]. In educational scenarios, the high costs of real robots often inhibit their usage. Simulations enable students to engage with robotics in a quick and easy manner and, in situations where a physical robot is not accessible, have proven to be a very valuable teaching tool [60, 18].

Some simulators were built to integrate with ROS, directly interacting with the node system. For 3D simulation, the most popular of these is Gazebo [12] while Flatland [13] is currently the best alternative for a simpler 2D simulation. Other popular simulators were not originally built to be used with ROS but adaptations were later developed to make the integration, such as MuJoCo [59] [14].

Outside of the ROS scope, there are a lot of other options to use in Robotics simulations. Some of them focus only on teaching young robotics students, such as the case of the Tactode robotic simulator [37]. Despite being always usable for educational purposes, most simulations also target more professional applications. Simulators such as Webots [33] or RoboDK [16] are very targeted to industrial settings. In some cases, simulations were created to fit physical robot platforms that already exist on the market, creating virtual environments for platforms like Duckietown or LEGO Mindstorms [21, 9].

For Robot Learning, a lot of simulations have the capability of being accelerated, which can drastically improve the learning times for RL algorithms [25]. But using simulators to train robots

[10] http://wiki.ros.org/Documentation
[11] https://docs.ros.org/en/foxy/index.html
[12] https://gazebosim.org/home
[13] https://flatland-simulator.readthedocs.io/en/latest/
[14] https://github.com/shadow-robot/mujoco_ros_pkgs

presents a major problem: the gap between the simulation environment to reality means that the knowledge is not necessarily transferable. This means that is usually not reliable to train a physical robot using simulations as policies may perform much worse in the real scenario [74].

2.8 Teaching Robotics with a Physical Robot

As the ultimate goal of Robotics is to create physical machines that can autonomously perform useful tasks for humans, the transition from simulation to real robots is a topic that has to be addressed when talking about Educational Robotics. As real robots tend to be very expensive, the usual answer is to develop robots with the sole purpose of being used in teaching scenarios. By building robots with simpler architectures and components, the costs of producing it can be reduced while creating a physical robot developing platform that still uses fundamental principles that need to be learned by students. Although the kit that was developed does not feature a physical robot platform, that inclusion could be made in the future.

Ventuzelos *et al.* [62] presented a kit to teach Robotics that includes a simple robot. The robot in question is a two-wheeled ground robot containing an Arduino micro-controller to control the motors, a wide lens camera as its sensor and a Raspberry Pi single board computer as its brain. The kit also contains an equivalent simulation and tutorials on both ROS 1 and 2 on how to control the robot using RL.

Other educational physical robot platforms follow a similar approach: small two-wheeled differential drive robots with some simple sensors. One example is AlphaBot2 which has several variations and can be controlled by either an Arduino or a Raspberry Pi. It can also include some common robot functionalities including line tracking or obstacle avoiding but it does not provide a simulation [32] or support for ROS. Another available platform that follows this pattern is PyBoKids. Its approach is very similar to AlphaBot2, having several versions as well but with more robust bodies. Another important distinction is that it has an accompanying simulation for Gazebo [61].

A slightly different approach to providing physical robots for educational purposes is the Duckietown project that started as a class at MIT. The base idea is to also develop simple robots, controlled either by a Raspberry Pi or a Jetson Nano, but it goes further than that by also providing complete kits with simple physical maps, with small roads and traffic lights, that the robots can use as their environment [45]. Despite its simplicity, the kits can become very expensive as they are meant to be used by an entire class of students.

A more ambitious project is the EUROPA Robot that, besides the differential drive also has a robotic arm mounted on it. The robot is controlled by a Raspberry Pi and also offers its simulation for Gazebo [22]. The robot is a lot more complex than other options, which means that it might not provide the best option to introduce students to Robotics.

Another different type of product that is available on the market, instead of providing the complete robot, gives the user modular components that allow the robot to be assembled in more than one way and complete different tasks. The assembling process usually does not require any

Table 2.1: Robot Learning query

Database	**Query**	**No. results**
Engineering Village	(((educat* OR pedagogic* OR student*) WN KY) AND ((robot*) WN KY))	9,707
Scopus	(TITLE-ABS-KEY(educat* OR pedagogic* OR student*) AND TITLE-ABS-KEY(robot*))	26,068
IEEE Xplore	("All Metadata":educat* OR "All Metadata": pedagogic* OR "All Metadata": student*) AND ("All Metadata":robot*)	21,195

soldering or other types of dangerous machinery, being adequate for children. One of the most popular product lines that uses this approach is the LEGO Mindstorms [23] but more products are available with similar characteristics, such as the Makeblock's mBot [15] or the IQ-KEY Perfect 1000 [16]. These types of robots tend to have much higher price points than simpler platforms.

2.9 Teaching Robot Learning with ROS

There is a lot of material available on the topic of Educational Robotics but a lot of it does not focus on the full scope that the kit to be developed will cover. Therefore, to effectively get a clear idea of similar previous work done, a more deterministic process to gather information on relevant projects. This process consisted of querying some of the most popular scientific databases for papers on the topic and then manually determining which ones are relevant. The initial query used to search in the title, abstract and keywords for Educational Robotics terms (Equation 2.3) and the results are presented in Table 2.1.

$$(educat* \ OR \ pedagogic* \ OR \ student*) \ AND \ robot* \tag{2.3}$$

As the number of results returned by the query was too high to perform a manual search, another condition was added to only show results that also contain RL in the search fields (Equation 2.4). Table 2.2 shows the results of this query.

$$(educat* \ OR \ pedagogic* \ OR \ student*) \ AND \ robot* \ AND \ ''reinforcement \ learning'' \tag{2.4}$$

For a result to be considered relevant, it has to present a novel way to teach Robotics paired with RL. Only projects with a physical robot or a simulation framework were considered. As the number of results was still very large, only the 50 first were manually reviewed. Table 2.3 shows all the results found and the technologies they use, along with some other entries already discussed in this Chapter.

[15] https://store.makeblock.com/products/diy-coding-robot-kits-ultimate
[16] http://www.iq-key.com/

Table 2.2: Robot Learning with RL query

Database	Query	No. results	Relevant results
Engineering Village	((((educat* OR pedagogic* OR student*) WN KY) AND ((robot*) WN KY)) AND (("reinforcement learning") WN KY))	87	4 [17, 19, 73, 76]
Scopus	(TITLE-ABS-KEY(educat* OR pedagogic* OR student*) AND TITLE-ABS-KEY(robot*) AND TITLE-ABS-KEY("reinforcement learning"))	701	2 [13, 62]
IEEE Xplore	("All Metadata":educat* OR "All Metadata": pedagogic* OR "All Metadata": student*) AND ("All Metadata":robot*) AND ("All Metadata":"reinforcement learning")	453	5 [17, 19, 26, 43, 57]

As they are more scarce than simulations, many of these projects attempt to create an affordable physical robot platform for new robotics students. This means that in many scenarios the simulation is not a concern. The problem is that the transition from simulation to reality is also a crucial step in the Robotics Industry as most projects start as virtual environments. Direct integration with ROS is usually not a priority and specifically ROS 2 is rarely utilized in Educational Robotics resources.

2.10 Summary

This chapter was dedicated to performing a literature review. It started by explaining the state of the Robot Learning field and the crucial role RL plays in it. A brief exploration of the most important RL algorithms followed. A short study on how path planning can be done using the RL approach was conducted, along with how the real-life scenario of IW can be integrated as well. It was also demonstrated how ROS, Robotic Simulations and Real Robots are typically approached in learning scenarios. Finally, an extensive survey was made on the topic of Teaching Robot Learning. This study showed that Robot Learning has a lot of educational materials that have been developed over the existence of the field. There is, however, a lack of a kit that unifies all the current relevant tools in the Robotics Industry.

Table 2.3: Related work on teaching RL with Robotics

Paper	**Simulation**	**Physical Robot**	**ROS version**	**RL algorithms**
Patil *et al.*, 2022 [43]	-	DeltaZ (low cost, simple robot)	-	Episodic Relative Entropy Policy Search (eREPS)
Ventuzelos *et al.*, 2022 [62]	Flatland Simulator	Two-wheeled ground robot	ROS 1/2	PPO
Zhang *et al.*, 2022 [73]	-	LEGO SPIKE Prime	-	Q-learning
Dreveck *et al.*, 2021 [13]	AWS DeepRacer console	AWS DeepRacer	-	PPO, Soft Actor Critic (SAC)
Giernacki *et al.*, 2020 [17]	Gazebo	Bebop 2	ROS1	-
Suenaga *et al.*, 2020 [57]	Fabot2D	Robot controlled in web-browser	ROS with Rowma	Double DQN
Haak *et al.*, 2019 [21]	Creating simulation in progress	LEGO Mindstorms EV3	-	-
Martínez-Tenor *et al.*, 2019 [76]	-	LEGO Mindstorms	-	Q-learning, SARSA
Chevalier-Boisvert *et al.*, 2018 [9]	Simulator written with Python and OpenGL	Duckietown Robots	-	DDPG
Newman, 2017 [38]	Examples for Gazebo	-	ROS1 concepts and structure	-
Goodspeed *et al.*, 2007 [19]	-	Sony AIBO	-	Matlab-based RL library for the robot
Lalonde *et al.*, 2006 [26]	-	Nomad Scout robot mounted with a Dell laptop	-	Q-learning

Chapter 3

Using Deep Reinforcement Learning for Navigation in Simulated Hallways

As previously mentioned in Section 1.3, to make sure that a RL system that was capable of training a robot to perform a path planning task was presented in the kit, some prior testing was required. This motivated the realization of an initial work that targeted using a very similar setup to the one that was planned to be used in the tutorials. One of the main constraints was that, at the time this work was done, Flatland did not have a working version that was compatible with ROS 2 so ROS 1 was used. Since the main objective was to test the RL components of the framework, there would be no major differences in using ROS 1. This section will present the first part of that work.

The goal of this first part was to explore using DQN [35] for navigation in a simulated 2D environment with a simple differential drive robot, equipped with a LiDAR sensor. The aim is to explore the specific advantages and challenges of applying DRL in this context, as there are already several classic navigation algorithms to solve this problem [34]. Several maps, resembling hallways, were used, with varying configurations and obstacles, including doors or turns. The robot's objective is to find a path from one end of the hallway to the opposite end.

3.1 Methodological Approach

the project consists of recreating the proposed scenario in a simulated environment and applying reinforcement learning in that environment. This section describes the methodological approach taken to achieve this goal.

3.1.1 Simulation

The simulation is built using Flatland and ROS 1. The simulation can be accelerated, allowing the tests to run at 10 times the normal speed. Figure 3.1 shows the most basic setup explored and this section will describe each of the simulation components that can be seen in it.

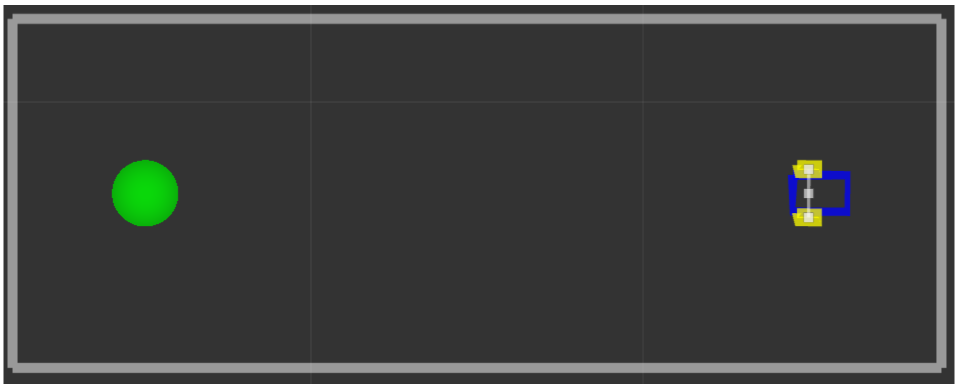


Figure 3.1: Straight hallway in the simulator with one robot and target beacon

3.1.1.1 Robot

In the simulation, the robot is represented by a simple model with a differential drive plugin. This robot can be seen at the bottom of Figure 3.1. The state of the robot is represented by the position in two axes and the rotations (Equation 3.1).

$$robot\_state = (x, y, \theta) \tag{3.1}$$

The differential drive plugin in Flatland allows the robot to have independent linear and angular velocities, allowing for three types of trajectories:

- Translation: moving either forwards or backwards in the direction it is facing.
- Rotation: rotate in the same place in either direction.
- Translation and Rotation: combining both to give the robot linear and angular velocity simultaneously.

The robot is also equipped with a LiDAR sensor that collects information from 90 rays spread evenly all around the model and bumpers to detect collisions.

3.1.1.2 Target Beacon

To detect that the robot has reached the end, there is a very simple model at the target position represented by the green circle (target beacon) in Figure 3.1. It is a basic static robot fitted with a LiDAR and which periodically checks the closest reading. By using a feature from Flatland called ‘layers’, the LiDAR only detects the differential drive robot and, if the lowest reading reaches below a certain threshold, then it signals that the robot has reached the target.

3.1.1.3 Maps

Figure 3.1 represents a simple empty hallway. New maps were created by adding turns, doors and obstacles. The resulting maps can be seen in Figure 3.2 and provide several degrees of complexity as well as different challenges to reach the end. These new maps can be divided into four groups:

- Group 1 - Doors: Top row; second, third and fourth maps in Figure 3.2.
- Group 2 - Obstacles: Top row; fifth, sixth and seventh maps in Figure 3.2.
- Group 3 - 90º turns (left or right): Bottom row; first and second maps in Figure 3.2.
- Group 4 - 180º turns (left or right): Bottom row; third and fourth maps in Figure 3.2.

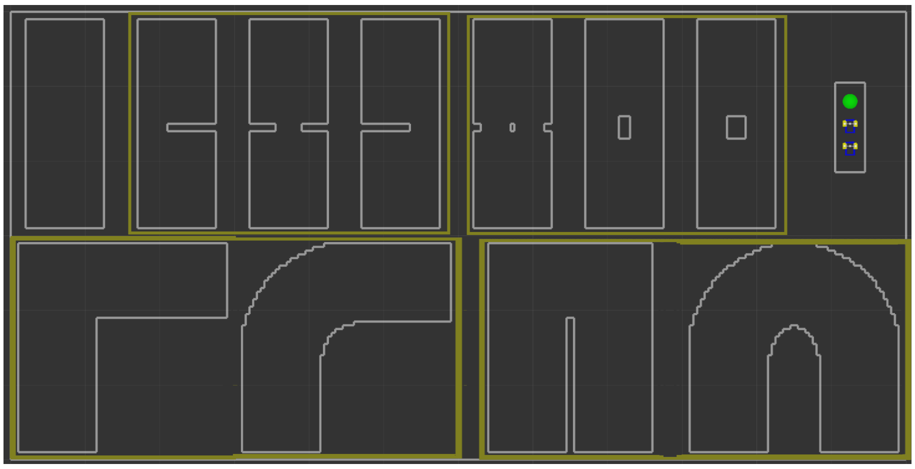


Figure 3.2: All maps used for training and testing

When creating the maps, some important details were taken into account. For Group 2, the obstacles are set in a way that, if the robot is learning all the maps at the same time, it cannot simply learn a single path, as a wider trajectory will not work on the first map and a more centered one will collide in the third. It is important to note that any of the maps with turns can either be a left or right turn depending on the start and end positions.

3.1.2 Reinforcement Learning Algorithm

The RL algorithm used to train the robot was the DQN provided by OpenAI gym [6]. The model used for the DQN was developed using Keras and TensorFlow and the training agent was created using the Keras-RL package[1]. The learning rate was 0.01 and the setup of the model and agent was the same for all tests. The action selection policy was the Boltzmann policy, with an initial temperature value, τ, of 1.0. Table 3.1 presents the architecture of the DQN used, where the input is passed through a set of layers, sequentially.

[1]https://github.com/keras-rl/keras-rl

Table 3.1: Architecture of the DQN

Layer	Input dimensions	Output dimensions	Activation function
Dense	17	24	ReLU
Dense	24	24	ReLU
Dense	24	7	Linear

3.1.2.1 Input Space

The state space for the algorithm is retrieved by the LiDAR mounted on the robot. Since the sensor contains 90 rays, if the algorithm were to use all of them, then the input space would be too large and it could take too long to learn. Therefore, the LiDAR's values are sampled as shown in Figure 3.3. The LiDAR's readings are divided into sections and, for each section, only the lowest value is stored, which indicates the closest obstacle in that direction. As the robot is supposed to move forward, the front of the sensor has a lower sampling ratio and is divided into 9 sections (yellow and orange in Figure 3.3), while the back is divided into 3 equal parts (blue sections in Figure 3.3).

For additional information, the state space also includes values from 3 rays in the front and 2 directly to the sides (red lines in Figure 3.3). Therefore, the input space has a size of 17 (Equation 3.2).

$$state\_space(17) = closest\_in\_region(12) + extra\_rays(5) \tag{3.2}$$

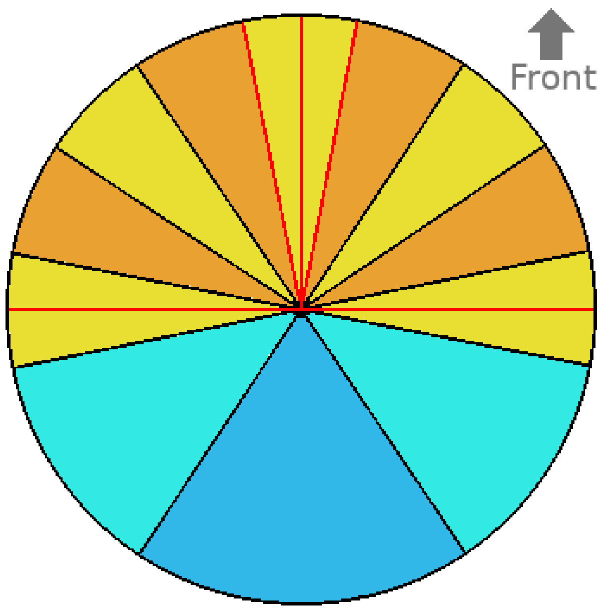


Figure 3.3: Visual representation of the LiDAR sampling strategy

3.1.2.2 Action Space

For the output, the RL algorithm can choose one of seven actions for the robot presented in Table 3.2. A larger action space was not used to avoid having to use a larger DQN, which would require more training time.

Table 3.2: Robot actions

Action	**Linear velocity (m/s)**	**Angular velocity (rad/s)**
Stop	0	0
Move forward	0.3	0
Move backwards	-0.3	0
Rotate left	0	1.05
Rotate right	0	-1.05
Move forward and left	0.3	1.05
Move forward and right	0.3	-1.05

3.1.2.3 Reward Function

The reward function created for this system is presented in Algorithm 1. The function was developed with the main goal of directing the robot to the target. A secondary goal was to complete the task as fast as possible, accomplished by trying to choose the shortest path. To make this possible, the following rules were applied:

- Give a large negative reward for any collisions (line 3).
- Give a large positive reward for reaching the target. The reward reflects how fast the robot reached the target (line 5).
- Give a large negative reward for exceeding the maximum time to reach the goal (line 7).
- Give a positive reward to actions that help the robot progress on the map. A larger reward for moving forward (line 9) and a smaller one for moving forward and rotating (line 12). These rewards are accumulated in a value to be given as a negative reward if the robot exceeds the time to reach the target to avoid the agent exploiting it without completing the main objective.
- Give a negative reward to actions that do not contribute to the robot's progression and that should only be used as a last resort: stopping and moving backwards (line 15).
- For rotation-only movements, give a positive reward if the robot had no space to move forward (line 20) and negative otherwise (line 18).
- Give a reward based on the space the robot detects it has directly in front of it, to discourage being too close to a wall right ahead (line 30).

- Give a negative reward for redundant pairs of consecutive actions (line 26).
- Give a negative reward for consecutive full rotations to the same side (line 28). As the robot uses angular velocity, the reward function keeps track of how much it has rotated and can detect when it makes a full 360º turn. The more full turns it does consecutively to the same side, the larger the penalty. This aims to help to prevent the robot from getting stuck in a section of a map.

3.1.3 Environment

The Flatland Simulator and the DQN are two separate layers in the system that work in parallel. A third layer exists to coordinate the actions and states used by the DQN with the simulation. This layer uses both ROS Topics and Services to control the robots in the simulation and the Gym Python package to translate the problem to an environment that can be used by a DQN agent. Figure 3.4 is a schematic representation of how the environment layer is used to connect the entire system.

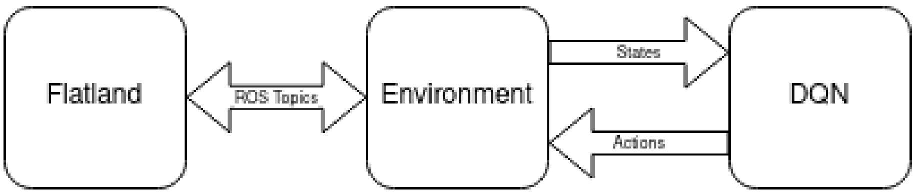


Figure 3.4: Visualization of how the system is organized

The environment defines the notion of step which consists of processing an action selected by the agent for the current state. The steps of that process are:

1. Execute the selected action in the simulation by changing the speed of the robot accordingly.
2. Wait for the action to be complete. Each action executes for 0.2 seconds.
3. Read values from the simulation and update the input space of the DQN accordingly.
4. Compute the reward for the action.
5. Return the values calculated so that the agent can update the DQN's weights and select a new action.

Another important concept of the environment is the episode. An episode is the set of sequential steps where the environment goes from an initial to a final state. There are three types of final states:

- Collisions: a wheelchair's bumpers detect a collision and signal the environment layer.
- Reaching the target position: the environment constantly receives and analyses the readings from the target beacon (Section 3.1.1.2) to determine if the goal was reached.

Algorithm 1 Reward function algorithm

```
Input:
    agent - RL agent
    action - Last action performed
    n_steps - Current step
    max_steps - Number of steps in an episode
1:  $basic\_reward \leftarrow \frac{300}{max\_steps}$
2:  if $collided()$ then
3:      $reward \leftarrow -200$
4:  else if $reachedDestination()$ then
5:      $reward \leftarrow 400 + max\_steps - n\_steps$
6:  else if $n\_steps > max\_steps$ then
7:      $reward \leftarrow -(300 + forward\_reward)$
8:  else if $action = FORWARD$ then
9:      $reward \leftarrow basic\_reward$
10:     $forward\_reward \leftarrow forward\_reward + reward$
11: else if $action \in \{FORWARD\_LEFT, FORWARD\_RIGHT\}$ then
12:     $reward \leftarrow 0.5 * basic\_reward$
13:     $forward\_reward \leftarrow forward\_reward + reward$
14: else if $action \in \{STOP, BACK\}$ then
15:     $reward \leftarrow -5 * basic\_reward$
16: else if $action \in \{ROT\_LEFT, ROT\_RIGHT\}$ then
17:     if $forward\_laser\_dist > 0.1$ then
18:         $reward \leftarrow -2 * basic\_reward$
19:     else
20:         $reward \leftarrow 0.5 * basic\_reward$
21: else
22:     $reward \leftarrow 0$
23: if $n\_steps > 0$ then
24:     $prev\_acts \leftarrow [previous\_action, action]$
25:     if $(prev\_acts = [FORWARD, BACK]) \lor (prev\_acts = [BACK, FORWARD]) \lor (prev\_acts = [ROT\_LEFT, ROT\_RIGHT]) \lor (prev\_acts = [ROT\_RIGHT, ROT\_LEFT])$ then
26:         $reward \leftarrow reward - 5 * basic\_reward$
27: $consec\_rot \leftarrow getFullConsecRotations()$
28: $reward \leftarrow reward - 75 * consec\_rot$
29: $forw\_laser \leftarrow forward\_laser\_dist - 0.5$
30: $reward \leftarrow reward + 2 * forw\_laser * basic\_reward$
31: if $forward\_laser\_dist > 0.5$ then
32:     $reward \leftarrow reward + ((forward\_left\_laser\_dist - 0.5) * (forward\_right\_laser\_dist - 0.5)) * basic\_reward$
33: return reward
```

- Time out: each episode has a maximum of 200 steps or 40 seconds. If no other state is met before, it ends when this time is exceeded.

If, at any step of an episode, one of these conditions is true, then the environment stops and re-positions the robot at the start and signals the DQN agent to start a new episode. To allow the robot to train and be tested in different scenarios complexity, when putting the robot back in a starting position, a small random shift in position and a random rotation is applied. Moreover, a small random shift is also applied to the target. The shifts are small to ensure that the robot has to traverse most of the hallway in each episode, while the rotations force the robot to orient itself correctly to be able to reach the end.

3.1.4 Training Strategies

The environment layer also defines how the agent is trained. The Keras-RL package provides functions to both test and train the agent. With the use of these functions, two different strategies were developed for training. The first strategy simply trains the agent for a given number of steps and then tests it for 100 episodes, returning an accuracy. The accuracy corresponds to a performance indicator for the proportion of episodes it reached the target position.

The second strategy involves the following steps:

1. Training the agent for a given number of steps (in all the tests, 10000 steps were used).
2. Test the agent after training for 100 episodes and determine the accuracy.
3. If a predetermined threshold for the accuracy is reached, the algorithm was successful and the weights are stored. If the accuracy is still too low, go back to step 1.

This second method was the one used to do most of the experiments as the first one revealed a major problem that will be discussed in section 3.2.1.

3.1.5 Model Classification

The model developed is mostly a descriptive model, as it aims to explore the usage of DQNs for path planning in Robotics and demonstrate the advantages and disadvantages of the approach. From the perspective of the Flatland Simulation layer, for each episode, the exogenous variables are:

- Map to be used (controllable)
- Starting position and rotation (controllable or uncontrollable, depending on the scenario)
- Action (one per step) (controllable)

The endogenous variables are:

- End condition of the episode

- Position and Rotation (at each step, only used as a metric for data analysis)
- LiDAR reading (one per step)

All the layers presented in Section 3.1.3 are Dynamic. The Flatland simulation is a Continuous model but, to be used in the DQN, it has to be transformed into a Discrete model with states and actions. In some of the scenarios, initial position and rotation have a random component, making the model Stochastic. Because there are no other random variables, in situations where the starting pose is constant, the model is Deterministic.

3.2 Results and Discussion

As a starting point, some experiments were done to help determine what was the most effective way to train the robot in the developed environment. Some important findings were made during these initial trials. Those conclusions were then used to improve the methodology for conducting more extensive tests.

3.2.1 Catastrophic Forgetting

In the scenario represented in Figure 3.1, the challenge is very simple: all the robot needs to do is to move forward and it will reach the target. Thus, it is expected that the DQN should be able to easily train the agent. However, some issues arose in this scenario.

Figures 3.5 and 3.6 depict the accuracy (percentage of episodes where the robot reached the goal) and average cumulative reward over the 100 test episodes with respect to the number of training sessions, each one composed of 10000 steps, for the straight hallway map.

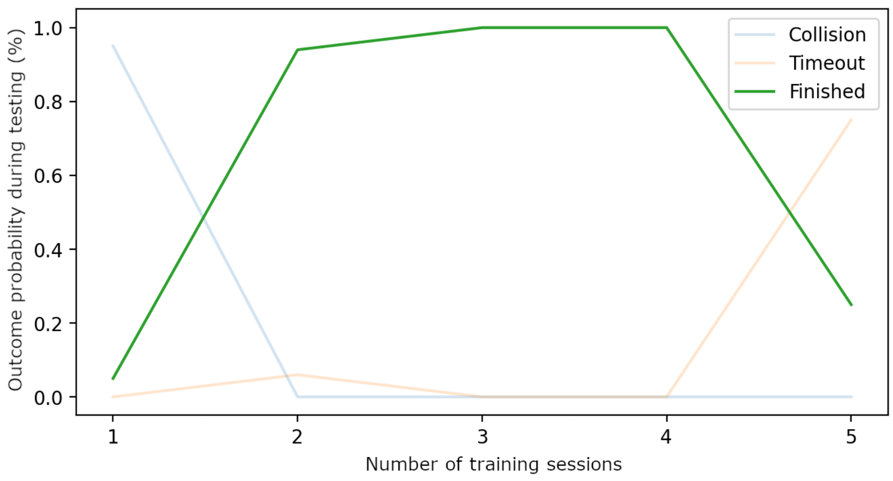


Figure 3.5: Straight hallway outcome probabilities of the testing episodes with respect to the number of training sessions

Figure 3.5 shows the accuracy reaching an almost perfect score with very few training sessions but, in the last session it dropped to below 0.3 (i.e. only around 30% of episodes led to the robot

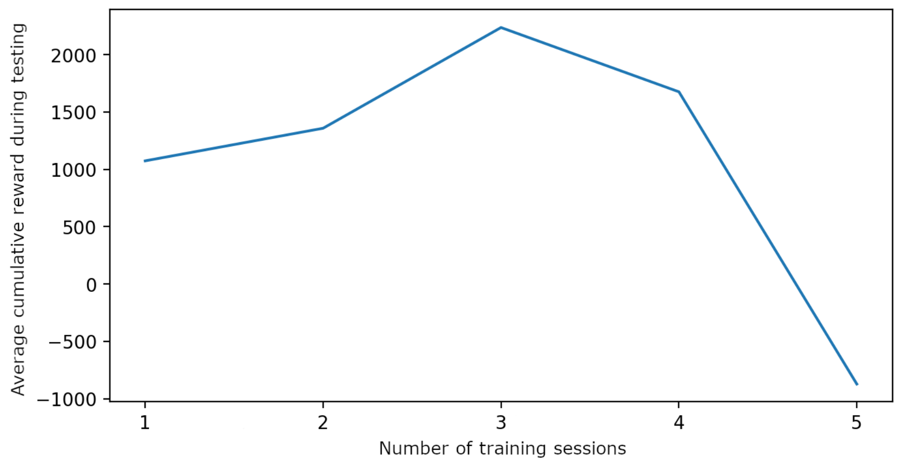


Figure 3.6: Straight hallway average cumulative reward of the testing episodes with respect to the number of training sessions

reaching the goal). In fact, after the fifth training session, the robot incurs a timeout in around 70% of the testing episodes. Moreover, in figure 3.6, the average cumulative reward drops significantly. The best explanation for these issues is a common phenomenon in RL called Catastrophic Forgetting, which occurs when the algorithm starts using a solution and small changes to incorporate new data end up affecting the entire network and disrupting the previously learned patterns, which are reflected on the network's weights [7].

This experiment demonstrated that the robot may run into problems if it is over-trained. To overcome this issue, the second method explained in Section 3.1.4, which trains the agent until it passes a certain threshold of accuracy, was used for the rest of the tests. This helps to ensure that the weights after training will be effective on, at least, completing the maps it was using.

3.2.2 Using Distance and Rotation to target

Another scenario that was tested involved adding information about the distance and the rotation from the robot to the target position, which was achieved using the ROS odometry topic. The distance and rotation were added to the state space, and the reward function included a positive reward if the robot was moving or rotating in the direction of the target, or a negative otherwise. To understand if adding this extra information had a positive impact on the agent's training, the robot was trained with all of the maps in the original setup and the one presented in this subsection until an accuracy of at least 0.8 is reached. Each scenario was attempted 3 times and the number of steps required to train was registered for each one. The results are displayed in Table 3.3.

The results show that using the distance and rotation to the target leads does not help to decrease the number of training steps needed to reach the accuracy threshold. Moreover, using this metric with a physical robot is not realistic as it would require using an expensive and accurate sensor, which may not be feasible. Therefore, this extra information was not used in the state space and reward function.

Table 3.3: Results from testing with or without distance and rotation to target

Attempt	**With dist and rot**		**Without dist and rot**	
	Train steps	**Acc**	**Train steps**	**Acc**
1	220000	0.800	590000	0.830
2	150000	0.810	140000	0.860
3	530000	0.800	110000	0.810
Mean	300000	0.803	280000	0.833

3.2.3 Group Tests

This section describes the most extensive tests done to this RL setup. The tests consist of training the robot with different sets of maps until an accuracy of 0.8 is reached and saving the obtained weights. Each map is then tested on 100 episodes for each different agent that was trained. Table 3.4 shows the steps needed and the accuracy obtained when training each group and Table 3.5 shows the accuracy when testing all the weights collected for each map.

Table 3.4: Training results for all groups

Group	**Train steps**	**Acc**
Doors	720000	0.8
Obstacles	270000	0.81
90º turns	50000	0.98
180º turns	130000	0.9
All maps	590000	0.83

From these results, several observations can be made:

- The robot very rarely used the stop action during testing, as it was able to learn during training that it prevented it from reaching the destination earlier, which is encouraged by the reward function. It is expected that the usage frequency of this action tends to 0 as more episodes are used for training.

- The robot occasionally used the ‘move backwards’ action during testing. The authors speculate this is due to the need to correct the robot’s path when it reaches a dead end, namely in maps with doors when the robot is too close to a wall adjacent to the door.

- The maps with the doors seem to be the most complex ones as the robot required a much larger amount of training steps to reach the accuracy threshold.

- There seems to be a correlation between better results and training for a longer time. For example, training only with the maps with doors has good results all around while training

Table 3.5: Testing accuracy for all groups

Tested map	**Group trained**				
	Doors	**Obst**	**90º turns**	**180º turns**	**All**
Straight	0.99	0.94	0.96	0.88	1.0
Left door	0.75	0.73	0.31	0.87	0.82
Center door	0.78	0.57	0.23	0.16	0.69
Right door	0.57	0.6	0.41	0.09	0.46
Two doors	0.34	0.88	0.54	0.44	0.85
Small obst	0.52	0.75	0.27	0.58	0.84
Large obst	0.54	0.7	0.13	0.42	0.65
Turn left(90º)	0.97	0.86	0.96	0.91	0.76
Turn right(90º)	0.99	0.15	0.99	0.90	0.97
Curve left(90º)	0.96	0.88	1.0	0.94	0.98
Curve right(90º)	0.94	0.93	0.98	0.85	0.95
Turn left(180º)	0.88	0.82	0.97	0.89	0.74
Turn right(180º)	0.96	0.04	0.99	0.87	0.81
Curve left(180º)	0.94	0.87	0.99	0.93	0.96
Curve right(180º)	0.82	0.39	0.93	0.93	0.9
Mean	0.797	0.674	0.711	0.711	0.825
Std Dev	0.202	0.271	0.334	0.284	0.144

the 90º turns, which only took 50000 steps, has a lower average and a much higher standard deviation.

- The tests with the weights trained for maps with obstacles show that the robot was able to complete turns to the left but not to the right. This illustrates that, just because the agent learns how to turn to one side, it does not mean that it is capable of turning to the other.
- The best performance (highest mean and lowest standard deviation) was obtained by training with all the maps, thus showing that there is always a lot of value in training with a larger variety of scenarios.
- Overall the results are very positive because the averages are very high and there are very few values below 0.5 in Table 3.5. This means that no matter what maps are used to train, the agent is always able to find patterns that are useful for most situations.

3.2.4 Training Patterns

To better understand the results from the group tests, it is also relevant to look at how the agent learns and how it improves the accuracy over training. The graph in Figure 3.7 depicts the accuracy obtained while training with all the maps at the same time. The figure shows a pattern that is very common while learning with this model: although the overall accuracy is increasing over time, the growth is not gradual as it presents several high and low points. This is consistent with what was described in Section 3.2.1, as the agent may learn a good solution and then drastically change its DQN weights.

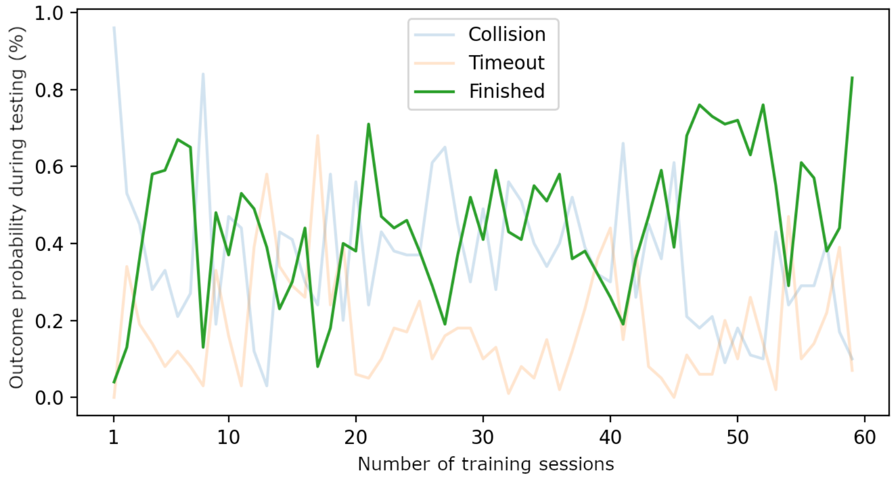


Figure 3.7: Outcome probabilities of the testing episodes while training for all maps until the accuracy threshold is reached

There is an exception to this: the learning progress of training maps with doors, as shown in Figure 3.8. Both this example and the one for all maps took a long time to train, but the evolution of the accuracy shows a much more gradual growth in this one. One hypothesis to explain this difference is the fact that, when training for all maps, the challenge comes from the variety of the states it needs to tune the weights to. As the agent tries to learn good results for one map, it might forget what it learned for others, thus explaining the sudden drops in accuracy. In the example with the doors, the challenge comes more from the precision required to pass through a door, so it is a lot harder to reach a good solution. As the agent trains the specific weights needed to complete the task, the accuracy increases gradually.

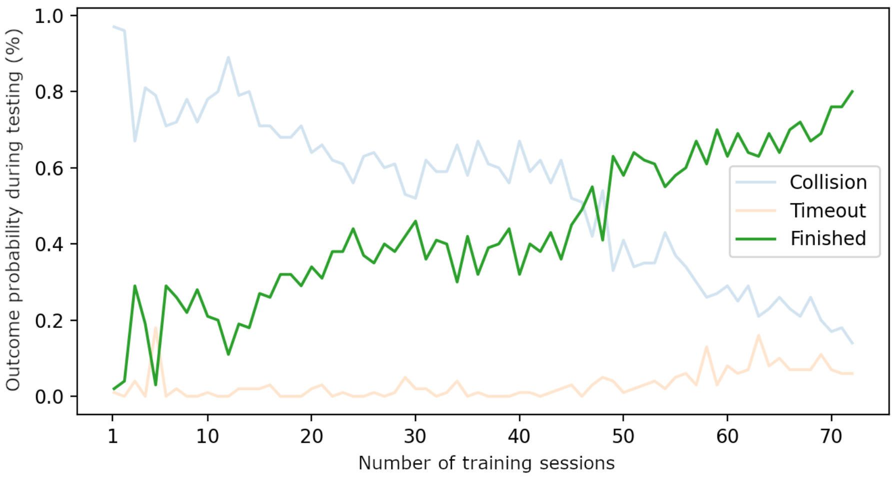


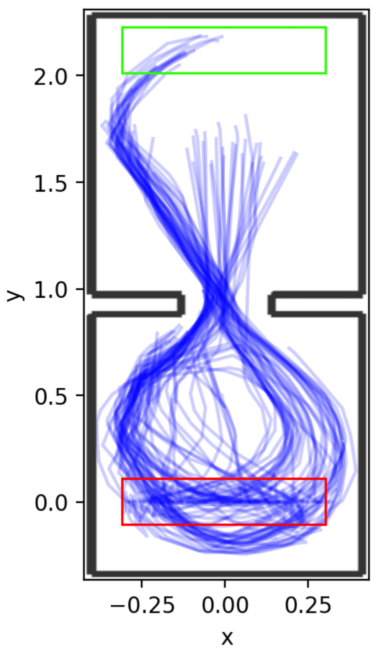
Figure 3.8: Outcome probabilities of the testing episodes while training for maps with doors until the accuracy threshold is reached

3.2.5 Wall Following Patterns

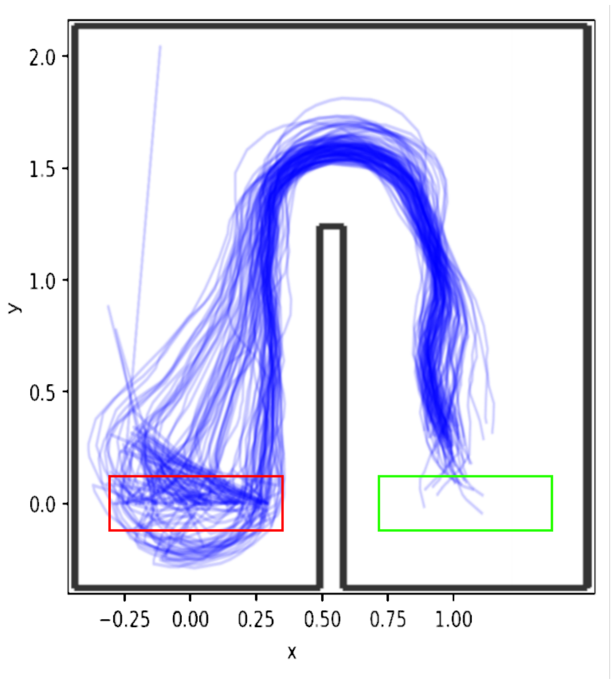
Figures 3.9a and 3.9b depict the paths followed by the robot in two of the maps, with the weights for the DQN obtained by training with all the maps. The red rectangle depicts the area where the robot is randomly placed at the beginning of an episode while the green rectangle depicts the area where the target may be randomly placed.

Despite not using any input or reward that would encourage the robot to do so, the agent seems to learn very often to choose paths that have it navigate near a wall. This is especially interesting if taking into consideration that moving close to a wall increases the risk of collision, which is associated with a negative reward. Figures 3.9a and 3.9b illustrate this phenomenon. The authors hypothesize that this wall-following strategy is coherent and gives the robot a higher chance of reaching the target regardless of the environment's layout since:

- The robot does not have any knowledge of where the target area is and thus needs to develop strategies to explore the map that have a high probability of reaching the goal in all scenarios.
- The target area is always at the end of the hallway, and therefore near a wall.



(a) Center door map



(b) 180º left curve map

Figure 3.9: Paths obtained during testing with weights from training all maps

3.3 Conclusions

The main contribution of this work is an illustration of the main advantages and challenges of using DRL for navigation in a simulated 2D environment. Regarding DRL's strengths, this line of work showed how it is possible for the agent to successfully navigate through a map it never interacted with by learning new behaviors (for instance, learning to follow walls). Regarding the main challenges, it was shown how Catastrophic Forgetting can cause issues when training agents as well as also showing a potential solution to overcome this issue of using a minimum threshold for the accuracy. In addition, it was shown that using some types of information in the agent's state may actually be counter-productive: the results showed that allowing the agent to know where the target position is relative to the robot does not bring any benefit to the performance, despite requiring a lot more complex sensors.

Overall, this first part of the initial work demonstrated that this RL framework can produce very interesting results to be used in a learning kit. It also provides an easy-to-understand structure that does not require a lot of knowledge of RL to start using and, for simpler tasks with an accelerated simulation, it can produce results reasonably quickly.

3.4 Summary

This chapter presented an initial work conducted to test the framework that will be used in the tutorials. It presented all the simulation components as well as the RL setup. It also provided results of various tests performed and concluded that the framework developed is suitable to be used in the learning kit.

Chapter 4

Navigation of Simulated Adjacent Wheelchairs using Deep Reinforcement Learning

The work presented in this section is the second part of the initial work and has a very similar framework and setup to the one presented in Chapter 3 but with some major changes in the scope and objectives it attempts to achieve. Even though the kit developed does not provide a physical robot, does not mean that it is inapplicable when studying real-life scenarios. If a scenario is appropriately depicted in a simulated environment, testing it there can be very valuable for an eventual transition into reality. For these reasons, one of the most important differences from the previous chapter is that the robots were re-imagined as IW to demonstrate how a real-life application can be studied in simulations.

Solving mobility limitations is a problem where Robotics has the potential of playing an important role [5]. For instance, in hospitals or nursing homes, wheelchairs are often used to move people around the facilities. In some cases, there is also a need for wheelchairs to travel together, which can bring problems in spaces that include obstacles or doors. Therefore, to improve the accessibility and independence of people with mobility limitations, wheelchairs can be enhanced with the capability of autonomous navigation.

This work aims to explore how IW can be trained to navigate autonomously using a DQN [35]. For a regular wheelchair to be able to acquire autonomous navigation, it needs to be equipped with additional components, such as electric motors to power the wheels (actuators), sensors to perceive the environment and a connection with the computer running the algorithm. Additionally to the costs and effort required by these tools, RL algorithms also tend to take a long time to learn and tune the parameters. For these reasons, it is very common that studies done on the navigation of IW are based on real scenarios but implemented and tested using simulations.

In the scenario presented in Chapter 3, a single simulated robot was successfully trained to traverse hallways using DRL. Given these positive results, its DQN configuration, the input and action spaces, the reward function and the simulation environment served as the basis for this work. Despite that, the whole system had to undergo major changes to accommodate the usage of two robots and an entirely new set of results was produced. Another key difference from previous work regarding the motivation is that the robots are now interpreted to be IW rather than abstract entities, which means that other concerns should also be considered, such as the comfort and viability of the paths taken.

The goal is to create a simulation environment where the IW must traverse areas with different layouts to reach a target destination. The tested maps are a subset of the ones presented in Section 3.1.1.3 since they already had a similar layout to hallways found in public spaces. The scenarios were explored using a pair of IW, with the additional objective of traveling side-by-side whenever possible. In addition, a more classical approach based on APF [64, 65, 70] was developed to serve as a baseline when assessing the performance of the RL-based solution.

One notable similar work that also performs RL on a pair of IW is by Rodrigues *et al.* [48]. Their goal was also to train a wheelchair to navigate through an area to reach a target position. It also had a second objective of creating another independent agent to train another chair to follow the first one. The paper focuses a lot on comparing the performance of Q-Learning and DQNs to complete this task, presenting results that point to DQNs being more efficient and effective. One important difference that it has from the work presented here is the approach to the second wheelchair. Instead of having a leader and a follower as separate agents, the scenario proposed in this work has both chairs attempting to navigate side by side and being trained by a single agent. Therefore, this method ensures that, by selecting a pair of actions, the movement of each chair is dependent on the other, as opposed to having two agents acting separately. Another relevant distinction is the questions trying to be answered: while Rodrigues *et al.* aim to compare the performance of several RL algorithms, the main objective here is to better understand how to tune the environment and the DQN to achieve the best possible results.

4.1 Methodological Approach

The approach to this work was very similar to the one presented in Section 3.1. This was possible since IW act very similarly to the differential drive robots, like the one from the previous approach. The main differences are the adjustments made to accommodate two wheelchairs.

4.1.1 Simulation

The same simulation setup from Section 3.1.1, using Flatland and ROS 1, again taking advantage of the fact that it can be accelerated to run the tests at 10 times the normal speed. To represent a simplified version of a wheelchair, the same robot presented in Section 3.1.1.1 was used. In all scenarios, there are IW that can move independently on the map, each one with its own LiDAR sensor. The state of the wheelchairs is represented by a pair of values representing their positions and rotations (Equation 4.1).

$$wheelchairs\_state = ((x_1, y_1, \theta_1), (x_2, y_2, \theta_2)) \tag{4.1}$$

To detect if the wheelchairs have reached the end, the same solution presented in Section 3.1.1.2 is used. The target beacon has the same appearance (green circle) with the only difference being that now it has two LiDARs instead of one. With the usage of the layers functionality from Flatland, each of the LiDARs detects only one of the wheelchairs, being able to determine the distance of both individually.

Since this task is a lot more complex, not all the maps that were used in the previous chapter were used in this case. Besides the straight hallway (Figure 3.1), the maps that were used were the turns divided in the same group configuration presented in Section 3.1.1.3. Figure 4.1 shows all the maps used and the groups are:

- Group 1 - 90° turns (left or right): First and second maps in Fig. 4.1.
- Group 2 - 180° turns (left or right): Third and fourth maps in Fig. 4.1.

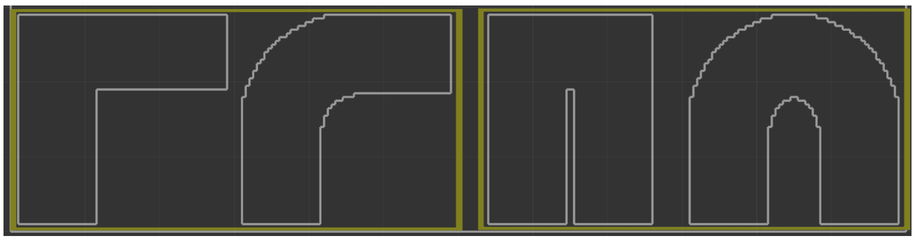


Figure 4.1: All hallway turns used to train and test

Some details about the maps need to be pointed out for some latter considerations. Some turns allow for the usage of faster and more direct paths that, when followed in other maps, would result in a collision. This will force the wheelchairs to learn patterns of navigation from the LiDAR readings instead of fixed paths. The hallways have constant width to allow for both chairs to maintain an adjacent path.

4.1.2 Wheelchair Adjacency

The system requires a well-defined criterion to determine when the wheelchairs are side by side. Initial formalization of the problem established the chairs being adjacent by meeting the following conditions:

1. The Euclidean distance between the two chairs is lower than a threshold (d):

$$r = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2} \tag{4.2}$$

$$r < d \tag{4.3}$$

2. Both chairs are approximately facing in the same direction, with a maximum error threshold (θ_t) (in radians):

$$|\theta_1 - \theta_2| < \theta_t \tag{4.4}$$

3. The line between them has to be approximately perpendicular to the direction they are facing, with a maximum error threshold (θ_t) (in radians):

$$\theta_{per} = \arctan\left(\frac{y_2 - y_1}{x_2 - x_1}\right) \tag{4.5}$$

$$\left|\theta_{per} - \left(\theta_1 - \frac{\pi}{2}\right)\right| < \theta_t \tag{4.6}$$

As there is no information on the absolute or relative position of the wheelchairs, only the LiDAR readings from both of them can be used to identify if they are next to each other. As the LiDARs also detect all other obstacles in the environment, it would be very difficult to determine the adjacency conditions with only these values.

To solve this issue, a simpler definition for wheelchair adjacency was proposed using the rays from the LiDAR represented in green in Figure 4.8. One of the chairs is labeled as the left one and uses two of the rays to its right while the chair labeled as the right one uses ones from its left. If they are side by side, these rays should point to each other, allowing for a new definition of adjacency:

1. All 4 rays have to read below a certain threshold (t):

$$top\_left, bot\_left, top\_right, bot\_right \leq t \tag{4.7}$$

2. Both pairs of top and bottom rays need to have similar readings to a threshold (t):

$$|top\_left - top\_right| \leq t \tag{4.8}$$

$$|bot\_left - bot\_right| \leq t \tag{4.9}$$

This definition allows for false positives when testing for adjacency, as correct values can also appear if both are detecting walls. To minimize potential mistakes, the starting positions place the wheelchairs side-by-side. The main task then becomes maintaining that position as they navigate through the map.

4.1.3 Artificial Potential Fields Algorithm

APF is a classic method used for robot path planning when the scenario contains stationary obstacles and will be used as a baseline to compare the RL approach. The idea is to use potential fields to repel the robot from possible collisions and attract it to the target location. This algorithm offers a relatively fast and effective way to guide robots around obstacles and has been successful in many realistic path planning situations [64]. Figure 4.2 shows a simple example of an APF scenario.

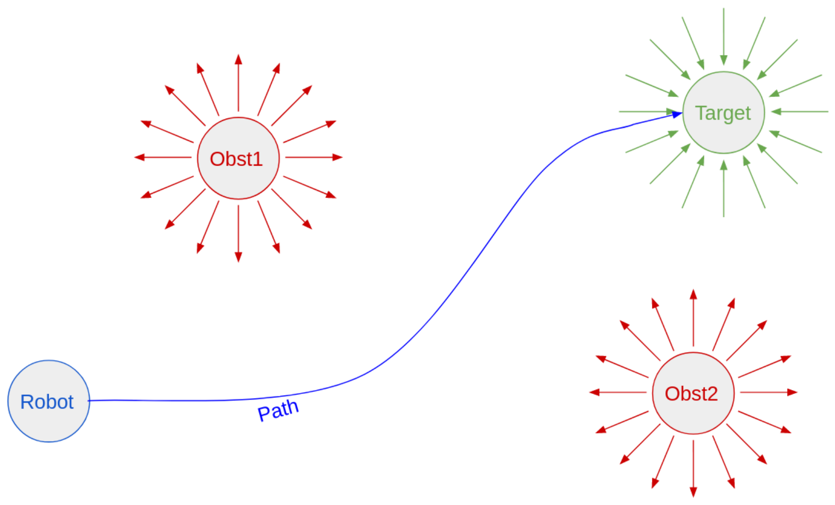


Figure 4.2: Visual representation of a simple APF example for path planning

Each obstacle will create a potential field around itself that will interact with the robot. The force that a charge applies in the robot can be determined using Coulomb's law [70], used to calculate the amount of force between two electrically charged particles:

$$|F| = k_e \frac{|q_1||q_2|}{r^2} \tag{4.10}$$

where k_e is Coulomb's constant, q_1 and q_2 are the two charges involved (robot and obstacle charges) and r is the Euclidean distance (Equation 4.2) between the charges. This is an inverse-square law, which means the force greatly increases the closer the charges are. The direction of the force is defined by the line that passes through both the center of the robot and the obstacle and the sense depends on the sign of both charges (equal signs repel and different ones attract).

The resulting force can be obtained by adding all the forces. Then, the acceleration can be computed using the robot's mass and Newton's second law:

$$\vec{a} = \frac{\sum \vec{F}}{m} \tag{4.11}$$

4.1.3.1 Charges on a Map

From a more practical perspective, the potential field from a real obstacle cannot be represented by a single charge at a point in space. A real obstacle has volume (or an area in a 2D simulation) which means that the closest point to the robot has to be the one considered to determine the distance (r). In addition to that, all the maps that are used in this project have the robots navigating in a space surrounded by walls. To correctly represent the potential fields produced by the maps, charges were divided into several groups based on how the force they applied on the robot was calculated. Each map is composed of a set of charges of one or multiple of these types:

- Point Charges
- Line Segment Charges
- Composite Charges

A **point charge** is defined by a point in space and its intensity (i). It represents the simplest type of charge and, although alone it can not be used to represent walls, an attractive point charge on the target position is used to guide the robot to the goal. Coulomb's law (4.10) can be used to determine the magnitude of the force produced at any point by the charge, but some simplifications can be made:

- Coulomb's constant (k_e) is necessary when calculating forces on actual electric forces but, in this case, the law serves only as an analogy to path planning. Since realistic values are not relevant, the constant can be cut from the equation.
- The charge from the robot itself can also be simplified and positive point charges attract while negative ones repel (essentially making the robot charge -1). This can be seen in Figure 4.3.

This way, a point charge, at a given point in space, produces a force of magnitude:

$$|F| = \frac{|i|}{r^2} \tag{4.12}$$

where i is the intensity of the charge and r is the Euclidean distance (Equation 4.2) between the charge and the point.

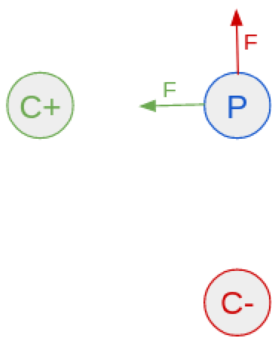


Figure 4.3: Visual representation of how positive (green) and negative (red) point charges apply forces to a point (blue) in space

A **line segment charge** is defined by two points in space that form a line segment (l) and their intensity (i). The idea for this type of charge is to represent the inside of a wall in the maps. To determine the force produced at any point P, first one needs to find the intersection between l and the line that passes through P and is perpendicular to l. If there is no intersection, the charge does not apply a force in P. The magnitude of the force can be determined the same way as in a point charge (Equation 4.12) with r the distance from P to the intersection. Figure 4.4 shows how a line segment interacts with different points in a map.

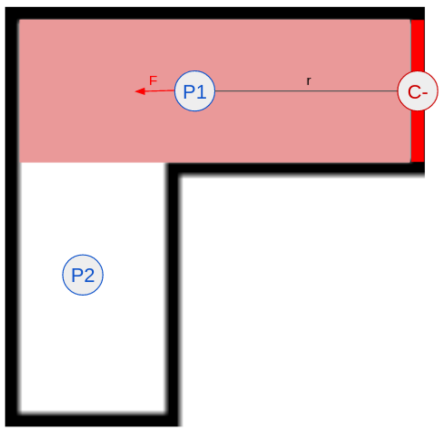
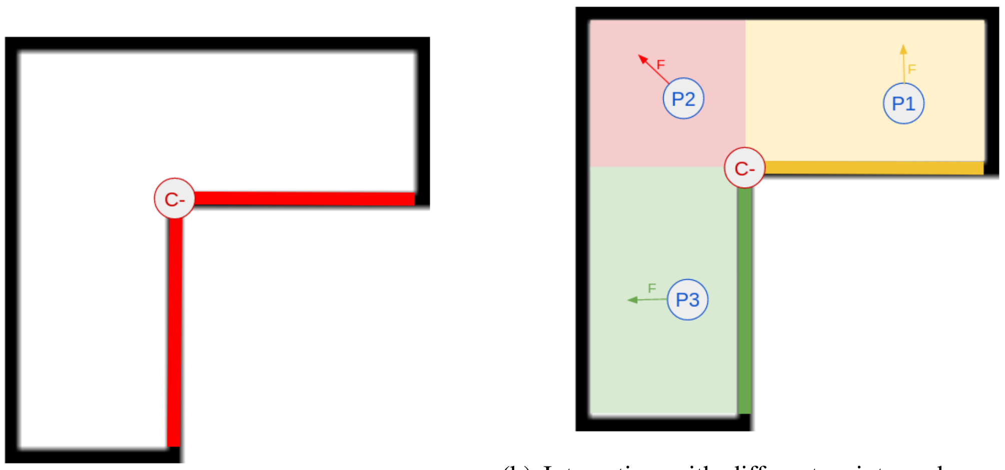


Figure 4.4: Example of how a line segment charge (red) affects different points in space (blue) in the left/right 90° turn map and the area where it applies forces in (light red)

A **composite charge** is defined by a set of line segment charges and point charge and their intensity (i). The force applied at any point is defined only by the charge that is closest to that point. This is used when one or more consecutive vertices of the map layout have an angle larger than 180° degrees facing the inside. A composite charge contains all those vertices (point charges) and all the edges (line segment charges) that are connected to them. Figure 4.5 shows an example of a composite charge and how they apply forces at different points.



(a) Set of charges in the composite (red)

(b) Interaction with different points and areas each charge applies forces (color-coded)

Figure 4.5: Example of a composite charge in the 90° turn map

4.1.3.2 Following Behavior

In an APF algorithm, when more than one robot is used, one of the typical solutions is to have a leader and a follower [65]. The leader chooses their path based on the potential fields from obstacles and the target. The follower also uses the obstacle fields to avoid collisions but, instead of having a charge on the target point, uses the position of the leader to create a Tension Point near the leader to define the path.

A **tension point** is almost identical to a point charge but it calculates the magnitude of the force using Hooke's law instead, which indicates the force needed to extend or compress a spring any distance:

$$|F| = kx \tag{4.13}$$

where k is a constant factor characteristic of the spring and x is the compression/extension of the spring. For the algorithm, k can be replaced by the intensity and x by the Euclidean distance (Equation 4.2) between the tension point and the robot. This means the magnitude of the force grows linearly with the distance, as is shown in Figure 4.6.

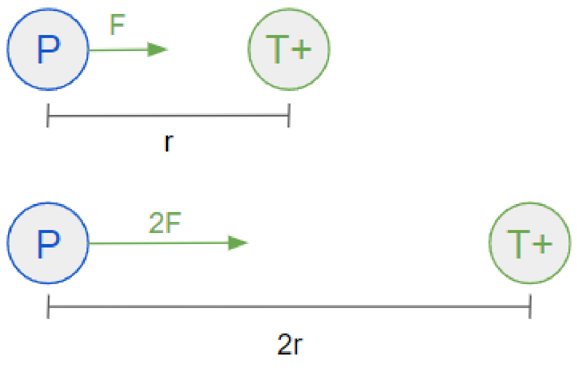


Figure 4.6: Visual representation of how a positive tension point (green) applies forces to a point (blue) in space and how the distance affects its magnitude

The main issue with the leader and follower approach, in this case, is that one of the goals is for the wheelchairs to be traveling side by side. One possible solution is for the follower to aim for a point in space where it would be adjacent to the leader. The problem with this is that the leader would end up traveling through the center of the hallway, potentially not leaving enough space for the other robot. It could be possible to add charges to the map to try to force the leader to navigate next to the walls on one side but the solution proposed here is to modify the leader and follower perspective. The idea is to start by thinking of both robots as a single system and use that system as the leader for both robots to follow. This is achieved by following these steps (also demonstrated in Figure 4.7):

1. Finding the middle point of the positions of both robots.

2. Calculate the resulting forces from all the map charges (with a positive charge in the target position) and add the force vector to get a target middle point.

3. From the target middle point, find the target point for each robot. This is done by shifting the target middle point by half the ideal distance between the robots in both ways of the direction perpendicular to the force vector found in step 2.

4. In the target points found, create a tension point for each of the robots.

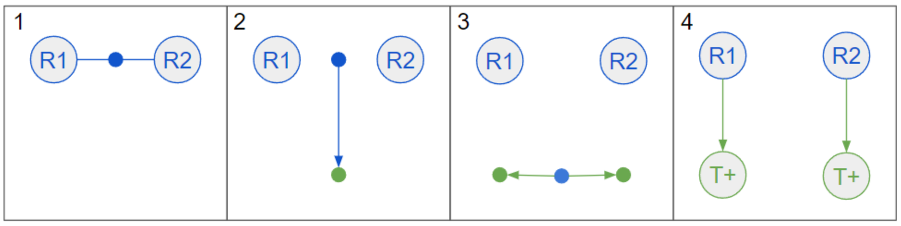


Figure 4.7: Example of the steps necessary to find the target points for both follower robots

After this, it is possible to determine what the resulting force for each robot at any position on the map should be, just by adding all the forces from:

- The charges placed on the walls of the map.
- A negative point charge in the position of the other robot, to help avoid collisions between them.
- Add the force from the newly found tension point.

4.1.3.3 Limitations of the IW

The fact that the goal is to simulate a pair of wheelchairs means that some limitations have to be considered. To maintain a real use case plausible, the simulation must respect movement restrictions and passenger comfort considerations. These problems condition the way the APF algorithm can function.

Just like a real wheelchair, the robots in the simulation use differential drive to move. This means that to move towards a target, they need to already be facing that direction. Upon knowing the resulting force that acts on them according to APF, to make sure they are following the correct path, the robots start by only having angular speed until they reach the correct location (with a small margin of error) and only then can use linear speed to move forward. To make sure the movements do not become unsynchronized, the robots only move forward if both are facing the correct way.

In addition, due to hardware limitations and danger to its passenger, wheelchairs have limited speed and acceleration. The most important component of the resulting force from the APF algorithm is the direction since that is what contributes the most to making the robots reach the end.

But the magnitude is still useful because, if one of the robots falls behind, the added force from the tension point can help it catch up. The solution found was to decide on a maximum linear speed and, at any point, give the robot with the largest force magnitude that speed and the other the same speed scaled down using the ratio of bot magnitudes:

$$v_1 = max\_speed \tag{4.14}$$

$$v_2 = max\_speed\frac{|F_2|}{|F_1|} \tag{4.15}$$

where v_1 and F_1 are, respectively, the linear velocity and the resulting force of the robot with the largest force magnitude ($|F|$), while v_2 and F_2 correspond to the other robot.

4.1.4 Reinforcement Learning Algorithm

The DQN agent used is the same as the one created in the work from the previous chapter, presented in Section 3.1.2. It maintains the same learning rate (0.01) and action selection policy (Boltzmann policy with an initial τ of 1.0). It also follows the same DQN architecture presented in Table 3.1. This agent is used to train both wheelchairs simultaneously.

The environment works very similarly to the one presented in Section 3.1.3, linking Flatland Simulator and the DQN agent to train the hyperref[abbrevs]IW. Collisions can happen with any of the chairs and, to reach a successful final state, both have to be in the target location. Since using two wheelchairs heavily increases the complexity of the task, there is no random shift to the starting positions of any of the robots. The exogenous variable starting position and rotation presented in Section 3.1.5 is now always controllable, making the model now exclusively Deterministic.

4.1.4.1 Input and Action Spaces

The sampling from the LiDAR sensors is done in the same manner as what is depicted in Figure 3.3. The main difference is that the input space is doubled seeing that there are two IW. Additionally, To detect the adjacency between them, two extra rays from each of the LiDARs are stored to help identify if the chairs are side by side. This new sampling setup is represented in Figure 4.8 where the green lines on the right side represent the ones for the wheelchair on the left while for the one on the right the rays would be on the left side. This means that the input space has a dimension of 38 (Equation 4.16).

$$space(38) = (samp(12) + extrarays(5) + siderays(2)) * 2 \tag{4.16}$$

Each wheelchair moves independently which means each one needs its action. The actions for one IW remain the same as the ones from Table 3.2, but the agent has to select one for each at every step. By choosing a pair out of the 7 original actions, the action space is now 49 (7^2).

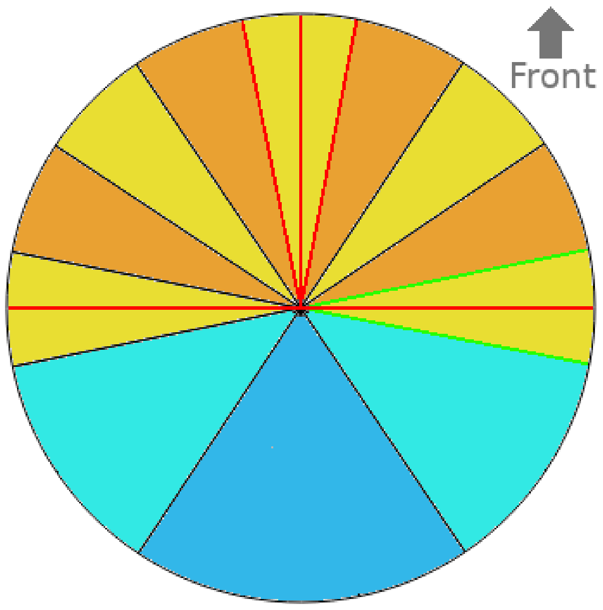


Figure 4.8: Visual representation of the LiDAR sampling strategy for the left wheelchair in a two-wheelchair setup

4.1.4.2 Reward Function

The reward function rules from Section 3.1.2.3 that were applied to one robot, still apply to each of the wheelchairs. Some rules had to be added or modified to adjust to the two IW traveling side by side task. Additionally, two secondary goals were to optimize the path to be both efficient and comfortable for the users of the wheelchairs. These new rules are:

- Collisions now can be caused by any of the wheelchairs and still give a large negative reward (line 3).
- To give the large positive reward for reaching the end both wheelchairs need to be in the target area (line 5).
- If a full 360° rotation from the starting position is completed, receive a negative reward (line 34). The magnitude of the penalty increases if consecutive complete rotations are performed in the same sense. This helps to prevent the chair from getting stuck in a section of the map and also has the user's comfort in mind (to avoid motion sickness).
- If one of the chairs reaches the target position, receive a large positive reward (line 13).
- If one of the wheelchairs leaves the target position after reaching it, receive a large negative reward (line 15).
- If after any action the wheelchairs are side by side, receive a positive reward (line 43). What determines if the chairs are considered to be adjacent to each other is explained in Section 4.1.2.

Algorithm 2 Reward function computation for two wheelchairs

```
Input:
    agent - RL agent for all wheelchairs
    n_steps - Current step
    max_steps - Number of steps in an episode
 1: $basic\_reward \leftarrow \frac{300}{max\_steps}$
 2: if $agent.collided()$ then
 3:     $reward \leftarrow -200$
 4: else if $agent.allReachedDestination()$ then
 5:     $reward \leftarrow 400 + max\_steps - n\_steps$
 6: else if $n\_steps > max\_steps$ then
 7:     $reward \leftarrow -(300 + agent.getForwardReward())$
 8: else
 9:     $reward \leftarrow 0$
10: for each $wc \leftarrow wheelchair \in agent$ do
11:     $action \leftarrow wc.getCurrentAction()$
12:     if $wc.reachedDestination()$ then
13:         $reward \leftarrow reward + 100$
14:     else if $wc.leftDestination()$ then
15:         $reward \leftarrow reward - 100$
16:     if $action = FORWARD$ then
17:         $reward \leftarrow reward + basic\_reward$
18:         $agent.incrementForwardReward(reward)$
19:     else if $action \in \{FORWARD\_LEFT, FORWARD\_RIGHT\}$ then
20:         $reward \leftarrow reward + 0.5 * basic\_reward$
21:         $agent.incrementForwardReward(reward)$
22:     else if $action \in \{STOP, BACK\}$ then
23:         $reward \leftarrow reward - 5 * basic\_reward$
24:     else if $action \in \{ROT\_LEFT, ROT\_RIGHT\}$ then
25:         if $wc.getForwardLaserDistance() > 0.1$ then
26:             $reward \leftarrow reward - 2 * basic\_reward$
27:         else
28:             $reward \leftarrow reward + 0.5 * basic\_reward$
29:     if $n\_steps > 0$ then
30:         $prev\_acts \leftarrow [wc.getPreviousAction(), action]$
31:         if $(prev\_acts = [FORWARD, BACK]) \vee (prev\_acts = [BACK, FORWARD]) \vee$
      $(prev\_acts = [ROT\_LEFT, ROT\_RIGHT]) \vee (prev\_acts = [ROT\_RIGHT, ROT\_LEFT])$ then
32:             $reward \leftarrow reward - 5 * basic\_reward$
33:     $consec\_rot \leftarrow wc.getFullConsecRotations()$
34:     $reward \leftarrow reward - 75 * consec\_rot$
35:     $forw\_laser \leftarrow wc.getForwardLaserDistance()$
36:     $reward \leftarrow reward + 2 * (forw\_laser - 0.5) * basic\_reward$
37:     if $forw\_laser > 0.5$ then
38:         $FL\_las \leftarrow wc.laserFLDistance()$
39:         $FR\_las \leftarrow wc.laserFRDistance()$
40:         $laser\_rew \leftarrow ((FL\_las - 0.5) * (FR\_las - 0.5))$
41:         $reward \leftarrow reward + laser\_rew * basic\_reward$
42: if $agent.wheelchairsAdjacent()$ then
43:     $reward \leftarrow reward + 10$
44: return reward
```

4.2 Results and Discussion

This section will present all the results obtained during testing. The initial goal was to perform simple tests to determine if the setup was working as intended. Since some problems were encountered with the initial settings, the system went through several iterations and the next subsection will explore the reasoning and results obtained in each of them. Once the setup was producing better results, more complex scenarios were explored. The training strategy used in all tests is the second one presented in Section 3.1.4 with the difference that, if the accuracy threshold (0.8 in all tests) was not reached until 1000000 steps of training, it stops and considers the best weights obtained up to that point.

4.2.1 Preliminary RL experiments with a straight hallway

The purpose of using a single agent to train both wheelchairs at the same time is to ease the process of coordinating both of them while traveling. This choice comes with a cost of a much more complex task for the agent to learn (larger input and action spaces). Initial tests for two wheelchairs were done with the simplest possible task: two chairs starting side by side in a straight hallway, already facing the target. Table 4.1 and the graph in Figure 4.9 show results with the number of steps to train, the accuracy achieved and an additional performance indicator for the wheelchairs' adjacency. Adjacency is the fraction of steps when testing where the wheelchairs were considered to be next to each other.

Table 4.1: Results from training two hyperref[abbrevs]IW in a straight hallway

Train steps	**Accuracy**	**Adjacency**
570000	0.77	0.21

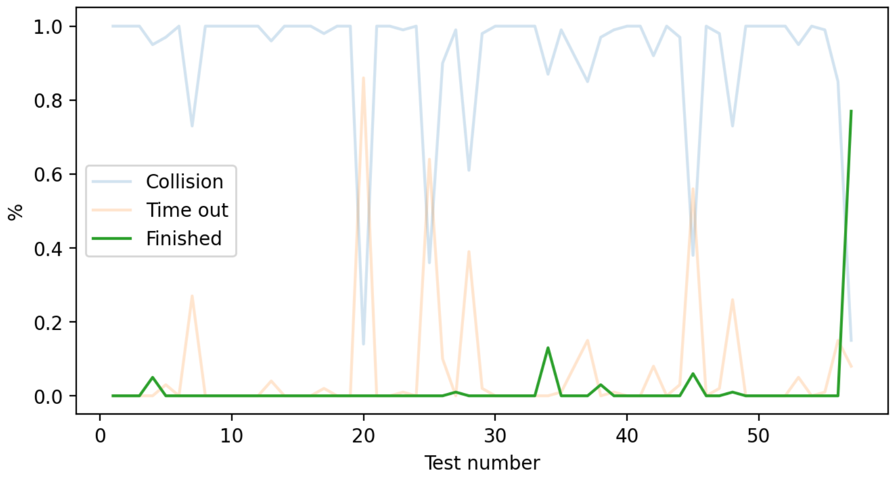


Figure 4.9: Two IW in a straight hallway end conditions of testing over 570000 steps of training

Despite taking a very long time on a simple task, the final accuracy was high and some adjacency was achieved. The main issue can be observed in the graph showing the accuracy: there seems to be no steady growth in the success rate and higher accuracy tests seem to be outliers. This probably means that, instead of learning useful patterns to navigate the map, the agent, after enough training sessions, is reaching weights that allow the chairs to finish the map by random chance. This means that it does not accumulate any knowledge between training sessions regardless of the accuracy achieved. To better understand the reason for this problem, several probable causes were proposed:

- Because there are two chairs, there is less space to move causing more collisions. From the graph in Figure 4.9 it is possible to observe that collisions are the most common end condition throughout training.
- Very large action space (7 for each chair means a total of 49 different actions for the agent).
- Large input space (dimension of 38).

This would cause problems in any further testing, so a solution to solve or at least mitigate the issue was necessary. Based on the proposed causes, several changes were made to the environment. The next sections will describe the experiments made.

4.2.1.1 Solving space issues

To try to solve issues caused by the lack of space for two wheelchairs, two changes were made. Additionally, a small change was done to the reward function. These changes were:

- Increasing the radius of the target area (from 0.25 to 0.4).
- The chairs now start further away from each other to try to reduce collisions between them (from 0.3 to 0.4).
- If a chair is in the target area, it now receives a positive reward for stopping and a negative one for any other action.

Using this new environment, the same methodology from Section 4.2.1 was applied to test it. Table 4.2 and the graph from Figure 4.10 show the results obtained.

Table 4.2: Results from training two hyperref[abbrevs]IW in a straight hallway with space improvements

Train steps	**Accuracy**	**Adjacency**
670000	0.79	0.17

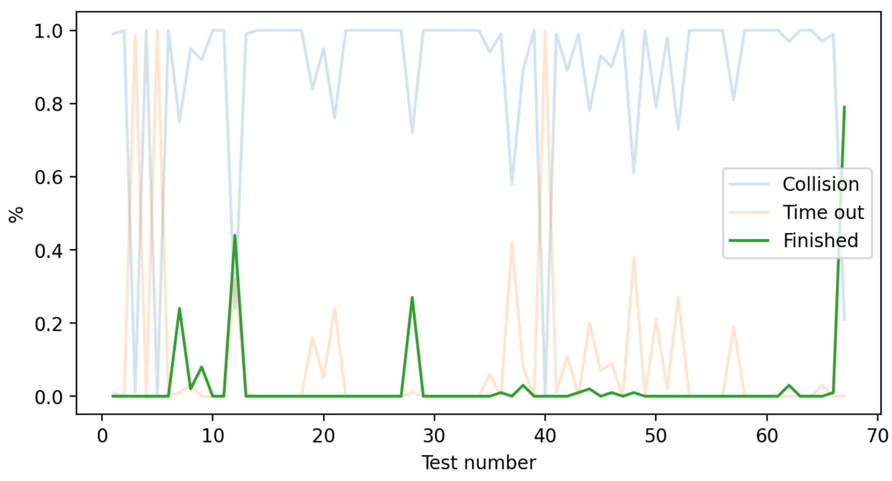


Figure 4.10: Two IW in a straight hallway with space improvements end conditions of testing over 670000 steps of training

Some small improvements can be seen in Figure 4.10, as more peaks of accuracy appear. The problem is not solved though because, when better results are achieved, the accuracy goes back to 0 in the next session, indicating the agent is not learning properly. Therefore, in addition to these, further changes were required before testing with more complex maps.

4.2.1.2 Reducing the number of actions

To try to improve the performance of the DQN, some actions were removed, leaving only four available (actions space from 49 to 16). The new set of possible actions can be seen in Table 4.3.

Table 4.3: New wheelchair actions

Action	Linear velocity (m/s)	Angular velocity (rad/s)
Stop	0	0
Move forward	0.3	0
Rotate left	0	1.05
Rotate right	0	-1.05

It is important to notice that, by removing actions, the wheelchairs lose some autonomy of movement. The actions removed were chosen with the purpose of not having a great effect on the number of possible paths but can have a greater effect on the time needed to reach the goal. The results are shown in Table 4.4 and graph from Figure 4.11. Figure 4.12 also shows a graph with the adjacency measured throughout training.

This change shows very significant positive changes. The most apparent one is how much better the accuracy is and how much faster it was reached, only taking 50000 to achieve a perfect score. There is also some improvement in adjacency. The most important difference is the steady

Table 4.4: Results from training two IW in a straight hallway with fewer actions

Train steps	Accuracy	Adjacency
50000	1.0	0.28

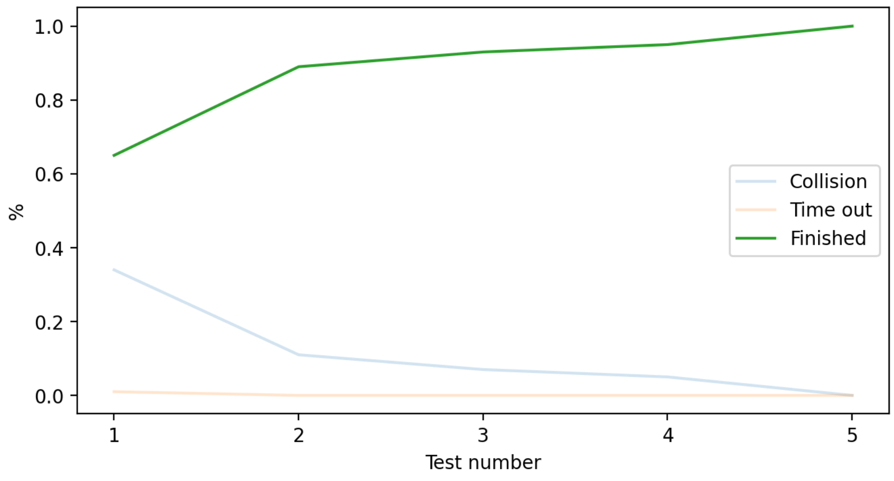


Figure 4.11: Two IW in a straight hallway with fewer actions end conditions of testing over 50000 steps of training

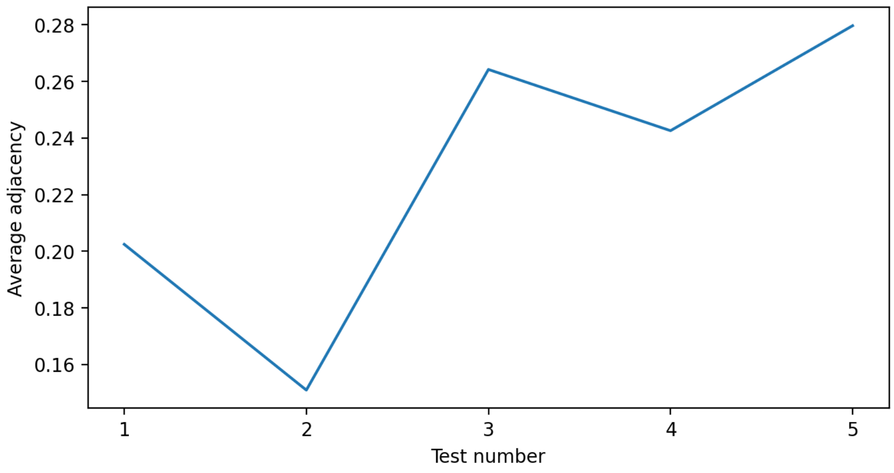


Figure 4.12: Two IW in a straight hallway with fewer actions adjacency of testing over 50000 steps of training

growth achieved throughout training, both in the accuracy (Figure 4.11) and adjacency (Figure 4.12).

The excessive number of possible different actions appeared to be the main reason for the poor performance of the DQN. After solving this issue, initial results start to show a lot more consistency and stability. For those reasons, the environment with the changes described in Sections 4.2.1.1 and 4.2.1.2 was the one used in all the remaining tests.

4.2.2 Experiments with non-straight hallways

To test if the single agent for two wheelchairs is capable of adapting to a map after training for a different one, an experiment where the agent learns the 90° turns and then tries to complete the 180° turns was conducted. The weights obtained in the training described in Section 4.2.1.2 were also used for comparison. For each of the scenarios tested, 100 episodes were run and the accuracy and adjacency were retrieved. The results from training are in Table 4.5 and the results from testing are in Table 4.6.

Table 4.6 also includes the results of applying APF to the straight and sharp turn hallways (90° and 180°), to establish a baseline. For all these maps, a charge layout was created, always using the same intensity values, and each of the different scenarios was tested 20 times, each time letting the algorithm run until the end is reached or a collision happens (the same as an episode in RL).

Table 4.5: Results from training two hyperref[abbrevs]IW in the 90° turns and the straight hallway

Trained	**Train steps**	**Accuracy**	**Adjacency**
Straight	50000	1.0	0.28
90° turns	100000	0.44	0.01

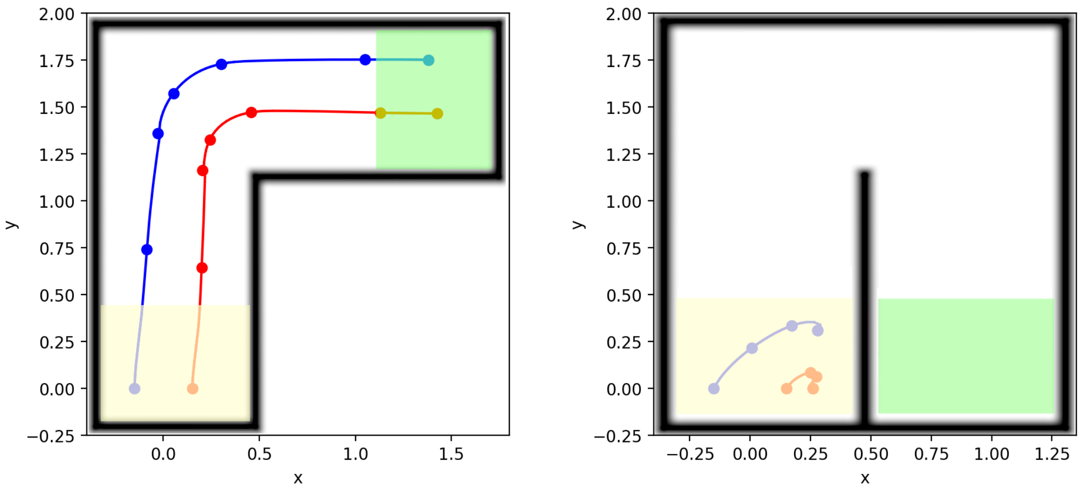
Table 4.6: Testing results from training straight hallway and 90° turns.

Tested map	**APF**		**RL trained with 90° Turns map**		**RL trained with Straight map**	
	Acc	**Adj**	**Acc**	**Adj**	**Acc**	**Adj**
Straight	1.0	1.0	0.96	0.0	1.0	0.28
Turn left (90°)	1.0	0.54	0.35	0.02	0.0	0.03
Turn right (90°)	1.0	0.62	0.13	0.0	0.0	0.13
Curve left (90°)	-	-	0.79	0.01	0.0	0.02
Curve right (90°)	-	-	0.34	0.0	0.0	0.09
Turn left (180°)	0.0	0.04	0.32	0.05	0.0	0.02
Turn right (180°)	0.0	0.07	0.07	0.0	0.0	0.11
Curve left (180°)	-	-	0.68	0.0	0.0	0.01
Curve right (180°)	-	-	0.35	0.01	0.0	0.04

4.2.2.1 Results analysis for APF

The first thing to notice is that the results are very polarizing across the maps. This happens because, in contrast with RL methods, this is a reactive algorithm meaning that, given the same conditions, it will take almost the same path. This path either succeeds or fails, depending on the scenario, which is why the accuracy is either 0 or 1. Adjacency was also very high when the algorithm was successful, even when completing 90° turns, demonstrating the viability of the method presented in Section 4.1.3.2. This behavior is demonstrated in Figure 4.13a.

The algorithm fails to complete 180° turns and always ends up colliding with the wall in the middle, as is shown in Figure 4.13b. This happens because it is very hard to find a balance for the intensity of the positive end goal charge as it needs to initially move away from the target to go around the wall. This means that the setup that was 100% effective for all other maps completely fails in this scenario.



(a) Successful test on the 90° turn (b) Failed test on the 180° turn

Figure 4.13: Paths taken by two IW on the 90° and 180° turns using the APF algorithm (left wheelchair in blue right in red, start yellow zone, finish green zone; dots in the lines represent positions in fixed time steps)

The algorithm was not assessed for the 90° and 180° curves, as that would require adapting the computation of the electric force for curves, which is more complex than with straight lines, and APF is merely being used to establish a baseline, with the main focus being on RL.

4.2.2.2 Results analysis for RL

A lot of observations can be done based on these results. Some of them include:

- Although some success was achieved, results still show an accuracy below 50%. This means that there is still a lot of room for improvement before the model can be used with real wheelchairs.

- For the results with the turns, the agent was able to use what was learned in the 90° turns and apply it to the 180° turns. Despite not having very high accuracy, it is consistent through the maps, showing it is learning useful patterns.
- The results for left turns seem to be better in all scenarios than for right turns. This shows that in this setup the agent learning how to turn to one side does not imply that it can turn as effectively to the other side.
- There also seems to be a tendency for better results in curved hallways compared to sharp turns. This could be because curves allow for a shorter path to be taken. The 180° sharp turn also has a thin wall across the map that can be harder for the LiDARs to detect from certain angles.
- The scores for the adjacency were very low. This is probably due to the conditions for the adjacency being very restricting and the wheelchair is not able to maintain it with the limited movements it can perform. A better solution could be having an adjacency score instead of a binary decision (adjacent or not).
- The weights from the straight hallway could not be used on other maps. This was caused by a very low amount of inputs being trained, causing the wheelchairs to run into completely new readings from the environment, having no knowledge of what actions to take.

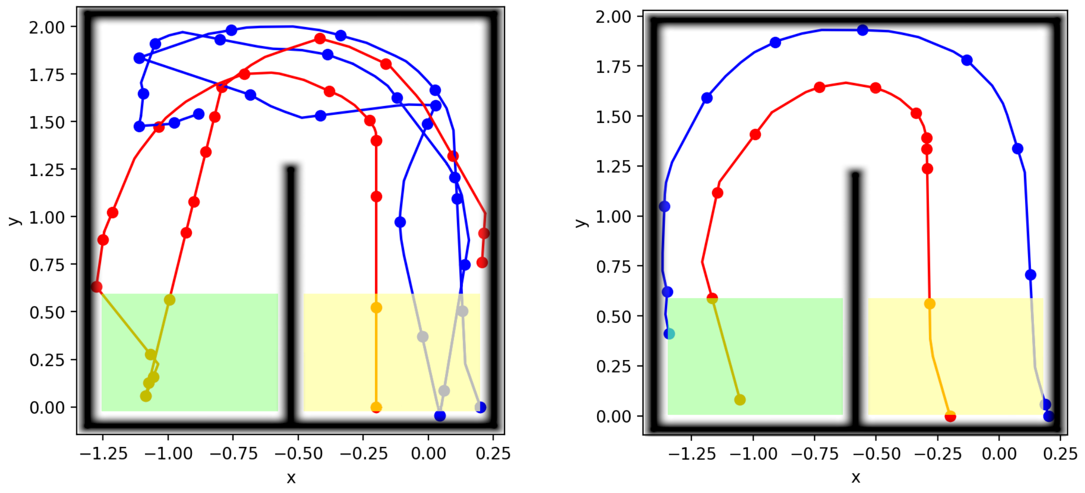
One possible issue that could be preventing a higher accuracy is the fact that the agent does not know when one of the wheelchairs reaches the end. This causes the chair to turn back and follow the hallway in the opposite direction. This behavior can be observed in the graphs from Figures 4.14a and 4.15a. In contrast, successful episodes usually happen when both wheelchairs can directly reach the end at similar times, as it is shown in Figures 4.14b and 4.15b. Adding the information that one of the IW has reached the end of the DQN input should solve the issue.

4.2.2.3 Comparison between RL and APF

When compared to RL, APF presents very competitive results, since it is very effective under the right circumstances. Particularly when it comes to the adjacency, it seems to have much better results. However, before any conclusions are made, one must take into consideration that APF needs a lot of knowledge about the environment, namely:

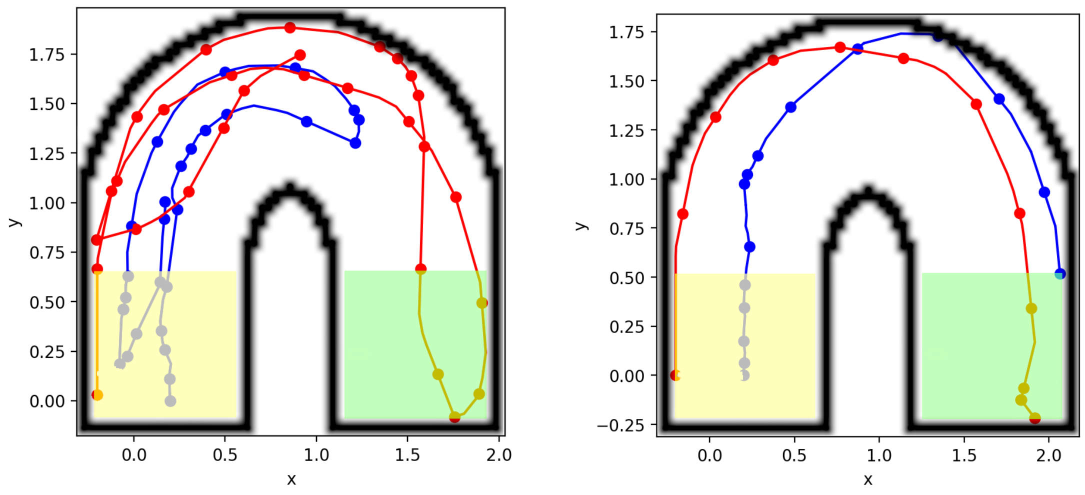
- The complete layout of the map to place charges on the obstacles;
- The position of both robots at all times;
- The position of the target area.

In a real scenario, this information will probably not be available, which immediately raises questions about the adaptability of the algorithm to new scenarios. With RL, on the other hand, the robot can adapt to new maps, distinct from the ones used during training.



(a) Failed test (b) Successful test

Figure 4.14: Paths taken by two IW trained to do 90° turns, on two tests on the 180° left turn (left wheelchair in blue and right in red, start yellow zone, finish green zone; dots in the lines represent positions in fixed time steps)



(a) Failed test (b) Successful test

Figure 4.15: Paths taken by two IW trained to do 90° turns, on two tests on the 180° right curve (left wheelchair in blue and right in red, start yellow zone, finish green zone; dots in the lines represent positions in fixed time steps)

Even if it was to be assumed that all the required information is available, APF offers no guarantee of working in a new scenario. This can be seen by how it was able to very effectively complete the straight hallway and the 90° turns but, with the same setup, it was not able to complete a 180° turn. While the RL might have shown lower accuracy on average, it was always able to find some success in new scenarios, showing an ability to learn patterns and adapt to its environment.

In conclusion, RL is much easier to use than APF in unexplored environments since it does not require any prior knowledge of the map to work. It also has a lot of room for improvement, either by training for a longer time or tuning learning parameters. These reasons make RL the best option between the two to perform autonomous navigation to IW since it provides the best chances to eventually use them in real-life scenarios.

4.3 Conclusions

As there is an increasing necessity for autonomous intelligent wheelchairs, new techniques to tackle the problem start to appear. RL still offers one of the best answers for this issue.

The main contribution of this work is the application of a DQN to train a single agent to control two IW for navigation along simulated curved hallways. This allowed for a detailed exploration of some advantages and obstacles of using this RL algorithm. APF was used as a baseline and proved to be very effective under the right circumstances but with some major issues that can be resolved with RL, such as the need to have complete and perfect information on the map, not being capable of improvement (reactive method) and very low adaptability to new scenarios. With RL, it was proven that it is possible for the two wheelchairs to successfully navigate through a map they never used before by learning patterns trained on other maps. As it is common with ML techniques, tests revealed the trained network to be usable in some other scenarios but usability in all future scenarios is not ensured.

This second part of the initial work was very valuable to further explore the RL approach for navigation. Several issues were found during the different scenarios explored with many solutions being proposed and some of them proving to be very effective. Perhaps as important as the RL conclusions is the re-interpretation of the problem to use IW. It would not be very productive to perform most of these tests in real wheelchairs as they would be very time-consuming and costly, demonstrating the value of using a simulation before using real robots. This proves the value of the kit being developed for studying real scenarios, despite not including a physical robot.

4.4 Summary

This chapter presented the second part of the initial work that explored path planning using two IW. It presented all the changes made from the setup presented in Chapter 3 to the simulation components and the RL setup. It also provided a new set of results from various tests performed and, by using simulated IW in Flatland, proved the value of using simulations in Robotics to study real scenarios.

Chapter 5

Developing a Kit to teach Robot Learning

This chapter will present the development of a kit that can be used to teach Robot Learning, which is the main goal of this dissertation. The initial work presented in Chapters 3 and 4 was used to understand how to develop the best possible framework with the tools available for RL and Robotics and what is the most effective way to teach it.

The kit developed is meant to be valuable for users with different levels of expertise in Robot Learning, with the only pre-requisite being some fundamental concepts of programming. It was designed to guide the user in a sequence of logical steps that provide a simple and replicable basis for developing Robotics applications that use RL. To explain how the kit was developed, this chapter will explain the process of creating the framework, designing the tutorials and performing user tests to evaluate its performance.

5.1 The Kit's Framework

The kit created has several components that need to work together. There were multiple options for all of them and each of the choices was made with multiple factors taken into account. This section will explain the decision process for all the components.

5.1.1 GitHub as a host for the tutorials

By using GitHub as the platform to host the tutorials, several problems are eliminated. Since this resource is already used by many Computer Science students, there is no need to learn a new platform and immediately makes the kit very easy to access and share. Each tutorial has a dedicated repository for the ROS package and the Markdown *README* file on the main page is used to display the instructions for the tutorial. The Markdown format is simple to read and understand the page's structure, while also being very useful to share terminal commands and example code snippets.

5.1.2 Virtual Machine

To run the packages provided by the kit on their machine, the user needs to install all the framework components. This can be a lengthy process and, if errors occur during any of the installations, it can not only take more time but also become very tedious and frustrating, since the user will need to search for how to solve the issue. These situations take the focus of the main topic of the tutorials.

To avoid intimidating the user with the framework setup, especially one that has no experience in Robotics, a Virtual Machine (VM) for VirtualBox [1] was created that is ready to run the packages provided by the tutorials. The VM is available through a Google Drive link [2] in the tutorials. The file can be directly opened in VirtualBox and contains a VM with Ubuntu 22.04.

For someone that will need to use the framework for a prolonged time, its installation is inevitable and instructions for it are available at the beginning of the first tutorial. These instructions try to cover as many potential issues as possible but there are no guarantees that it will work on all machines.

5.1.3 ROS 2

Unlike the initial work that had to use ROS 1 for reasons that will be explained in Section 5.1.4, all contents from the kit will be developed under the ROS 2 framework. Using ROS is the best option since it is expected that it will be the base for most commercial robots in the near future. Due to its modularity, ROS also promotes good software engineering principles [1].

Other than the possibility of being done in different OSs, none of the new features of ROS 2 will be focused on the tutorials. This means that the kit could also use ROS 1 since, for the more basic features, they function very similarly. Despite this, ROS 2 is still the best choice as the end-of-life for ROS 1 is approaching and there is still a lack of learning materials to help with the transition.

5.1.4 Flatland Simulator

As it was already mentioned, since it is meant to only use a simple differential drive robot, Flatland was the simulator chosen to be used to develop the kit. At the time the work from Chapters 3 and 4 was done, Flatland was not available in a version that was compatible with ROS 2 which meant ROS 1 had to be used. When this kit was being developed, a version for Flatland had recently been released for the latest ROS 2 distribution (Humble) [3].

Flatland is a performance-centric 2D robot simulator and was designed to be a lightweight alternative to other ROS integrated simulators. It heavily facilitates the framework's comprehension for the user since, by removing one of the axes, a 2D simulator simplifies the setup and any necessary geometrical calculations. It also allows for the simulation to be accelerated without

[1] `https://www.virtualbox.org/`

[2] `https://drive.google.com/file/d/1N6N4pSjVlnStYj-vKl02eGlmH9EECybP/view`

[3] `https://github.com/JoaoCostaIFG/flatland`

many performance issues, which is very useful for RL purposes. By improving the performance, Flatland will allow the kit to work on more machines and to be used inside the VM presented in Section 5.1.2.

5.1.5 Python Programming Language

To create robot controllers, a programming language that is supported by the framework has to be chosen. ROS 2 currently offers support for programming both in Python and C++. The decision for all the tutorials was to use Python and the reasons for that will be explained in this section.

5.1.5.1 Python and C++ comparison

Most of the resources to learn ROS usually offer examples for both Python and C++ and the ROS 2 documentation follows this pattern. The functioning of ROS packages are very similar for both and differences mainly come from language-specific syntax or semantics. Since, from the perspective of teaching ROS, there seems to be very little difference between Python and C++, this component was not considered in the decision.

In more recent times, Python has been gaining a lot of popularity and is particularly useful for Computer Science students that are starting to learn how to code, which is one of the major target audiences for this kit, when compared with other programming languages[55, 3]. When it comes to packages to work with RL, Python also seems to be gaining an advantage over other alternatives as more and more new libraries for AI and ML appear[11]. For these reasons, the choice was made to focus the kit on teaching Python, as it seems that, in the near future, it is going to be an essential skill to work in the field of Robot Learning.

5.1.5.2 RL packages for Python

When it comes to RL resources in particular, the Gym [4] package developed by OpenAI offers a simple interface capable of representing environments that can be used to train RL agents. This is done by providing a class that can be inherited from and serves as a template to create RL environments. This class emphasizes more fundamental principles of RL, such as observation/action spaces and the reward function, instead of algorithm-specific concepts, making it ideal for an introductory lesson.

The Gym package does not actually provide any of the RL algorithms, but serves as the basis for many others that do. Some of the compatible packages were already presented in Section 2.3.5, such as Stable-Baselines3 and Keras-RL, the latter being used for the initial work. Using the Keras-RL package would require the users to build their own NNs which can be too demanding for new users. Stable-Baselines3 on the other hand provides default parameters for the policy while also letting a more experienced user create their own. For this reason, when creating the framework, it was decided to change the RL agent package to Stable-Baselines3. Aside from

[4] https://www.gymlibrary.dev/

being more beginner friendly, it also provides a wide range of the most popular algorithms, such as DQN, PPO or DDPG.

5.2 Tutorials for Robot Learning

The tutorials were created in a way that guides the user through the process of constructing a ROS 2 package for Flatland and eventually uses a RL agent to train a robot to perform a simple task. This way, the user only needs to have some basic knowledge of Python programming. Each tutorial starts by showing the user how to run an example package, also providing visual examples of the package running by showing images and animations, and then explaining all the relevant components and files it is using. This is done by showing code snippets and explaining some theoretical concepts while providing the user with links to more information on the topics. Throughout the tutorial, users are presented with some suggestions to try to modify the code and explore the package for themselves. The code also contains several comments to help understand it.

5.2.1 Tutorial 1 - Teleop Keys

The first tutorial [5] will focus on teaching how to setup a Flatland package for ROS 2. It will also show how a robot can be controlled through the use of a Python script. The full tutorial is included in Appendix A.1. After completing the tutorial, the user should be able to:

1. Prepare a machine to develop basic ROS 2 packages with Flatland.
2. Analyse the progress of a virtual robot in 2D simulation with visualization tools.
3. Analyse the code controlling a virtual robot in 2D simulation.
4. Modify the code controlling a virtual robot in 2D simulation.

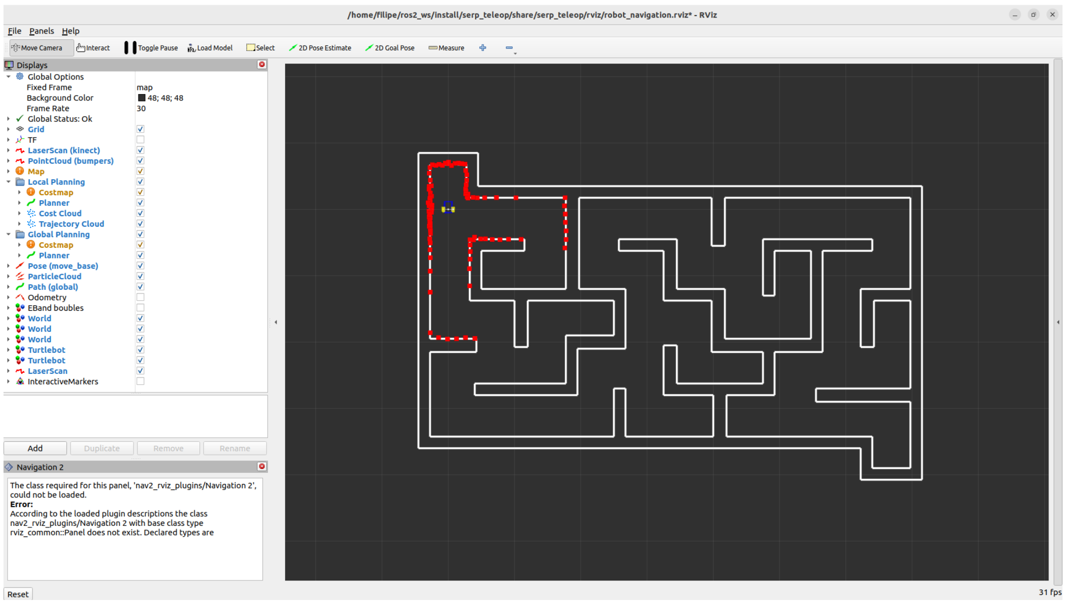
5.2.1.1 Example package

When the user runs the package, a window with the configured Flatland world will appear, just like is shown in Figure 5.1. This will show a simple robot inside a map in the shape of a maze [6]. The robot is the same as presented in Section 3.1.1.1 and contains the following Flatland plugins:

- Bumper - Detects collisions and publishes them to a topic.
- Diff Drive - Subscribes to a topic that receives messages to modify the model's velocity.
- Laser - Simulates a LiDAR sensor and publishes the readings to a topic.

[5] `https://github.com/FilipeAlmeidaFEUP/ros2_teleopkeys_tutorial`
[6] `https://en.wikipedia.org/wiki/File:Maze_simple.svg`

The eventual goal of the tutorial will be to control it with the keyboard but, initially, the robot moves forward until it detects a wall in front of it using its radar. When a wall is detected, it randomly chooses one direction to rotate 90°, essentially following a random path. If a collision happens, the robot goes back to its original position.



```
Global Options
  Fixed Frame           map
  Background Color      48; 48; 48
  Frame Rate            30
Global Status: Ok
Grid                    ✓
TF
LaserScan (kinect)      ✓
PointCloud (bumpers)    ✓
Map                     ✓
Local Planning          ✓
  Costmap               ✓
  Planner               ✓
  Cost Cloud            ✓
  Trajectory Cloud      ✓
Global Planning         ✓
  Costmap               ✓
  Planner               ✓
Pose (move_base)        ✓
ParticleCloud           ✓
Path (global)           ✓
Odometry
EBand boubles
World                   ✓
World                   ✓
World                   ✓
Turtlebot               ✓
Turtlebot               ✓
LaserScan               ✓
InteractiveMarkers
```

```
The class required for this panel, 'nav2_rviz_plugins/Navigation 2',
could not be loaded.
Error:
According to the loaded plugin descriptions the class
nav2_rviz_plugins/Navigation 2 with base class type
rviz_common::Panel does not exist. Declared types are
```

Figure 5.1: Robot inside a maze in the Flatland window from tutorial 1

Later in the tutorial, the user is asked to run a different ROS 2 package[7] in another console. This package was also created for this tutorial but can be reused for other projects. It reads keystrokes from the keyboard and publishes them to a ROS Topic. Instructions on how to use this package are included in Appendix A.2. By changing a Boolean variable inside the robot controller, it will now subscribe to that topic and the user will be able to control the robot using the arrow or the *WASD* keys. If it gets too close to a wall, a warning message appears on the terminal and the robot moves slower. Just like in the previous scenario, any collisions send the robot to the original position.

5.2.1.2 Theoretical concepts

This tutorial explains the user how ROS is structured in Nodes and how each of them should have a singular task. It shows how Nodes communicate between them using either Topics (subscriber and publisher model) or Services (server and client model). It also reinforces the importance and value of the modular structure of ROS by showing how it can be used effectively.

5.2.1.3 Practical concepts

This tutorial shows the user how to:

[7]https://github.com/FilipeAlmeidaFEUP/ros2_teleopkeys_publisher

- Use ROS 2 launch files to run several nodes.
- Change the Flatland's launch parameters.
- Create an instance of the class Node in Python.
- Publishing and subscribing to Topics in a Python script using the Node class.
- Make a service request in a Python script using the Node class.
- Use ROS 2 commands to see information about active Nodes, Topics and Services.
- Add new models to the visualization file (ROS RViz file).
- Modify relevant setup files, such as the Python setup and the Extensible Markup Language (XML) package file.
- Configure Flatland world and layer files.
- Manage Flatland layers.
- Configure Flatland models and their plugins.
- Use plugins such as the ones in the robot from this package.
- Create two separate Nodes that communicate with each other.
- Control a robot in ROS 2 using the keyboard.

The concept of layers makes Flatland essentially a 2.5D simulator since each layer can contain different components of the world that work independently in terms of physics. This means objects in different layers will not collide with each other and a Flatland world can only have up to 16 layers. Managing layers can quickly become a very complex task, even for relatively simple projects. To provide the user some extra help with this task, a Google Slides presentation [8] was created to demonstrate a method based on Graph Theory to determine the minimum number of layers necessary in a Flatland world and which components need to interact with them. The full presentation is included in Appendix A.6.

5.2.1.4 Self-Learning/Autonomous work

By the end of the tutorial, the user is encouraged to attempt to create a new controller for the robot. As the example package in this tutorial already has a lot of code, it is not very inviting to make changes. To combat this issue, a template package [9] is also provided. This package has a similar setup to the one presented, but the Python controller written for the robot only contains a few commented basic code examples to make the robot:

[8] https://docs.google.com/presentation/d/1KqJDQR_PBaGtS-kA5KgsTbaRThu_uP2U8NJeoSP0GDE/edit?usp=sharing
[9] https://github.com/FilipeAlmeidaFEUP/ros2_flatland_robot_controller

- Move forward.
- Move in a circle.
- Move forward and turn to the right when it finds an obstacle in front of it.

The package also offers other pre-prepared maps along with the maze that the user can choose from, as can be seen in Figure 5.2. Some of these maps were created to resemble real-life scenarios to try to captivate the user's interest in Robotics such as a racetrack [10] and an example of house plans [11]. Instructions on how to use this package are included in Appendix A.3.

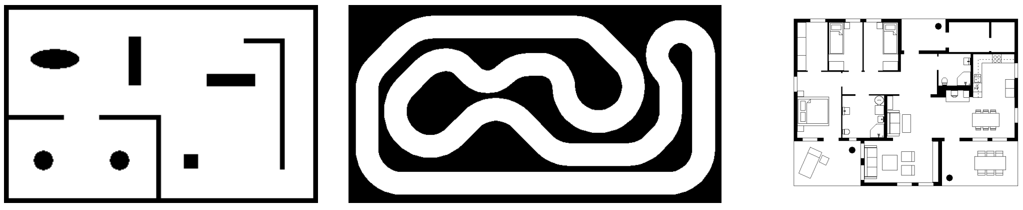

(a) Basic map (b) Racetrack (c) House Plans

Figure 5.2: Example maps on the robot controller template package

5.2.2 Tutorial 2 - Reinforcement Learning

The second tutorial [12] focuses on applying RL to a setup very similar to the first one with some small changes that are also explained. This tutorial serves as an introduction to RL and how to apply it in Robotics scenarios. The full tutorial is included in Appendix A.4. After completing the tutorial, the user should be able to:

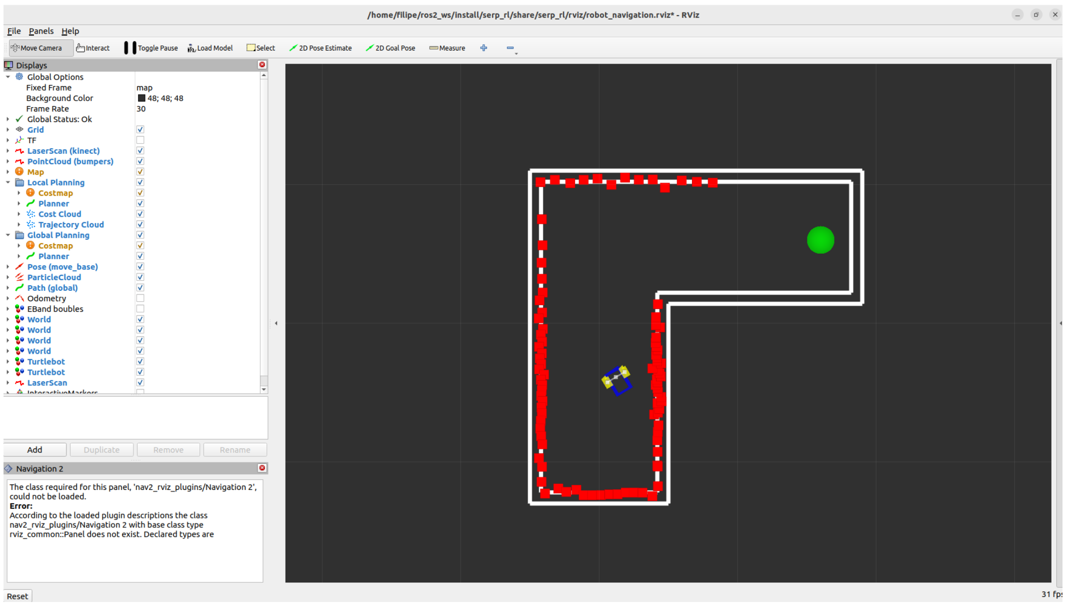
1. Prepare a machine to run a RL system.
2. Analyse the behavior of a RL agent.
3. Modify the learning parameters of a RL agent.

5.2.2.1 Example package

Right after the package is launched, a RL agent immediately starts training to perform a task and the Flatland window will look like Figure 5.3. The robot uses the PPO algorithm to learn how to navigate the hallway turn from one end to the other. The target area is represented by the green circle, which is a new model added that can detect its distance to the robot (equivalent to the target beacon from Section 3.1.1.2). Every time the task is restarted, the initial and final positions swap

[10]`https://ru.pinterest.com/pin/ho-slot-car-racing-ho-slot-car-track-layouts-2-and-4lane-race-tracks--637118678514235810/?amp_client_id=CLIENT_ID(_)&mweb_unauth_id=&simplified=true`
[11]`https://www.roomsketcher.com/house-plans/`
[12]`https://github.com/FilipeAlmeidaFEUP/ros2_flatland_rl_tutorial`

so the robot learns how to turn to both the left and the right. The task is restarted if it fails if there are any collisions, if it takes too much time or if it succeeds by reaching the end. The simulation was also sped up so that it reduces the time needed to learn the task and the user can see the progress more rapidly. The agent follows the same training strategy from the tests done in the initial work, by training in batches until a certain accuracy is reached.



Global Options	
Fixed Frame	map
Background Color	48; 48; 48
Frame Rate	30
Global Status: Ok	
Grid	✓
TF	☐
LaserScan (kinect)	✓
PointCloud (bumpers)	✓
Map	✓
Local Planning	✓
Costmap	✓
Planner	✓
Cost Cloud	✓
Trajectory Cloud	✓
Global Planning	✓
Costmap	✓
Planner	✓
Pose (move_base)	✓
ParticleCloud	✓
Path (global)	✓
Odometry	☐
EBand boubles	☐
World	✓
World	✓
World	✓
World	✓
Turtlebot	✓
Turtlebot	✓
LaserScan	✓

The class required for this panel, 'nav2_rviz_plugins/Navigation 2', could not be loaded.
Error:
According to the loaded plugin descriptions the class nav2_rviz_plugins/Navigation 2 with base class type rviz_common::Panel does not exist. Declared types are

Figure 5.3: Robot inside a hallway turn in the Flatland window from tutorial 2

5.2.2.2 Theoretical concepts

This tutorial explains several RL concepts, such as the agent, the environment, the action space, the observation space, the state, the reward function, the step and the episode. It also explains the importance of each one and how they all need to be used together to create a RL system.

5.2.2.3 Practical concepts

This tutorial shows the user how to:

- Create a class that inherits from the Env class from the Gym Python package to create a standard RL environment.
- Initialize the action and observation spaces and the state.
- Override the step, reset, render and close functions from the Env class to define the behavior of the agent and the environment.
- Use the environment class created to run any RL algorithm from the Stable-Baselines3 package.

- Test a trained agent.
- Save and load the weights of a trained agent.
- Define a training strategy to improve the chances of the agent learning the task.

5.2.2.4 Self-Learning/Autonomous work

During the tutorial, the users are presented with the following challenges:

- Trying to improve the reward function to make the agent learn even faster.
- Test different RL algorithms from the ones available in the Stable-Baselines3 package.
- Developing their own RL projects and experimenting with different maps, setups or tasks.

5.2.2.5 RL Setup

As this is meant to be an introductory lesson, the RL setup is a simplified version of the one developed in Section 3.1.2 that also ensures the agent is capable of learning the task. The PPO agent uses the Multilayer Perceptron (MLP) Policy [13] and has a learning rate of $3e{-}4$.

The action space is composed of three different actions presented in Table 5.1. The actions are performed by modifying the robot's linear and angular velocities for a given time period. The actions are:

Table 5.1: Robot actions

Action	**Linear velocity (m/s)**	**Angular velocity (rad/s)**
Move forward	0.5	0
Rotate left	0	1.57
Rotate right	0	-1.57

The observation space is given by the robot's LiDAR sensor. A similar sampling from the one depicted in Figure 3.3, which proved to be effective, was used but, since the task is simpler and this is an introductory tutorial, there is no need to make the code more complicated than necessary.

The sampling used consisted in simply dividing the LiDAR into 9 equal sections, and from each, we get the closest reading. This means that the observation space has a size of nine 9, with each value representing the closest obstacle in one of the sections. Figure 5.4 illustrates the sampling process.

The reward function is also a simplified version of previous examples (Section 3.1.2.3). The function created is presented in Algorithm 3 and respects the following rules were applied:

[13] https://stable-baselines.readthedocs.io/en/master/modules/policies.html#mlp-policies

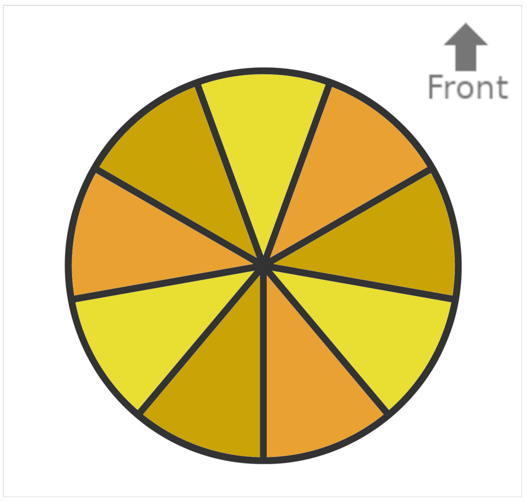


Figure 5.4: Visual representation of the LiDAR sampling strategy in the RL tutorial

- Give a large negative reward to any collisions(line 2).
- Give a large positive reward for reaching the target. The reward reflects how fast the robot reached the target (line 4).
- Give a large negative reward for exceeding the maximum time to reach the goal (line 6).
- Give a small positive reward for moving forward (line 8).

Algorithm 3 Reward function algorithm

```
Input:
    $action$ - Last action performed
    $n\_steps$ - Current step
    $max\_steps$ - Max number of steps in an episode
1:  if $collided()$ then
2:      $reward \leftarrow -200$
3:  else if $reachedDestination()$ then
4:      $reward \leftarrow 400 + max\_steps - n\_steps$
5:  else if $n\_steps > max\_steps$ then
6:      $reward \leftarrow -(300 + forward\_reward)$
7:  else if $action = FORWARD$ then
8:      $reward \leftarrow basic\_reward$
9:  else
10:     $reward \leftarrow 0$
11: return reward
```

5.2.2.6 Expected Results

To make sure this setup was able to learn the task, the agent was tested three times and, each time, the number of steps required and the final accuracy was retrieved. The training was done in batches of 5000 steps and tested for 20 episodes each time until an accuracy of 0.8 was achieved. The results of those tests are in Table 5.2 and since the simulation is sped up 10 times (1 step $= 0.1/10 = 0.01$ seconds), demonstrate that the agent is capable of training in an acceptable time for the user to wait.

Table 5.2: Results for the RL algorithm

Test Number	Train steps	Accuracy
1	35000	0.8
2	20000	0.8
3	45000	1.0

5.2.3 Tutorial 3 - Artificial Potential Fields

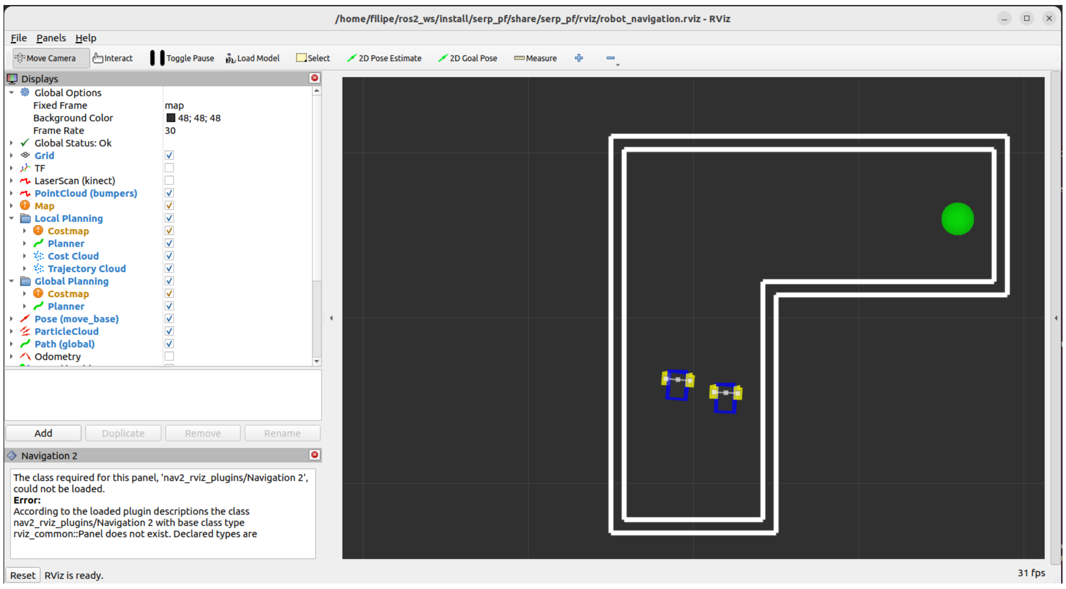
The third tutorial [14] shows how the APF algorithm can also be used to perform the same task from the second tutorial. Additionally, instead of using just one robot, another one was added and both need to travel side by side. The goal is to present a reactive algorithm, which is another viable strategy for path planning, as an alternative to RL while also presenting some other different details about this package that were required for this new setup. The full tutorial is included in Appendix A.5. After completing the tutorial, the user should be able to:

1. Understand and analyze the behavior of the APF algorithm.
2. Setup their own map to be used with the APF algorithm.
3. Modify the intensity of the charges to adjust the APF algorithm behavior.

5.2.3.1 Example package

When running the package, the user will see a Flatland window that looks like Figure 5.5. Just like in tutorial 2, the start is at one end of the hallway and the target position is on the other, but now there are two robots traveling side by side. Collisions or reaching the end, will reset the environment and switch the starting and target positions. Since APF is a reactive algorithm, the robot does not need to train and is instantly able to perform the task. The APF algorithm setup is exactly the same as the one presented in Section 4.1.3.

[14] https://github.com/FilipeAlmeidaFEUP/ros2_flatland_pf_tutorial



```
Global Options
  Fixed Frame          map
  Background Color     48; 48; 48
  Frame Rate           30
Global Status: Ok
Grid                   ✓
TF
LaserScan (kinect)
PointCloud (bumpers)   ✓
Map                    ✓
Local Planning         ✓
  Costmap              ✓
  Planner              ✓
  Cost Cloud           ✓
  Trajectory Cloud     ✓
Global Planning        ✓
  Costmap              ✓
  Planner              ✓
Pose (move_base)       ✓
ParticleCloud          ✓
Path (global)          ✓
Odometry
```

```
The class required for this panel, 'nav2_rviz_plugins/Navigation 2',
could not be loaded.
Error:
According to the loaded plugin descriptions the class
nav2_rviz_plugins/Navigation 2 with base class type
rviz_common::Panel does not exist. Declared types are
```

```
RViz is ready.
```

Figure 5.5: Two robots inside a hallway turn in the Flatland window from tutorial 3

5.2.3.2 Theoretical concepts

The tutorial explains the basic functioning and purposes of the APF algorithm. It presents Coulomb's law (Equation 4.10) that can be used to calculate the forces exerted by repulsive (obstacles) or attractive (target position) charges at any point of the map. It also shows how tension points can be created using Hooke's law (Equation 4.13). It explains how Hooke's law is better to implement a following behavior since, unlike Coulomb which is an inverse-square law, Hooke is linear and increases with the distance.

5.2.3.3 Practical concepts

This tutorial shows the user how to:

- Use local Python packages in ROS 2 by adding them to the setup file.
- Use callback methods with multiple arguments for subscribing to Topics in ROS 2.
- Represent the layout of a map with walls as obstacles for the APF algorithm by creating charges in the shape of line segments.
- Determine the forces generated by line segment charges at any point of the map.
- Use both robots as a singular system to travel to the target position side by side.
- Use tension points to implement a following behavior in a robot.
- Manage the movement limitations of a differential drive robot to still be able to use the APF algorithm.

5.2.3.4 Self-Learning/Autonomous work

At the end of the tutorial, the user is presented with the challenge of trying to create a new map. This means creating a whole new charge layout and adjusting the intensity values, testing how well the tutorial was able to explain the concept behind the APF algorithm.

5.2.3.5 Expected Results

By running the package without making any changes, the user is expected to see results similar to the ones in Table 5.3. Each of the turns was tested 20 times and the accuracy and adjacency indicators (same indicators from Chapters 3 and 4) were retrieved, averaging all the values.

Table 5.3: Results for the APF algorithm

Map	Accuracy	Adjacency
Right turn	1.0	0.62
Left turn	1.0	0.54

In this specific scenario, the algorithm presents very good results, which might lead the user to think that is clearly better than using RL. So the tutorial also points out some of the problems with APF relative to RL, such as requiring perfect knowledge of the map, the robots' positions and target location and the need to create a whole new charge layout for every new map.

5.3 Results and Discussion

To ensure that the kit is effective at teaching the subjects it covers, the solution proposed is to perform user tests. In these tests, subjects with different backgrounds, but all with some programming knowledge, were asked to complete the tutorials and then answer a form about their experience. As testing the entire learning kit would be very time-consuming and finding test subjects that are available and meet the requirements can be difficult, tutorial 3 was not used in the tests. Tutorials 1 and 2 were considered more important since they are the ones that are essential for the subject of Robot Learning, while tutorial 3 provides only complementary information. At the end of the form, the user is asked to answer a short quiz with questions about the different theoretical and practical concepts presented by the tutorials.

The form was created using Google Forms [15] and mostly asks the subjects for quantitative feedback, by evaluating a statement in a scale from 1 to 7. Subjects were also asked if there were any problems running the packages and were given the option of leaving qualitative feedback in the form of open-ended questions. This will serve to retrieve data that can help understand what

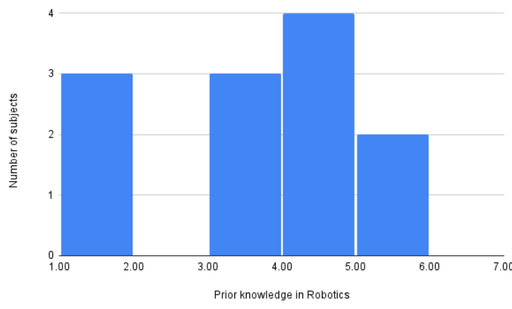
[15] https://docs.google.com/forms/d/e/1FAIpQLSf4Tfr1wfZtkZ4nsdfKQ6DLCVWZZcqDTL3W1yjLbXKUXAWSFQ/viewform?usp=sf_link

features of the kit are useful and what changes should be made. The full form is included in Appendix B.1 and the quiz in Appendix B.2. All the answers for both are in Appendix B.3.

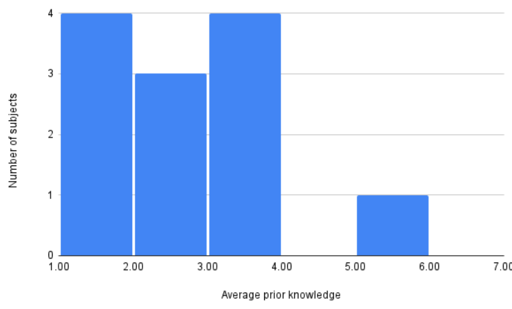
The user tests were performed by a total of 12 participants. Out of those 12, only one did not use the VM provided to follow the tutorials and claimed to have no problems installing the setup or running the packages. The following sections will describe and analyze the results obtained. Since this kit has a similar approach as the one developed by Ventuzelos *et al.*[62], comparisons between results will be made whenever possible.

5.3.1 Prior Subject Knowledge

To have a better grasp on where each subject stood in terms of knowledge on each of the relevant topics for the tutorials, they were asked to provide quantitative data that evaluates their experience before completing the tutorials on each of the following fields: Robotics, ROS 1, ROS 2, Robotic Simulators, Flatland and RL. The distribution of subjects according to their prior knowledge in Robotics and on average can be observed in the histograms from Figure 5.6. From the histograms it is possible to infer that most of them estimate themselves to have a medium to high knowledge of Robotics (M = 3.17 and SD = 1.47) but those values decrease when all the topics approached are considered (M = 2.64 and SD = 1.39). This is probably because Robotic applications do not necessarily imply the usage of all the other components of the kit which demonstrates the diversity of approaches in the Robotics field.



(a) Distribution per prior knowledge in Robotics

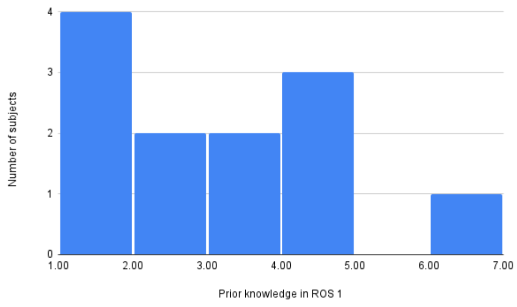


(b) Distribution per average prior knowledge

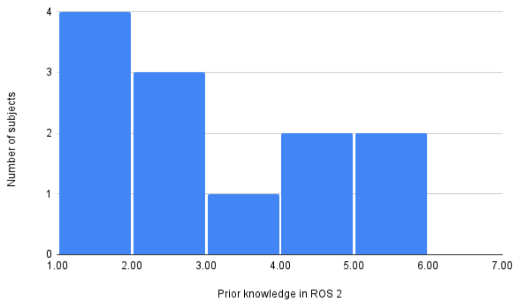
Figure 5.6: Histograms of the distribution of test subjects on their prior knowledge

Although lower prior experience, even among people with knowledge in Robotics, is expected for some of the less popular components, such as Flatland, ROS is meant to be used as a unifying structure for Robotic applications [46, 38]. Despite that, histograms from Figure 5.7 show that prior knowledge for both ROS 1 (M = 2.67 and SD = 1.61) and ROS 2 (M = 2.58 and SD = 1.56) is considerably lower than for Robotics in general 5.6a. This demonstrates that the adoption of ROS is not very high despite the crucial role it has in Robotics.

As the user tests for the kit developed by Ventuzelos *et al.* split their subjects into beginners and experts, the same was made in this work to facilitate the comparison. To establish the experience of the subjects, an expertise score that enhances the most relevant topics was determined for each one.



(a) ROS 1



(b) ROS 2

Figure 5.7: Histograms of the distribution of test subjects on their prior knowledge in ROS

This score was calculated based on prior knowledge of Robotics, ROS 2, ROS in general, Flatland, RL and overall knowledge, by order of importance. Algorithm 4 demonstrates the process of calculating this score for each subject and the resulting values (M = 2.69 and SD = 1.38) are in table 5.4 in descending order.

Algorithm 4 Function to determine the expertise score for each subject

```
Input:
    $robotics$ - Robotics prior knowledge
    $ros1$ - ROS 1 prior knowledge
    $ros2$ - ROS 2 prior knowledge
    $simulators$ - Robotic Simulators prior knowledge
    $flatland$ - Flatland prior knowledge
    $rl$ - RL prior knowledge

1: $overall \leftarrow mean(robotics, ros1, ros2, simulators, flatland, rl)$
2: $ros \leftarrow max(ros1, ros2)$
3: $robotics\_ros \leftarrow mean(ros, robotics)$
4: $expertise\_score \leftarrow mean(robotics, ros2, flatland, rl, ros, robotics\_ros, overall)$
5: return $expertise\_score$
```

Based on the scores calculated, subjects were divided into beginners and experts. To make this decision, several division criteria were proposed. The first was to divide using the threshold of 3.5 which is half of the maximum possible score (7). The second one was to divide the subjects into 2 equal groups, meaning the 6 higher scores are experts and the rest are beginners. The third option was to use the mean of the scores (2.69) as the threshold. Table 5.4 shows the divisions on each of the methods and Table 5.5 shows data of both groups created with each method.

Using a threshold of 3.5, despite there being a big difference in the mean expertise score for beginners and experts (2.25 and 4.87), which reflects a big knowledge difference between them, creates very uneven groups in terms of size. Since it only assigned 2 subjects as experts, the sample size of data would be too small for any valuable comparison with beginners, which was the main reason why it was discarded. Among the other two options, the subject distribution and the mean scores are very similar in the two groups but the SD is lower in the method that uses the mean

Table 5.4: Subjects expertise scores and their evaluation (beginner or experienced) based on different criteria

User	**Score**	**Division at 3.5**	**Division in half**	**Division at M**
1	5.76	Expert	Expert	Expert
2	3.98	Expert	Expert	Expert
3	3.33	Beginner	Expert	Expert
4	3.26	Beginner	Expert	Expert
5	3.00	Beginner	Expert	Expert
6	2.83	Beginner	Expert	Expert
7	2.79	Beginner	Beginner	Expert
8	2.24	Beginner	Beginner	Beginner
9	1.88	Beginner	Beginner	Beginner
10	1.17	Beginner	Beginner	Beginner
11	1.00	Beginner	Beginner	Beginner
12	1.00	Beginner	Beginner	Beginner

Table 5.5: Number of subjects, M and SD of the beginner and expert groups generated with different criteria

Division criteria	**Beginners**			**Expert**		
	Number	M	SD	Number	M	SD
Division at 3.5	10	2.25	0.93	2	4.87	1.26
Division in half	6	1.68	0.74	6	3.69	1.09
Division at M	5	1.46	0.57	7	3.56	1.05

score as the threshold, meaning that it provides better clusters. It is important to note that the SD in the expert groups are always larger due to an outlier score much higher than the average (5.76). Additionally to that, the difference between the lowest expert and the highest beginner scores in this method ($2.79 - 2.24 = 0.55$) is much larger than in the result of splitting the subjects into two groups of the same size ($2.83 - 2.79 = 0.04$), providing a much clearer division point. These reasons resulted in the choice of using the mean of all scores as a threshold to create the expert and beginner groups that were used in any further analysis. The knowledge on average for each of the groups by topic can be observed in Figure 5.8.

5.3.2 Prior and Gained Knowledge comparison

The subjects are also asked to provide quantitative data on how much knowledge they perceived to gain after completing the tutorials in all the same topics from the questions in Section 5.3.1, with the exception of ROS 1 that was not used in the kit. Figures 5.8 and 5.9 show a direct comparison between the previous and gained knowledge that was perceived by the subjects. From these graphs, it is possible to conclude that:

- The overall perceived gained knowledge for experts (M = 3.77 and SD = 1.49) has a higher SD than the one for beginners (M = 3.32 and SD = 1.21). This is probably due to the value of the kit depending on the type of prior experience in Robotics for expert users.
- Flatland had the lowest prior experience on average and had one of the highest scores in gained knowledge. This is expected since it had a lot more room for improvement.
- Unlike Flatland, RL, which also had one of the lowest prior knowledge, had the lowest scores in gained knowledge.
- Both Robotics and Robotics simulators scored some of the lowest values in perceived gained knowledge. This might be due to being more generic topics and having a wider range of material to be explored.
- The highest gained knowledge overall was for ROS 2. One of the main contributing factors for this probably was the animations of the ROS 2 structure, provided by the documentation, that are recommended by the first tutorial.

Despite these small details, results for the perceived gained knowledge are overall (M = 3.58 and SD = 1.24) very positive. On average, both beginner and expert subjects seemed to retrieve some value out of the tutorials in all the topics addressed.

One other important detail to point out is that, somewhat surprisingly, expert subjects claim to gain more knowledge than beginners. This is counter-intuitive since it was to be expected that beginners would have more to learn, therefore retrieving more value from the kit. The graph in Figure 5.10 demonstrates a possible explanation. It is possible to observe that the gained knowledge seems to peak for users with a medium prior knowledge level, which is where a lot of the

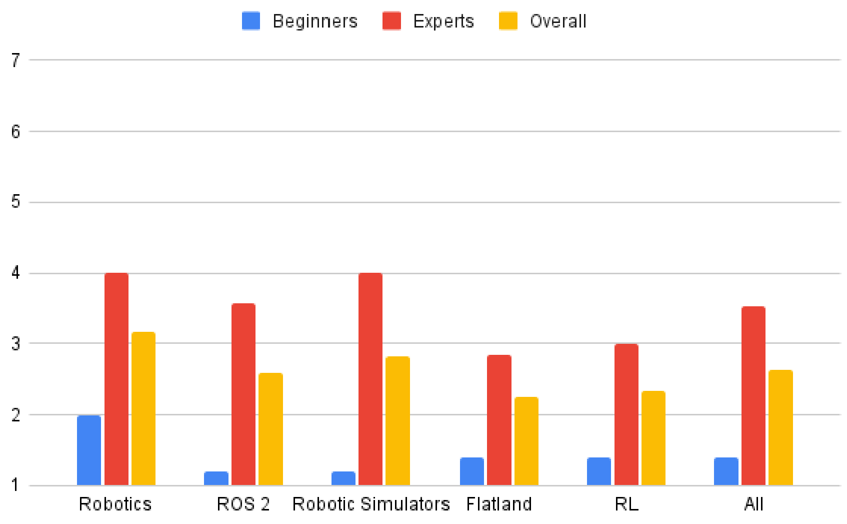


Figure 5.8: Bar graph of the prior knowledge by topic

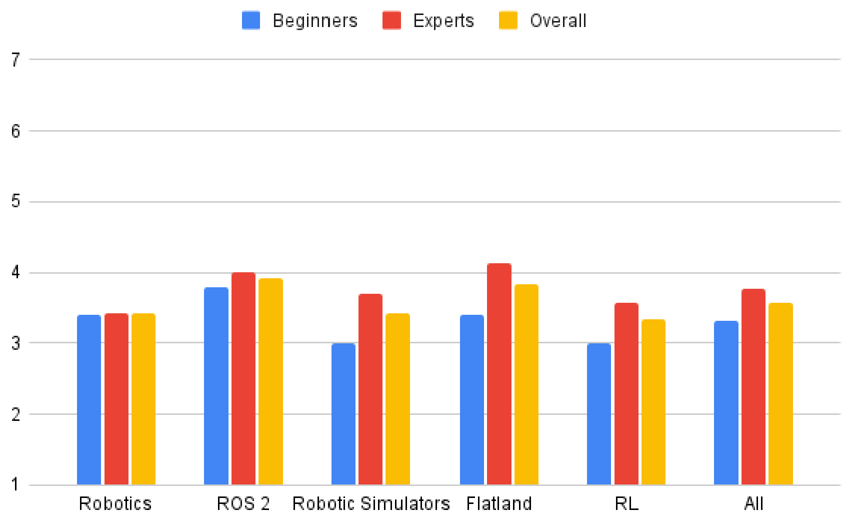


Figure 5.9: Bar graph of the perceived gained knowledge by topic

expert users fall under, and decreases outside that range. This appears to point out that the subjects who can retrieve the most value from the kit are the ones with some prior experience in Robotics since it helps to understand the topic presented on first contact while still having a lot to learn.

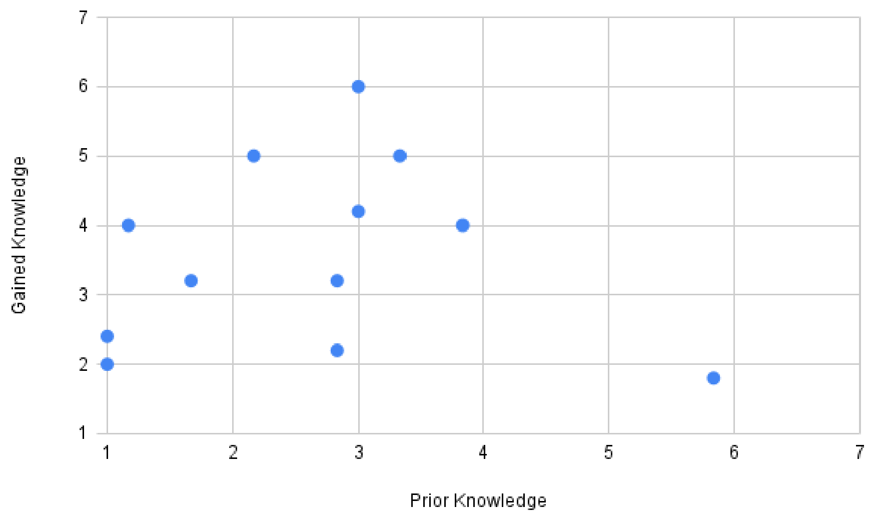


Figure 5.10: Plot graph comparing the prior and perceived gained knowledge of all subjects

5.3.3 Learning Outcomes

Regarding the learning outcomes presented in Sections 5.2.1 and 5.2.2, subjects were asked to give quantitative feedback on each of them. The results from the learning outcomes for the Teleop keys tutorial are in Table 5.6 and Figure 5.11 and the ones for the RL tutorial are in Table 5.7 and Figure 5.12. Some conclusions can be drawn from these results:

- Only 1 out of the 12 subjects claimed to have problems running the packages. This is consistent with the overall positive results obtained on the outcomes.
- There seems to be no significant distinction between the two groups, showing the tutorials can be followed by both beginners and experts.
- The SD values appear to be low in most cases, with the only notable exception being the first on the Teleop keys tutorial, relative to the preparation of the ROS 2 and Flatland package
- The outcomes that seem to score the lowest on average are the ones that require the subject to modify the code. As they require more effort, it is possible that better guidance in the tutorials could help.

As the analysis in this section is very close to the one done by Ventuzelos *et al.*, using the same learning outcomes, a comparison was made with their results. Both sets of experiments present positive values with high means and low SDs. In both scenarios, prior knowledge of Robotics

Table 5.6: Results for the learning outcomes for the Teleop keys tutorial

Group	**Value**	**Learning Outcome**			
		1	**2**	**3**	**4**
Beginner	M	5.60	5.20	5.00	4.00
	SD	1.34	1.10	1.00	0.71
Expert	M	5.57	5.43	5.00	5.00
	SD	1.51	1.13	1.15	1.29
Overall	M	5.58	5.33	5.00	4.58
	SD	1.38	1.07	1.04	1.16

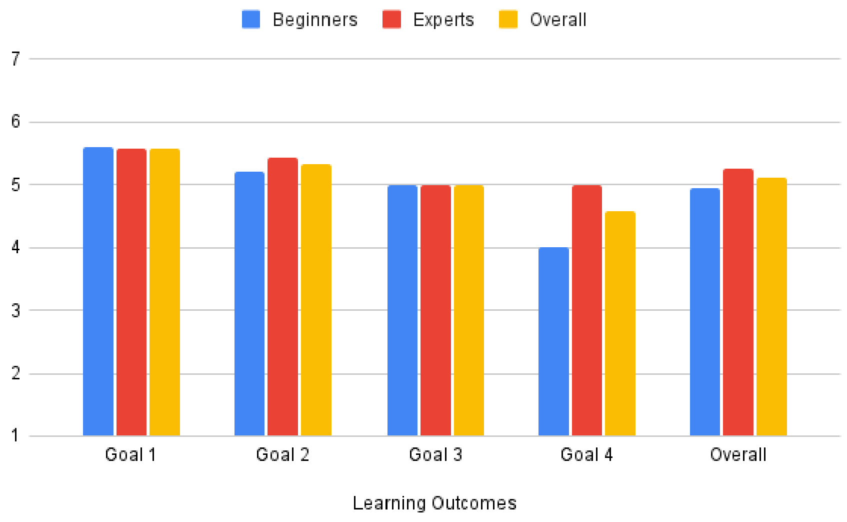


Figure 5.11: Bar graph of the results for the learning outcomes for the Teleop keys tutorial

Table 5.7: Results for the learning outcomes for the RL tutorial

Group	**Value**	**Learning Outcome**		
		1	**2**	**3**
Beginner	M	6.00	5.40	4.80
	SD	0.71	0.89	1.10
Expert	M	5.57	5.14	5.00
	SD	1.27	1.07	1.00
Overall	M	5.75	5.25	4.92
	SD	1.06	0.97	1.00

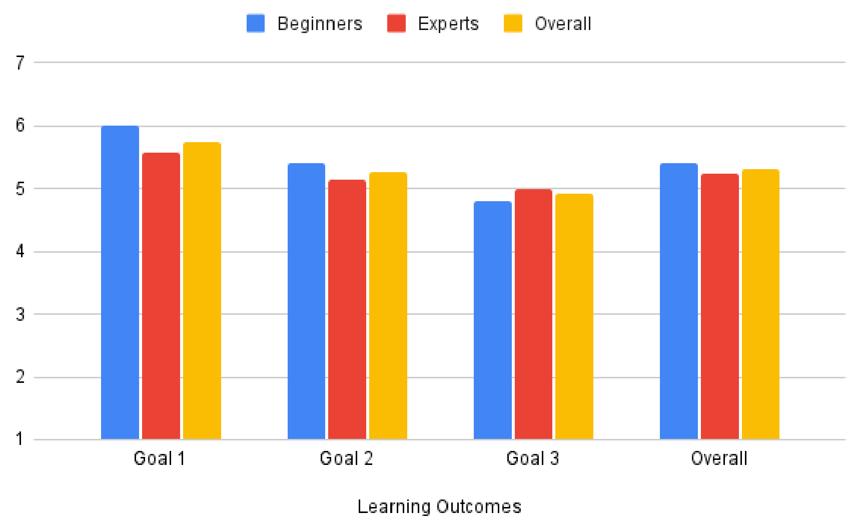


Figure 5.12: Bar graph of the results for the learning outcomes for the RL tutorial

seems to not have any considerable influence on the results. Ventuzelos *et al.* claimed that there were problems for the subjects in setting up the RL environment, causing worse results in some of the learning outcomes. This kit, on the other hand, takes advantage of a simpler and more intuitive RL framework which seems to resolve most of these issues.

5.3.4 Difficulty and Explanation Quality comparison

Another part of the quantitative data collected was regarding the more theoretical concepts present in the tutorials. For the main components of the kit, ROS 2, Flatland and RL, subjects were asked to evaluate how difficult several concepts from each one were. The results for these questions are displayed in Figure 5.13. Additionally, they were asked how well did they perceive those concepts to be explained in the tutorials. The results for these questions are displayed in Figure 5.14. These results show that:

- In terms of difficulty, subjects considered the hardest topic to be RL (M = 3.69 and SD = 1.37) and the easiest to be ROS 2 (M = 3.11 and SD = 1.42). Compared to the perceived gained knowledge, previously explored in Section 5.3.2, subjects claimed that they gained the most expertise in ROS 2 and the least in RL. This demonstrates that the harder the topic is to grasp, the more difficult it is to retain that information from the tutorials.

- Overall beginner subjects seem to perceive the topics as being harder to understand in general (M = 3.61 and SD = 1.67) compared to experts (M = 3.28 and SD = 0.83).

- When analyzing how well the subjects perceived the topics to be explained, there seems to not be any major differences between beginners (M = 4.90 and SD = 0.87) and experts (M = 4.96 and SD = 1.24).

- Overall feedback for how well the topics were explained is very positive (M = 4.93 and SD = 1.06).
- For the explanation quality evaluations, SD was overall low with ROS 2 being the exception (M = 5.00 and SD = 1.41). This is probably due to it being a very extensive topic and the tutorials often linked to its documentation to help explain the concepts. The perception of how well it was explained might depend on the subject's willingness to go through the documentation.
- There seems to be a direct correlation between the overall perceived explanation quality and the perceived gained knowledge, as can be observed in Figure 5.15. The trend line on the graph shows that the better the subjects perceived the information was explained in the tutorials, the more value they claim to have gathered from the kit.

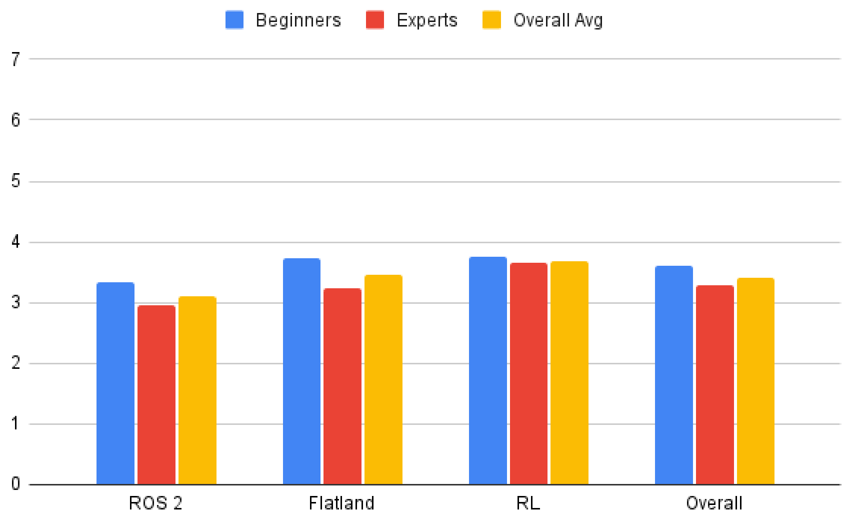


Figure 5.13: Bar graph of how difficult the subjects perceived the topics to be

5.3.5 Qualitative Feedback

The form also asked subjects to give some qualitative feedback in the form of open-ended questions. Some suggestions regarding the tutorials were to try to make them more interactive with videos or games. This seems to be consistent with the previous observation (Section 5.3.2) claiming that animations of the ROS 2 structure provided by the documentation appeared to help its comprehension.

For the Google Slides presentation on how to manage Flatland layers, subjects considered the method presented to be difficult to understand. The slideshow uses a complex layer graph as an example that shows what to do in every possible scenario that can be encountered. Although this more complex scenario is necessary to cover all basis, there should also be a simpler example to provide the viewer with a softer introduction to the method.

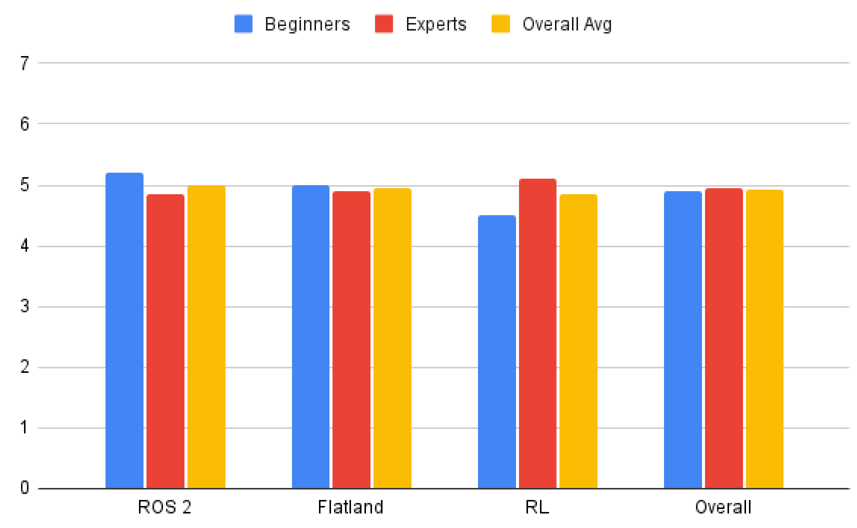


Figure 5.14: Bar graph of how well the subjects perceived the topics to be explained in the tutorials

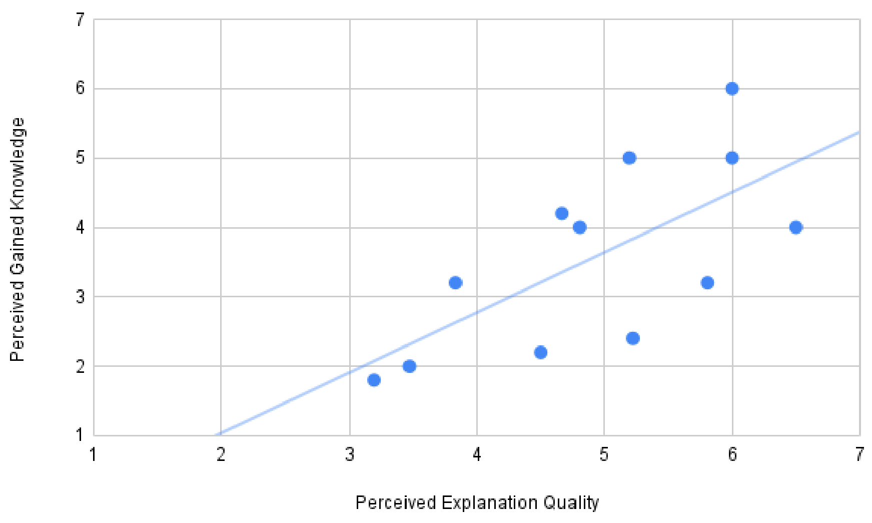


Figure 5.15: Plot graph comparing how well the subjects perceived the topics to be explained in the tutorials with their perceived gained knowledge with a trend line

Subjects that had prior experience in ROS 1 were also asked to give their opinion of what they thought were the main differences when using ROS 2. The ones that also had prior experience with ROS 2 claimed that "ROS 2 is slightly easier and more intuitive from a beginner perspective, however, it is more difficult to transition from ROS 1 to ROS 2" and, for some of them, one of the most valuable new features was being "supported on Windows and macOS". Since the tutorials do not use most of the new ROS 2 features, subjects with no prior experience with it said that it "seemed similar to ROS 1".

5.3.6 Quiz results

To be able to compare the subjects' perception of how much they learned from the tutorials with a more objective evaluation, at the end of the form, subjects are asked to answer a short quiz. The quiz contains a total of 12 multiple choice questions divided into 3 sections, one for each of the major topics, which are ROS 2, Flatland and RL. As the questions are meant to prioritize evaluating comprehension over memorization, the subjects were instructed that consultation is allowed, whether that be the tutorials or any documentation. The results from the answers to that quiz are displayed in Tables 5.8 and 5.9 and in the bar graph in Figure 5.16. The score of each subject is determined by the fraction of questions correctly answered. Some of the main takeaways from these results are:

- Even though the subjects were able to mostly answer correctly, the overall score can still be improved. This means that there is still room to complement the learning process either by revisiting the tutorials or consulting other materials.

- Experts scored much higher on average than beginners on all topics. This demonstrates that prior experience has a lot of influence on these results and beginners may need more time with the kit to be better acquainted with the concepts.

- Beginners also have much higher SD values compared to experts. This is expected since for beginners this is their first experience with most of the components of the kit and their individual scores will greatly depend on their ability to learn on first contact.

- The question that the least subjects were able to correctly answer was the first question on the Flatland section that is related to the concept of layers. As was already mentioned in Section 5.2.1.3, this is one of the most difficult concepts to understand in Flatland.

- Subjects seem to have a good understanding of what is a RL agent is, since they were mostly capable of answering the question about it (RL section, question 4). The question about the RL environment (RL section, question 2) appeared to be much harder to get right. The probable cause for this is that it is possible to interpret the robot in the tutorial as the agent, which helps in understanding it. The most intuitive thing is to equate the environment to the map where the robot is. The issue is that this is not an accurate way of understanding a RL environment.

- Beginners had their highest scores on average on the RL sections while experts performed the best in the ROS 2 section.
- Even though the sample size is small, the distribution of the subjects' scores seems to follow a standard normal distribution. This phenomenon can be observed in the histogram in Figure 5.17.
- In the plot in Figure 5.18 it is possible to observe that the subjects either considered the topics very simple or very hard to understand scored the lowest in the quiz. This probably happens because the ones that considered the concepts easy, might not have actually grasped their complexity, leading to a false sense of confidence, while the ones who considered them very difficult probably struggled to understand them. Subjects with the best scores tend to consider the topics somewhat difficult probably due to better comprehension of their nuances.
- It appears that the subjects were not very effective in self-evaluating how much they learned with the kit. This is possible to verify in the plot graph in Figure 5.19, where the perceived gained knowledge seems to not be correlated with the quiz scores. It is possible that answering the quiz gave subjects more awareness of how much information they retained from the tutorials so this question should also have been asked after.

Table 5.8: Fraction of subjects that correctly answered each question

Topic	**Question**	**Beginners**	**Experts**	**Overall**
ROS 2	Q1	0.60	0.86	0.75
	Q2	0.40	0.86	0.67
	Q3	0.00	0.57	0.33
	Q4	0.00	0.71	0.42
Flatland	Q1	0.20	0.00	0.08
	Q2	0.40	0.86	0.67
	Q3	0.60	1.00	0.83
	Q4	0.20	0.57	0.42
RL	Q1	0.40	0.43	0.42
	Q2	0.00	0.57	0.33
	Q3	0.60	0.86	0.75
	Q4	0.60	0.86	0.75

5.4 Conclusions

As IR appears to be an evergrowing field of research in the present day, the demand for tools and alternatives to teaching it increases. This work set out to develop a kit that could provide a

Table 5.9: Overall results of the quiz scores

Group	Value	Topic			
		ROS 2	**Flatland**	**RL**	**Total**
Beginner	M	0.25	0.35	0.40	0.33
	SD	0.25	0.42	0.29	0.26
Expert	M	0.75	0.61	0.68	0.68
	SD	0.20	0.20	0.24	0.16
Overall	M	0.54	0.50	0.56	0.53
	SD	0.33	0.32	0.28	0.26

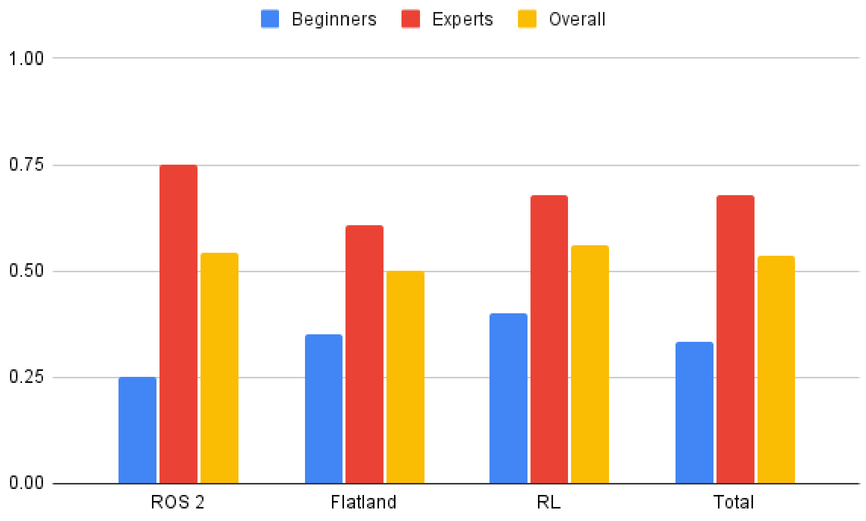


Figure 5.16: Bar graph of the overall scores on the quiz

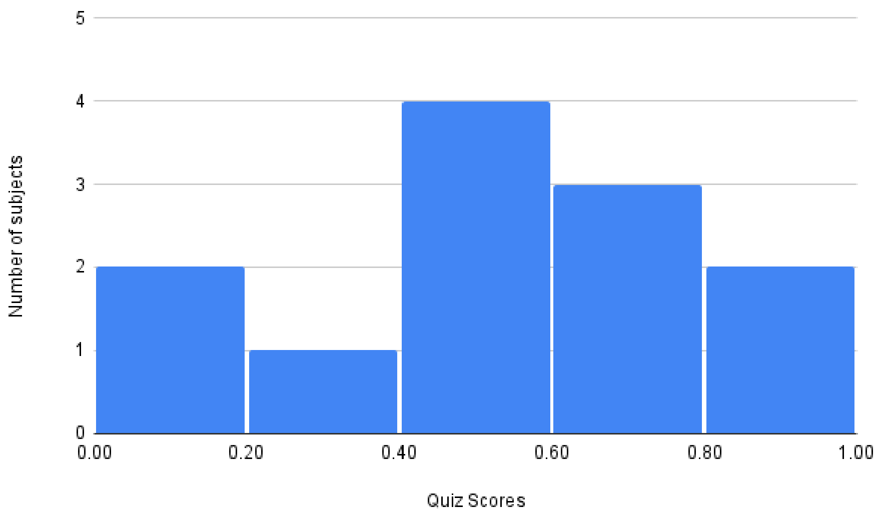


Figure 5.17: Histogram of the distribution of the quiz scores

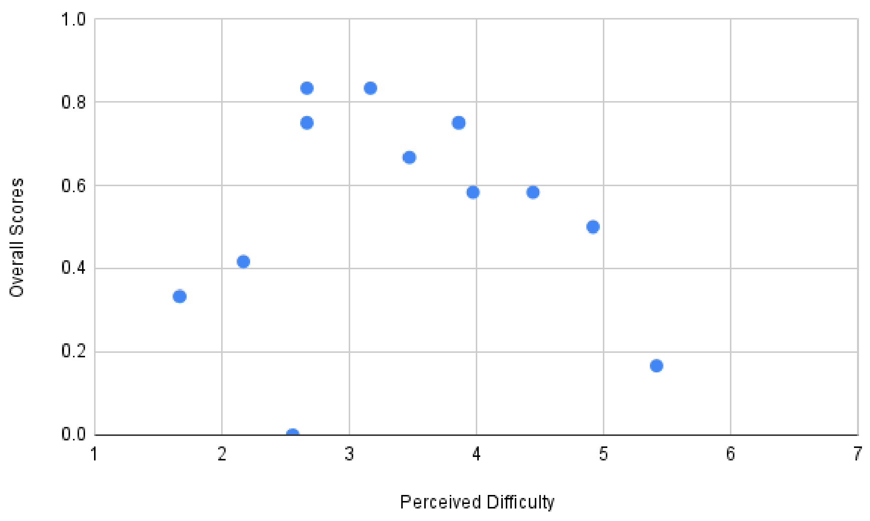


Figure 5.18: Plot graph comparing the overall perceived difficulty and the quiz score of all subjects

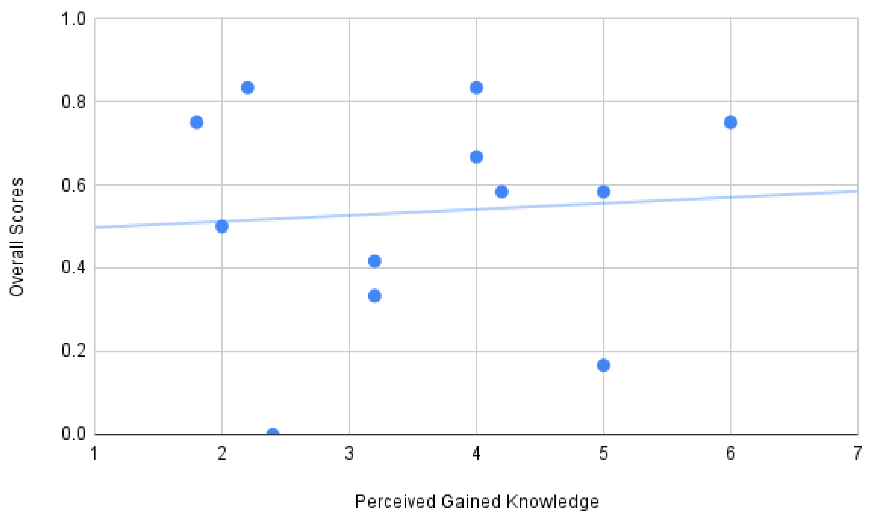


Figure 5.19: Plot graph comparing the overall perceived gained knowledge and the quiz score of all subjects with a trend line

teaching environment for Robot Learning inside a simulator that would benefit users with different backgrounds. The kit was designed to be simple and intuitive to use with its framework only taking advantage of open-source free software, meaning there are no costs associated with its usage. The tutorials that are contained in it provide its users with relevant examples of Robotic applications, such as keyboard control and autonomous navigation using RL or APF, explained in detail how each of them can be achieved.

The user tests were conducted by users with different levels of expertise in Robotics. These tests demonstrated that, although prior experience had an impact on the interaction with it, it was not a deciding factor on whether it was possible to get value from the kit. The main learning outcomes the tutorials were trying to achieve were effective for almost all subjects tested and overall knowledge of practical and theoretical concepts seem to have improved. Although there are still details that could be improved, it is possible to claim that the kit received generally very positive feedback and reached most goals it set out to accomplish.

5.5 Summary

This chapter demonstrated the development of the learning kit. It started by explaining the process of developing the framework and then presented the tutorials developed. Finally, it displayed all the results produced by the user tests performed on the kit and concluded that the feedback was very positive overall.

Chapter 6

Conclusions and Future Work

This chapter will present the conclusions relative to this dissertation and answer the research questions proposed in Section 1.4. It will also propose some possible future work to build upon the current state of the kit developed.

6.1 Conclusions

The main goal for this project was to develop a kit that could be used to teach Robot Learning that could be used in various stages of education. As it currently stands, the kit has proven to be effective for users with different backgrounds and levels of expertise. It provides a very detailed introduction to concepts related to Robotics and RL and could serve as a reliable foundation in the process of learning IR. The kit can be used as a self-learning tool, as it is designed in a way that allows its users to follow its tutorials autonomously, but it can also be used to complement existing Robotics courses.

To be able to answer research question 1, test subjects were divided into experts and beginners. Tests showed that prior knowledge of the topics seems to impact the way some of the subjects interacted with the tutorial but also that it was not a relevant factor in how much value they were able to get from the tutorials. For beginners, they serve as an introduction to IR and for more experienced users it is helpful to expand their knowledge in the field.

In regards to research question 2, it seemed clear that, to an extent, self-learning is a viable option for students to be introduced to IR. The tested subjects were able to acquire basic knowledge of many important components of Robot Learning while working autonomously. But it is also important to acknowledge that there is still a need for in-person classes and courses, specifically when it comes to explaining more theoretical concepts.

As discussed in Section 5.3.2, subjects that had a medium prior experience level in the topics covered by the tutorials, seemed to be the ones who benefited the most from the kit. This provides a good answer for research question 3, demonstrating that the kit provides the most value when used to augment the knowledge and connect all the concepts. Although it is perfectly possible to teach Robotics, ROS 2 and RL simultaneously, providing separate introductions for each of the fields appears to be the best course of action for better results.

6.2 Future Work

Despite the positive feedback, several improvements can still be made to the current contents of the kit. Furthermore, the scope of the topics addressed can also be expanded and new features could be added to allow for a better learning process.

Regarding the simulation in Flatland, the working version for ROS 2 used in the kit is currently a separate branch from the original Flatland repository. In the future it would be advantageous if this branch was merged with the main one so that any future updates would be available for the ROS 2 version as well. In that case, installation instructions for Flatland would have to be updated in the tutorials.

Despite the tutorials receiving positive feedback, some modifications could be made to the tutorials to make them more user-friendly. The ROS 2 documentation uses animations to represent its structure, which some test subjects used to improve the learning process. The tutorials could also contain a more appealing interface with more animations and videos or even some interactive exercises.

One type of scenario that was studied in Chapter 4 but never directly discussed in any of the tutorials is the idea of using the simulation to depict real-life Robotic applications. If new tutorials are added in the future, this could be interesting to explore since it could motivate the user by showing real-life scenarios, such as IW or autonomous robot vacuum cleaners. New tutorials could also present more technical challenges in conjunction with providing more appealing scenarios. A lot of other interesting features from Flatland and ROS 2 could be explored but perhaps the area that needs more follow-up after the tutorials is RL. In the introductory tutorial, users are told that knowing how the RL algorithms work is not necessary to build the environment and train the agent, which is possible thanks to the Stable-Baselines3 package providing default policies. To have a complete curriculum in Robot Learning, a more in-depth knowledge of RL is required than the one provided, meaning new tutorials could be created to teach users how different RL algorithms work and how to tune all their parameters and policies.

For now, user tests have been made with a total of 12 subjects. Although a lot of interesting patterns seem to emerge, the sample size is not large enough to be completely sure of some conclusions proposed based on the results. Further testing would be advisable to be able to assess the kit's quality more accurately, including tests for the APF tutorial as well. Since most participants had an education level of a bachelor's degree or above, it would also be important to try to validate the usage of the kit with students below the University education level.

Another important addition to increasing the scope of the kit could be to include a physical robot. This would be very valuable since it would allow for the addition of a tutorial that helps the users transition from simulation to reality. It is possible to either attempt to develop a new robot or to re-purpose a preexisting one, such as the options presented in Section 2.8. In this situation, the robot in the simulation would resemble the real one. One very interesting possibility with this setup would be to prepare a tutorial that would have the user create a real map for the robot, take a picture of that map and use the image to recreate the scenario in Flatland. After using RL to train the robot to perform a task in the simulation, the user would try to test the agent with the physical robot map created.

Appendix A

Robot Learning tutorials

A.1 Flatland Teleop keys Tutorial using ROS 2

This appendix contains the full Teleop keys Tutorial presented in Section 5.2.1 which is also available at https://github.com/FilipeAlmeidaFEUP/ros2_teleopkeys_tutorial.

Videos with examples from this tutorial are available at https://github.com/FilipeAlmeidaFEUP/dissertation_videos/tree/main/teaching_robot_learning_in_ros2/teleop_keys_tutorial.

Flatland Teleopkeys Tutorial using ROS 2

This tutorial will show you how it is possible to use ROS 2 and the Flatland Simulator to create a simulated world with a robot and how to control it using the keyboard. It also briefly explains how these tools function but, if you wish to learn more about them, please visit the ROS 2 Documentation and the Flatland Documentation. All the code in this tutorial is written in python.

Setup and Pre-requisites

Using a VM

To allow for a faster and simpler start, there is a Virtual Machine for VirtualBox with every requirement already installed and ready to build the packages and run the code. To get it running, you can download this file and import it in the VirtualBox Manager window. The next sections will explain how to get this setup on your own machine but if you only want to use the VM for now you can skip to this section.

VM user: ros2

VM password: ros2

NOTE: To make sure you have the most current version of all the tutorials on the machine, you should do `git pull` in the folders of all cloned repositories.

ROS 2 Humble Installation

This tutorial was built to work with ROS 2 Humble which is, at the time of writing, the latest distribution. Follow the instructions in the official ROS 2 installation guide. It is recommended to at least install the desktop version, as some tools that will be used here are already contained within.

Workspace setup

To be able to build all the dependencies and run your projects, ROS needs a designated folder known as the workspace. To set up your workspace follow the official ROS 2 tutorial on how to create a workspace.

You will probably need to install some colcon packages to build the workspace. Follow the proper instructions for your own machine in the documentation.

To make sure you don't need to source the setup.bash script in every new terminal, run these two commands:

```
echo "source /opt/ros/humble/setup.bash" >> ~/.bashrc
source ~/.bashrc
```

Flatland 2

A fork of Flatland modified to work on ROS 2 natively is available on this repository. Follow the installation instructions to have this version available on your machine.

Clone the repository

After everything is set up, run the following commands inside your ROS 2 workspace folder to clone this repository as a package:

```
cd src/
```

```
git clone https://github.com/FilipeAlmeidaFEUP/ros2_teleopkeys_tutorial.git
```

Building and installing dependencies

Before running the package for the first time, you need to execute all these commands in the workspace folder (named "ros2_ws" in the VM):

1. Resolve dependencies: `rosdep install -i --from-path src --rosdistro humble -y`
2. Build the workspace: `colcon build`
3. Source the setup script: `source install/setup.bash`

After the first run, these commands might be useful for this or any other package in your workspace:

- For every new terminal that you open that accesses the workspace, run command 3. As an alternative, you can add the command to .bashrc so that it is automatically executed every time a new terminal is opened. For users running the tutorial in the VM, the commands are:

```
echo "source /home/ros2/ros2_ws/install/setup.bash" >> ~/.bashrc
source ~/.bashrc
```

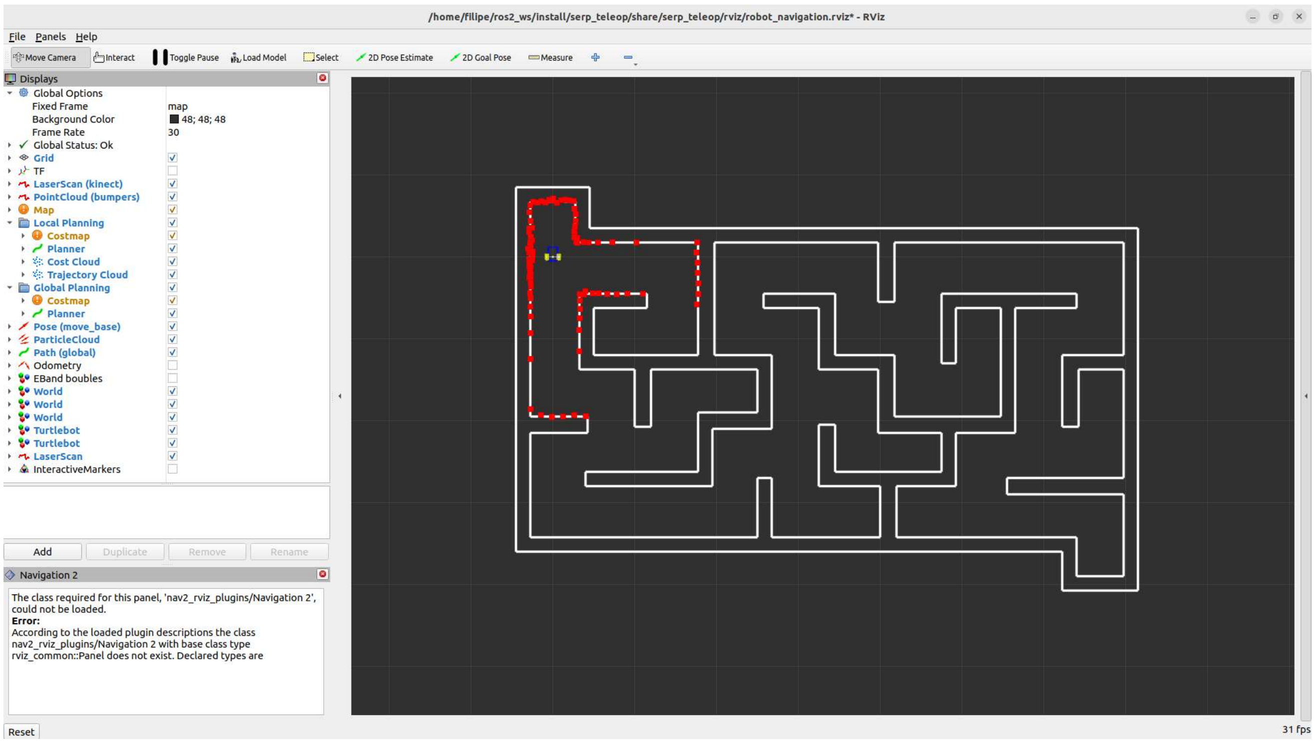
- If you made changes in any file of any package, you need to either build the entire workspace with command 2 or use the same command with arguments to build only specific packages.
- If there are new dependencies on any of your packages (ex.:new python imports), you should run command 1 to make sure all of them are resolved.
- If you are creating a new package or cloning one, run all the commands.

Running the package

Once everything is built, running the package can be done with the command:

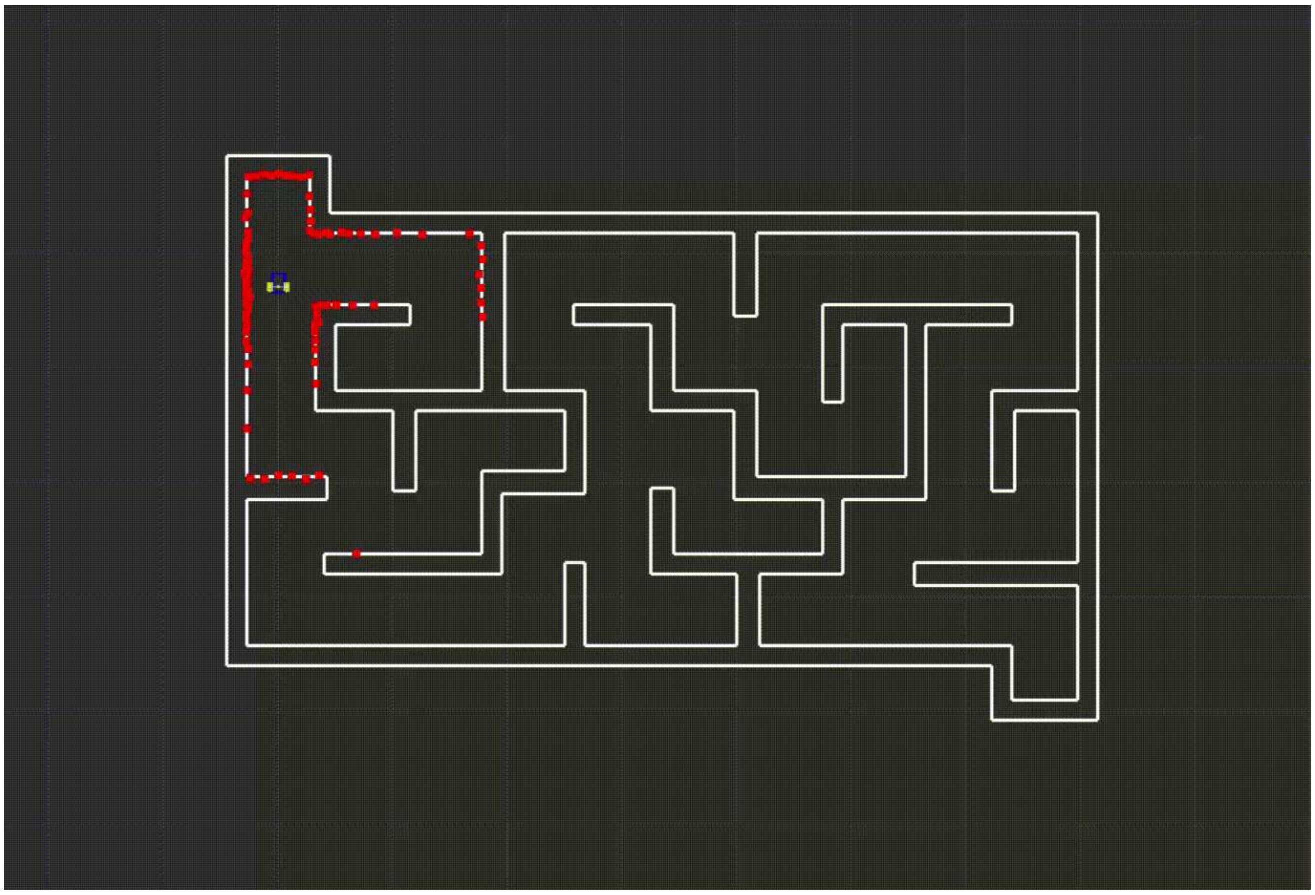
```
ros2 launch serp_teleop serp_teleop.launch.py
```

If everything worked as intended, you should be seeing a window that looks like this:



```
Global Options
    Fixed Frame          map
    Background Color     48; 48; 48
    Frame Rate           30
Global Status: Ok
Grid                     ✓
TF
LaserScan (kinect)       ✓
PointCloud (bumpers)     ✓
Map                      ✓
Local Planning           ✓
    Costmap              ✓
    Planner              ✓
    Cost Cloud           ✓
    Trajectory Cloud     ✓
Global Planning          ✓
    Costmap              ✓
    Planner              ✓
Pose (move_base)         ✓
ParticleCloud            ✓
Path (global)            ✓
Odometry
EBand boubles
World                    ✓
World                    ✓
World                    ✓
Turtlebot                ✓
Turtlebot                ✓
LaserScan                ✓
InteractiveMarkers
```

The class required for this panel, 'nav2_rviz_plugins/Navigation 2', could not be loaded.
Error:
According to the loaded plugin descriptions the class nav2_rviz_plugins/Navigation 2 with base class type rviz_common::Panel does not exist. Declared types are



Right now you are seeing the robot move inside the map built for this package but, although the space is not being used, the world extends beyond the walls on the screen. To navigate through the flatland world use these mouse controls:

- Scroll button + drag: move the window.
- Scroll up/down: zoom in/out.
- Left click + drag: rotate the window.

The window on the left called Displays allows you to control some aspects of the visualization. For example, try to unselect the checkbox called `LaserScan (kinect)` . Once you do, the red squares around the robot will disappear. These squares were representing the collisions from the robot's radar with the walls. Turning this off does not mean the radar is no longer working, only that it is not appearing in the visualization.

For now, the robot can not be controlled with the keyboard. It moves forward until it detects a wall in front of it using its radar. When a wall is detected, it randomly chooses one direction to rotate 90º, essentially following a random path. The next section will briefly explain how a Flatland ROS 2 package is structured. If you wish to skip this part and go directly to the keyboard control, go to this section.

The Flatland ROS 2 package

ROS 2

To help you understand the structure of ROS, take a quick look at the documentation sections Nodes, Topics and Services. You don't need to perform any of the tasks, just read the first few sections in each and watch the animations that are provided.

Launch file

ROS 2 provides the `run` run command that starts one node but, in some cases, like in this package, you might want to start several nodes at once. To do that you can use a launch file such as the serp_teleop.launch.py in this repository. For more information on this type of file, consult the documentation on creating launch files. A total of 4 nodes are launched by this file but, for now, let us focus on only 2:

- The `flatland_server` node: runs the Flatland simulation, including the robot and the world. This node has several arguments that modify the functioning of the simulator. For a more detailed explanation of these parameters go to Flatland documentation on how to launch Flatland server node.

- The `serp_teleop` node: runs the code in the file __init__.py that is used to control the robot. This node needs to have an instance of the class `rclpy.node.Node` created and initialized. This can be done by:

NOTE: From the Flatland parameters, the `update_rate` allows you to change the speed of the simulation with 100 being the regular speed. Keep this parameter in mind, since it will be important in the following tutorial.

Using Topics and Services

Before you can use topics and services, you need an initialized instance of the `rclpy.node.Node` class. This can be achieved in several ways, such as:

1. Using `rclpy.create_node` :

```
node = rclpy.create_node('node_name')
```

2. Creating a class that inherits from the `Node` and accessing as `self` inside the class (used in this package):

```
class NewNode(Node):
    def __init__(self) -> None:
        super().__init__("node_name")
```

Both methods are equally valid but the second one might be better to keep the code organized in more complex projects. For any other code examples in this tutorial, consider the variable `node` as an initialized instance of the `rclpy.node.Node` class.

NOTE: In the following code examples, <Msg_Type> is only a placeholder for these examples and each topic has a different format for its messages. For each case look for the proper documentation to help you.

The `flatland_server` node publishes to several topics the `serp_teleop` node needs to subscribe to. To do so, the following code is necessary for each subscription:

```
#create a subscription
node.create_subscription(<Msg_Type>, "/topic_name", handling_function, queue_size)

#each time a message is published, this function is executed and the arg data is the message as a Msg_Type instance
def handling_function(data):
    [...]
```

The `flatland_server` also subscribes to some topics the `serp_teleop` node will publish to. The following code shows how a message can be subscribed to a topic:

```
#create a publisher
node.pub:Publisher = self.create_publisher(<Msg_Type>, "/topic_name", queue_size)

#call this function to send a message to the topic
msg = <Msg_Type>()
publisher.publish(msg)
```

The `serp_teleop` node also needs to use Flatland services. This code sends a request to a service:

```
client = node.create_client(<Msg_Type>, "/service_name")
client.wait_for_service()
request = <Msg_Type>()
client.call_async(request)
```

ROS 2 provides several commands to help you see details about every communication between nodes inside the ROS 2 platform. With the package running, you can experiment with the following commands:

1. List of active nodes/topics/services:

```
ros2 node list
ros2 topic list
ros2 service list
```

2. Information about all subscribers, publishers and services (servers and clients) of an active node:

```
ros2 node info node_name
```

3. Print all messages published to a topic:

```
ros2 topic echo topic_name
```

RViz file

The window that contains the Flatland worl is configured by the RViz plugin, a visualization tool for ROS in the robot_navigation.rviz file. To keep things simple, you can keep this file mostly unchanged, except when you need to add a new model or a new layer to your world (more about Flatland models and layers in the next section). Inside the list `Visualization Manager/Displays` you need the entries:

```
# for each model in the world
- Class: rviz_default_plugins/MarkerArray
  Enabled: true
  Topic:
    Value: /models/m_<model_name>
  Name: Turtlebot
  Namespaces:
    "": true
  Queue Size: 100
  Value: true

# for each layer in the world
- Class: rviz_default_plugins/MarkerArray
  Enabled: true
  Topic:
    Value: /layers/l_<layer_name>
    Depth: 1
    History Policy: Keep Last
    Reliability Policy: Reliable
    Durability Policy: Transient Local
  Name: World
  Namespaces:
    "": true
  Queue Size: 100
  Value: true
```

Although there is no influence in the physics component of simulation, if a model/layer is not added to this list, it won't appear in the visualization. Some other possible changes to this file are signaled in the file through comments.

Other setup files

Although most setup files don't require a lot of attention and are very simple, there are some configurations that you need to check in your packages.

One of them is to check that you have all the dependencies in the package.xml file.

You also may need to pay some attention to the setup.py file. To help you build this file in future packages look at the example with comments on this project or for a more detailed explanation go to the guide on how to develop a ROS 2 python package.

Flatland

Flatland configures its worlds through the use of YAML files. These files follow a simple structure that should be simple to understand just by looking at the examples provided in this repository. If you need any extra help, take a look at the YAML Syntax.

World file

The world.yaml file is where the Flatland world is configured and is sourced directly by the launch file.

To understand this file, you need to be familiar with the concept of layers. This makes Flatland essentially a 2.5D simulator since each layer can contain different components of the world that work independently in terms of physics. This means objects in different layers won't collide with each other. The world file can configure up to 16 layers.

There is also a list of models that are included in the world. Each one needs a name, the initial position and their own configuration file

For more information on how to configure this file go to the configuring world page of the documentation.

Layer file

In this package, there is only one layer file (maze.yaml). This configuration works by taking an image (maze.png) and using a threshold to turn it into a binary image. It then builds a map by placing walls where the image transitions from 0 to 1 or 1 to 0.

Another possible configuration for this file is to manually define line segments in a .dat file and use it to define the walls of the map.

For more information on how to configure this file go to the configuring layers page of the documentation.

Managing layers can quickly become a very complex task, even for relatively simple projects. If you need extra help managing this task in future projects, this Google Slides presentation (pdf also available in this repository) demonstrates a method that uses graphs to determine the minimum number of layers and which components need to interact with them. The second to last slide shows the layer graph for this project.

Model file

Each model needs their own configuration file. In this package, you can look at the example from the SERP robot (a simple differential drive robot) simulation model in the file serp.model.yaml.

This file starts by defining a list of bodies with predefined shapes or customizable polygons that create the shape of the model. It can also have a list of joints to connect the bodies.

For the model to interact with the world, it needs to configure a list of plugins. Flatland offers several built-in plugins that usually interact with topics. Of those, the SERP model uses:

- Bumper. Detects collisions and publishes them to a topic using flatland_msgs.msg.Collisions.
- Diff Drive. Subscribes to a topic that receives geometry_msgs.msg.Twist that modify the model's velocity.
- Laser. Simulates a LiDAR sensor and publishes the readings to a topic using sensor_msgs.msg.LaserScan.

For more information on how to configure this file go to the configuring models page of the documentation.

Keyboard control

Now it's time to control the robot using the keyboard. Although it is possible to read keystrokes and control the robot all inside the same node, ROS works better if we keep things modular. We'll try to have another node running in parallel reading keyboard inputs and publishing them to a topic.

You can already find a ready-to-use package to do that in this repository. You can follow the instructions there to get it running. It's a very simple package so you can inspect it later and figure out how it works but, for now, all you need to know is that it reads keystrokes from certain keys and it publishes them to the topic '/teleopkeys' as messages of the type String. Here's the list of available keys and the messages they produce.

NOTE: If you are using the VM, this package is already installed.

Once you have the keyboard reading node running, you need to go to the __init__.py file and change this variable to `True` :

```
# if you didn't change anything, this is on line 20
self.use_keyboard = False
```

This will change the flow of the code so that it now will read from the topic '/teleopkeys' and control the robot accordingly. Now all you need to do is to open a new terminal and run the Flatland world again. However, before that, do not forget that you might need to run some of the commands from this section again. In particular, you will need to rebuild the workspace:

```
colcon build
```

If everything is working, you should be able to control the robot using the arrow or the WASD keys. All the plugins are still being used. If you get too close to a wall, a warning message appears on the terminal and the robot moves slower. If you collide with a wall you go back to the original position.

Make your own robot controller

Now that you understand how to use ROS2 and Flatland, you can try to modify the code to develop your own controller for the robot. Since this project is set up to control the robot with the keyboard, the code can be a little confusing to make changes to.

To make it easier for you to get started writing a controller, use this package instead, which is an empty template for a controller with comments in the code to help you. To get it set up and running, just follow the instructions in the repository.

Next steps

Now that you are finished with this tutorial, move on to the next one where you will use a similar setup to teach the robot a simple task using Reinforcement Learning.

A.2 Teleop keys Keystroke Publisher for ROS 2

This appendix contains the instructions to install and use the Keystroke Publisher for ROS 2 (used in the Teleop keys tutorial) which are also available at `https://github.com/FilipeAlmeidaFEUP/ros2_teleopkeys_publisher`.

Teleopkeys Keystroke Publisher for ROS 2

Simple node that detects keystrokes from certain keys and interprets them as commands to move robots. It publishes them to the topic '/teleopkeys' as messages of the type String.

How to Run

To run this note, you need to install ROS 2 (Humble is the latest distribution at the time of writing) and setup the workspace.

After everything is setup, run the following commands inside your ROS 2 workspace folder to clone the repository:

```
cd src/
git clone https://github.com/FilipeAlmeidaFEUP/ros2_teleopkeys_publisher.git
```

Again inside the workspace folder, to build and install the dependencies, run the following commands:

```
rosdep install -i --from-path src --rosdistro humble -y
colcon build
source install/setup.bash
```

To run the node execute:

```
ros2 run teleopkeys_publisher teleopkeys_publisher
```

When the node is running, make sure the keyboard inputs are not blocked by the terminal or other window by de-selecting or minimizing them. Once you start pressing the available keys, the terminal will display messages in the following format:

```
[INFO] [1684004330.241555735] [TeleOpKeys]: Ready to read keyboard inputs.
[INFO] [1684004338.782670910] [TeleOpKeys]: Pressed up.
[INFO] [1684004338.896476342] [TeleOpKeys]: Released up.
[INFO] [1684004339.831320590] [TeleOpKeys]: Pressed left.
```

```
[INFO] [1684004340.002965885] [TeleOpKeys]: Released left.
[INFO] [1684004340.369623949] [TeleOpKeys]: Pressed right.
[INFO] [1684004340.467356998] [TeleOpKeys]: Released right.
[INFO] [1684004340.948777665] [TeleOpKeys]: Pressed down.
[INFO] [1684004348.055793108] [TeleOpKeys]: Pressed q.
[INFO] [1684004348.261234340] [TeleOpKeys]: Released q.
[INFO] [1684004348.830487471] [TeleOpKeys]: Pressed e.
[INFO] [1684004349.011708976] [TeleOpKeys]: Released e.
[INFO] [1684004349.588772532] [TeleOpKeys]: Pressed space.
[INFO] [1684004349.709970417] [TeleOpKeys]: Released space.
[INFO] [1684004350.251092621] [TeleOpKeys]: Pressed ctrl_l.
[INFO] [1684004351.078519518] [TeleOpKeys]: Pressed shift_l.
[INFO] [1684004352.784016082] [TeleOpKeys]: Released shift_l.
[INFO] [1684004353.349115364] [TeleOpKeys]: Released ctrl_l.
```

Available Keys

Not all keys are interpreted by the note. The following table shows the ones available and the messages that are published for each one. Keystrokes from other keys are ignored.

Keys	Pressed message	Released message
Arrow Up / W	`'p_up'`	`'r_up'`
Arrow Down / S	`'p_down'`	`'r_down'`
Arrow Left / A	`'p_left'`	`'r_left'`
Arrow Right / D	`'p_right'`	`'r_right'`
Q	`'p_q'`	`'r_q'`
E	`'p_e'`	`'r_e'`
F	`'p_f'`	`'r_f'`
Space	`'p_space'`	`'r_space'`
Left Shift	`'p_shift_l'`	`'r_shift_l'`
Right Shift	`'p_shift_r'`	`'r_shift_r'`
Left Ctrl	`'p_ctrl_l'`	`'r_ctrl_l'`
Right Ctrl	`'p_ctrl_r'`	`'r_ctrl_r'`
Esc	`'p_esc'`	`'r_esc'`

NOTE: If you want to add more keys, just follow the instructions in the `TODO` comments in the file __init__.py.

A.3 Base robot controller for ROS 2 and Flatland

This appendix contains the instructions to install and use the base robot controller for ROS 2 and Flatland (used in the Teleop keys tutorial) which are also available at `https://github.com/FilipeAlmeidaFEUP/ros2_flatland_robot_controller`.

Videos of the examples provided are available at `https://github.com/FilipeAlmeidaFEUP/dissertation_videos/tree/main/teaching_robot_learning_in_ros2/base_controller`.

FilipeAlmeidaFEUP /
ros2_flatland_robot_controller

Code Issues Pull requests Actions Projects Wiki Security Insights Settings

ros2_flatland_robot_controller / README.md

FilipeAlmeidaFEUP last week

38 lines (25 loc) · 930 Bytes

Preview Code Blame

Base robot controller for ROS2 and Flatland

Base robot controller project for Flatland packages using ROS2.

Inside the file __init__.py there are comments with intructions and code examples on how and where to add your code to create a controller.

Setup the package

Clone the repository inside `workspace/src` :

```
git clone https://github.com/FilipeAlmeidaFEUP/ros2_flatland_robot_controller.git
```

Build inside the `workspace` folder:

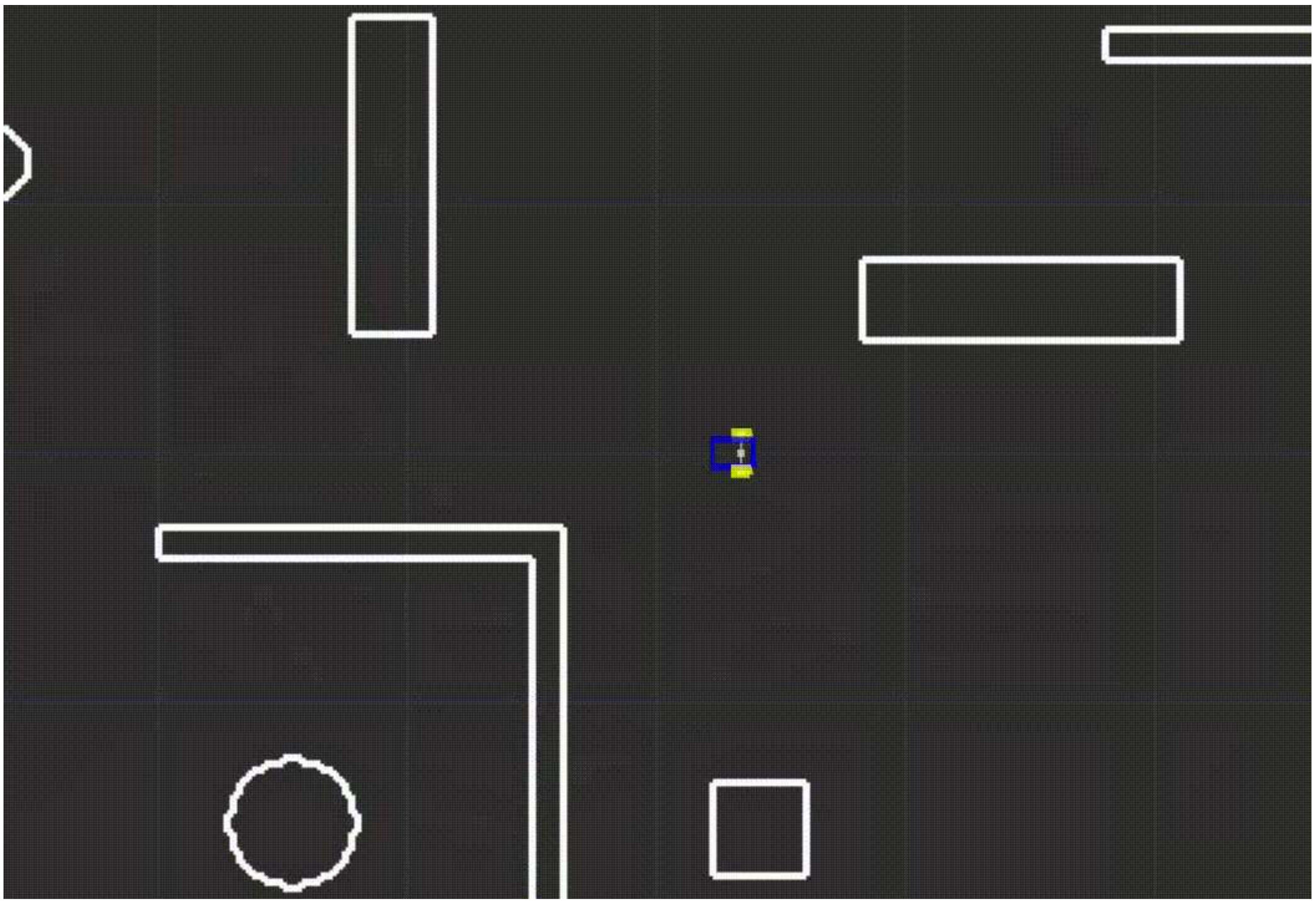
```
rosdep install -i --from-path src --rosdistro humble -y
colcon build
source install/setup.bash
```

Run the package:

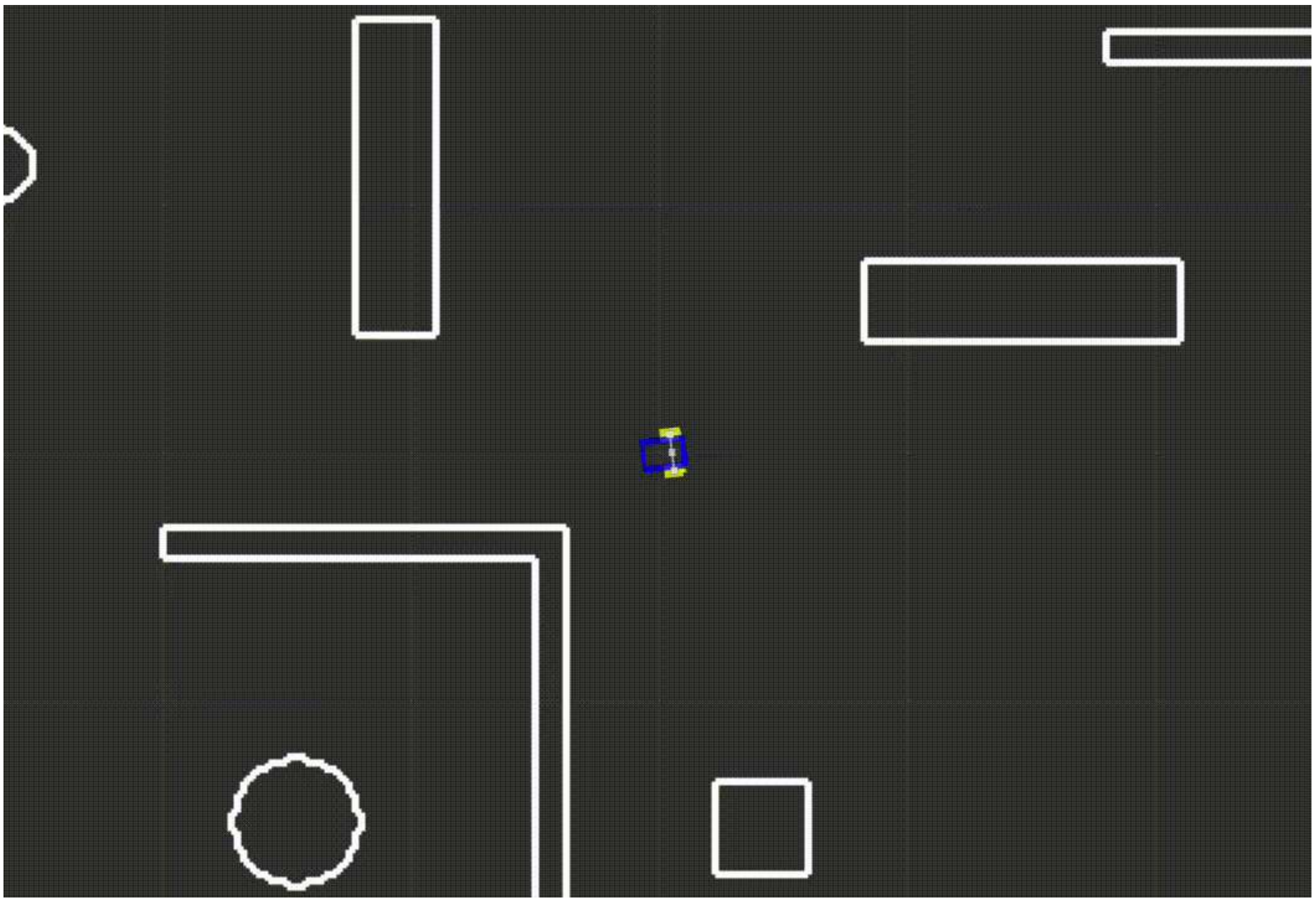
```
ros2 launch serp_controller serp_controller.launch.py
```

Code examples

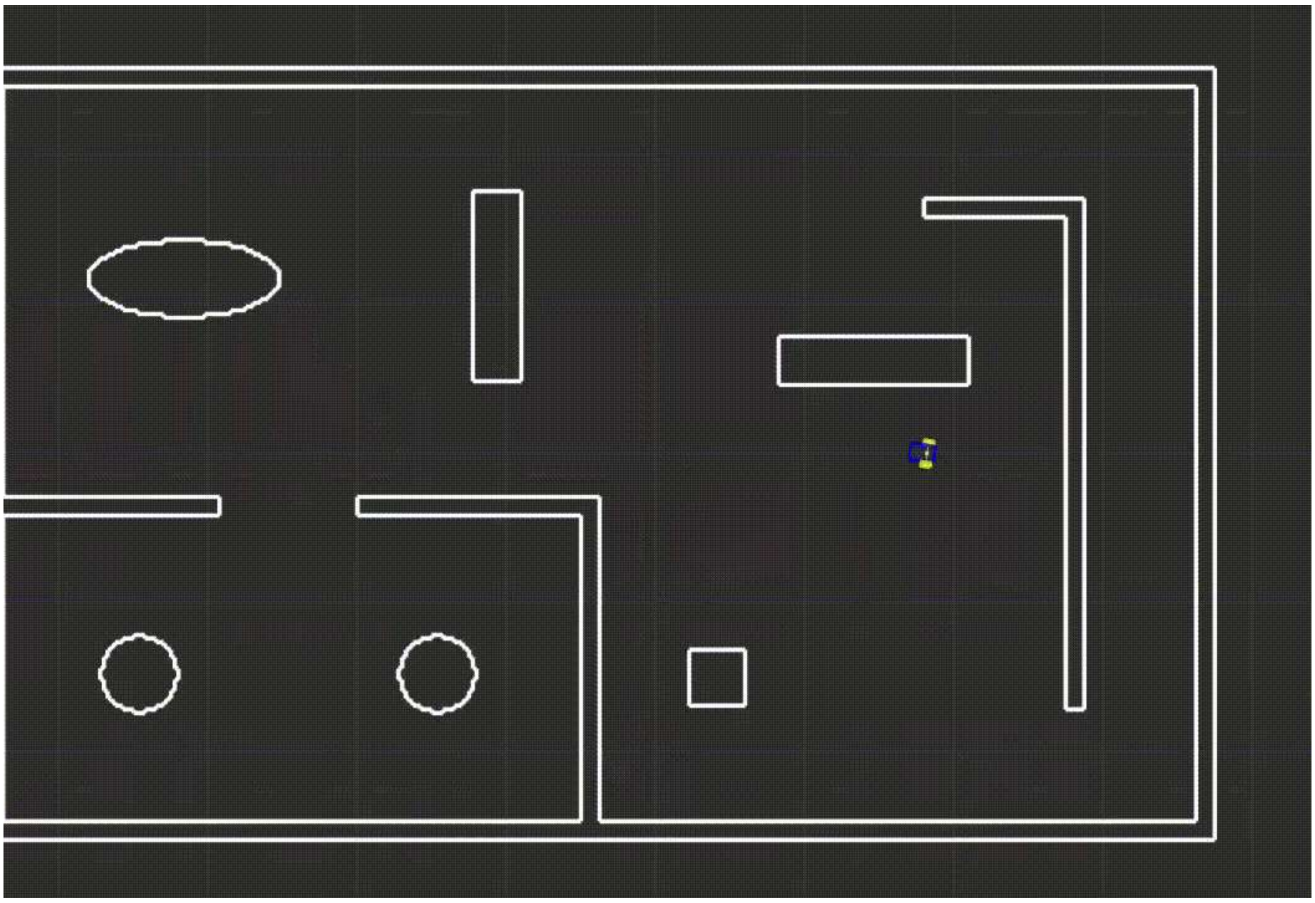
Example 1: A robot that moves forward



Example 2: A robot that moves in a circle



Example 3: A robot that moves forward and turns to the right when it finds an obstacle in front of it



A.4 Flatland Reinforcement Learning Tutorial using ROS 2

This appendix contains the full Reinforcement Learning Tutorial presented in Section 5.2.2 which is also available at `https://github.com/FilipeAlmeidaFEUP/ros2_flatland_rl_tutorial`.

Videos with examples from this tutorial are available at `https://github.com/FilipeAlmeidaFEUP/dissertation_videos/tree/main/teaching_robot_learning_in_ros2/reinforcement_learning_tutorial`.

Flatland Reinforcement Learning Tutorial using ROS 2

The previous tutorial focused on explaining how to use ROS 2 and Flatland to create a robot and control it. In this tutorial you will learn how to use Reinforcement Learning (RL) inside the same setup to teach the robot how to perform a simple task. The packages used for the RL algorithms are the Stable-Baselines3 and OpenAI's Gym.

First Run

Pre-requisites

All the prerequisites are carried over from the previous tutorial. In addition, you also need to install the Stable-Baselines3 package. Follow the installation guide from the documentation. For this example, you can install the most basic version either for Windows or using pip for other OS's.

The Stable-Baselines3 installation should automatically install all missing dependencies, including the Gym package. Nevertheless, pay attention during the installation and make sure there were no errors. If any dependency fails to install try to do it manually.

If you are using the VM provided, the package is ready to run.

Running the code

Clone the repository to your `<ros2_workspace>/src` folder:

```
git clone https://github.com/FilipeAlmeidaFEUP/ros2_flatland_rl_tutorial.git
```

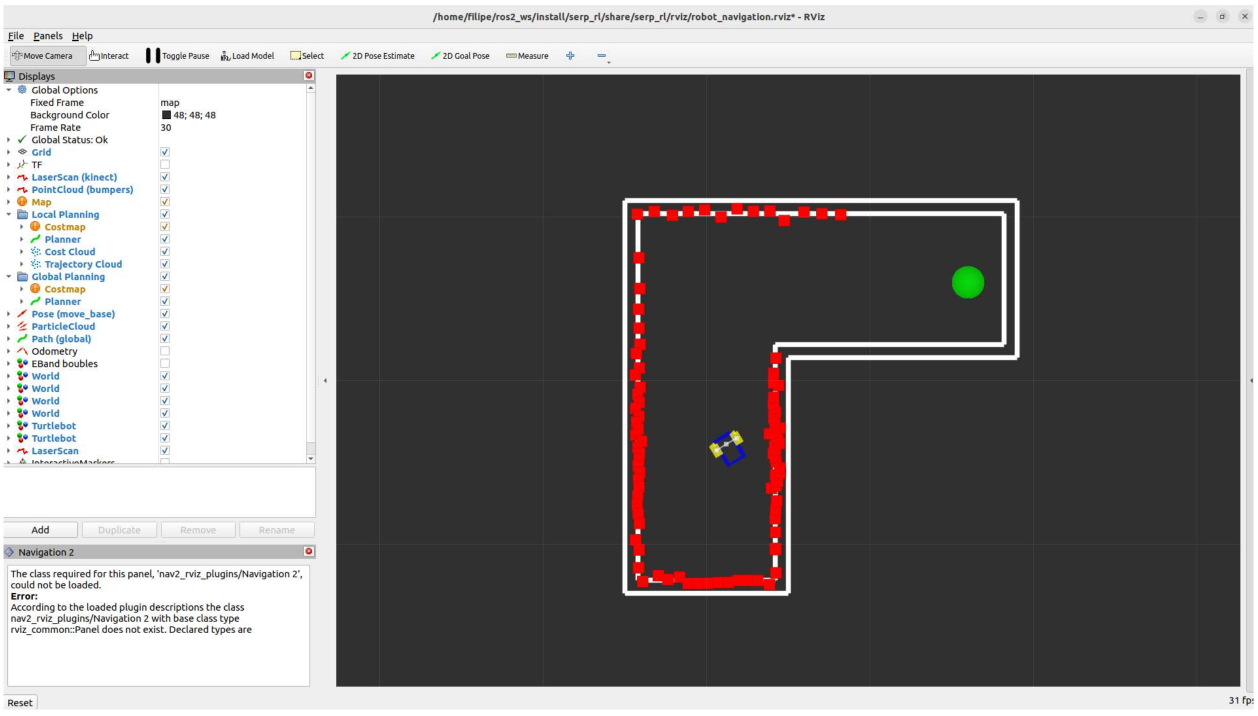
Build the project and install dependencies:

```
rosdep install -i --from-path src --rosdistro humble -y
colcon build
source install/setup.bash
```

Run the launch file:

```
ros2 launch serp_rl serp_rl.launch.py
```

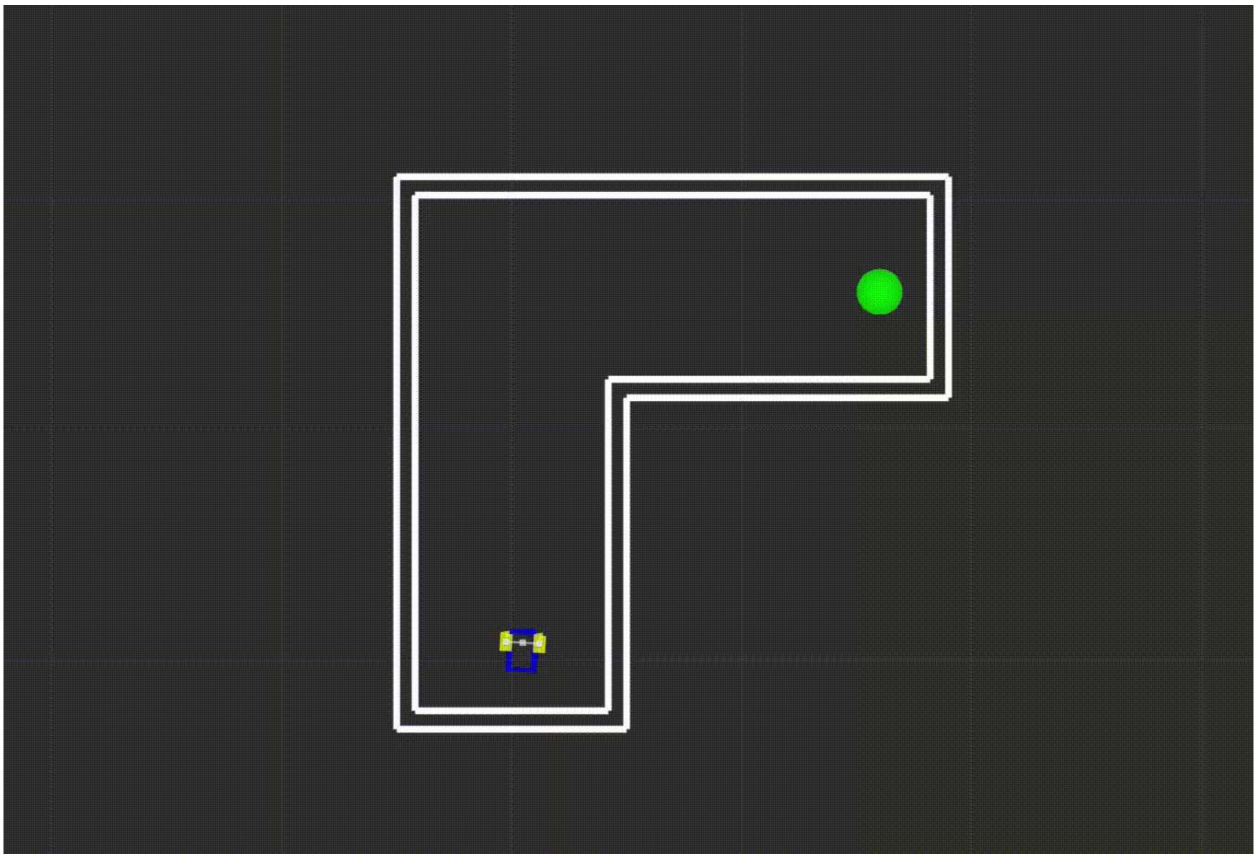
At this point, if no errors occur, you should be seeing the following world:



Global Options
Fixed Frame map
Background Color 48; 48; 48
Frame Rate 30
Global Status: Ok
Grid ✓
TF
LaserScan (kinect) ✓
PointCloud (bumpers) ✓
Map ✓
Local Planning ✓
Costmap ✓
Planner ✓
Cost Cloud ✓
Trajectory Cloud ✓
Global Planning ✓
Costmap ✓
Planner ✓
Pose (move_base) ✓
ParticleCloud ✓
Path (global) ✓
Odometry
EBand boubles
World ✓
World ✓
World ✓
World ✓
Turtlebot ✓
Turtlebot ✓
LaserScan ✓

Navigation 2

The class required for this panel, 'nav2_rviz_plugins/Navigation 2', could not be loaded.
Error:
According to the loaded plugin descriptions the class nav2_rviz_plugins/Navigation 2 with base class type rviz_common::Panel does not exist. Declared types are



The robot is currently using the Proximal Policy Optimization (PPO) algorithm to learn how to navigate the hallway from one end to the other. In the beginning, the robot is just exploring the environment and practically taking random actions. Over time, it starts to learn what to do in the different scenarios and it will improve at the task.

The target area is represented by the green circle and, every time the task is restarted, the initial and final positions swap so the robot learns how to turn to both the left and the right. The task is restarted if it fails if there are any collisions, if it takes too much time or if it succeeds (reaches the end).

These are all the components in the world and how they changed from the previous tutorial:

- Map: now is a simpler layout, representing a hallway with a 90º turn. Same configuration with a different image.
- SERP robot model with LiDAR: this model is exactly the same as before.
- End Beacon: new model added to detect if the SERP robot reached the end. It's represented by a green circle and its body has no collision with any other component.
- End Beacon LiDAR: laser plugin inside the end beacon model that ignores walls and only detects the SERP robot. The smallest value read is the distance from the robot to the end and, if below a threshold, the target area was reached.

NOTE: The layers were set up in order to accommodate the necessary collisions and lasers. If you're having trouble understanding the logic, revisit this Google Slides presentation on how to manage layers. The last slide shows the layer graph for this world.

As using RL can be very time-consuming, this simulation was sped up. This can be done by modifying the `update_rate` in Flatland, as mentioned in the previous tutorial.

NOTE: If you are using the VM and are running into performance issues, it might be a good idea to lower the `update_rate`.

The next sections will explain how the code from the controller needs to be organized to use any RL algorithm provided by Stable-Baselines3.

Setup the Environment

Important definitions before you start:

- RL Agent: component that decides what action to take based on the state of its environment.
- RL Environment: component that contains information about all possible actions for every state and the respective reward (how good the action is for any given state).

Every RL algorithm has the same goal: find the best action for the agent to take to every possible state of the environment. The first step to using RL in robotics (also known as Robot Learning) is to turn the robot (and its sensors) and the map into the environment for the agent to use.

This is exactly what the OpenAI's Gym package allows us to do with an `Env` class that can be inherited. In this package, we can reuse the node class:

```
from gym import Env

class SerpControllerEnv(Node, Env):
  [...]
```

This will then allow us to use variables and override functions that define the environment. The next sections will explain how to configure this environment but you can also check the documentation on how to make your own custom environment.

__init__ fuction

Inside the init function you need to initialize the `action_space`, `observation_space` and `state` variables.

Let us start by choosing the actions the robot can perform. For this project, there are three possible actions, executable by changing the robot's linear and angular velocity:

1. Move Forward
2. Rotate Left
3. Rotate Right

Then we need to choose what a state of our environment looks like (the `observation_space`). For that, we'll use the readings from the LiDAR. Since using all 90 rays would give a state space too large that would take too long for the agent to train in, we need to sample the readings. To do that, The LiDAR was divided into 9 equal sections, and from each we get the closest reading. This means that our observation space is composed of 9 floating point values.

NOTE: In more conventional definitions of a state in RL, the agent has total knowledge of the environment (position of all entities, shape of the map, etc.). In this case (and a lot of other Robot Learning applications), the agent can only know what the robot knows, which is the LiDAR readings.

To actually initialize the variables, we need to define their shape using Gym Spaces. For this project, these shapes are used:

```
from gym.spaces import Discrete, Box

# action is an integer between 0 and 2 (total of 3 actions)
self.action_space = Discrete(3)

# state is represented by a numpy.Array with size 9 and values between 0 and 2
self.observation_space = Box(0, 2, shape=(9,), dtype=numpy.float64)

# after the observation space is defined, you can declare the initial state
self.state = numpy.array(self.lidar_sample)
```

step function

In RL a step is the process that is constantly being repeated and consists of:

1. Receiving the action decided by the agent for the current state and executing it.
2. Determining the new state.
3. Calculate the reward based on the old and new states and the action.
4. Check if a final state was reached and the environment needs to be reset.

In the code for this package, this equates to:

1. Changing the robot speed based on the action chosen.
2. Wait for the action to be completed. In this case, the action finishes when the next LiDAR reading is published. This means each step lasts the same as the update rate of the LiDAR, 0.1 seconds.
3. Calculate the reward based on events detected in the simulation while performing the action and other available information.
4. Determines if it is a final state based on events from subscribed topics during the action (collisions or end reached) or the number of actions already taken (times out at 200

steps).

Returns: The current state, the reward, and if it reached a final state (boolean).

Calculating the Reward

The events that trigger rewards for an action are:

- Detecting the end was reached (large positive reward)
- Detecting a collision (large negative reward)
- Exceeding the maximum number of steps (large negative reward)
- Choosing the action to move forward (small positive reward)

NOTE: The reward calculation can still be improved, allowing the agent to train faster. A good exercise for you would be to think of other things that can be added to help the robot complete the task. Hint: take advantage of all information available, like old and new lidar readings and the action taken.

reset function

An episode, in the context of RL, is the set of steps between an initial state and a final state. The reset function has the task of starting a new episode by setting the environment back to an initial state. In this case, this means:

- Placing all the models (SERP and End beacon) in starting positions.
- Resetting all variables that need to.
- Determine the new initial state (next reading from the Lidar).

Returns: The initial state.

Other functions

- render: Used to render your environment. Not needed here since it is already rendered in Flatland.
- close: Runs when the environment is no longer needed. Can be used to close no longer necessary ROS 2 processes.

Running the RL algorithm

The more recently developed RL algorithms are very effective but also very hard to understand. The good news is that, thanks to the Stable-Baselines3 package, you can completely abstract from how the algorithm works. In fact, if you have a properly setup environment, getting an algorithm to run an agent on it only takes a few lines of code:

```
from stable_baselines3 import PPO
from stable_baselines3.common.env_checker import check_env
```

```
env = Env()

# optional but launches exceptions with helpful messages debug your
environment if it has errors
check_env(env)

# create the agent for the PPO algorithm and assign one of the predefined
policies and the environment
agent = PPO("MlpPolicy", env)

# letting the agent learn for 25000 steps
agent.learn(total_timesteps=25000)
```

Go to the documentation to see all available RL algorithms in this package. Notice that you can swap between algorithms with very few changes to the code.

After the training is complete, you can test your model by manually calling the environment functions:

```
obs = self.reset()
while True:
  # returns an action based on what it learned
  action, _states = agent.predict(obs)

  obs, rewards, done = self.step(action)
  if done:
    self.reset()
```

You can easily store your trained agent in a file and load it later with the functions:

```
agent.save("ppo")
agent = PPO.load("ppo")
```

This code needs to run in parallel with the ROS2 processes so threading was used.

The function `learn` trains the agent for a given number of steps, which does not guarantee that by the end of training the agent will be capable of performing the task. To make sure your agent is properly trained, you need to think what is the best strategy to train it. This section will present you with an option to resolve this issue.

Training Strategies

The goal is to make sure you end up with an agent capable of completing the task, meaning that it needs to somehow be validated. One possible solution, which is used in this project, is to follow these steps:

1. Train for a given number of steps.

2. Test the agent for a set number of episodes and determine the accuracy (number of successful episodes divided by the total number of episodes)
3. If the accuracy is above a given threshold finish training, otherwise go back to step 1.

Next steps

Now that you know how both Flatland, ROS2 and RL can work together, you can now start developing your own projects and experimenting with different maps, setups or tasks.

A.5 Flatland Artificial Potential Fields Tutorial using ROS 2

This appendix contains the full Artificial Potential Fields Tutorial presented in Section 5.2.3 which is also available at `https://github.com/FilipeAlmeidaFEUP/ros2_flatland_pf_tutorial`.

Videos with examples from this tutorial are available at `https://github.com/FilipeAlmeidaFEUP/dissertation_videos/tree/main/teaching_robot_learning_in_ros2/artificial_potential_fields_tutorial`.

Flatland Artificial Potential Fields Tutorial using ROS 2

This tutorial will teach you a reactive alterative to using Reinforcement Learning that was explained in the previous tutorial for robot navigation. Here you will learn how to use the Artificial Potential Fields algorithm to make a pair of robots navigate from one side of a hallway with a 90º turn to the other while travelling side by side.

There are some differences in the setup that will be explained in this section

Running the code

First make sure you have the transforms3d module install by running:

```
sudo pip install transforms3d
```

Then clone the repository to your `<ros2_workspace>/src` folder:

```
git clone https://github.com/FilipeAlmeidaFEUP/ros2_flatland_pf_tutorial.git
```

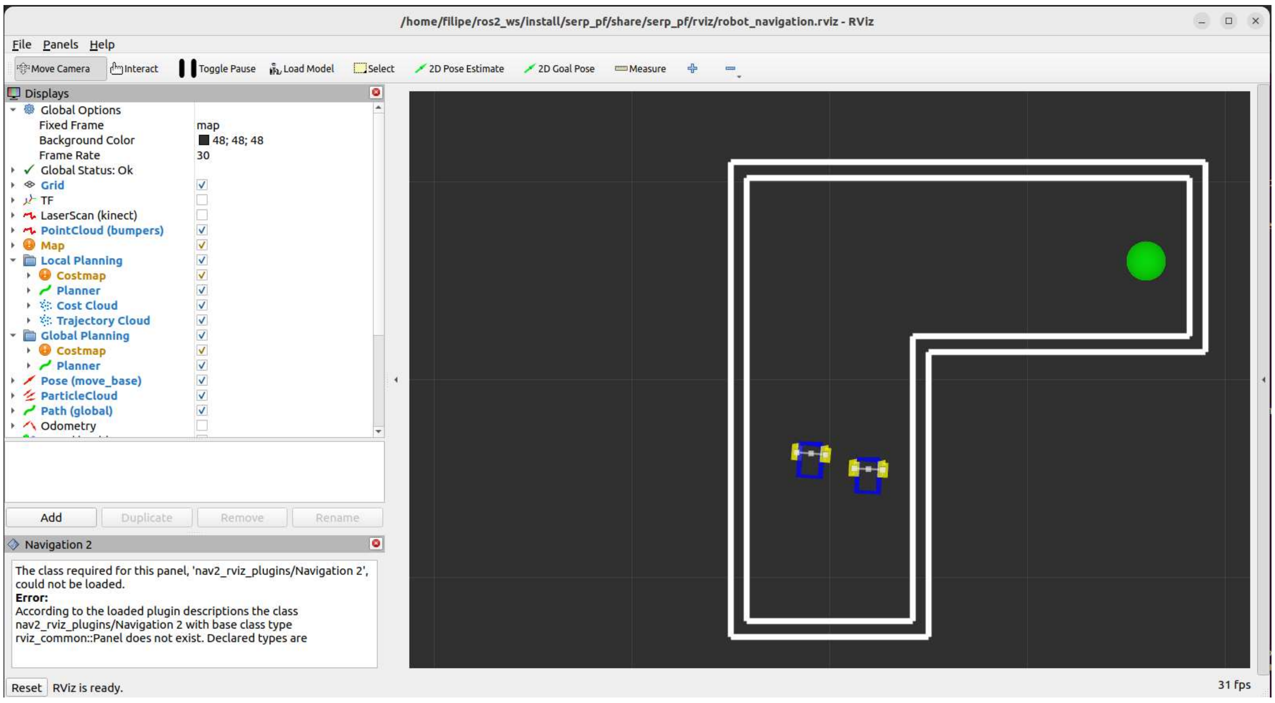
Build the project and install dependencies:

```
rosdep install -i --from-path src --rosdistro humble -y
colcon build
source install/setup.bash
```

Run the launch file:

```
ros2 launch serp_pf serp_pf.launch.py
```

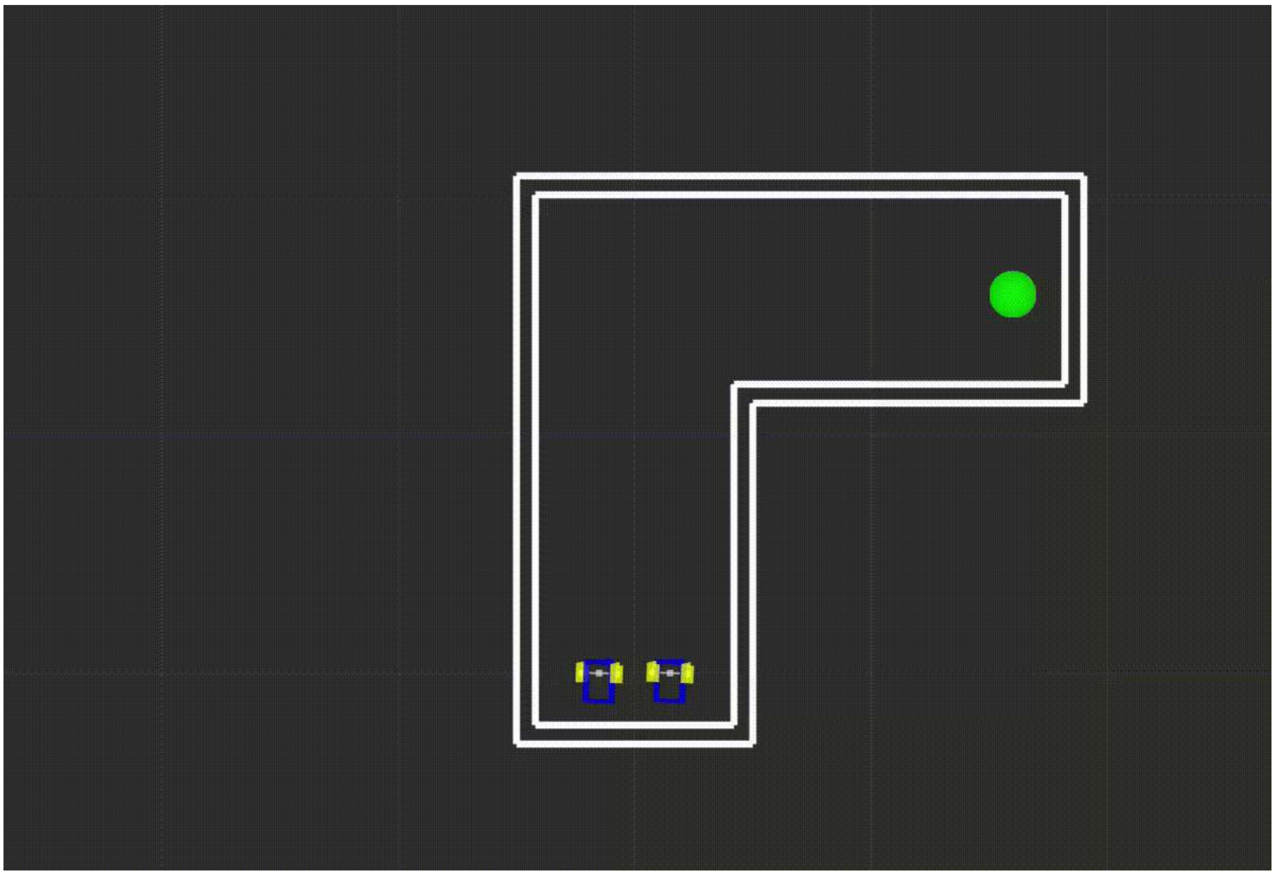
At this point, if no errors occur, you should be seeing the following world:



Fixed Frame map
Background Color 48; 48; 48
Frame Rate 30
Global Status: Ok

The class required for this panel, 'nav2_rviz_plugins/Navigation 2', could not be loaded.
Error:
According to the loaded plugin descriptions the class nav2_rviz_plugins/Navigation 2 with base class type rviz_common::Panel does not exist. Declared types are

RViz is ready.



Just like in the RL tutorial, the start is in one end of the hallway and the target position is on the other, but now there are two robots travelling side by side. Collisions or reaching the end, will reset the environment and switch the starting and target positions.

The main difference is the way the robot is deciding where to move at any point. Instead of using RL algorithm to decide, it's using the Artificial Potential Fields algorithm. More on this in this section.

New Flatland and ROS 2 setup

This package is very similar to the one on the previous tutorial but without the RL components. That being said, there are some important small changes that should be highlighted.

Using local python modules

In your python code, you can import any module that you would be able to import in any other python project. You can also import modules from local files on your own machine but that will take a few extra steps to work in ROS 2.

First you need to add the folder of the module to the folder of the controller python code (which is the folder with the same name as the package; example in this package). Each new folder also needs to contain a file called __init__.py, which can be left empty or used as a regular python file. Your code folder should look something like this:

```
├── <ros_package_name>
│   ├── __init__.py *
│   ├── <another_py_file>.py
│   ├── <py_module1>
│   │   ├── __init__.py *
│   │   ├── <py_file1>
│   │   ├── <py_file2>
│   ├── <py_module2>
│   │   ├── __init__.py *
│   │   ├── <py_file3>

* required files
```

After that, the setup file needs to be updated. First the `packages` variable needs to contain all added modules and then folders need to be shared by adding them to `data_files` . For the example presented before:

```
setup(
    [...]
    packages=['<ros_package_name>', '<ros_package_name>/<py_module1>',
'<ros_package_name>/<py_module2>'],
    data_files=[
        [...]
        ('share/<ros_package_name>/<ros_package_name>/<py_module1>/ ,
glob('<ros_package_name>/<py_module1>/*')),
        ('share/<ros_package_name>/<ros_package_name>/<py_module2>/ ,
glob('<ros_package_name>/<py_module2>/*')),
    ],
    [...]
)
```

Now you can access the modules inside you controller python code. One local module called charges was added in this package and contains auxiliary classes and methods for the Artificial Potential Fields algorithm.

Callback method with arguments

Callback methods are the function that is passed as an argument when a subscriction to a topic is made. Each time the topic publishes a message, this method is called. By default it can only have one agument, which is the message published. But, in ROS 2, through the use of the python lambda functions, it is possible to send more arguments.

```
self.create_subscription(<MsgType>, topic, lambda msg: callback_method(msg, arg), 1)

def callback_method(msg, arg):
    [...]
```

In this package this was used in the callback function for odometry subscriptions (robot position). Since there are two robots, the same function could be used and the extra argument served to know which robot the message came from.

```
[...]

self.create_subscription(Odometry, "/odom1", lambda msg: self.updatePosition(msg,
"serp1"), 1)
self.create_subscription(Odometry, "/odom2", lambda msg: self.updatePosition(msg,
"serp2"), 1)

[...]

def updatePosition(self, data, model):
    [...]
```

Using multiple robots

Using multiple robot models in Flatland is for the most part not a problem since you just need to add model entries in the world file. The only issue in the Flatland configuration is that only in very rare situations the same model file can be used to generate several models in the world. This happens because the model file configures its interactions with layers and the topics it publishes and subscribes to. For example, you can never control 2 robots independently if they were generated by the same file.

In this package, the two Serp models have exactly the same chape and same type of plugins, but they need different:

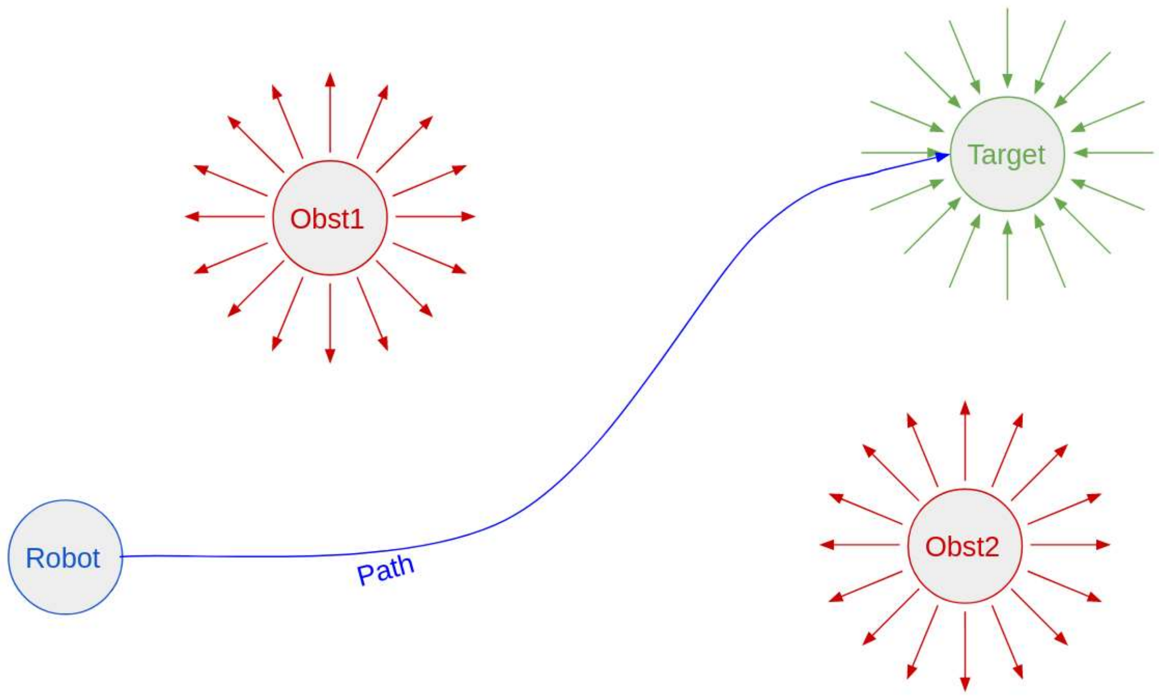
- topics to subscribe to Twist messages(move the robot).
- topics to publish Odometry messages(position of the robot).
- interactions with layers, since the end beacon LiDAR can only detect one of them.

For these reasons, two model files had to be configured, serp1 and serp2.

Artificial Potential Fields

Unlike RL algorithms, Artificial Potential Fields(APF) is a reactive method. This means that, similarly to RL, it chooses an action for each state that it encounters and the difference is in a way it chooses an action. While RL algorithms use trial and error to learn over time the best action for each state, APF works like function that receives the state and computes an action. Since the same state will allways produce the same action, it does not require any training.

APF is used for robot path planning by determining, at any point on the map, where the robot needs to go. This is possible by filling the map with electrical charges, adding repulsive charges where there are obstacles and an attractive charge to the target. The sum of all the forces determines the force and, from that, the acceleration for the robot, eventually leading it to the goal. The following image shows a representation of the path planning in APF.



Placing charges on the map

For APF to work, it's necessary to define where the charges need to be placed and how they interact with the robots. Depending on shape of the obstacle, the electric field produced will be different. For this map, three different types of charges were created:

- Point Charge
- Line Segment Charge
- Composite Charge

These types of charges are represented by python classes in the file charges.py.

NOTE: The robots themselves should also have a charge but for simplicity lets consider a positive charge attracts the robot and a negative one repels it.

Point Charge

A charge represented by:

- a point in space
- an intensity

For any point in space P and point charge C the Force in P is defined by its orientation and magnitude.

The orientation is given by:

$$tan^{-1}\left(\frac{C.y - P.y}{C.x - P.x}\right)$$

And the magnitude is given by Coulomb's law:

$$F = k\frac{q_1 q_2}{r^2}$$

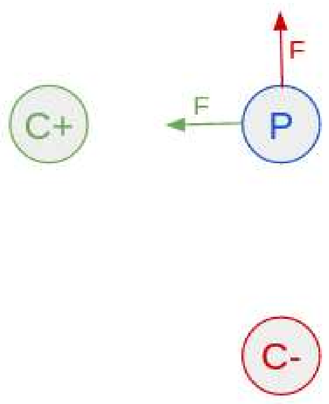
where:

- k is the Coulomb constant. Essential when using actual electrical fields but here can be simplified to 1.
- q1 is the intensity of the charge in P, which in this case will be the position of the robot. As stated before, the robot charge will be ignored so it can also be simplified to 1.
- q2 is the intensity of the charge.
- r is the distance from P to C

So the final magnitude formula is:

$$F = \frac{C.intensity}{Euclidean_distance(P, C)}$$

F will be positive or negative based on signal the intensity. If positive, when returning the x and y components, they will be in the direction of the charge and , if negative, they will be in the opposite direction. The following image demonstrates how Point Charges a point in space.



A positive Point Charge is used on the target position to attract the robot to it.

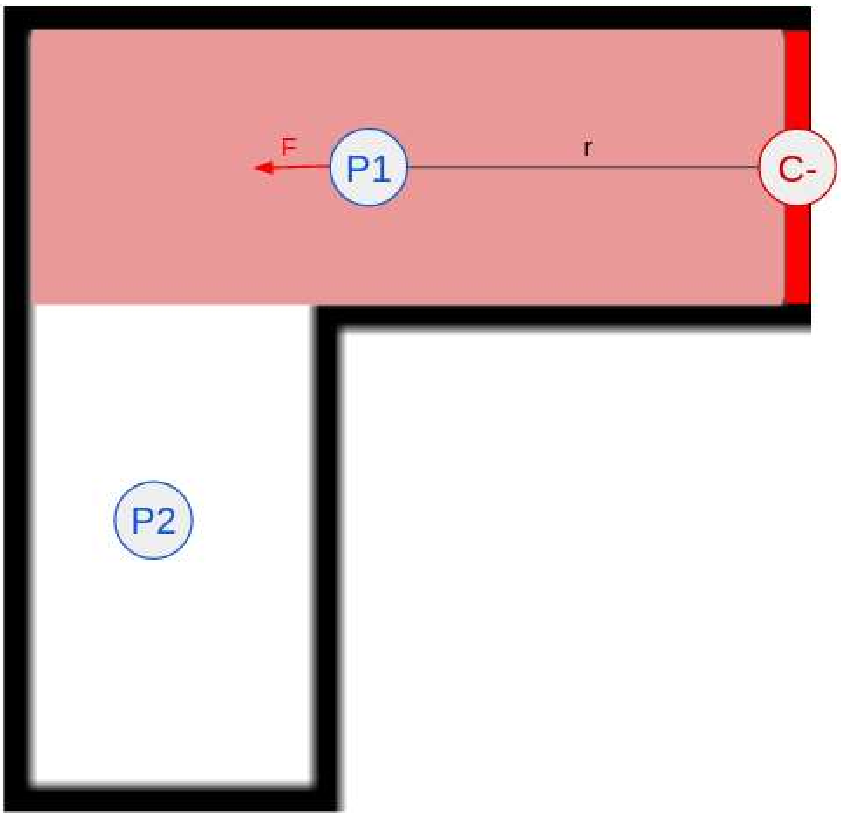
Line Segment Charge

A charge represented by:

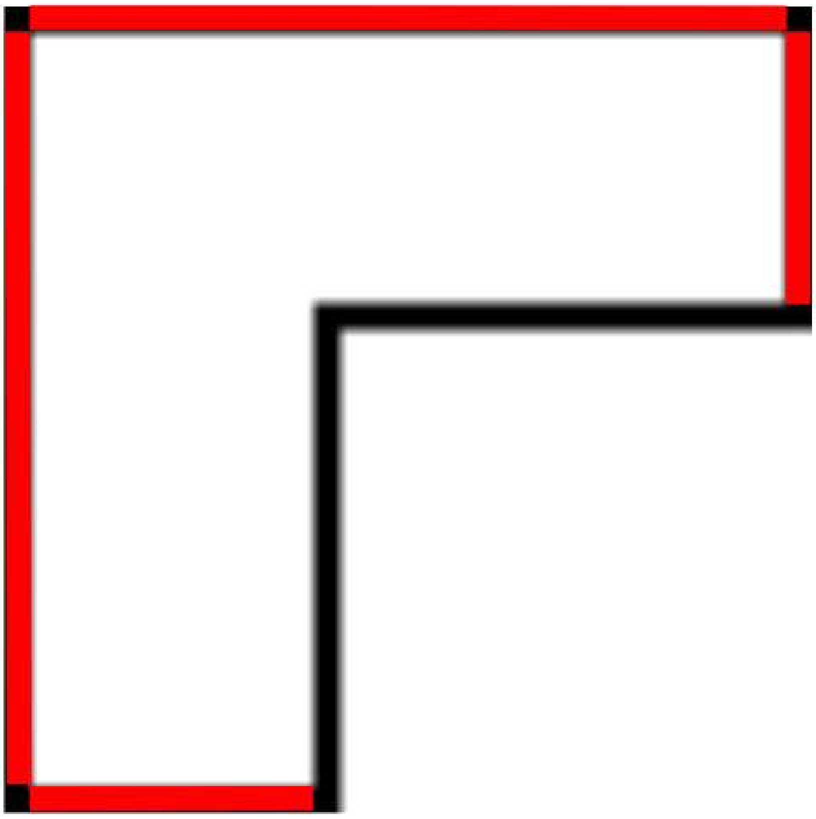
- a pair of point in space that define the line segment
- an intensity

For any point in space P and line segment charge C the Force in P is calculated by determining the intersection between the line segment and the line perpendicular to C that passes through P. Then a temporary charge is placed in that point with C.intensity and is used to determine the force. If the line doesn't intersect, the forrce has magnitude of 0 (no force).

The following image demonstrates how a Line Segment Charge affect points in space.



These Line Segments Charges are the ones used to represent the walls of the map:



Composite Charge

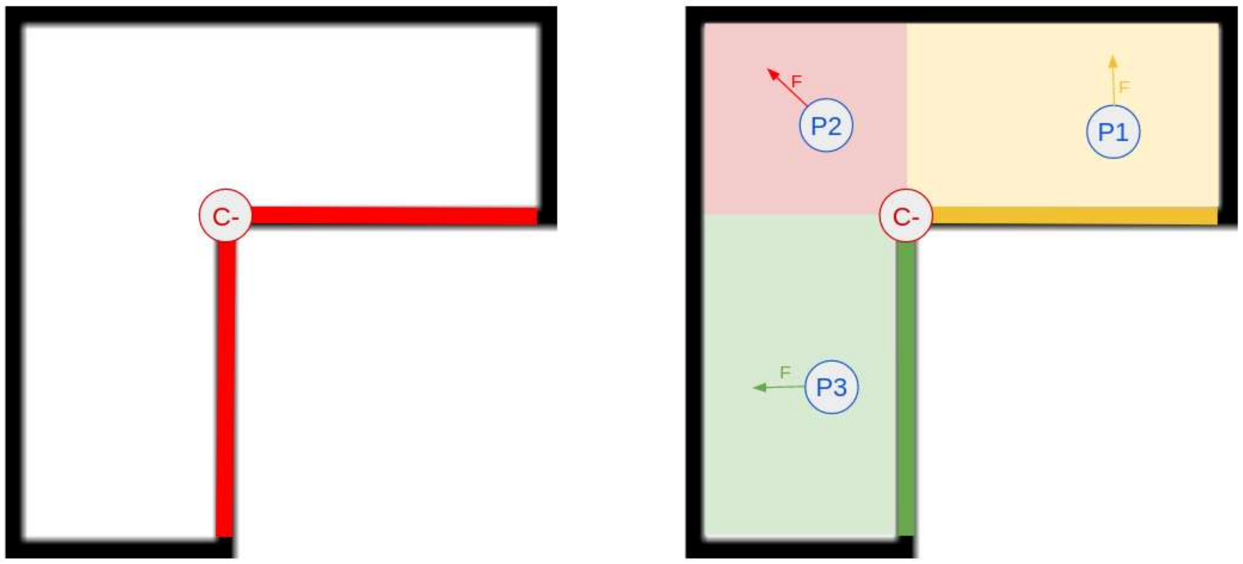
A charge represented by:

- a set of Line Segment Charges and Point Charges

- an intensity

For any point in space P and composite charge C the Force in P is determined by only one of the charges contained in C. If P is in one of the zones affected by one of the line segments, then that line segment is the charge that determines the force. If not, It is determined by the closest point charge.

The following figure represents the only composite charge contained in the map on the left and how it affects points in space on the right.



Calculate the Robots Speeds

After placing the charges on the map, the goal is to determine how the robots should be moving at any given time. The APF algorithm says that to determine the trajectory for the robot, we only need to add all the forces from the charges on the current robot position and that will give us the acceleration. This is not taking into aaccount some problems in this specific setup:

- We have two robots that need to reach the target while travelling side by side.
- Because the robots are differential drive robots, they need to be facing the target before they can start moving.
- The Diff Drive plugin in Flatland does not have an acceleration, only linear speed.

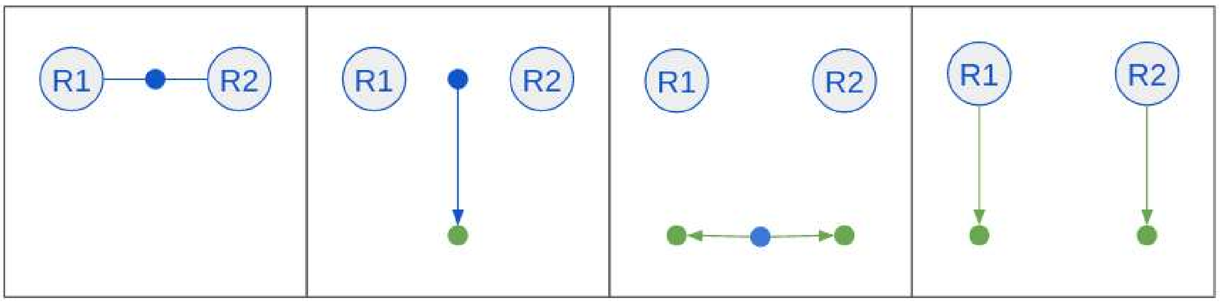
Lets tackle these issues one at a time.

Moving Robots side by side

To allow for the robots to move side by side, we need to make sure they can navigate through the map independently while also leaving room the other one. To do this, we need to start by thinking of the pair of robots as one unified component and determine the position where each one needs to be in order to move towards the target while being next to eachother. This can be achieved by:

1. Finding the middle point of the positions of both robots.
2. Calculate the resulting forces from all the map charges (with a positive charge in the target position) and add the force vector to get a target middle point.
3. From the target middle point, find the target point for each robot. This is done by shifting the target middle point by half the ideal distance between the robots in both ways of the direction perpendicular to the force vector found in step 2.

The following image ilustrates these steps.



In this to new points found, we can create Tension Points. Tension Points work exactly like Point Charges but the magnitude of the force is calculated by Hooke's law (used for spring extension) instead:

$$F = kr$$

Where:

- k is a constant factor characteristic of the spring. Can be replaced by the intensity.
- r is the distance from the robot to the tension point

Opposed to Coulomb's law, the larger the distance, the greater the force.

To determine the force in each robot in the point where the robot is:

- Add the forces from all the wall charges.
- Add force from a negative charge in the position of the other robot. This is to avoid collisions between them.
- Add the force from the newly found Tension points.

Rotate to face the target

For differential drive robots to move in a specific direction, they need to be facing the right way first. This means that if a robot does not have the same rotation as the angle of the force that was calculated to it, it only has angular speed. Only when it is facing the right way, it can move forward, using only linear speed.

To make sure they won't get to far appart from each other, the robots can only have linear speed if both are facing the right direction and are ready to move forward.

Using the Force magnitude for the linear speed

The angle of the force on each robot determines the direction they need to be moving to. The magnitude should determine the acceleration of the linear velocity but Flatland does not use acceleration. The magnitude should still be taken into account though, since if one of the robots is behind, it needs to catch up to the other one. In this example, the magnitude of both forces was used to determine the ratio of both linear speeds. The robot with the strongest Force sum is assinged a maximum linear speed, while the other has a smaller one that maintains the ratio of the forces.

Limitations of APF

At first glance, the APF algorith might look better than using RL, since it was able to perfectly perform a task without requiring any training. This is why it is important to recognize a major issue with APF: it requires perfect knowledge of the map, robots positions and the target location.

A RL agent can train in a map and then be placed in a new one and, if the new map shares some features with the one he trained in, he might be able to still perform the task.

In APF, if you want to use a new map, the whole setup needs to be changed (new charge layout).

Next steps

Now that you understand how the APF algorithm works, you can make your own map and create a new charge layout for it. Then try to adjust the intensity values so that the robot can reach the end goal.

A.6 Managing Layers in Flatland Presentation

This appendix contains the full Google Slides presentation on how to manage layers in Flatland which is also available at https://docs.google.com/presentation/d/1KqJDQR_PBaGtS-kA5KgsTbaRThu_uP2U8NJeoSP0GDE/edit?usp=sharing.

Managing Layers in Flatland

Flatland Layers

- Flatland was created to be the main 2D robotics simulator to be used with ROS.
- The concept of layers turns Flatland into a “2.5D simulator” by simplifying the 3rd dimension to a discrete space.
- Layers work independently from each other in terms of physics.
- Each component of the simulator is contained in one to many layers and two components only collide if they have at least one layer in common.
- Flatland worlds allow up to 16 layers.

Problems managing layers

- The only default types of components that need to use layers are the maps, the models and the laser plugin.
- Models may collide or not with each other or the map and radars need to define what needs to be detected by them.
- Lasers can detect the model they are contained in. Example:
    - Square box model with radar inside. The radar can only detect its own model.
- The different bodies of a model may not all have the same collisions.
- Managing layers can quickly become a very complex task very prone to errors.

How to manage layers

The goal is to respect all collisions with the minimum number of layers possible.

Methodological approach:

- Build an undirected graph to represent all collisions between models and maps.
- Group the nodes in layers.
- Add the lasers one by one and add new layers if necessary.
- Apply the necessary layers to each component.

Building the Collision Graph - Manage Maps

- In the world.yaml file, each layer is assigned to a map.
- Maps are defined by an image or a .dat file with line segments.
- If a layer is inside of a map, it interacts with his walls.

Example of the layer configuration in the world.yaml file:

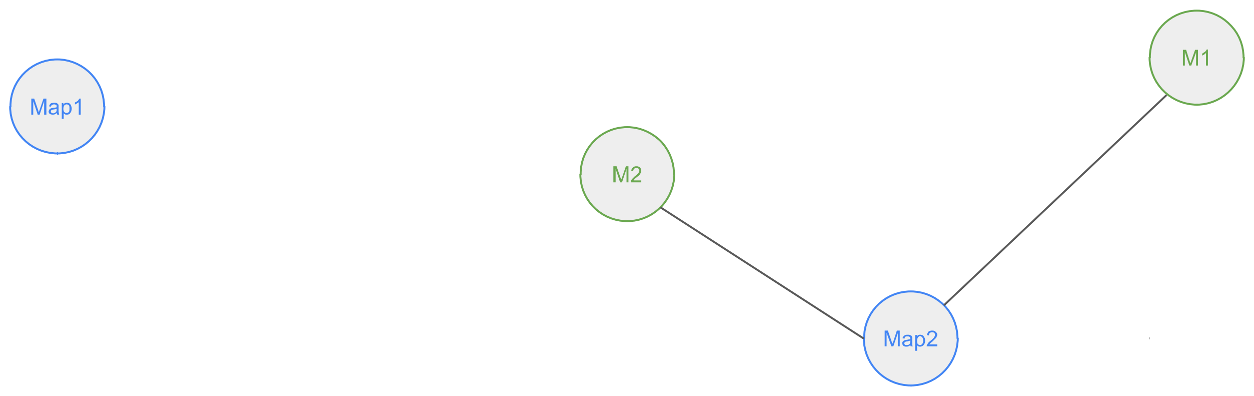
```
layers:
 # map 1
 - name: [<list of layers that interact with this map>]
   map: <image or dat file>
 # map 2, only needs more than one map if there are parts with different interactions
 - name: [<list of layers that interact with this map>]
   map: <image or dat file>
 # map 3, empty map; add if there are components that don't interact with any maps
 - name: [<list of layers that interact with this map>]
   map: <empty image or dat file>
```

Building the Collision Graph - Add Maps

Add one node for each map except for the empty map.

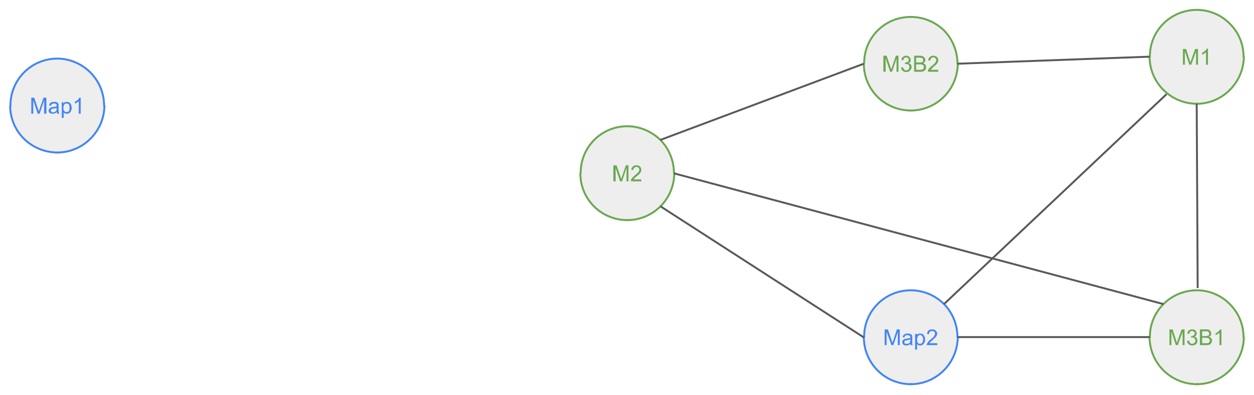
Building the Collision Graph - Add Models

Add the models one by one and connect them to the nodes they collide with.



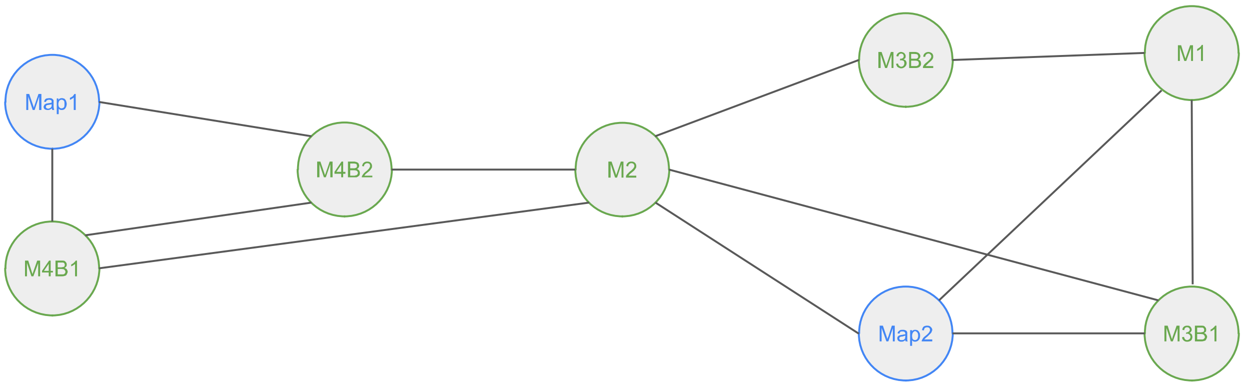
Building the Collision Graph - Add Models

Model 3 has 2 groups of bodies that have different collisions so they need to be separated. There are no collisions inside the same model, even if the layer is the same but, to avoid unnecessary layers, they should not be connected.



Building the Collision Graph - Add Models

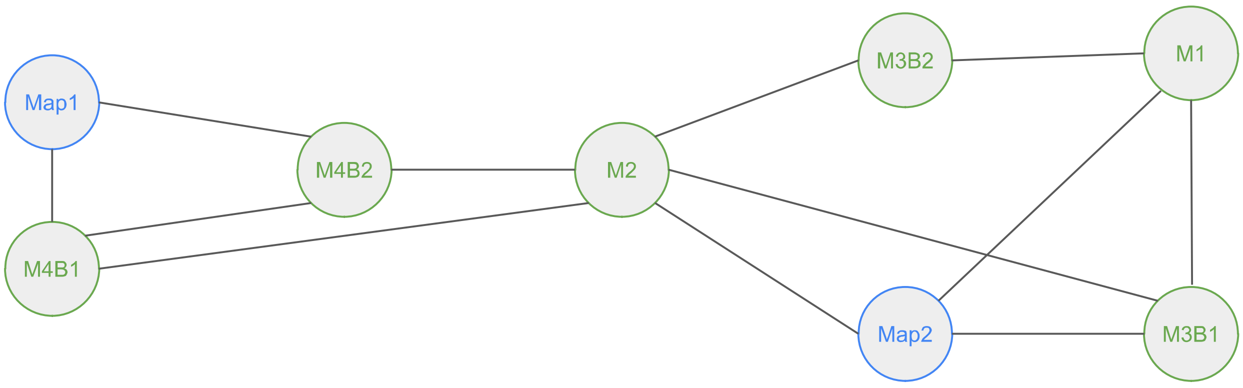
Model 4 has 1 group of bodies that needs to be detected by a laser and other that doesn't. They also need to be separated but, in this scenario, they interact the same way in terms of collisions so, to save layers, they should be connected.



Building the Collision Graph - Finding layers

The goal now is to find the minimum number of layers needed. This is done by obtaining the minimum number of complete subgraphs* that contain all the edges. Start by getting a list of all edges.

Edges:
Map1-M4B1
Map1-M4B2
M4B1-M4B2
M4B1-M2
M4B2-M2
M2-Map2
M2-M3B1
M2-M3B2
Map2-M3B1
Map2-M1
M3B1-M1
M3B2-M1

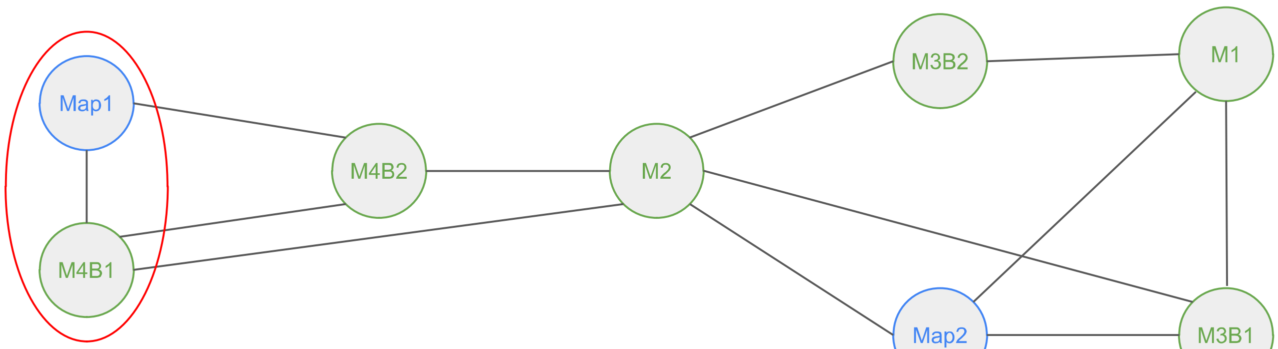


*Complete graph - each vertex is connected to every other vertex

Building the Collision Graph - Finding layers

Choose one of the edges and group their vertices.

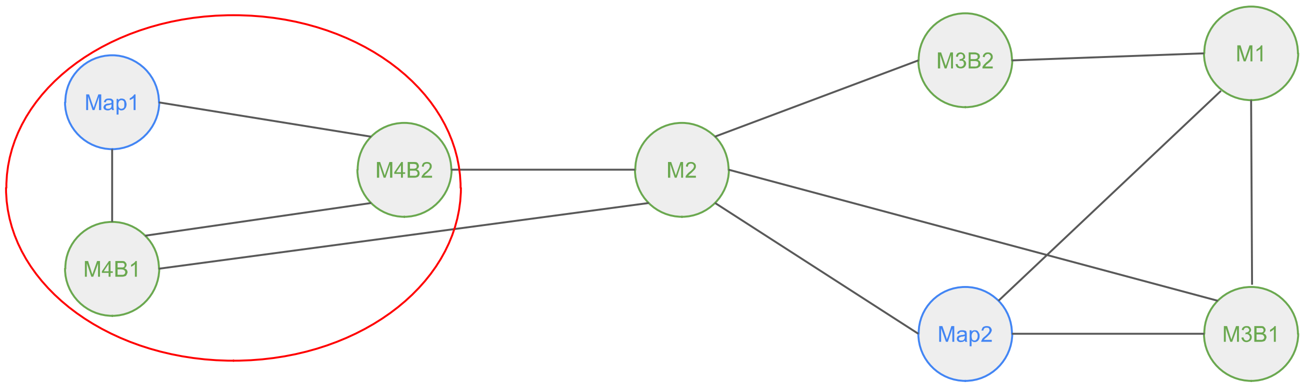
Edges:
Map1-M4B1
Map1-M4B2
M4B1-M4B2
M4B1-M2
M4B2-M2
M2-Map2
M2-M3B1
M2-M3B2
Map2-M3B1
Map2-M1
M3B1-M1
M3B2-M1



Building the Collision Graph - Finding layers

Add to the group all possible vertices so that the group remains a complete graph.

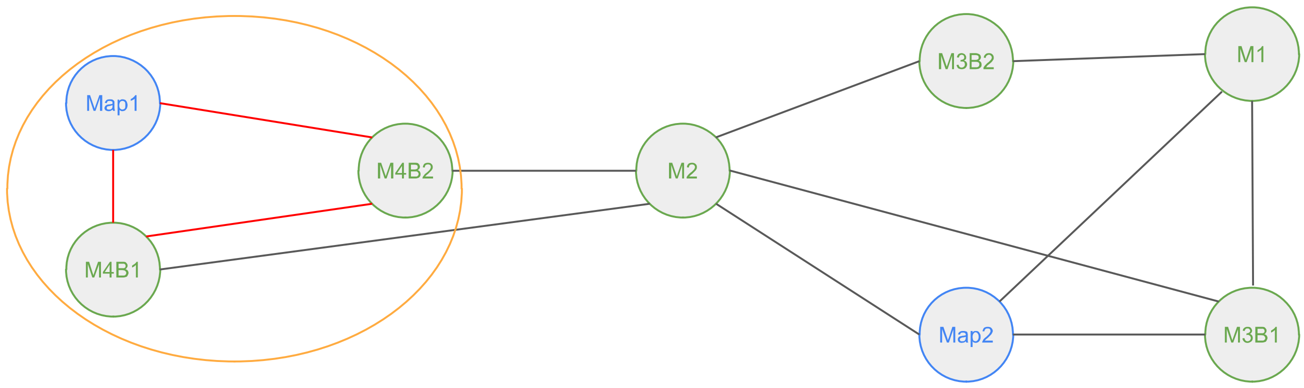
Edges:
Map1-M4B1
Map1-M4B2
M4B1-M4B2
M4B1-M2
M4B2-M2
M2-Map2
M2-M3B1
M2-M3B2
Map2-M3B1
Map2-M1
M3B1-M1
M3B2-M1



Building the Collision Graph - Finding layers

Scratch all edges contained in the group from the list.

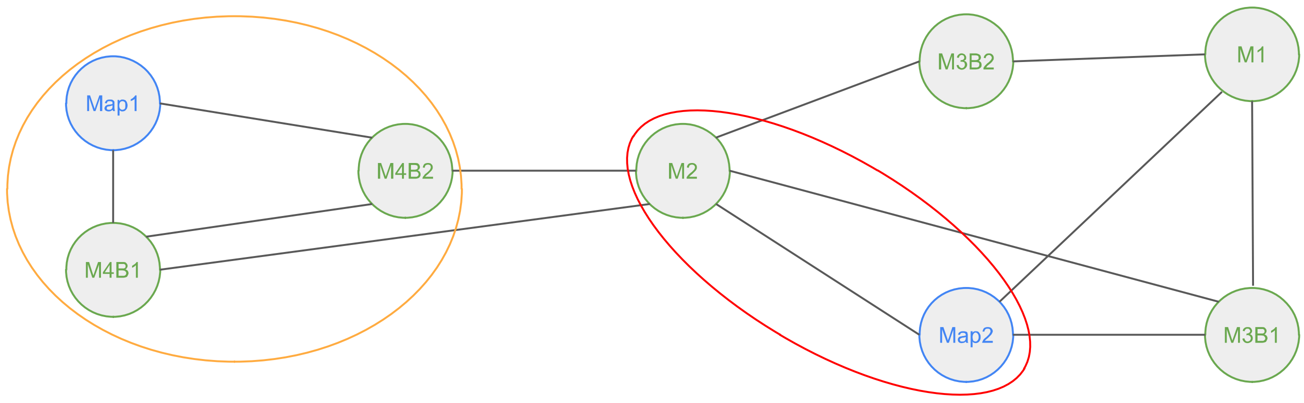
Edges:
~~Map1-M4B1~~
~~Map1-M4B2~~
~~M4B1-M4B2~~
M4B1-M2
M4B2-M2
M2-Map2
M2-M3B1
M2-M3B2
Map2-M3B1
Map2-M1
M3B1-M1
M3B2-M1



Building the Collision Graph - Finding layers

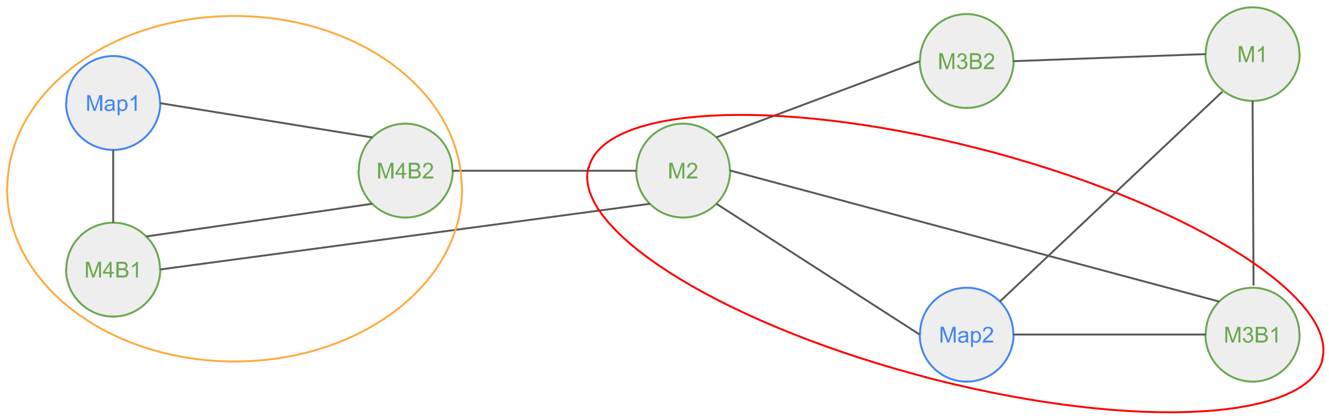
Choose another edge and repeat the process until the list is empty.

Edges:
~~Map1-M4B1~~
~~Map1-M4B2~~
~~M4B1-M4B2~~
M4B1-M2
M4B2-M2
M2-Map2
M2-M3B1
M2-M3B2
Map2-M3B1
Map2-M1
M3B1-M1
M3B2-M1



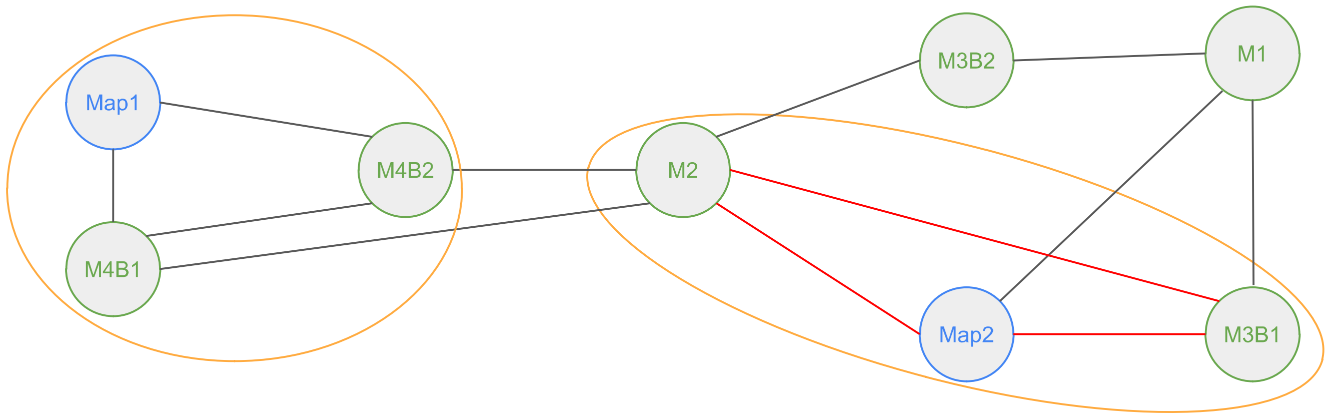
Building the Collision Graph - Finding layers

Edges:
~~Map1-M4B1~~
~~Map1-M4B2~~
~~M4B1-M4B2~~
M4B1-M2
M4B2-M2
M2-Map2
M2-M3B1
M2-M3B2
Map2-M3B1
Map2-M1
M3B1-M1
M3B2-M1



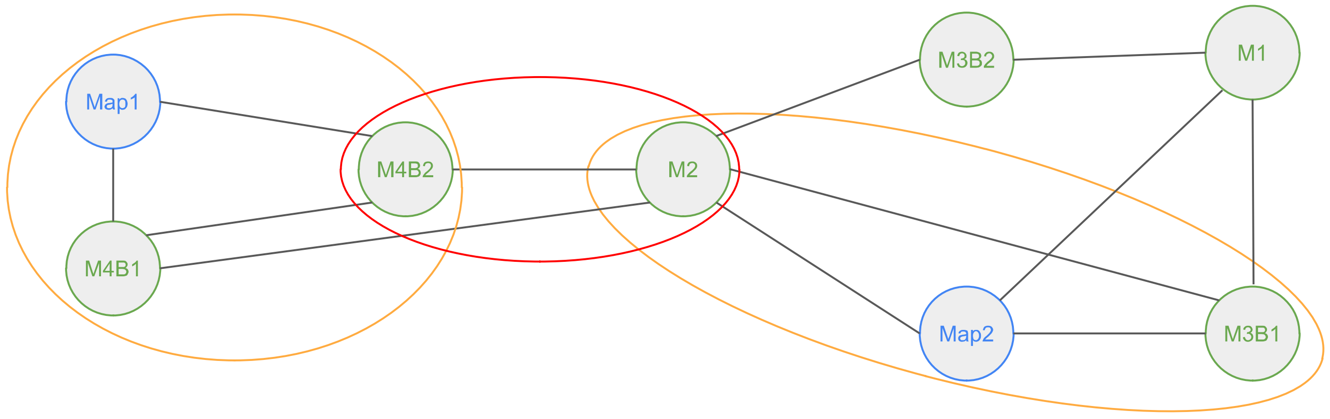
Building the Collision Graph - Finding layers

Edges:
~~Map1-M4B1~~
~~Map1-M4B2~~
~~M4B1-M4B2~~
M4B1-M2
M4B2-M2
~~M2-Map2~~
~~M2-M3B1~~
M2-M3B2
~~Map2-M3B1~~
Map2-M1
M3B1-M1
M3B2-M1



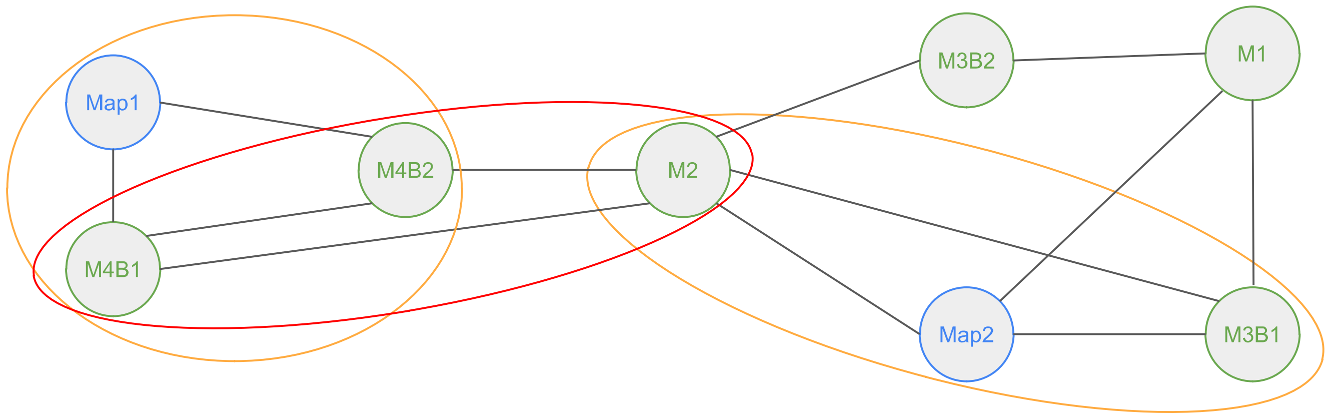
Building the Collision Graph - Finding layers

Edges:
~~Map1-M4B1~~
~~Map1-M4B2~~
~~M4B1-M4B2~~
M4B1-M2
M4B2-M2
~~M2-Map2~~
~~M2-M3B1~~
M2-M3B2
~~Map2-M3B1~~
Map2-M1
M3B1-M1
M3B2-M1



Building the Collision Graph - Finding layers

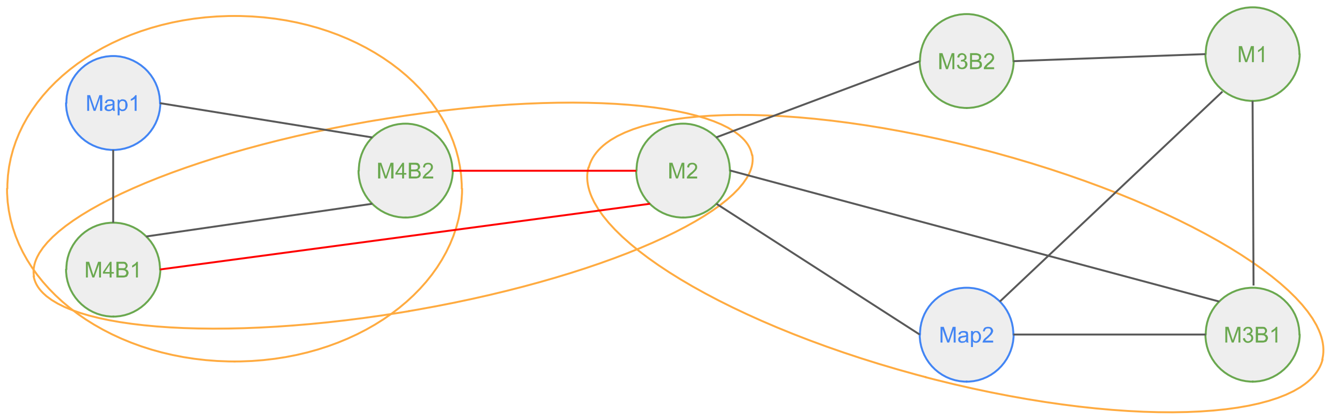
Edges:
~~Map1-M4B1~~
~~Map1-M4B2~~
~~M4B1-M4B2~~
M4B1-M2
M4B2-M2
~~M2-Map2~~
~~M2-M3B1~~
M2-M3B2
~~Map2-M3B1~~
Map2-M1
M3B1-M1
M3B2-M1



Building the Collision Graph - Finding layers

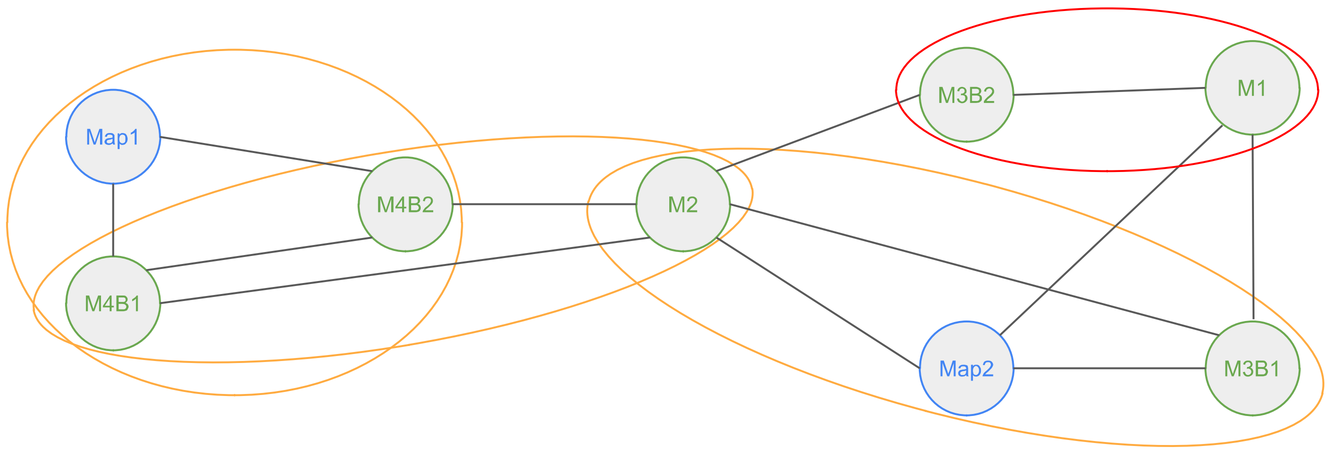
NOTE: Edges that were already removed from the list can still be used on new subgraphs. Example: M4B1-M4B2.

Edges:
~~Map1-M4B1~~
~~Map1-M4B2~~
~~M4B1-M4B2~~
~~M4B1-M2~~
~~M4B2-M2~~
~~M2-Map2~~
~~M2-M3B1~~
M2-M3B2
~~Map2-M3B1~~
Map2-M1
M3B1-M1
M3B2-M1



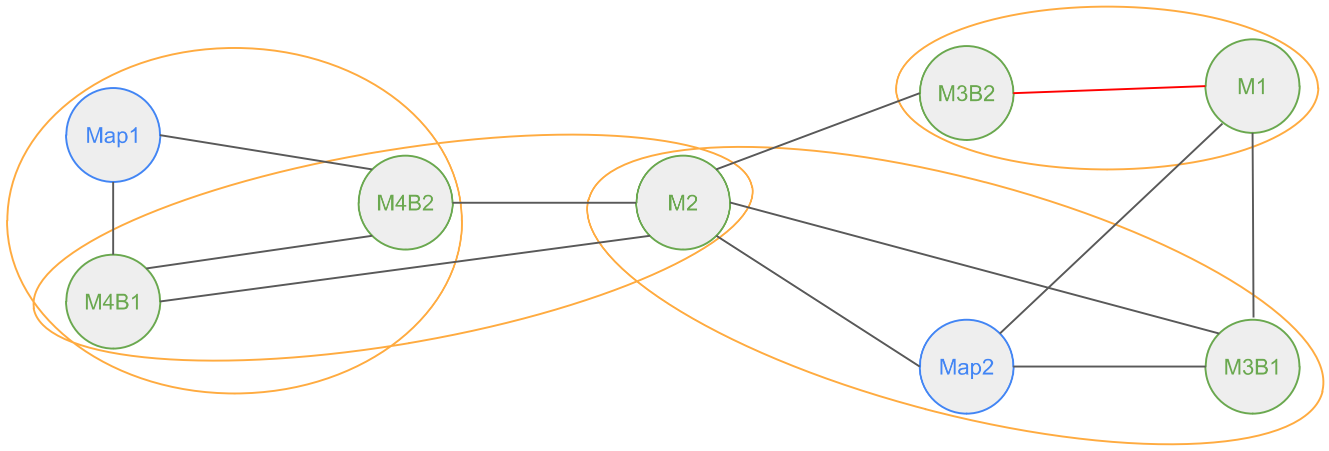
Building the Collision Graph - Finding layers

Edges:
~~Map1-M4B1~~
~~Map1-M4B2~~
~~M4B1-M4B2~~
~~M4B1-M2~~
~~M4B2-M2~~
~~M2-Map2~~
~~M2-M3B1~~
M2-M3B2
~~Map2-M3B1~~
Map2-M1
M3B1-M1
M3B2-M1



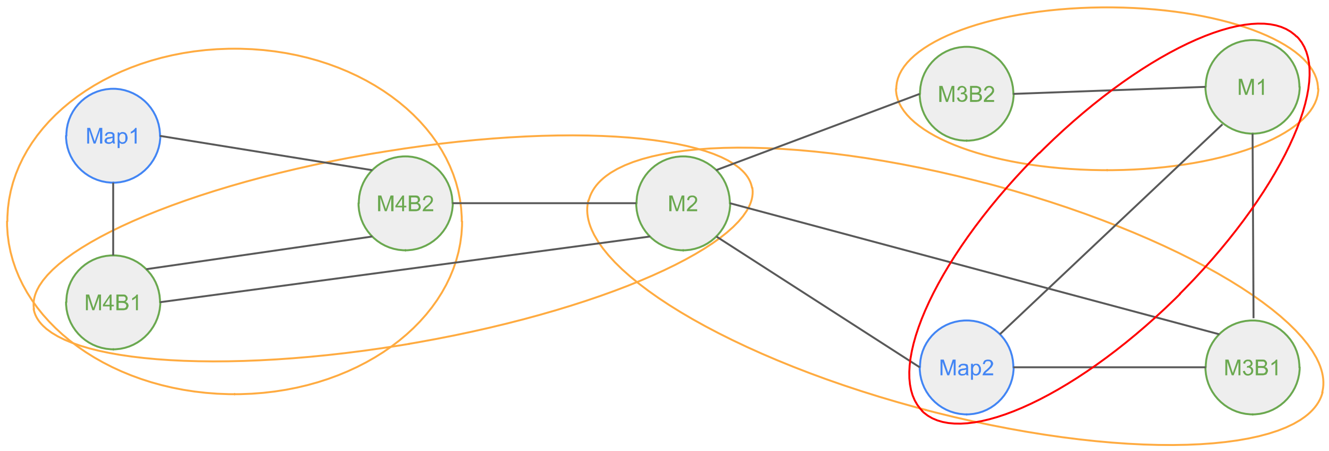
Building the Collision Graph - Finding layers

Edges:
~~Map1-M4B1~~
~~Map1-M4B2~~
~~M4B1-M4B2~~
~~M4B1-M2~~
~~M4B2-M2~~
~~M2-Map2~~
~~M2-M3B1~~
M2-M3B2
~~Map2-M3B1~~
Map2-M1
M3B1-M1
~~M3B2-M1~~



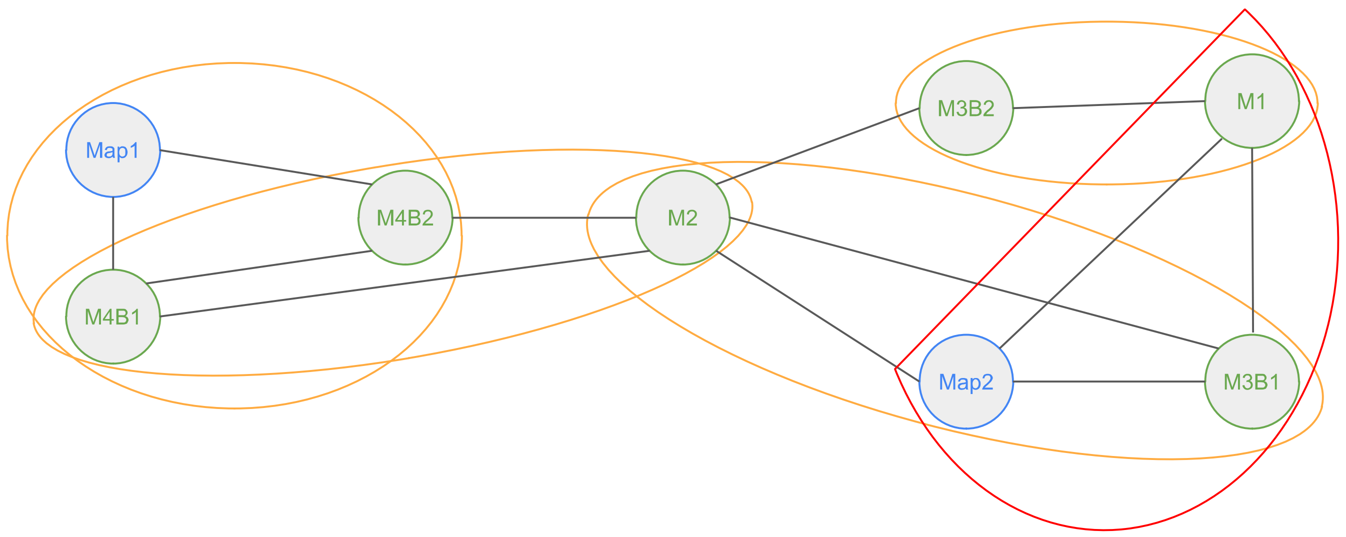
Building the Collision Graph - Finding layers

Edges:
~~Map1-M4B1~~
~~Map1-M4B2~~
~~M4B1-M4B2~~
~~M4B1-M2~~
~~M4B2-M2~~
~~M2-Map2~~
~~M2-M3B1~~
M2-M3B2
~~Map2-M3B1~~
Map2-M1
M3B1-M1
~~M3B2-M1~~



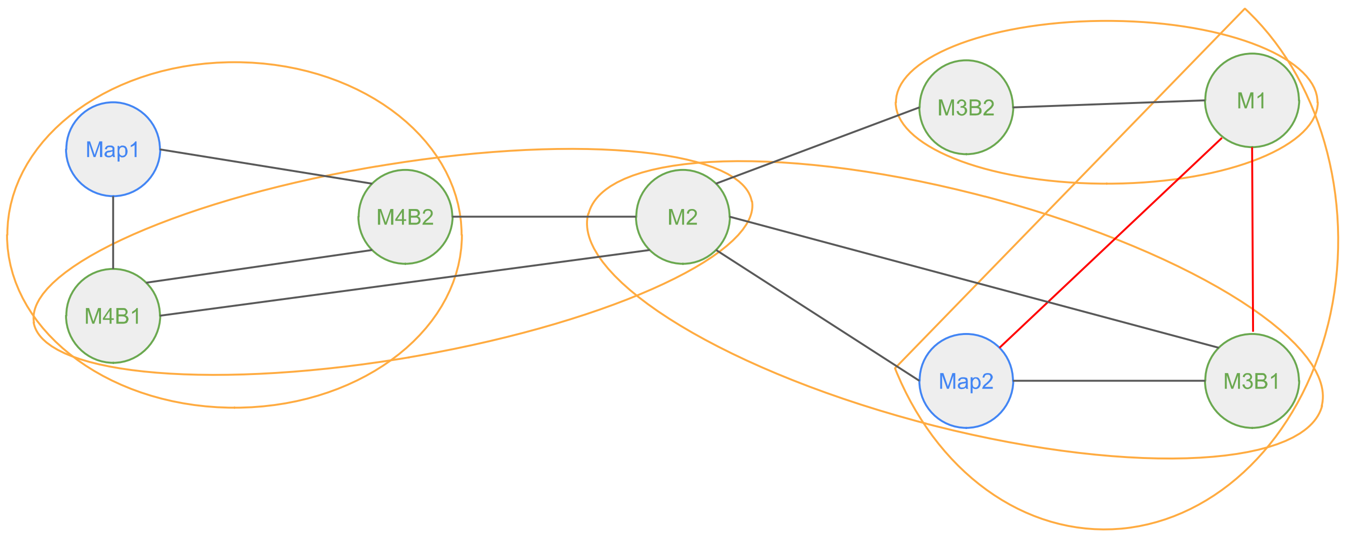
Building the Collision Graph - Finding layers

Edges:
~~Map1-M4B1~~
~~Map1-M4B2~~
~~M4B1-M4B2~~
~~M4B1-M2~~
~~M4B2-M2~~
~~M2-Map2~~
~~M2-M3B1~~
M2-M3B2
~~Map2-M3B1~~
Map2-M1
M3B1-M1
~~M3B2-M1~~



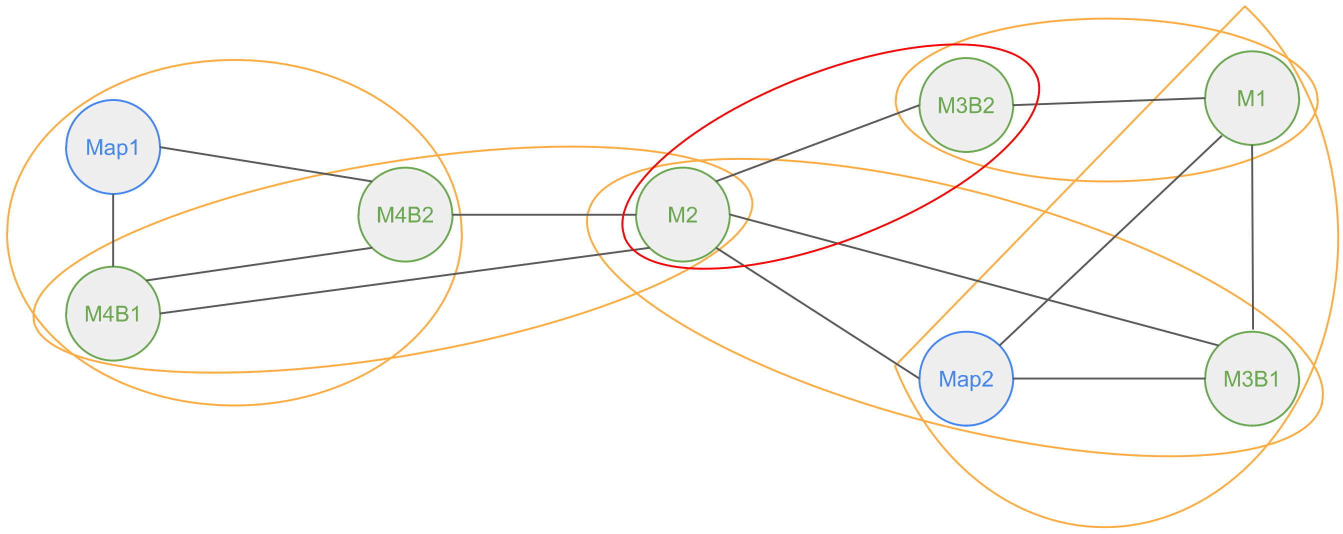
Building the Collision Graph - Finding layers

Edges:
~~Map1-M4B1~~
~~Map1-M4B2~~
~~M4B1-M4B2~~
~~M4B1-M2~~
~~M4B2-M2~~
~~M2-Map2~~
~~M2-M3B1~~
M2-M3B2
~~Map2-M3B1~~
~~Map2-M1~~
~~M3B1-M1~~
~~M3B2-M1~~



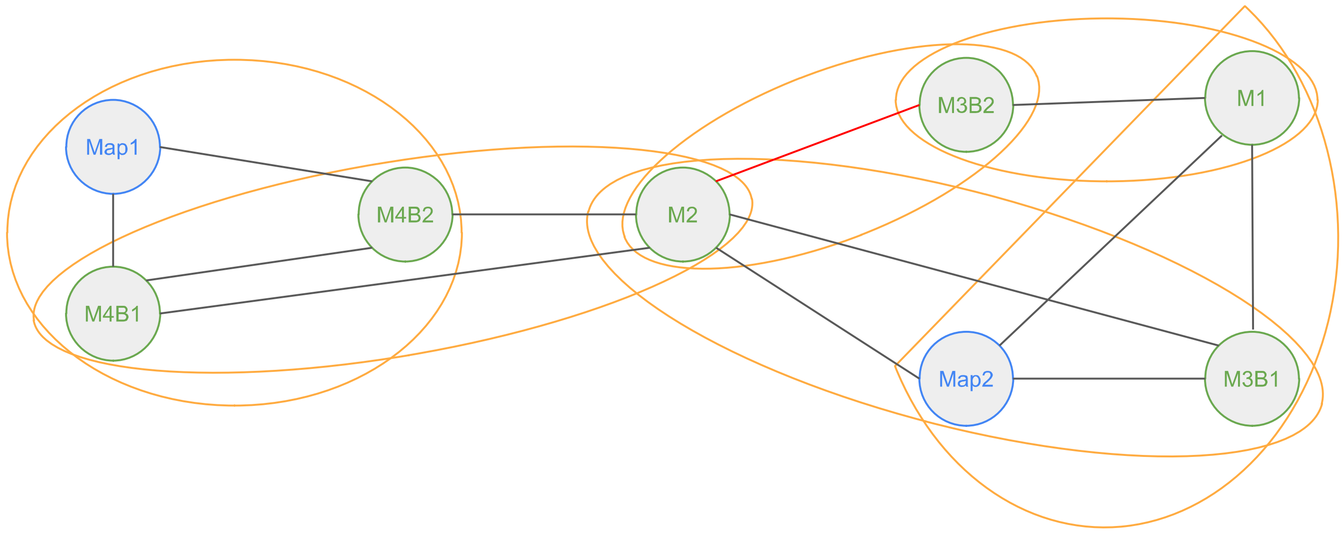
Building the Collision Graph - Finding layers

Edges:
~~Map1-M4B1~~
~~Map1-M4B2~~
~~M4B1-M4B2~~
~~M4B1-M2~~
~~M4B2-M2~~
~~M2-Map2~~
~~M2-M3B1~~
M2-M3B2
~~Map2-M3B1~~
~~Map2-M1~~
~~M3B1-M1~~
~~M3B2-M1~~

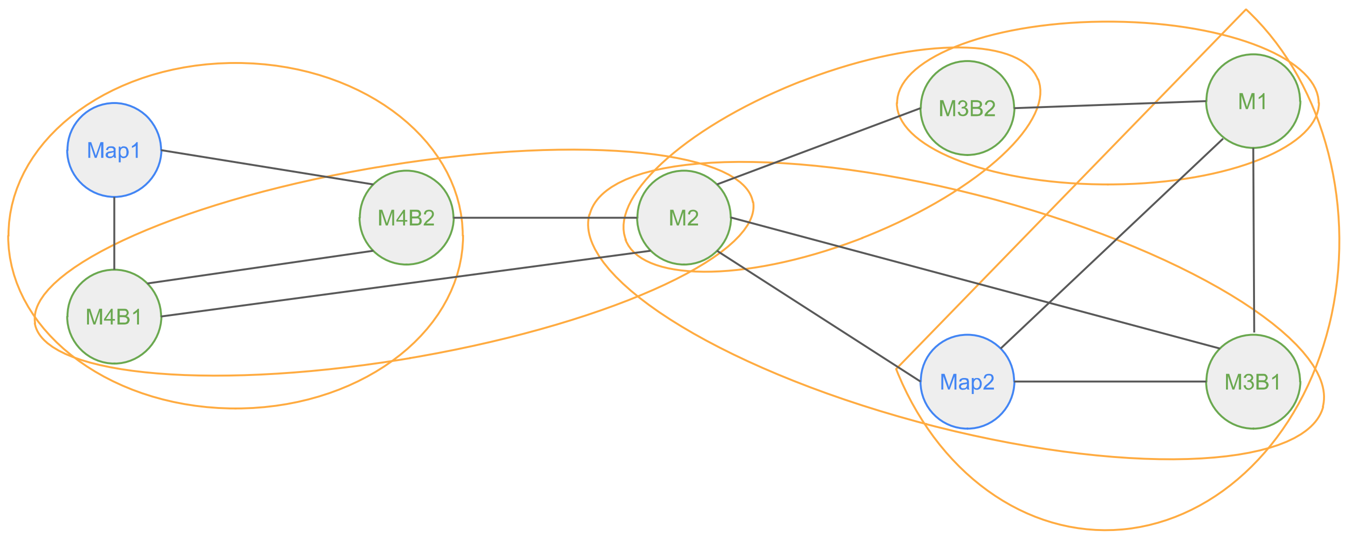


Building the Collision Graph - Finding layers

Edges:
~~Map1-M4B1~~
~~Map1-M4B2~~
~~M4B1-M4B2~~
~~M4B1-M2~~
~~M4B2-M2~~
~~M2-Map2~~
~~M2-M3B1~~
~~M2-M3B2~~
~~Map2-M3B1~~
~~Map2-M1~~
~~M3B1-M1~~
~~M3B2-M1~~

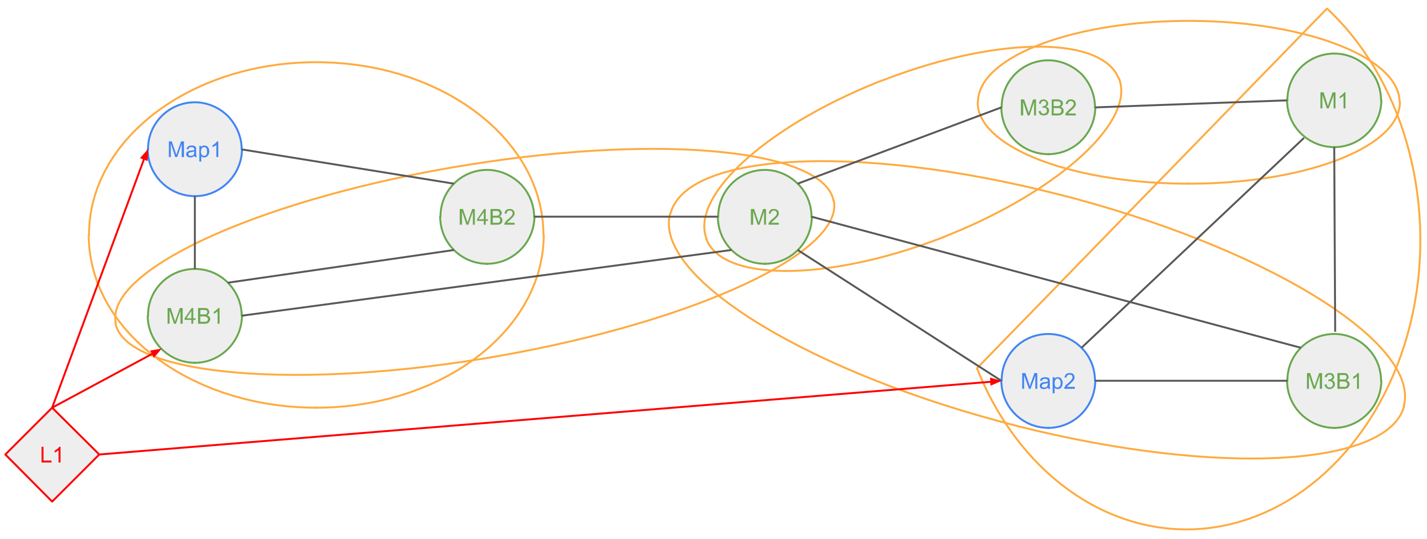


Final Collision Graph



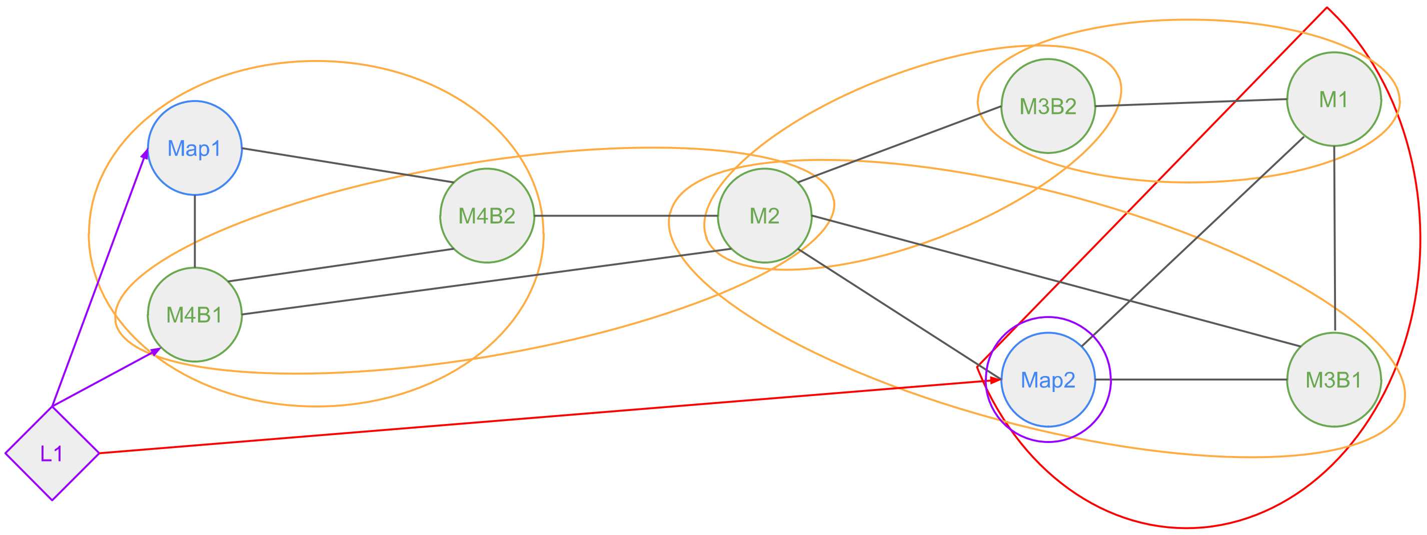
Adding Lasers to the Graph

Start adding lasers one at a time and connecting them to all nodes they must detect.

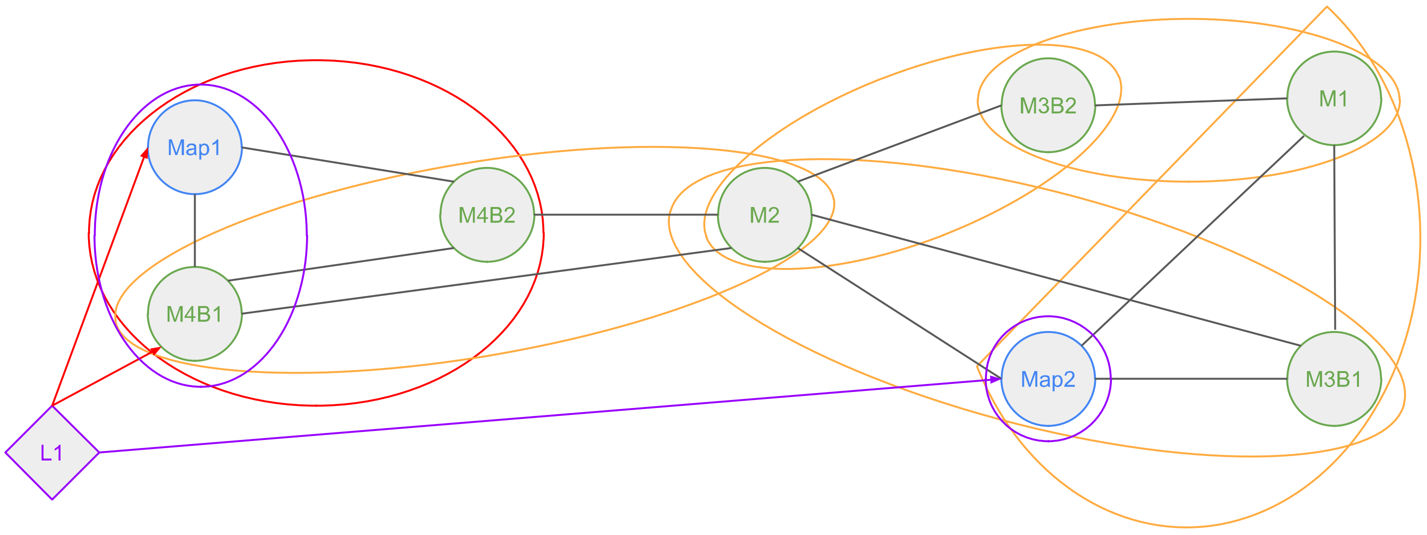


Adding Lasers to the Graph

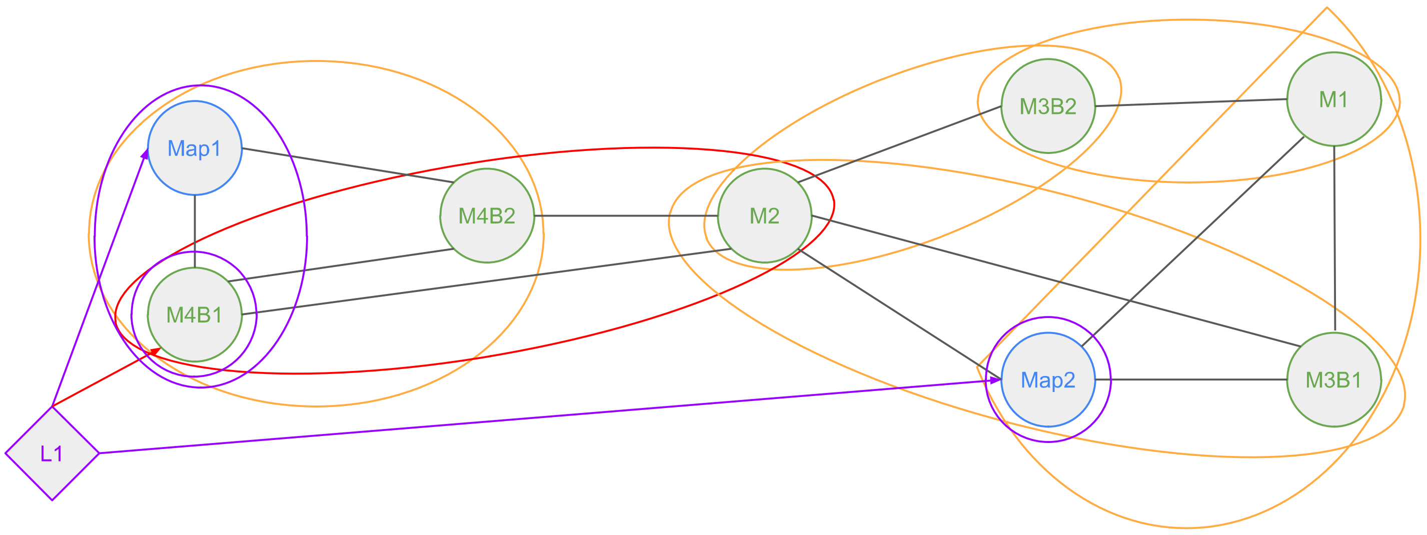
For each collision layer the laser interacts with, group the vertices inside it that are detected.



Adding Lasers to the Graph

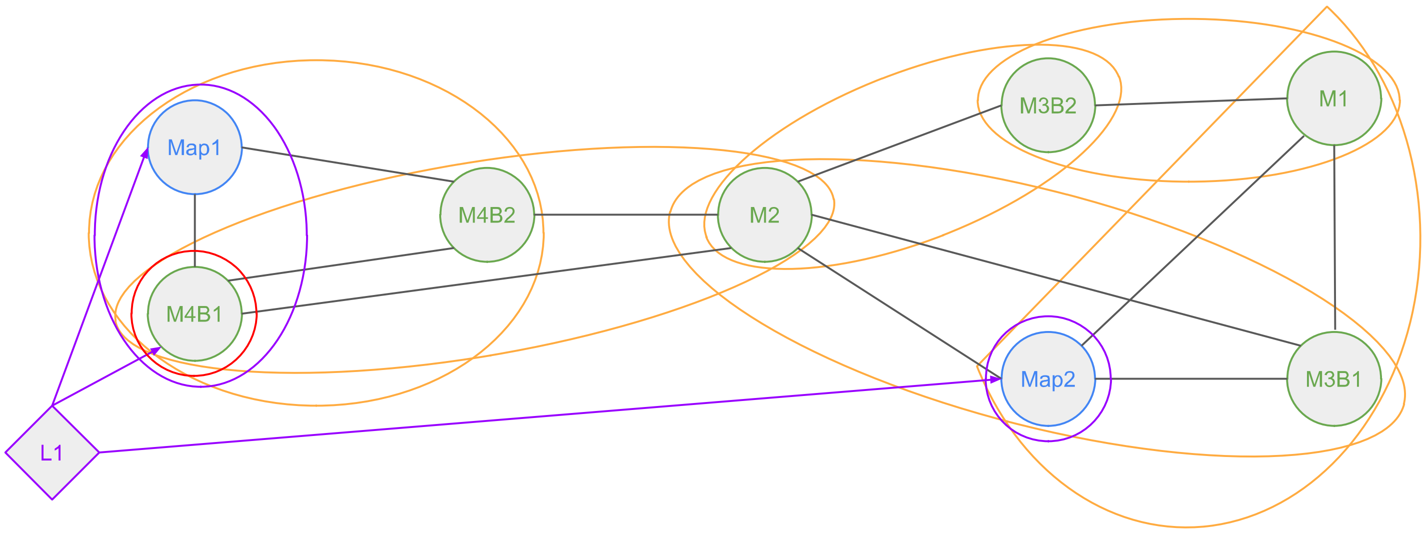


Adding Lasers to the Graph



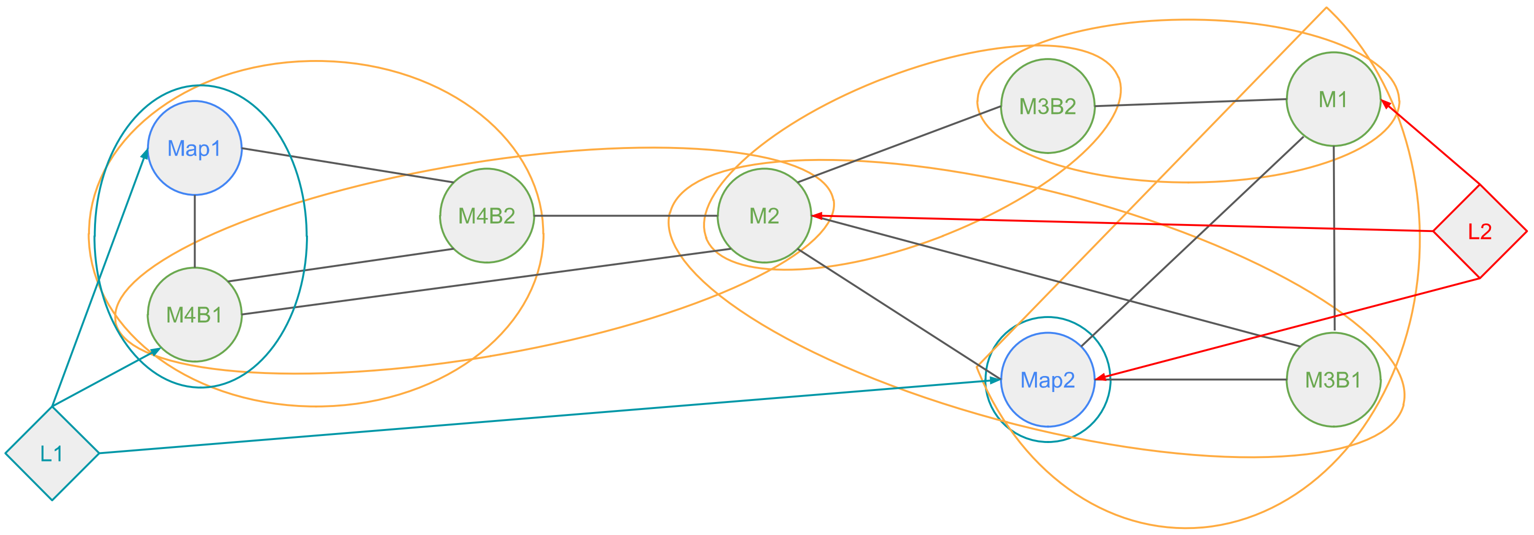
Adding Lasers to the Graph

After checking all layers, if all the vertices of one of the new groups created is contained inside another of the new ones, it can be removed.

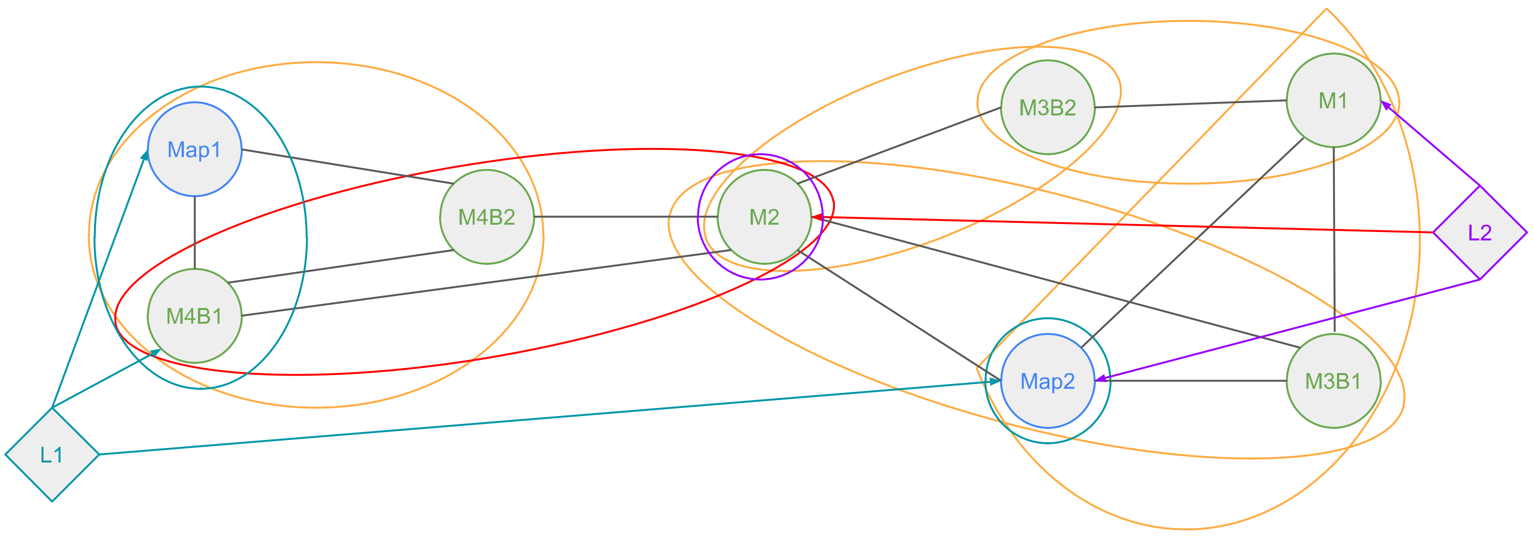


Adding Lasers to the Graph

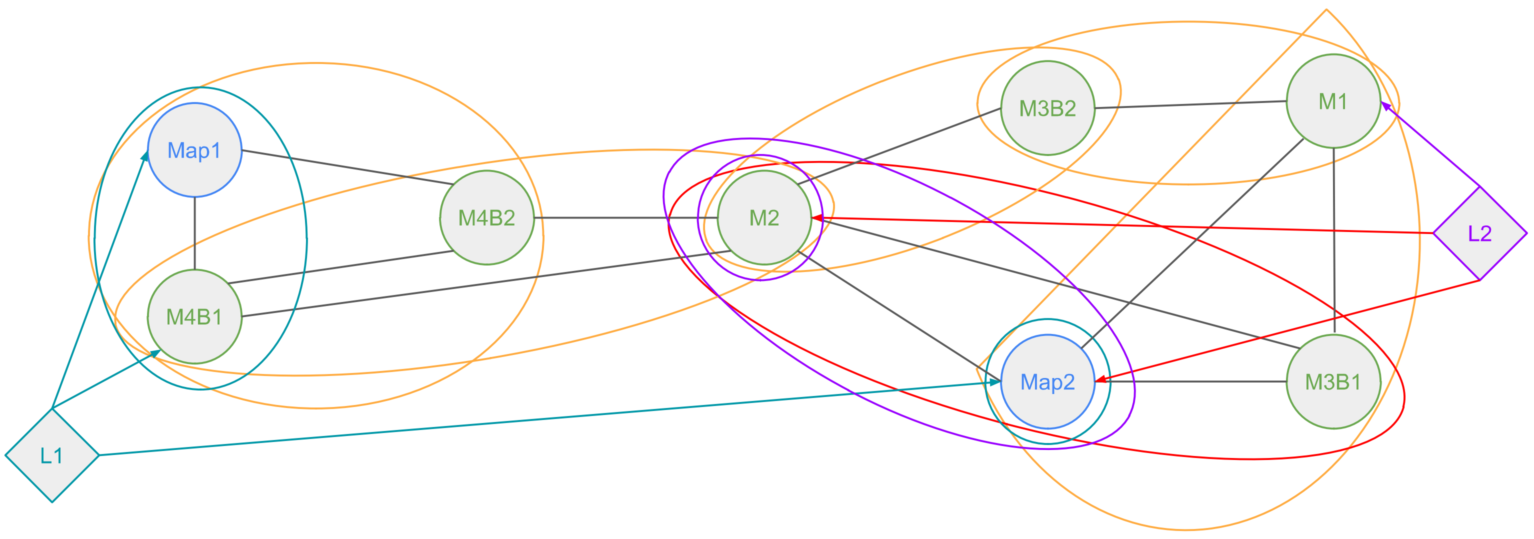
Repeat the same process for all the lasers.



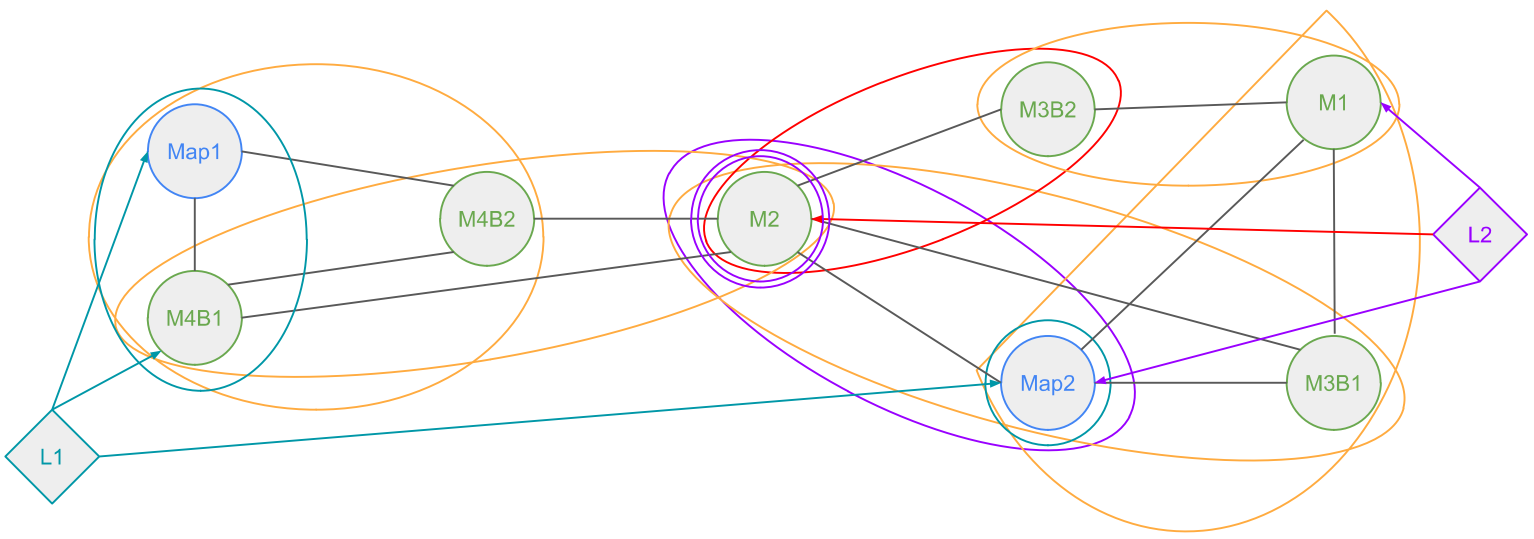
Adding Lasers to the Graph



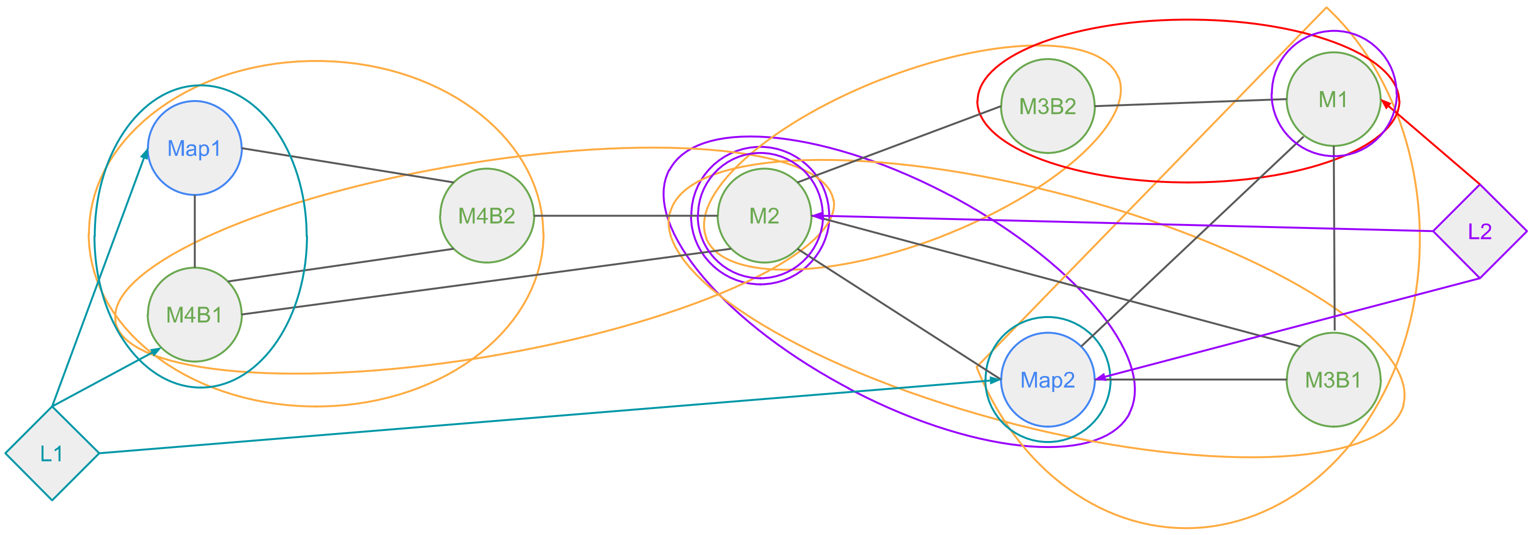
Adding Lasers to the Graph



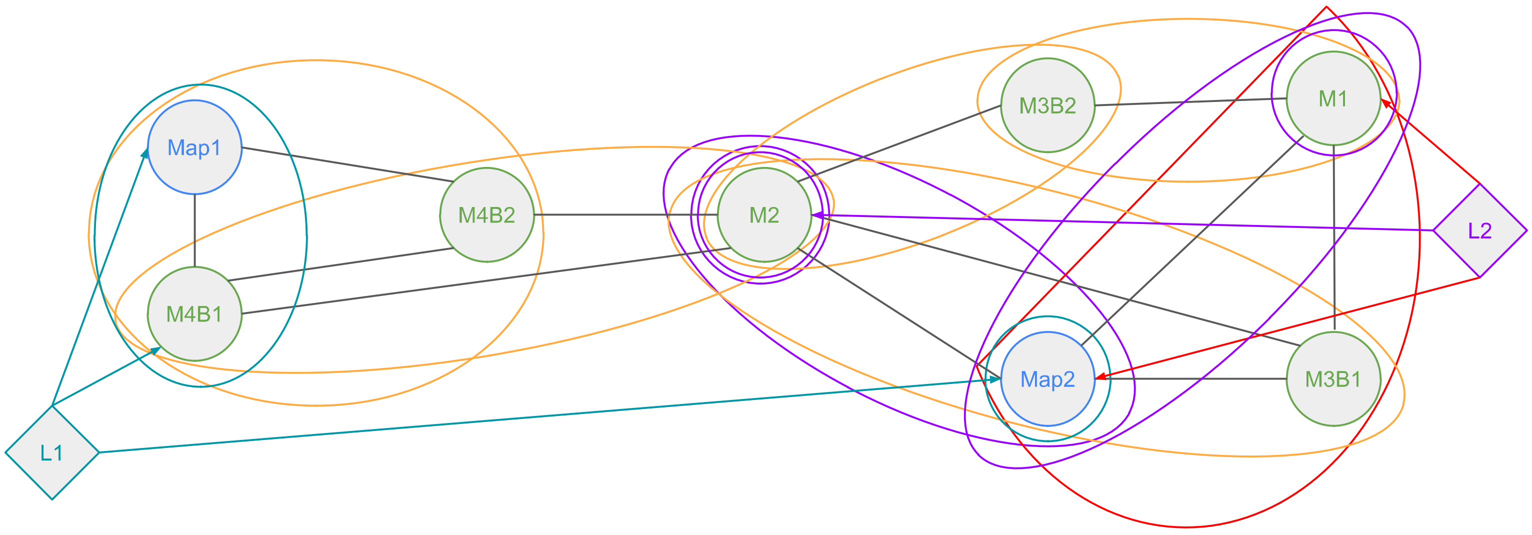
Adding Lasers to the Graph



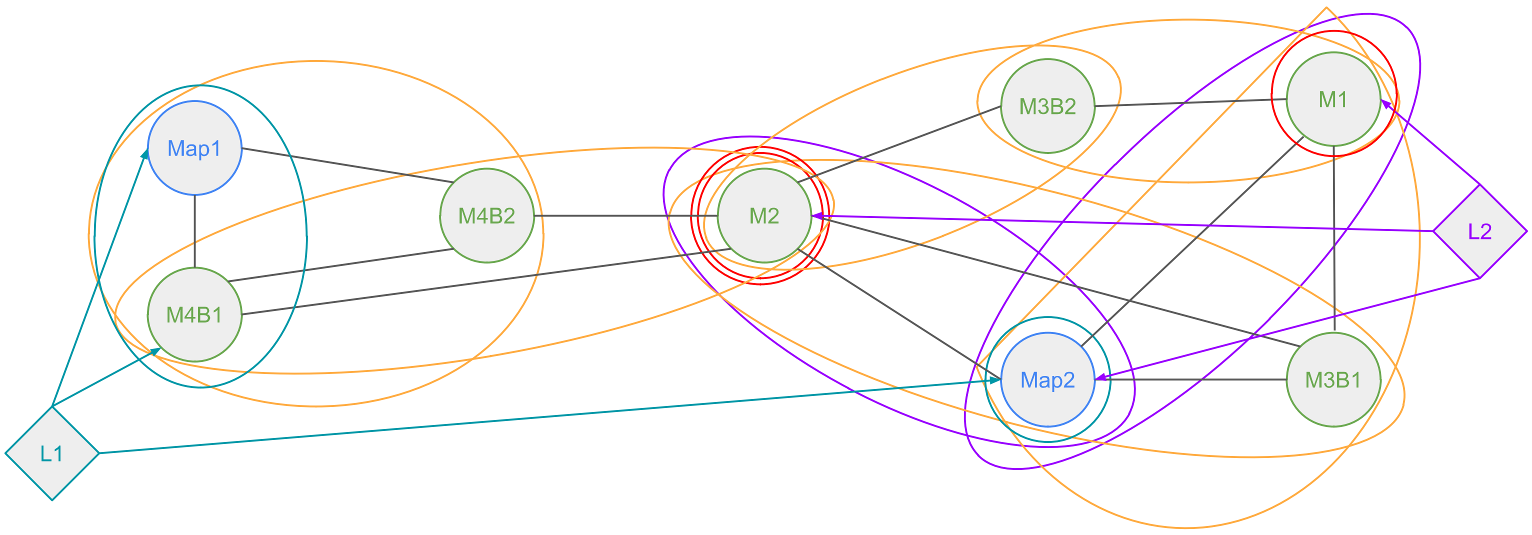
Adding Lasers to the Graph



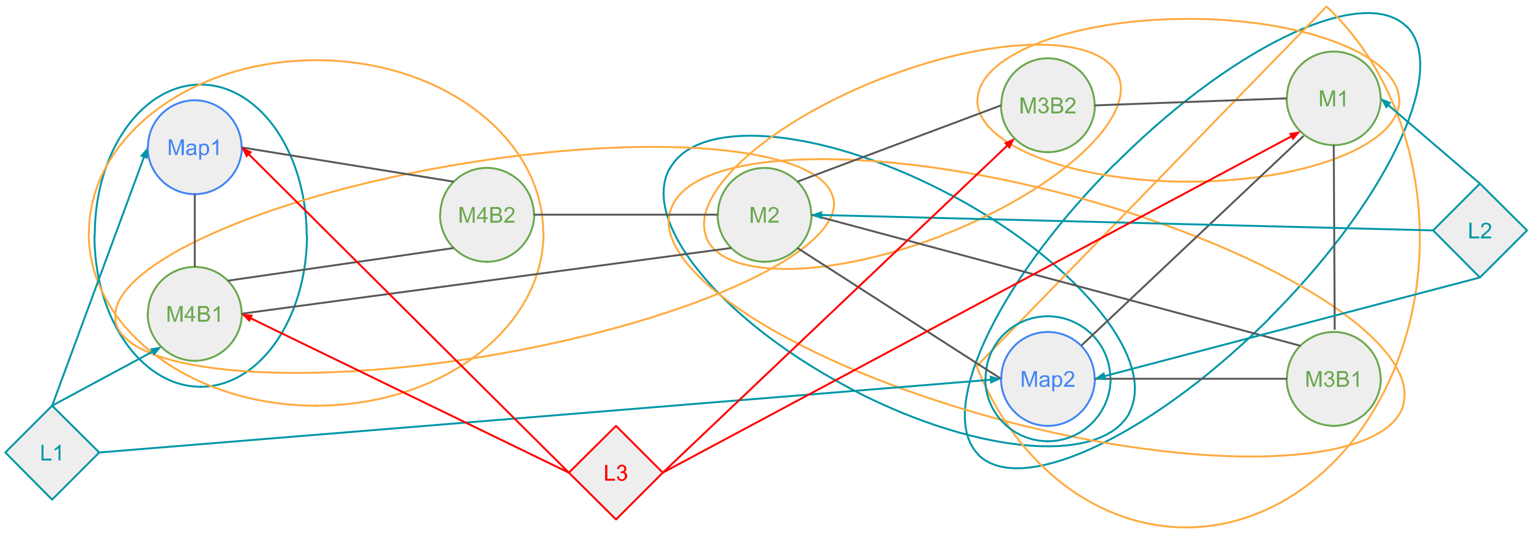
Adding Lasers to the Graph



Adding Lasers to the Graph

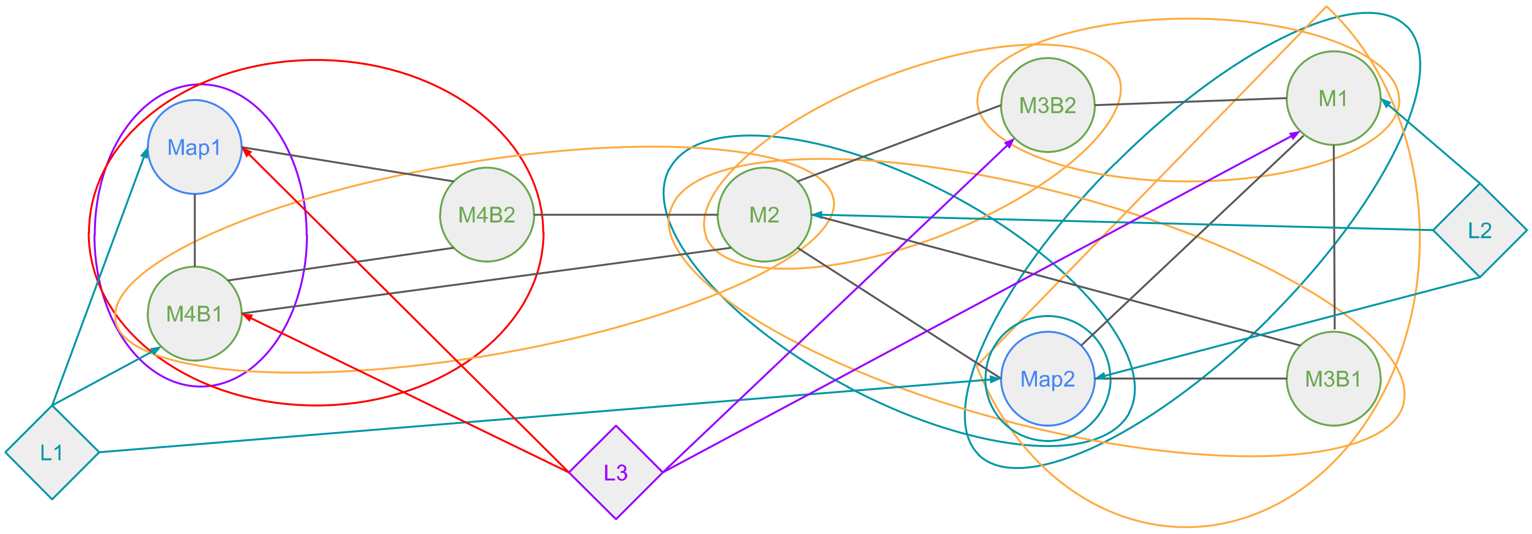


Adding Lasers to the Graph

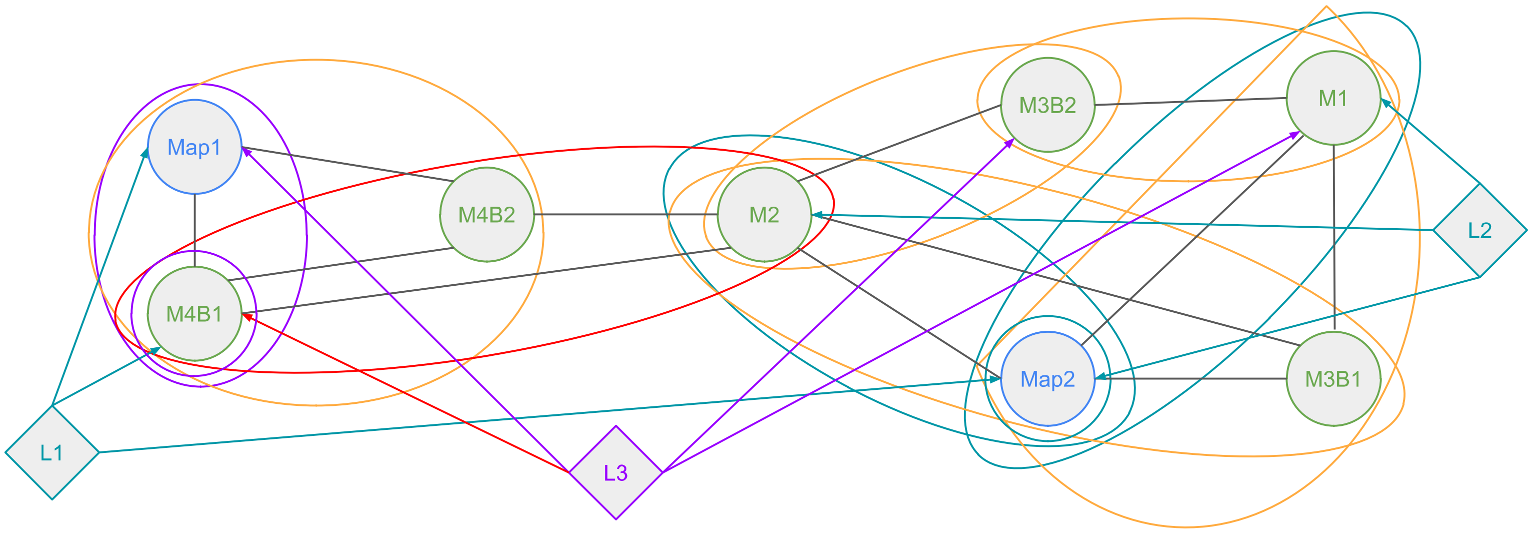


Adding Lasers to the Graph

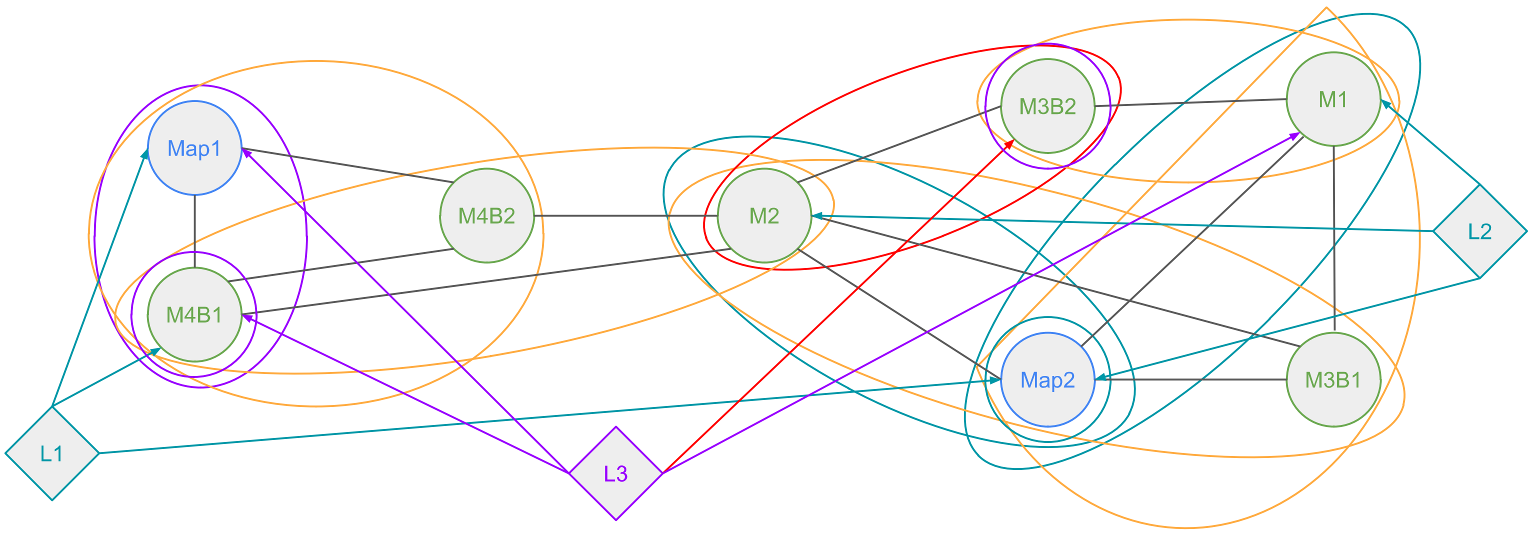
If at any point you get a group that already exist from a previous laser, it can be reused.



Adding Lasers to the Graph

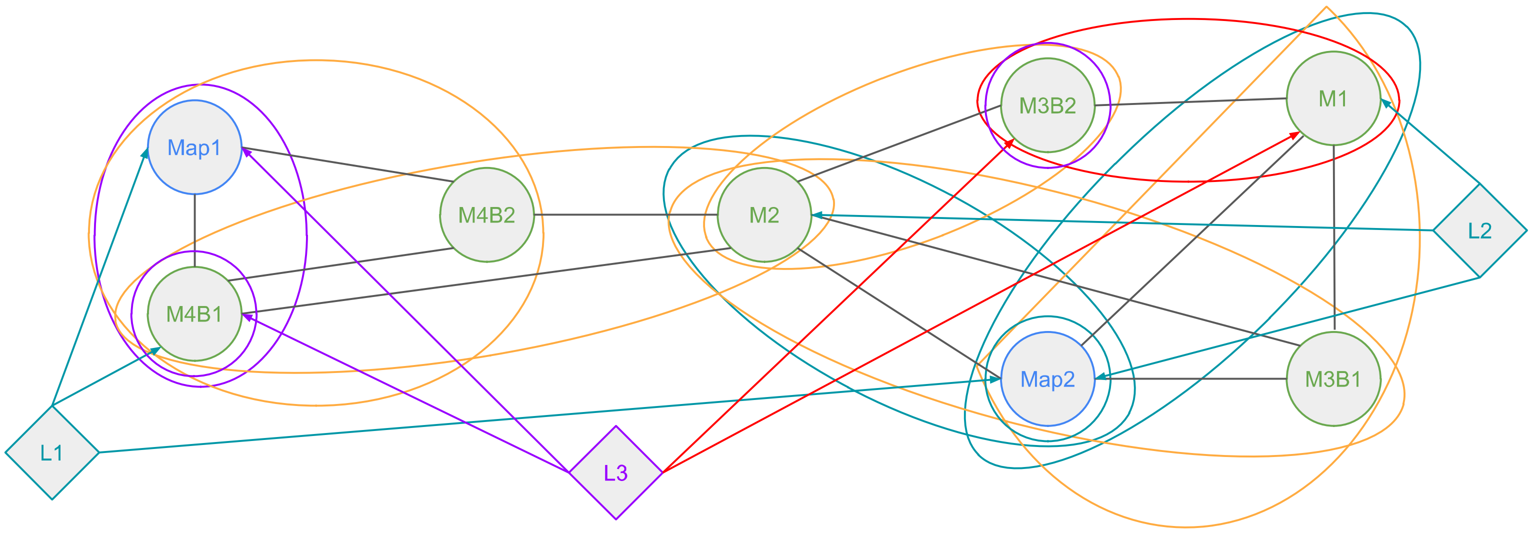


Adding Lasers to the Graph

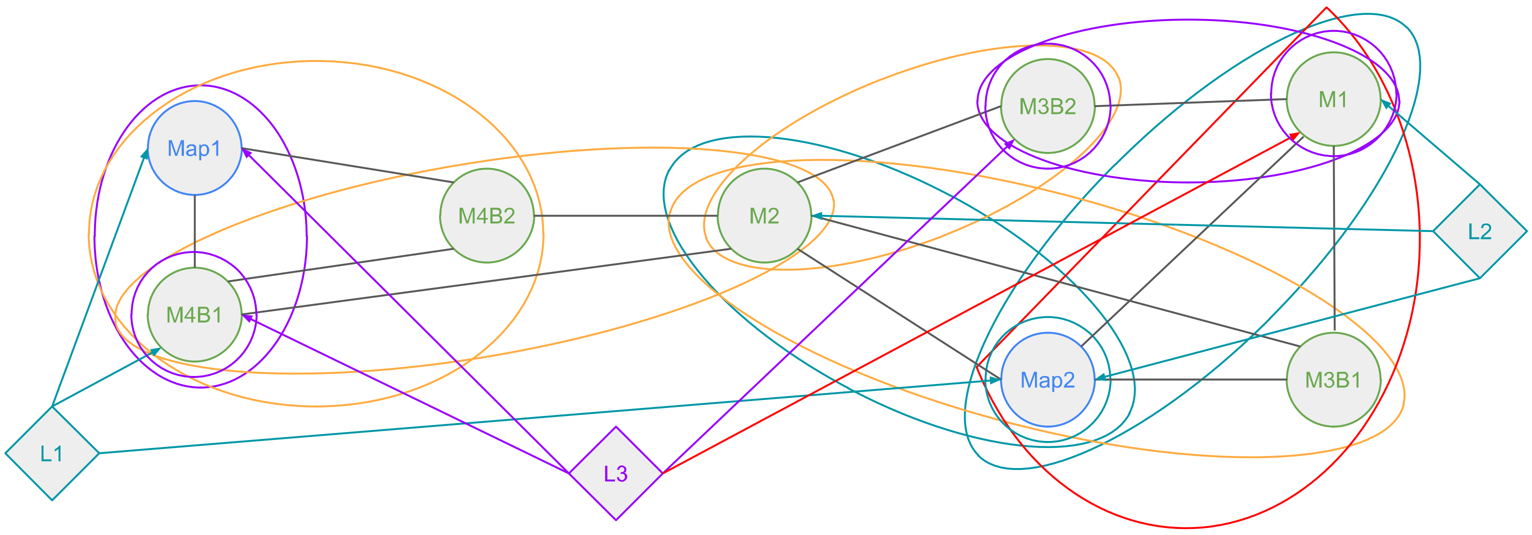


Adding Lasers to the Graph

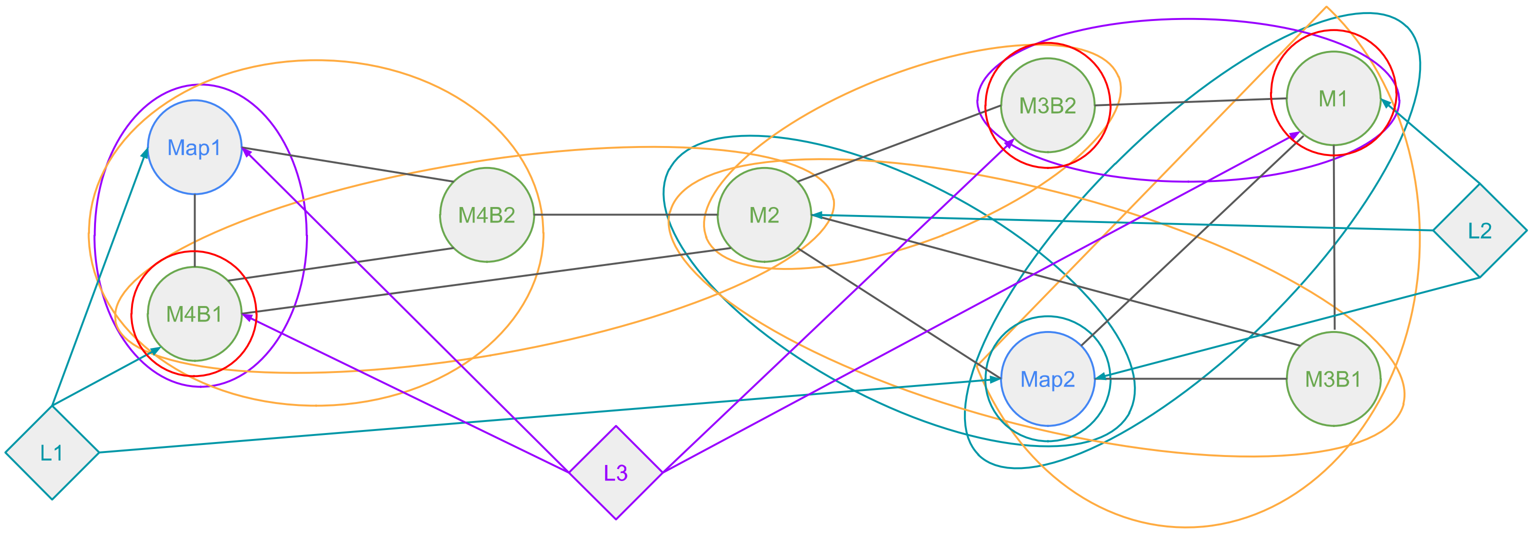
Collision groups can also be reused if all its elements need to be detected.



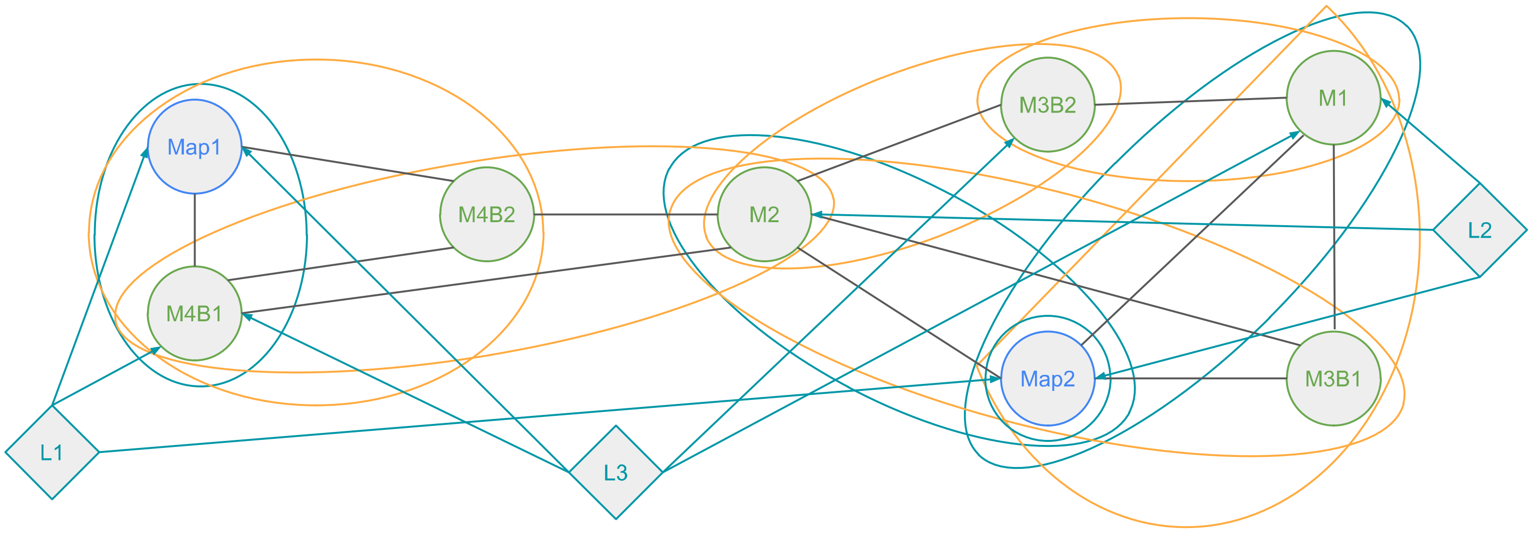
Adding Lasers to the Graph



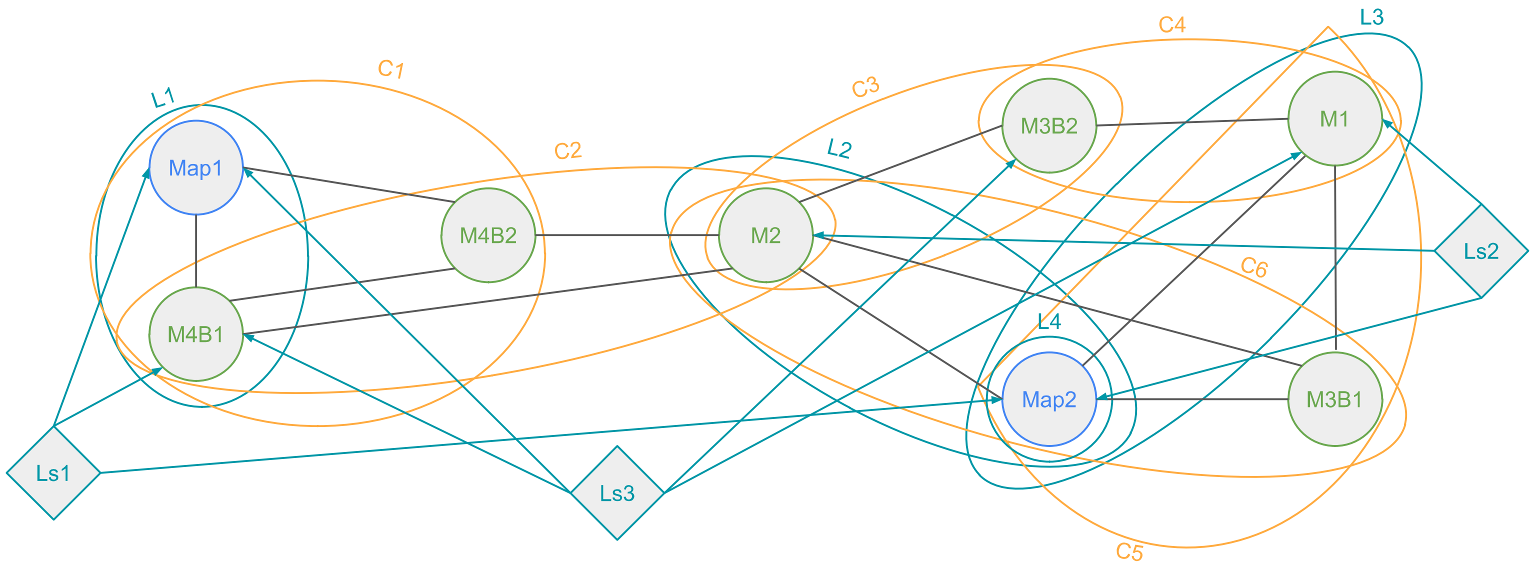
Adding Lasers to the Graph



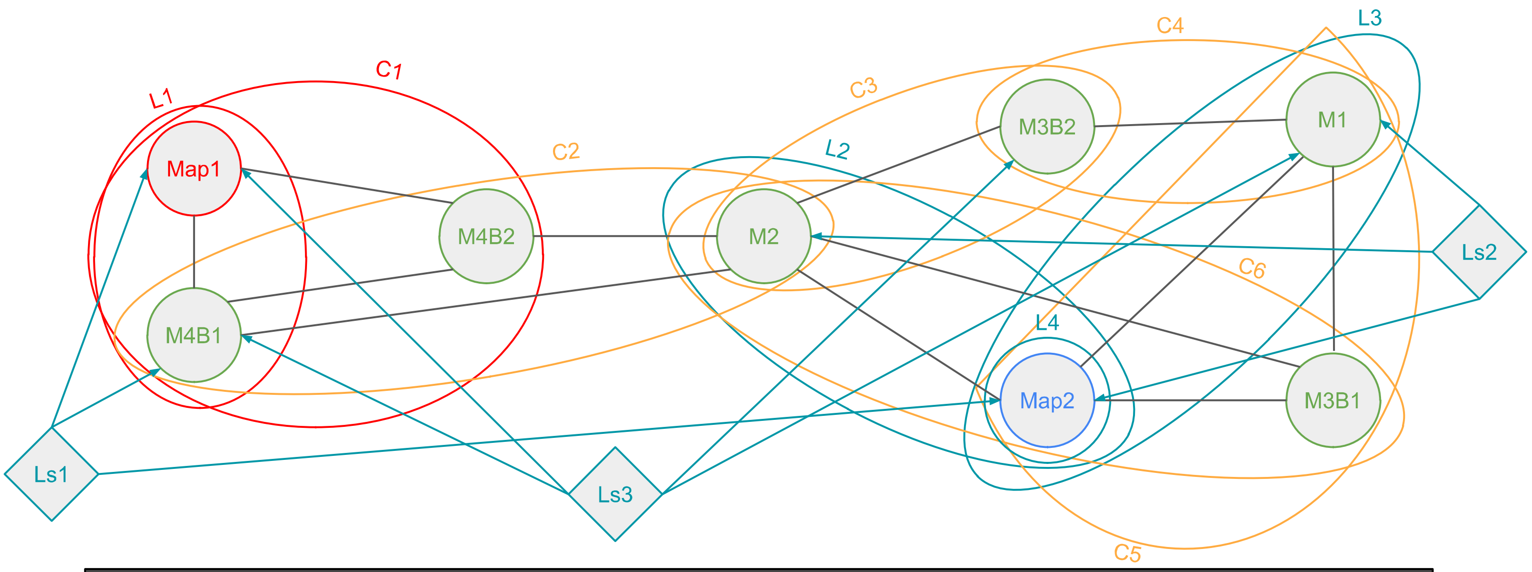
Final Graph



Label the layers

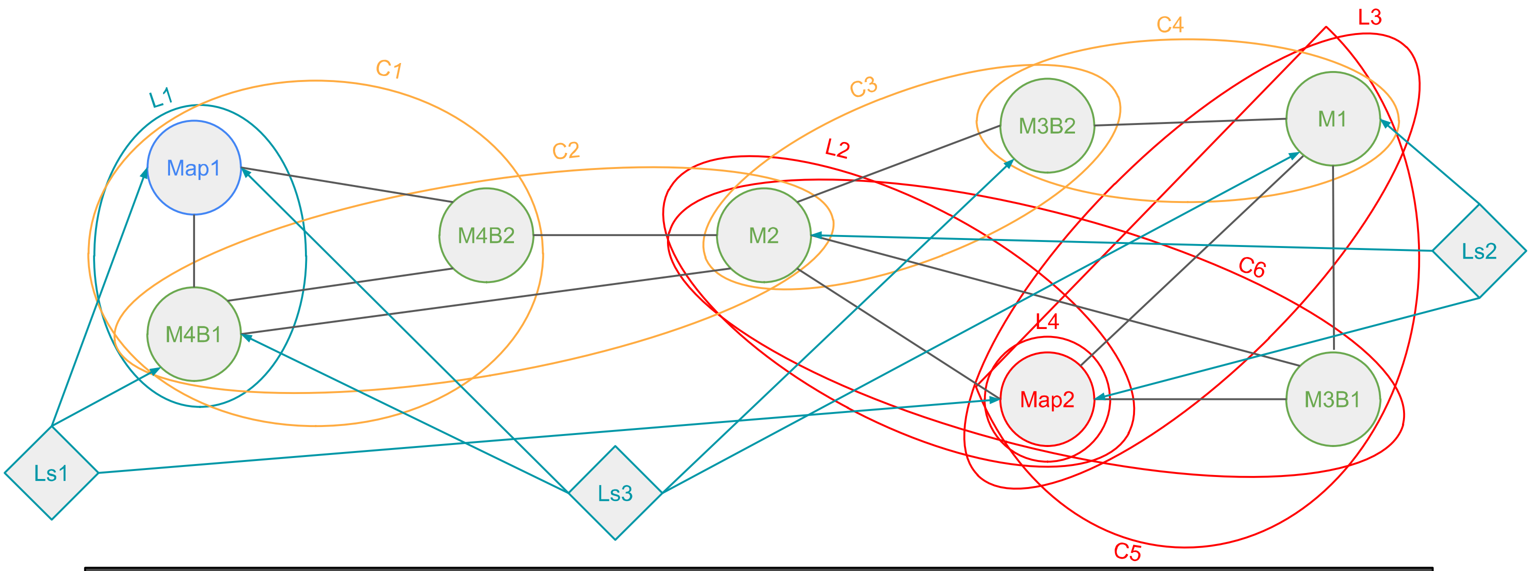


Add layers to the World file - Map1



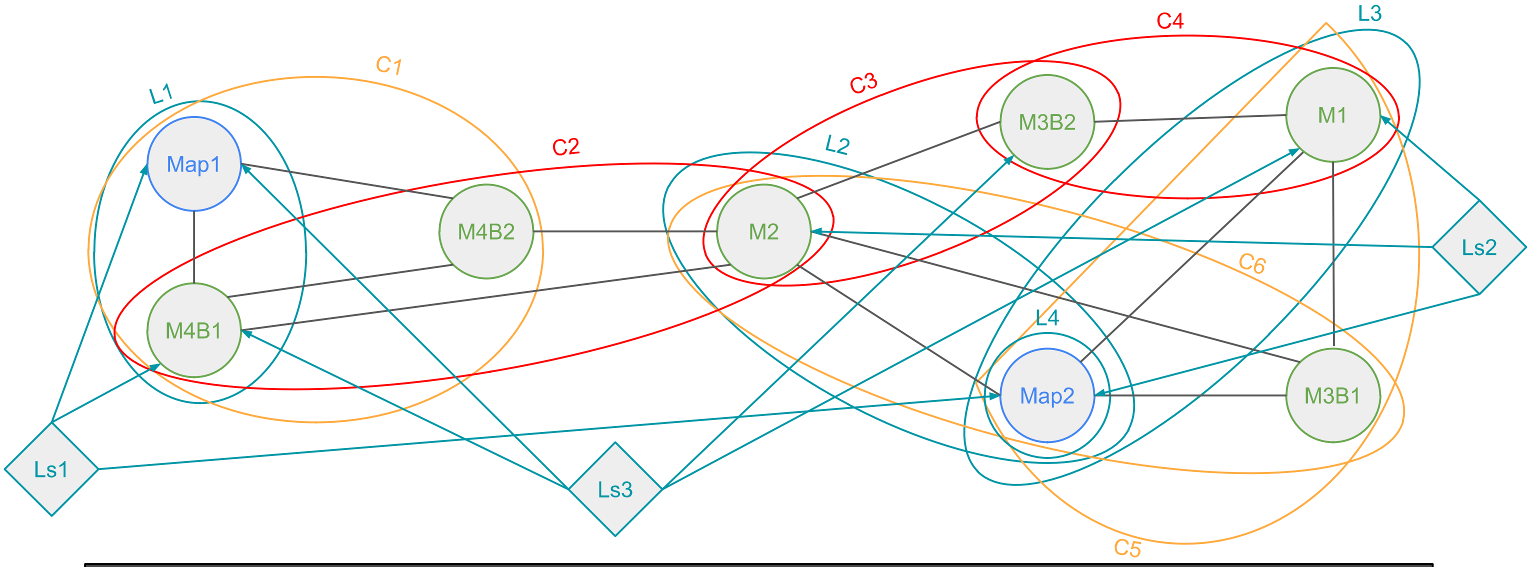
```
# map 1, add the layers that contain the node
- name: ["C1", "L1"]
  map: <image or dat file>
```

Add layers to the World file - Map2



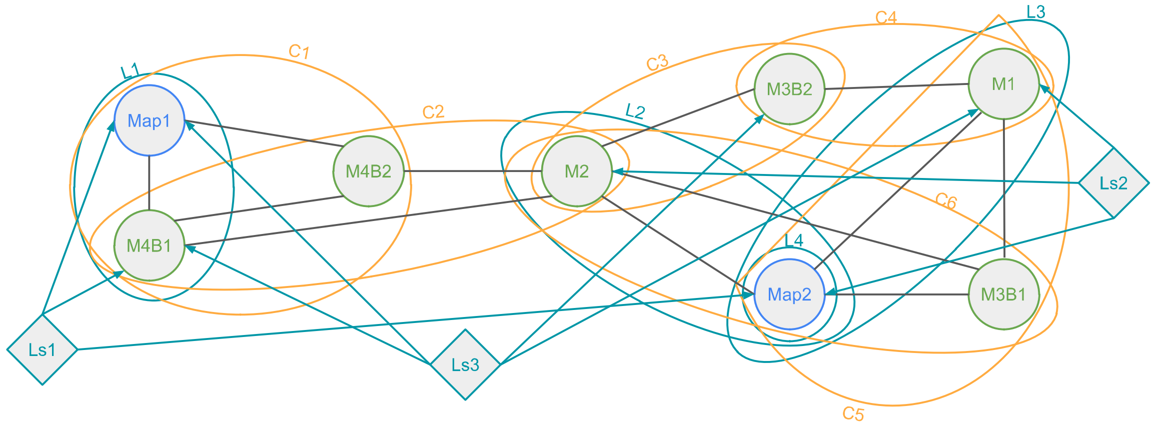
```
# map 2, add the layers that contain the node
- name: ["C5", "C6", "L2", "L3", "L4"]
  map: <image or dat file>
```

Add layers to the World file - Empty Map



```
# empty map, add the layers that are not contained in any other map
- name: [“C2”, “C3”, “C4”]
  map: <image or dat file>
```

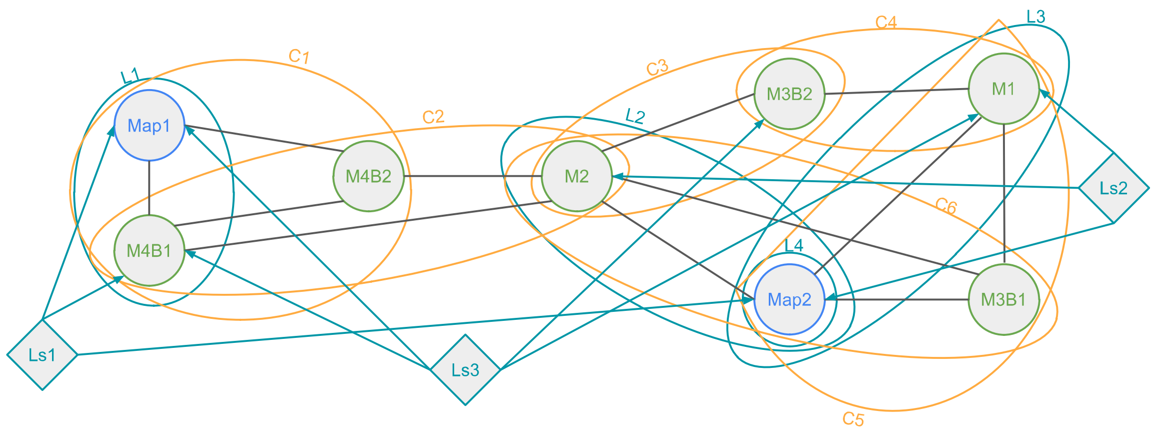
Assign layers to the models



Models (or bodies of the model) belong to all layers they are contained in:

- M1: C4, C5, L3
- M2: C2, C3, C6, L2
- M3:
  - B1: C5, C6
  - B2: C3, C4
- M4:
  - B1: C1, C2, L1
  - B2: C1, C2

Assign layers to the lasers

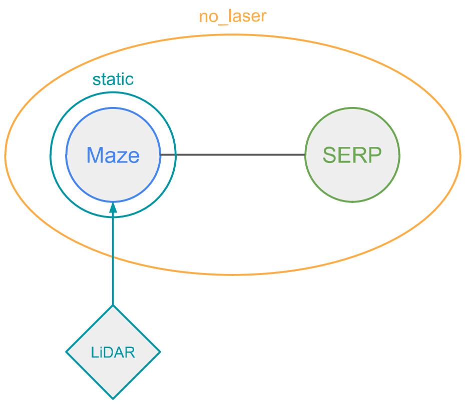


Lasers need to detect layers they are pointing to:

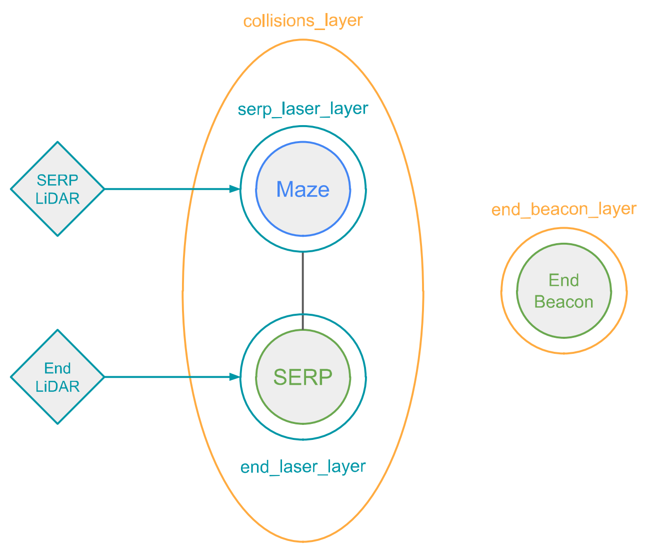
- Ls1: L1, L4
- Ls2: L2, L3
- Ls3: L1, C4

NOTE: There can be some redundancies in this step. Example: Ls2 is also pointing to L4 but since L2 (or L3) contains L4, it can be removed.

Graphs for the tutorial examples - Teleopkeys



Graphs for the tutorial examples - Reinforcement Learning



Appendix B

User Tests

This appendix contains the full Form and Quiz answered by the test subjects as well as the answers each one gave.

B.1 Kit Evaluation Forms

Flatland ROS 2 tutorials Evaluation

* Indicates required question

1. What is your current education level? (Highest degree you have already obtained) *

Mark only one oval.

- ◯ Grade 9
- ◯ Grade 10
- ◯ Grade 11
- ◯ Grade 12
- ◯ Bachelor's degree
- ◯ Master's degree
- ◯ PhD

Prior and gained knowledge

2. Before completing the tutorials, how much experience did you have in the following areas: *

Mark only one oval per row.

	1 - No experience	2	3	4	5	6	7 - A lot of experience
Robotics	◯	◯	◯	◯	◯	◯	◯
ROS 1	◯	◯	◯	◯	◯	◯	◯
ROS 2	◯	◯	◯	◯	◯	◯	◯
Robotic Simulators	◯	◯	◯	◯	◯	◯	◯
Flatland Simulator	◯	◯	◯	◯	◯	◯	◯
Reinforcement Learning	◯	◯	◯	◯	◯	◯	◯

3. After completing the tutorials, how much do you think you have learned in the following areas: *

Mark only one oval per row.

	1 - Nothing learned	2	3	4	5	6	7 - A lot learned
Robotics	◯	◯	◯	◯	◯	◯	◯
ROS 1	◯	◯	◯	◯	◯	◯	◯
ROS 2	◯	◯	◯	◯	◯	◯	◯
Robotic Simulators	◯	◯	◯	◯	◯	◯	◯
Flatland Simulator	◯	◯	◯	◯	◯	◯	◯
Reinforcement Learning	◯	◯	◯	◯	◯	◯	◯

Virtual Machine

4. Did you use the VM to follow the tutorials? *

Mark only one oval.

◯ Yes *Skip to question 6*

◯ No *Skip to question 5*

Setup and Pre-requisites

5. How difficult was to install and setup the pre-requisites? *

Mark only one oval per row.

	1 - Very Easy	2	3	4	5	6	7 - Very Hard
Install ROS 2 Humble	⬭	⬭	⬭	⬭	⬭	⬭	⬭
Setup Workspace	⬭	⬭	⬭	⬭	⬭	⬭	⬭
Install Flatland 2	⬭	⬭	⬭	⬭	⬭	⬭	⬭
Stable-Baselines3	⬭	⬭	⬭	⬭	⬭	⬭	⬭

Flatland Teleopkeys Tutorial using ROS 2

https://github.com/FilipeAlmeidaFEUP/ros2_flatland_rl_tutorial

6. Were you able to run the package and see the robot move on its own? *

Mark only one oval.

⬭ Yes
⬭ No

7. Were you able to run the second node to read keystrokes and control the robot with the keyboard? *

Mark only one oval.

⬭ Yes
⬭ No

8. How useful did you find this tutorial to fulfill the following goals? *

Mark only one oval per row.

	1 - Not helpful at all	2	3	4	5	6	7 - Extremely helpful
Prepare a machine to develop basic ROS 2 packages with Flatland.	◯	◯	◯	◯	◯	◯	◯
Analyse the progress of a virtual robot in 2D simulation with visualization tools.	◯	◯	◯	◯	◯	◯	◯
Analyse the code controlling a virtual robot in 2D simulation.	◯	◯	◯	◯	◯	◯	◯
Modify the code controlling a virtual robot in 2D simulation.	◯	◯	◯	◯	◯	◯	◯

9. How easy do you think these subjects are to understand? *

Mark only one oval per row.

	1 - Very simple	2	3	4	5	6	7 - Very complicated
ROS 2 Node structure	◯	◯	◯	◯	◯	◯	◯
ROS Topics and Services	◯	◯	◯	◯	◯	◯	◯
ROS 2 launch file	◯	◯	◯	◯	◯	◯	◯
Flatland worlds	◯	◯	◯	◯	◯	◯	◯
Flatland layers	◯	◯	◯	◯	◯	◯	◯
Flatland models and plugins	◯	◯	◯	◯	◯	◯	◯

10. How well do you think these subjects were explained in the tutorials? *

Mark only one oval per row.

	1 - Not explained	2	3	4	5	6	7 - Very well explained
ROS 2 Node structure	◯	◯	◯	◯	◯	◯	◯
ROS Topics and Services	◯	◯	◯	◯	◯	◯	◯
ROS 2 launch file	◯	◯	◯	◯	◯	◯	◯
Flatland worlds	◯	◯	◯	◯	◯	◯	◯
Flatland layers	◯	◯	◯	◯	◯	◯	◯
Flatland models and plugins	◯	◯	◯	◯	◯	◯	◯

11. Do you have any suggestions to improve the tutorial or problems you ran into while doing it?

Flatland Reinforcement Learning Tutorial using ROS 2

https://github.com/FilipeAlmeidaFEUP/ros2_teleopkeys_tutorial

12. Were you able to run the package? *

Mark only one oval.

- Yes
- No

13. Were you able to train the robot to complete the task? *

Mark only one oval.

- Yes
- No

14. How useful did you find this tutorial to fulfill the following goals? *

Mark only one oval per row.

	1 - Not helpful at all	2	3	4	5	6	7 - Extremely helpful
Prepare a machine to run a reinforcement learning system.							
Analyse the behavior of a reinforcement learning agent.							
Modify the learning parameters of a reinforcement learning agent.							

15. How easy do you think these subjects are to understand? *

Mark only one oval per row.

	1 - Very simple	2	3	4	5	6	7 - Very complicated
RL Agent	◯	◯	◯	◯	◯	◯	◯
RL Environment	◯	◯	◯	◯	◯	◯	◯
State and Action spaces	◯	◯	◯	◯	◯	◯	◯
Steps and Episodes	◯	◯	◯	◯	◯	◯	◯

16. How well do you think these subjects were explained in the tutorials? *

Mark only one oval per row.

	1 - Not explained	2	3	4	5	6	7 - Very well explained
RL Agent	◯	◯	◯	◯	◯	◯	◯
RL Environment	◯	◯	◯	◯	◯	◯	◯
State and Action spaces	◯	◯	◯	◯	◯	◯	◯
Steps and Episodes	◯	◯	◯	◯	◯	◯	◯

17. Do you have any suggestions to improve the tutorial or problems you ran into while doing it?

____

____

____

____

____

Managing Layers in Flatland slideshow

https://docs.google.com/presentation/d/1KqJDQR_PBaGtS-kA5KgsTbaRThu_uP2U8NJeoSP0GDE/edit?usp=sharing

18. Did you read the slideshow? *

Mark only one oval.

- ◯ Yes
- ◯ No *Skip to question 21*

Managing Layers in Flatland slideshow

https://docs.google.com/presentation/d/1KqJDQR_PBaGtS-kA5KgsTbaRThu_uP2U8NJeoSP0GDE/edit?usp=sharing

19. How simple was it to understand the method to determine the layers? *

Mark only one oval.

	Very simple
1	◯
2	◯
3	◯
4	◯
5	◯
6	◯
7	◯
	Very hard

20. Do you have any suggestions to improve the slideshow?

ROS 1 and ROS 2 comparison

21. Do you have any prior experience using ROS 1? *

Mark only one oval.

- Yes
- No

ROS 1 and ROS 2 comparison

22. What are the main differences you encountered between using ROS 1 and ROS 2? Do you think any of them is better or easier to use?

B.2 Quiz

ROS 2, Flatland and Reinforcement Learning Quiz

ROS 2

1. If you make modifications in one of your packages, inside which folder do you need to run the command build the changes?

   Mark only one oval.

   - Workspace folder
   - Root folder
   - ROS 2 installation folder
   - The modified package folder

2. Imagine you are developing a controller for a robot and need to know its position but only when a collision with a wall is detected. The most efficient way for the robot to share this information is to use a:

   Mark only one oval.

   - ROS Service
   - ROS Setup file
   - ROS Node
   - ROS Topic

3. What should be used to communicate with a Node that only has the task of controlling the speed of a motor in a robot?

Mark only one oval.

- ROS Topic
- ROS Setup file
- ROS Node
- ROS Service

4. Which statement is false about ROS Nodes:

Mark only one oval.

- A Node can be a client for a Service and a server for other simultaniously.
- Nodes can only comunicate with other Nodes that were launched by the same launch file.
- Nodes are the vertices of the ROS graph.
- A single robot can be controlled by many Nodes.

Flatland

5. The image that is used to represent the walls of the map needs to be sourced in which file?

Mark only one oval.

- Layer file
- RViz file
- World file
- Model file

6. In the following layer setup:
Robot 1 -> Layer 1, Layer 3
Robot 2 -> Layer 2, Layer 3
Robot 3 -> Layer 2

Which robots collide with each other?

Mark only one oval.

- ◯ Robot 1 collides with 2, Robot 2 collides with 3.
- ◯ All the robots collide.
- ◯ There are no collisions.
- ◯ Robot 3 collides with 2, Robot 3 collides with 1.

7. In the following layer setup:
Robot 1 -> Layer 1, Layer 3
Robot 2 -> Layer 2, Layer 3
Robot 3 -> Layer 2

Laser -> Layer 2, Layer 3

The Laser detects:

Mark only one oval.

- ◯ All the Robots.
- ◯ None of the Robots.
- ◯ Only Robot 2.
- ◯ Robots 1 and 2.

8. Which of these Flatland plugins subscribes to a topic?

Mark only one oval.

- ◯ Diff Drive
- ◯ Bumper
- ◯ Laser
- ◯ GPS

Reinforcement Learning

9. The possible actions for any given state are defined by the:

Mark only one oval.

- Environment
- Agent
- Observation space
- Reward

10. When a final state is reached, this means that:

Mark only one oval.

- The environment needs to be reset to an initial state.
- The agent failed to complete the task.
- The next action chosen by the agent depens on that state.
- The agent was succesful in completing the task.

11. Considering the concepts of step and episode in reinforcement learning, which of these statements is false?

Mark only one oval.

- Each episode needs to have the same number of steps.
- An episode is a set of steps.
- In each step, an action is performed and a reward is atributed.
- Steps can have different time durations.

12. Each algorithm defines its own policy to decide which action to take based on the state. The entity that makes this decision is also known as the:

Mark only one oval.

- Agent
- Neural Network
- State space
- Environment

B.3 Form and Quiz answers

Table B.1: Form questions numbered (1/3)

Type	Question	N
Multiple Choice	What is your current education level? (Highest degree you have already obtained)	1
1 - 7 Eval	Before completing the tutorials, how much experience did you have in the following areas:	
	Robotics	2
	ROS 1	3
	ROS 2	4
	Simulators	5
	Flatland	6
	RL	7
	After completing the tutorials, how much do you think you have learned in the following areas:	
	Robotics	8
	ROS 1	9
	ROS 2	10
	Simulators	11
	Flatland	12
	RL	13
	How difficult was to install and setup the pre-requisites?	
	Install ROS 2 Humble	14
	Setup Workspace	15
	Install Flatland 2	16
	Stable-Baselines3	17
	How useful did you find this tutorial to fulfill the following goals? (tutorial 1)	
	Prepare a machine to develop basic ROS 2 packages with Flatland.	18
	Analyse the progress of a virtual robot in 2D simulation with visualization tools.	19
	Analyse the code controlling a virtual robot in 2D simulation.	20
	Modify the code controlling a virtual robot in 2D simulation.	21

Table B.2: Form questions numbered (2/3)

Type	**Question**	**N**
1 - 7 Eval	How easy do you think these subjects are to understand? (tutorial 1)	
	ROS 2 Node structure	22
	ROS Topics and Services	23
	ROS 2 launch file	24
	Flatland worlds	25
	Flatland layers	26
	Flatland models and plugins	27
	How well do you think these subjects were explained in the tutorials? (tutorial 1)	
	ROS 2 Node structure	28
	ROS Topics and Services	29
	ROS 2 launch file	30
	Flatland worlds	31
	Flatland layers	32
	Flatland models and plugins	33
	How useful did you find this tutorial to fulfill the following goals? (tutorial 2)	
	Prepare a machine to run a reinforcement learning system.	34
	Analyse the behavior of a reinforcement learning agent.	35
	Modify the learning parameters of a reinforcement learning agent.	36
	How easy do you think these subjects are to understand? (tutorial 2)	
	RL Agent	37
	RL Environment	38
	State and Action spaces	39
	Steps and Episodes	40
	How well do you think these subjects were explained in the tutorials? (tutorial 2)	
	RL Agent	41
	RL Environment	42
	State and Action spaces	43
	Steps and Episodes	44
	How simple was it to understand the method to determine the layers?	45

Table B.3: Form questions numbered (3/3)

Type	Question	N
Yes/No	Did you use the VM to follow the tutorials?	46
	Were you able to run the package and see the robot move on its own?	47
	Were you able to run the second node to read keystrokes and control the robot with the keyboard?	48
	Were you able to run the package?	49
	Were you able to train the robot to complete the task?	50
	Did you read the slideshow?	51
	Do you have any prior experience using ROS 1?	52
Open Ended	Do you have any suggestions to improve the tutorial or problems you ran into while doing it? (tutorial 1)	53
	Do you have any suggestions to improve the tutorial or problems you ran into while doing it? (tutorial 2)	54
	Do you have any suggestions to improve the slideshow?	55
	What are the main differences you encountered between using ROS 1 and ROS 2? Do you think any of them is better or easier to use?	56

Table B.4: Form answers to multiple choice and yes/no questions

Subject	Question							
	1	46	47	48	49	50	51	52
1	Master's degree	Yes	Yes	Yes	Yes	Yes	Yes	Yes
2	Bachelor's degree	Yes	Yes	Yes	Yes	Yes	Yes	No
3	Master's degree	Yes	Yes	Yes	Yes	Yes	No	Yes
4	Bachelor's degree	Yes	Yes	No	Yes	Yes	No	No
5	Master's degree	Yes	Yes	Yes	Yes	Yes	Yes	No
6	Master's degree	Yes	Yes	Yes	Yes	Yes	Yes	No
7	Bachelor's degree	Yes	Yes	Yes	Yes	Yes	Yes	Yes
8	Master's degree	Yes	Yes	Yes	Yes	No	No	Yes
9	Master's degree	No	Yes	Yes	Yes	Yes	Yes	No
10	Grade 12	Yes	Yes	Yes	Yes	Yes	Yes	Yes
11	Master's degree	Yes	Yes	Yes	Yes	Yes	No	No
12	Master's degree	Yes	Yes	Yes	Yes	Yes	Yes	Yes

Table B.5: Form answers to 1 - 7 evaluation questions (1/2)

Q	**Subject**											
	1	2	3	4	5	6	7	8	9	10	11	12
2	5	3	3	1	4	5	3	4	1	4	1	4
3	6	3	2	1	1	2	3	4	1	4	1	4
4	5	3	2	1	1	2	5	2	1	4	1	4
5	6	3	2	1	1	5	3	4	1	3	1	4
6	7	3	2	1	2	1	1	3	1	4	1	1
7	6	2	2	1	1	3	2	3	1	4	2	1
8	1	3	5	2	4	5	1	5	2	3	4	6
9	1	4	5	1	3	4	1	1	2	3	1	6
10	4	4	5	3	3	4	1	5	3	4	5	6
11	1	4	5	2	3	5	2	4	2	4	3	6
12	1	3	5	2	4	4	4	6	2	5	4	6
13	2	2	5	1	2	3	3	5	3	4	4	6
14	-	-	-	-	-	-	-	-	2	-	-	-
15	-	-	-	-	-	-	-	-	2	-	-	-
16	-	-	-	-	-	-	-	-	2	-	-	-
17	-	-	-	-	-	-	-	-	2	-	-	-
18	6	3	5	7	7	6	4	7	4	6	5	7
19	5	4	4	6	6	6	4	6	4	6	6	7
20	5	4	5	6	4	5	4	6	4	4	6	7
21	5	3	5	5	4	5	4	6	3	5	4	7
22	3	2	6	6	1	3	2	5	2	4	2	2

Table B.6: Form answers to 1 - 7 evaluation questions (2/2)

Q	**Subject**											
	1	2	3	4	5	6	7	8	9	10	11	12
23	2	2	5	6	2	3	2	5	2	4	2	2
24	2	2	6	2	1	4	2	5	2	4	5	2
25	4	2	6	3	4	4	3	4	2	2	4	2
26	6	2	5	5	1	4	3	4	2	1	5	2
27	5	2	5	5	3	5	3	5	1	3	5	2
28	4	4	6	1	6	5	2	6	6	6	6	6
29	3	4	5	1	6	5	3	6	7	7	7	6
30	3	4	5	6	7	5	4	6	6	7	3	6
31	2	4	5	5	7	5	5	6	6	7	4	6
32	2	4	5	5	5	5	5	6	4	6	4	6
33	2	4	5	5	4	5	5	6	6	6	5	6
34	5	3	6	6	7	6	6	7	6	6	5	6
35	5	3	5	5	7	5	5	6	5	6	5	6
36	5	3	6	5	5	5	5	6	5	5	3	6
37	3	3	6	5	1	4	4	4	4	3	4	4
38	5	3	5	5	1	5	4	4	4	3	2	4
39	5	2	5	7	1	4	5	4	4	1	3	4
40	4	2	5	6	1	4	5	4	4	1	2	4
41	4	4	6	3	6	5	5	6	4	6	4	6
42	4	4	6	3	6	5	5	6	4	6	4	6
43	4	3	5	1	6	3	6	6	4	7	5	6
44	5	3	4	4	5	3	6	6	4	7	6	6
45	6	4	-	-	3	3	5	-	5	3	-	4

Table B.7: Form answers to open ended questions

Subject	**Q**	**Question**
1	53	Consider reading up on python package structure and the usage of the __init__.py file.
	55	Simplify it somehow.
	56	The main differences relate to QoS features in messages, launch file structure, node initialization and the decentralized architecture of ROS2. ROS2 is slightly easier and more intuitive from a beginner perspective, however it's more difficult to transition from ROS1 to ROS2. Additionally, this question is in no way relevant to your work, as it only deals with ROS2, providing no solid comparison to ROS1.
8	53	I'm lazy. I'd like to have something more interactable to learn like youtube videos or games.
10	56	Ros 2 is supported on windows and macOs that fact for me helps a lot in my workspace .
12	56	It was the first and only time I used ROS2, seemed similar to ROS1

Table B.8: Quiz questions and correct answers

Topic	**Question**		**Right answer**	
ROS 2	Q1	If you make modifications in one of your packages, inside which folder do you need to run the command to build the changes?	1	Workspace folder
	Q2	Imagine you are developing a controller for a robot and need to know its position but only when a collision with a wall is detected. The most efficient way for the robot to share this information is to use a:	1/4	ROS Service/ROS Topic
	Q3	What should be used to communicate with a Node that only has the task of controlling the speed of a motor in a robot?	1	ROS Topic
	Q4	Which statement is false about ROS Nodes:	2	Nodes can only communicate with other Nodes that were launched by the same launch file.
Flatland	Q1	The image that is used to represent the walls of the map needs to be sourced in which file?	1	Layer file
	Q2	In the following layer setup: Robot 1 -> Layer 1, Layer 3 Robot 2 -> Layer 2, Layer 3 Robot 3 -> Layer 2 Which robots collide with each other?	1	Robot 1 collides with 2, Robot 2 collides with 3.
	Q3	In the following layer setup: Robot 1 -> Layer 1, Layer 3 Robot 2 -> Layer 2, Layer 3 Robot 3 -> Layer 2 Laser -> Layer 2, Layer 3 The Laser detects:	1	All the Robots.
	Q4	Which of these Flatland plugins subscribes to a topic?	1	Diff Drive
RL	Q1	The possible actions for any given state are defined by the:	1	Environment
	Q2	When a final state is reached, this means that:	1	The environment needs to be reset to an initial state.
	Q3	Considering the concepts of step and episode in reinforcement learning, which of these statements is false?	1	Each episode needs to have the same number of steps.
	Q4	Each algorithm defines its own policy to decide which action to take based on the state. The entity that makes this decision is also known as the:	1	Agent

Table B.9: Quiz answers

Topic	**Question**	**Subject**											
		1	2	3	4	5	6	7	8	9	10	11	12
ROS 2	Q1	1	-	1	1	4	1	1	1	-	1	1	1
	Q2	1	3	-	4	3	4	4	4	-	4	1	1
	Q3	1	1	3	4	2	4	1	4	-	3	3	1
	Q4	2	2	4	1	-	2	2	3	-	2	-	3
Flatland	Q1	3	2	4	3	2	3	3	3	-	2	1	3
	Q2	1	3	-	1	2	1	1	1	2	1	1	1
	Q3	1	1	4	1	1	1	1	1	-	1	1	1
	Q4	1	3	2	-	-	3	1	1	-	1	1	3
RL	Q1	1	4	3	2	1	2	2	2	2	1	1	1
	Q2	-	1	3	4	4	4	1	4	-	1	4	1
	Q3	4	1	-	1	1	1	1	1	-	1	1	1
	Q4	1	3	1	1	1	1	1	1	-	1	2	1

References

[1] Michel Albonico, Milica Đorđević, Engel Hamer, and Ivano Malavolta. Software engineering research on the robot operating system: A systematic mapping study. *Journal of Systems and Software*, 197, 3 2023.

[2] Jafar Alzubi, Anand Nayyar, and Akshi Kumar. Machine learning from theory to algorithms: An overview. In *Journal of Physics: Conference Series*, volume 1142. Institute of Physics Publishing, 11 2018.

[3] Muhammad Ateeq, Hina Habib, Adnan Umer, and Muzammil Ul Rehman. C++ or python? which one to begin with: A learner's perspective. In *2014 International Conference on Teaching and Learning in Computing and Engineering*, pages 64–69. IEEE, 2014.

[4] Fahiem Bacchus and Froduald Kabanza. Planning for temporally extended goals. *Annals of Mathematics and Artificial Intelligence*, 22:5–27, 1998.

[5] Rodrigo Antonio Marques Braga, Marcelo Petry, António Paulo Moreira, and Luis Paulo Reis. Intellwheels: A development platform for intelligent wheelchairs for disabled people. In *International Conference on Informatics in Control, Automation and Robotics (ICINCO)*, volume 2, 2008.

[6] Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. Openai gym. *arXiv preprint arXiv:1606.01540*, 2016.

[7] Andy Cahill. *Catastrophic forgetting in reinforcement-learning environments*. PhD thesis, University of Otago, 2011.

[8] Pengzhan Chen, Jiean Pei, Weiqing Lu, and Mingzhen Li. A deep reinforcement learning based method for real-time path planning and dynamic obstacle avoidance. *Neurocomputing*, 497:64–75, 8 2022.

[9] Maxime Chevalier-Boisvert, Florian Golemo, Yanjun Cao, Bhairav Mehta, and Liam Paull. Duckietown environments for openai gym. *GitHub repository*, 2018.

[10] Heesun Choi, Cindy Crump, Christian Duriez, Asher Elmquist, Gregory Hager, David Han, Frank Hearl, Jessica Hodgins, Abhinandan Jain, Frederick Leve, Chen Li, Franziska Meier, Dan Negrut, Ludovic Righetti, Alberto Rodriguez, Jie Tan, Jeff Trinkle, J Tan, and J Trinkle. On the use of simulation in robotics: Opportunities, challenges, and suggestions for moving forward. *Proceedings of the National Academy of Sciences*, 118, 2020.

[11] Vineesh Cutting and Nehemiah Stephen. A review on using python as a preferred programming language for beginners. *International Research Journal of Engineering and Technology*, 2021.

[12] Vincenzo DiLuoffo, William R. Michalson, and Berk Sunar. Robot operating system 2: The need for a holistic security approach to robotic architectures. *International Journal of Advanced Robotic Systems*, 15, 5 2018.

[13] Elson Almeida Dreveck, Alex V. Salgado, Esteban W.Gonzales Clua, and Luiz Marcos Garcia Goncalves. Easy learning of reinforcement learning with a gamified tool. In *2021 Latin American Robotics Symposium, 2021 Brazilian Symposium on Robotics, and 2021 Workshop on Robotics in Education, LARS-SBR-WRE 2021*, pages 360–365. Institute of Electrical and Electronics Engineers Inc., 2021.

[14] Giovanni Falzone, Gianluca Giuffrida, Silvia Panicacci, Massimiliano Donati, and Luca Fanucci. Simulation framework to train intelligent agents towards an assisted driving power wheelchair for people with disability. In *International Conference on Agents and Artificial Intelligence (ICAART)*, volume 1, pages 189–196, 2021.

[15] Brígida Mónica Faria, Luís Paulo Reis, and Nuno Lau. A survey on intelligent wheelchair prototypes and simulators. *Advances in Intelligent Systems and Computing*, 1:545–557, 2014.

[16] Atanas Garbev and Atanas Atanassov. Comparative analysis of robodk and robot operating system for solving diagnostics tasks in off-line programming. In *2020 International Conference Automatics and Informatics, ICAI 2020 - Proceedings*. Institute of Electrical and Electronics Engineers Inc., 10 2020.

[17] Wojciech Giernacki, Piotr Kozierski, Jacek Michalski, Marek Retinger, Rafal Madonski, and Pascual Campoy. Bebop 2 quadrotor as a platform for research and education in robotics and control engineering. In *2020 International Conference on Unmanned Aircraft Systems (ICUAS)*, pages 1733–1741. IEEE, 2020.

[18] Andresa Shirley Alves Gomes, Joice Felix Da Silva, and Leonardo Rodrigues De Lima Teixeira. Educational robotics in times of pandemic: Challenges and possibilities. In *2020 Latin American Robotics Symposium, 2020 Brazilian Symposium on Robotics and 2020 Workshop on Robotics in Education, LARS-SBR-WRE 2020*. Institute of Electrical and Electronics Engineers Inc., 11 2020.

[19] Travis Goodspeed, Richard Wunderlich, and Itamar Elhanany. Work in progress-enhancing reinforcement learning class curriculum using a matlab interface library for use with the sony aibo robot. In *2007 37th Annual Frontiers In Education Conference-Global Engineering: Knowledge Without Borders, Opportunities Without Passports*, pages F3J–13. IEEE, 2007.

[20] Joshua Green, Joshua Clounie, Rafael Galarza, Shawn Anderson, Jash Campell-Smith, and Razvan Cristian Voicu. Optimization of an intelligent wheelchair: Lidar and camera vision for obstacle avoidance. In *International Conference on Control, Automation and Systems (ICCAS)*, pages 313–318. IEEE Computer Society, 2022.

[21] Valentin Haak, Joerg Abke, and Kai Borgeest. Work-in-progress: development of a lego mindstorms ev3 simulation for programming in c. In *The Challenges of the Digital Transformation in Education: Proceedings of the 21st International Conference on Interactive Collaborative Learning (ICL2018)-Volume 2*, pages 667–674. Springer, 2019.

[22] Georgios Karalekas, Stavros Vologiannidis, and John Kalomiros. Europa: A case study for teaching sensors, data acquisition and robotics via a ros-based educational robot. *Sensors (Switzerland)*, 20, 5 2020.

[23] Frank Klassner and Scott D Anderson. Lego mindstorms: Not just for k-12 anymore. *IEEE robotics & automation magazine*, 10:12–18, 2003.

[24] Jian Kong and Peng Li. Path planning of a multifunctional elderly intelligent wheelchair based on the sensor and fuzzy bayesian network algorithm. *Journal of Sensors*, 2022, 2022.

[25] Marian Körber, Johann Lange, Stephan Rediske, Simon Steinmann, and Roland Glück. Comparing popular simulation environments in the scope of robotics and reinforcement learning. *arXiv preprint arXiv:2103.04616*, 3 2021.

[26] Jean François Lalonde, Christopher P. Bartley, and Illah Nourbakhsh. Mobile robot programming in education. In *Proceedings - IEEE International Conference on Robotics and Automation*, volume 2006, pages 345–350. IEEE, 2006.

[27] Gonçalo Leão, Filipe Almeida, Emanuel Trigo, Henrique Ferreira, Armando Sousa, and Luís Paulo Reis. Using deep reinforcement learning for navigation in simulated hallways. In *2023 IEEE International Conference on Autonomous Robot Systems and Competitions (ICARSC)*, pages 207–213. IEEE, 4 2023.

[28] Yuxi Li. Deep reinforcement learning: An overview. *arXiv preprint arXiv:1701.07274*, 1 2017.

[29] Timothy P. Lillicrap, Jonathan J. Hunt, Alexander Pritzel, Nicolas Heess, Tom Erez, Yuval Tassa, David Silver, and Daan Wierstra. Continuous control with deep reinforcement learning. *arXiv preprint arXiv:1509.02971*, 9 2015.

[30] Yamato Maekawa, Naoki Akai, Takatsugu Hirayama, Luis Yoichi Morales, Daisuke Deguchi, Yasutomo Kawanishi, Ichiro Ide, and Hiroshi Murase. Modeling eye-gaze behavior of electric wheelchair drivers via inverse reinforcement learning. In *IEEE International Conference on Intelligent Transportation Systems (ITSC)*. IEEE, 9 2020.

[31] Yuya Maruyama, Shinpei Kato, and Takuya Azumi. Exploring the performance of ros2. In *Proceedings of the 13th International Conference on Embedded Software, EMSOFT 2016*. Association for Computing Machinery, Inc, 10 2016.

[32] Benjamim Medeiros, Ricardo Mousinho, José Cascalho, and Matthias Funk. Alphabot2 revisited: An educational tool to program and learn robotics. In *ROBOT2022: Fifth Iberian Robotics Conference: Advances in Robotics, Volume 1*, pages 562–574. Springer, 2022.

[33] Olivier Michel. Cyberbotics ltd. webots tm : Professional mobile robot simulation. *International Journal of Advanced Robotic Systems*, 1:5, 2004.

[34] Javier Minguez, Florent Lamiraux, and Jean-Paul Laumond. Motion planning and obstacle avoidance. *Springer Handbook of Robotics*, pages 1177–1202, 2016.

[35] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A. Rusu, Joel Veness, Marc G. Bellemare, Alex Graves, Martin Riedmiller, Andreas K. Fidjeland, Georg Ostrovski, Stig Petersen, Charles Beattie, Amir Sadik, Ioannis Antonoglou, Helen King, Dharshan Kumaran, Daan Wierstra, Shane Legg, and Demis Hassabis. Human-level control through deep reinforcement learning. *Nature*, 518:529–533, 2 2015.

[36] Yoichi Morales, Nagasrikanth Kallakuri, Kazuhiro Shinozawa, Takahiro Miyashita, and Norihiro Hagita. Human-comfortable navigation for an autonomous robotic wheelchair. In

IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS), pages 2737–2743. IEEE, 2013.

[37] Alves Márcia, Armando Sousa, and Ângela Cardoso. Web based robotic simulator for tactode tangible block programming system. In *Robot 2019: Fourth Iberian Robotics Conference: Advances in Robotics, Volume 1*, volume Volume 1, pages 490–501. Springer, 2020.

[38] Wyatt S. Newman. *A Systematic Approach to Learning Robot Programming with ROS*. Chapman and Hall/CRC, 1st edition, 2017.

[39] Jason M. O'Kane. *A gentle introduction to ROS*. Jason M. O'Kane, 2014.

[40] Lifan Pan, Anyi Li, Jun Ma, and Jianmin Ji. Learning navigation policies for mobile robots in deep reinforcement learning with random network distillation. In *International Conference on Innovation in Artificial Intelligence (ICIAI)*, volume PartF17154, pages 151–157. ACM, 2021.

[41] Vassilis A Papavassiliou and Stuart Russell. Convergence of reinforcement learning with general function approximators. In *IJCAI*, pages 748–755, 1999.

[42] Sarangi P. Parikh, Valdir Grassi, Vijay Kumar, and Jun Okamoto. Integrating human inputs with autonomous behaviors on an intelligent wheelchair platform. *IEEE Intelligent Systems*, 22:33–41, 2007.

[43] Sarvesh Patil, Samuel C. Alvares, Pragna Mannam, Oliver Kroemer, and F. Zeynep Temel. Deltaz: An accessible compliant delta robot manipulator for research and education. In *2022 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 13213–13219. IEEE, 10 2022.

[44] B. K. Patle, Ganesh Babu L, Anish Pandey, D. R.K. Parhi, and A. Jagadeesh. A review: On path planning strategies for navigation of mobile robot. *Defence Technology*, 15:582–606, 8 2019.

[45] Liam Paull, Jacopo Tani, Heejin Ahn, Javier Alonso-Mora, Luca Carlone, Michal Cap, Yu Fan Chen, Changhyun Choi, Jeff Dusek, Yajun Fang, Daniel Hoehener, Shih-Yuan Liu, Michael Novitzky, Igor Franzoni Okuyama, Jason Pazis, Guy Rosman, Valerio Varricchio, Hsueh-Cheng Wang, Dmitry Yershov, Hang Zhao, Michael Benjamin, Christopher Carr, Maria Zuber, Sertac Karaman, Emilio Frazzoli, Domitilla Del Vecchio, Daniela Rus, Jonathan How, John Leonard, and Andrea Censi. Duckietown: an open, inexpensive and flexible platform for autonomy education and research. In *2017 IEEE International Conference on Robotics and Automation (ICRA)*, pages 1497–1504. IEEE, 2017.

[46] Morgan Quigley, Brian Gerkey, Ken Conley, Josh Faust, Tully Foote, Jeremy Leibs, Eric Berger, Rob Wheeler, and Andrew Ng. Ros: an open-source robot operating system. In *ICRA workshop on open source software*, page 5, 2009.

[47] Andrew G. Sutton Richard S., Barto. *Reinforcement Learning: An Introduction*. MIT press, second edi edition, 2018.

[48] Nelson Rodrigues, Armando Sousa, Luís Paulo Reis, and António Coelho. Intelligent wheelchairs rolling in pairs using reinforcement learning. In *Iberian Robotics Conference (ROBOT)*, page 274–285. Springer, 2022.

[49] G A Rummery and M Niranjan. *On-line Q-learning using connectionist systems*, volume 37. University of Cambridge, Department of Engineering Cambridge, UK, 1994.

[50] Halil İbrahim Şahin and Ahmet Reşit Kavsaoğlu. Autonomously controlled intelligent wheelchair system for indoor areas. In *2021 3rd International Congress on Human-Computer Interaction, Optimization and Robotic Applications (HORA)*, pages 1–6. IEEE, 2021.

[51] John Schulman, Sergey Levine, Philipp Moritz, Michael I. Jordan, and Pieter Abbeel. Trust region policy optimization. In *International conference on machine learning*, pages 1889–1897. PMLR, 2 2015.

[52] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*, 7 2017.

[53] Cheng Shen and Qiuping Bi. Path optimization of intelligent wheelchair based on an improved ant colony algorithm. In *IEEE Conference on Industrial Electronics and Applications (ICIEA)*, pages 1862–1867. IEEE, 8 2021.

[54] Sivashankar Sivakanthan, Jeremy Castagno, Jorge L. Candiotti, Jie Zhou, Satish A. Sundaram, Ella M. Atkins, and Rory A. Cooper. Automated curb recognition and negotiation for robotic wheelchairs. *Sensors*, 21:7810, 11 2021.

[55] K R Srinath. Python-the fastest growing programming language. *International Research Journal of Engineering and Technology*, 2017.

[56] George Stavrinos. Ros2 for ros1 users. *Robot Operating System (ROS) The Complete Reference (Volume 5)*, pages 31–42, 2021.

[57] Ryota Suenaga and Kazuyuki Morioka. Development of a web-based education system for deep reinforcementlearning-based autonomous mobile robot navigation in real world. In *2020 IEEE/SICE International Symposium on System Integration (SII)*, pages 1040–1045. IEEE, 2020.

[58] Lei Tai, Giuseppe Paolo, and Ming Liu. Virtual-to-real deep reinforcement learning: Continuous control of mobile robots for mapless navigation. In *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, volume 2017-Septe, pages 31–36. IEEE, 2017.

[59] Emanuel Todorov, Tom Erez, and Yuval Tassa. Mujoco: A physics engine for model-based control. In *2012 IEEE/RSJ international conference on intelligent robots and systems*, pages 5026–5033. IEEE, 2012.

[60] Sokratis Tselegkaridis and Theodosios Sapounidis. Simulators in educational robotics: A review. *Education Sciences*, 11:11, 1 2021.

[61] Julio Vega and José M. Cañas. Pybokids: An innovative python-based educational framework using real and simulated arduino robots. *Electronics (Switzerland)*, 8, 8 2019.

[62] Vítor Ventuzelos, Gonçalo Leão, and Armando Sousa. Teaching ros1/2 and reinforcement learning using a mobile robot and its simulation. In *Iberian Robotics Conference (ROBOT)*, pages 586–598. Springer, 2022.

[63] Wei Wang, Zhenkui Wu, Huafu Luo, and Bin Zhang. Path planning method of mobile robot using improved deep reinforcement learning. *Journal of Electrical and Computer Engineering*, 2022, 2022.

[64] Charles W Warren. Global path planning using artificial potential fields. In *IEEE International Conference on Robotics and Automation (ICRA)*, pages 316–317. IEEE, 1989.

[65] Charles W Warren. Multiple robot path coordination using artificial potential fields. In *IEEE International Conference on Robotics and Automation (ICRA)*, pages 500–505. IEEE, 1990.

[66] Christopher J C H Watkins and Peter Dayan. Q-learning. *Machine learning*, 8:279–292, 1992.

[67] Lele Xi and Motoki Shino. Shared control design methodologies of an electric wheelchair for individuals with severe disabilities using reinforcement learning. *Journal of Advanced Simulation in Science and Engineering*, 7:300–319, 2020.

[68] Jing Xin, Huan Zhao, Ding Liu, and Minqi Li. Application of deep reinforcement learning in mobile robot path planning. In *Chinese Automation Congress (CAC)*, volume 2017-Janua, pages 7112–7116. IEEE, 2017.

[69] Jiachen Yang, Jingfei Ni, Yang Li, Jiabao Wen, and Desheng Chen. The intelligent path planning system of agricultural robot via reinforcement learning. *Sensors*, 22:4316, 6 2022.

[70] Changlong Ye, Deli Hu, Shugen Ma, and Huaiyong Li. Motion planning of a snake-like robot based on artificial potential method. In *IEEE International Conference on Robotics and Biomimetics (ROBIO)*, pages 1496–1501. IEEE, 12 2010.

[71] Pengyu Yue, Jing Xin, Huan Zhao, Ding Liu, Mao Shan, and Jian Zhang. Experimental research on deep reinforcement learning in autonomous navigation of mobile robot. In *IEEE Conference on Industrial Electronics and Applications (ICIEA)*, pages 1612–1616. IEEE, 2019.

[72] Chiyuan Zhang, Oriol Vinyals, Remi Munos, and Samy Bengio. A study on overfitting in deep reinforcement learning. *arXiv preprint arXiv:1804.06893*, 2018.

[73] Ziyi Zhang, Sara Willner-Giwerc, Jivko Sinapov, Jennifer Cross, and Chris Rogers. An interactive robot platform for introducing reinforcement learning to k-12 students. In *Robotics in Education: RiE 2021 12*, pages 288–301. Springer, 2022.

[74] Wenshuai Zhao, Jorge Pena Queralta, and Tomi Westerlund. Sim-to-real transfer in deep reinforcement learning for robotics: A survey. In *2020 IEEE Symposium Series on Computational Intelligence, SSCI 2020*, pages 737–744. Institute of Electrical and Electronics Engineers Inc., 12 2020.

[75] Jianfeng Zheng, Shuren Mao, Zhenyu Wu, Pengcheng Kong, and Hao Qiang. Improved path planning for indoor patrol robot based on deep reinforcement learning. *Symmetry*, 14:132, 1 2022.

[76] Ángel Martínez-Tenor, Ana Cruz-Martín, and Juan Antonio Fernández-Madrigal. Teaching machine learning in robotics interactively: the case of reinforcement learning with lego ® mindstorms. *Interactive Learning Environments*, 27:293–306, 4 2019.