Rheinische
Friedrich-Wilhelms-
Universität Bonn

Institute for Computer Science
Department VI
Autonomous Intelligent Systems

Rheinische
Friedrich-Wilhelms-Universität Bonn

Master Thesis

Indoor Robot Navigation using Lane Division and Future Prediction

Author:
Oleg Kosenko

First Examiner:
Prof. Dr. Sven Behnke

Second Examiner:
Prof. Dr. Maren Bennewitz

Advisor:
Marius Beul

Date: April 26, 2021

Declaration

I hereby declare that I am the sole author of this thesis and that none other than the specified sources and aids have been used. Passages and figures quoted from other works have been marked with appropriate mention of the source.

Place, Date

Signature

Abstract

The motivation of this thesis is to create a system that navigates multiple robots around in a known indoor environment. An application example of such a system would be medicine delivery in a hospital, document delivery, or visitor accompaniment in an office complex.

The problem of indoor robot navigation includes such challenges as localization, path planning, and collision avoidance. In our case, robots have to move in the same environment with humans and other robots. This complicates localization: a human can appear between the robot and a tracking base station; and collision avoidance: the robot must predict human behavior to drive around humans.

We developed the navigation software that implements concepts of lane division and future prediction. The lane division module divides each corridor into lanes. This allows making collision avoidance easier. Future prediction relies on a probabilistic human motion model. Given that the robot knows how humans usually move, it predicts which lanes may be blocked in the future and chooses the safest lane to follow.

We implemented a hardware prototype using a DJI Robomaster S1 robot and SteamVR tracking and evaluated our system in simulation and with real-world experiments.

Contents

1 Introduction

Robot navigation is a challenging field of robotic systems development and it requires new and more efficient solutions while the amount of applications of the mobile robots is increasing constantly. In warehouses, hospitals, industrial and office premises application of mobile robots has become more and more advantageous with the continuous improvement of robotic systems.

Recent works in this field are devoted to the development and improvement of algorithms that are able to predict accurately the consequences of robot movements and actions, including interaction with people and other robots.

The realization of the motion planning algorithms requires significant computational resources for assuring fast and safe path planning. Efficient path planning algorithms should be realized using minimal computational costs. At the same time, the strongest priority of such systems is the safety of all actors: people, the robot itself, other robots, other objects and property.

Road lane division techniques are rapidly developing in the last years with the introduction of self-driving cars. These techniques are widely used in autonomous driving for navigation and path planning as far as they are naturally inherited from roads with human-driven cars. Lane division dramatically simplifies the navigation and path planning problems in self-driving. While similar navigation tasks for robots are arising in indoor environments, the velocity rates and scene complexity are generally lower. The navigation techniques used for self-driving cars can be also successfully applied to indoor-navigating robots.

In this thesis, we developed a robot navigation system with the ability to predict the behavior of other actors and plan a safe and efficient trajectory. As far as the system has a reasonable human motion model for human motion prediction and is itself predictable, it can be used for the creation of a people-friendly multiple-robot environment.

Solutions for localization, path planning, and collision avoidance problems are introduced in this thesis. The novelty of this work is the adaption of the lane-division and future prediction techniques to the indoor environment. The offered system is called up to assure safe and efficient robot path planning.

In Chapter 2 we first introduce related works regarding the techniques used in this thesis.

In Chapter 3 we describe the proposed method. The first section is related to the hardware prototype and the control pipeline. The second one to the navigation system and the simulator.

Chapter 4 is dedicated to the description of system tests and their results. The tests include software tests of the simulator and hardware tests of the prototype.

We conclude the thesis in Chapter 5 where we summarize the evaluation results and point out the drawbacks and possible improvements.

2 Related Work

In this section, we review the related works from several research domains, i.e. robot navigation, path planning, collision avoidance, future prediction, self-driving, lane-changing, and robot operating system.

2.1 Path Planning Algorithms

Traditional path planning algorithms have been developing since 1959 when E.W. Dijkstra published his paper (Dijkstra 1959). The now called Dijkstra Algorithm finds the shortest path between two given nodes in the graph.

The A* search algorithm is an extension of Dijkstra's algorithm. A* reduces the total number of explored states using a heuristic estimate of the cost to get to the target point from a given starting point. Hart et al. (1968) draw together the mathematical approach and the heuristic approach of finding a path through a graph.

Artificial potential field concept (Khatib 1985) represents the moving robot as an object influenced by a potential field, which attracts the robot to its target while repelling it from the obstacles. The method is applicable to the moving obstacles. For processing the moving obstacles the authors use a time-varying artificial potential field.

Probabilistic Road Maps (PRM) (Kavraki et al. 1996) is a planning method for robots in static workspaces which effectively explores large and complex spaces.

PRM method is implemented in two steps: a learning step and a query step. In the learning step, a probabilistic road map is constructed and stored as a graph. The nodes of the graph correspond to collision-free configurations and the edges correspond to paths between these configurations. In the query step, any given start and target configurations of the robot are represented by two nodes of the roadmap. The roadmap is then searched for a path that leads from one node to the other.

The concept of Rapidly-exploring Random Trees (RRT) was introduced by LaValle (1998). RRT algorithm randomly builds a tree, which branches randomly towards unsearched areas. The path is extracted from that tree. RRT is implemented as a path planning module, which can be incorporated into a wide variety of planning systems.

Karaman and Frazzoli (2010) further developed the RRT algorithm and introduced RRT*. The RRT* algorithm rebuilds the tree structure after inserting new nodes as it looks for new lower-cost paths between the nodes that are already in the tree.

LaValle (2006) provides a survey of the motion planning algorithms. The author gives a definition of the basic motion planning problem as to find a path from a start point to a target point in an obstacle-free space and describes fundamentals of the robot and obstacle modeling in 2D and 3D worlds.

Among the other LaValle (2006) reviews following general search schemes: best first, iterative deepening approach, backward search, bidirectional search.

Grid-based methods are widely used for cost-optimal robot path planning. Usage of these methods becomes intractable when the resolution of the grid is high. A local multiresolution path planning algorithm (Behnke 2004) provides significant savings in computational costs combined with high accuracy of planning of the initial parts of the paths.

2.2 Lane Division and Lane Changing

As far as one of the main ideas of this thesis is using lane division and lane changing for indoor robot navigation, we have studied existing papers in the related field of self-driving vehicles, where lane changing is one of the core techniques. Self-driving vehicles are an important application of robotics and one of the main industry drivers. Self-driving technology requires automation of driving tasks such as safe lane following, obstacle avoidance, overtaking slower traffic, following the vehicle ahead, assessing and avoiding dangerous situations, and determining the route. According to Chakraborty (2021), the most noteworthy are self-driving technologies created by Tesla, Waymo, Pony.ai, Volvo, and Voyage.

The problems that arise in a self-driving domain are similar to indoor navigation problems, but they appear much stronger, taking into consideration the complexity of the roads, the variety of the involved actors and their behaviors, high speeds and masses of the vehicles, and emerging risks. At the same time, a lot of solutions in the field are already designed, tested, approved, and used. "An intelligent vehicle able to assess the driving scenario and react in case of danger would allow up to 90% of traffic accidents that are caused by human errors to be eliminated" (Broggi et al. 2008).

The authors describe the following technologies which are indispensable for implementation in self-driving vehicles: recognition of the other vehicles, work with digital maps and satellite navigation, communication with intelligent transportation infrastructure, road scene understanding, collision avoidance, lane keeping, and parking assistance. These technologies can be realized by combining two concepts: gathering the information by the cameras/sensors built into the vehicle, and exchanging the information coming from infrastructure and other vehicles.

The abovementioned concepts are useful for our work as far as they are well transferred to the indoor environment, where information from cameras and sensors of the robot can be combined with information obtained from the external sources, e.g. SteamVR base-stations.

As far as the self-driving industry is rapidly developing, its solutions can be adopted in the related domains, including indoor robot navigation or seamless indoor to outdoor robot navigation. (*Robotics 2020. Multi-Annual Roadmap for Robotics in Europe. Horizon 2020* 2015) among the wide range of reviewed technologies pays attention to both indoor and outdoor navigation domains. Among other topics, this work covers the operation of robots in close proximity to humans.

During the last 10 years, there were published a lot of papers in the field, addressing specific problems and offering solutions. E.g. Brechtel et al. (2011) present a method for high-level decision making in traffic environments. The method employs Markov Decision Process (MDP) to plan the optimal decision-making policy by assessing the outcomes of actions. Decisions are deduced from the knowledge about the behavior of the road users.

Ulbrich and Maurer (2015) present an approach for tactical behavior planning for lane changing in a planning horizon in between 100 ms and 30 s (see Fig. 2.1).

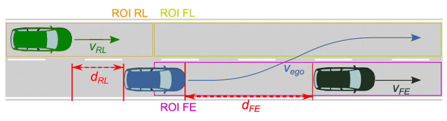


Figure 2.1: Typical scenario for lane change decision making with two dynamic objects and three regions of interest rear left (RL), front left (FL) and front ego (FE) (Ulbrich and Maurer 2015)

Xu et al. (2012) propose a dynamic cooperative lane-change maneuver, aiming to maintain safety both on the early stage of lane-change maneuver and during the lane-change process. We have considered such complex maneuvering, but in the case of indoor robot navigation, this is an overcomplication. Nevertheless, it is a possible extension, that can be used in some indoor environments.

Scheel et al. (2018) studied understanding of complex and dynamic scenes by self-driving vehicles for the planning of maneuvers, especially for the lane changing. Scheel et al. (2018) proposed a situation assessment algorithm for classifying driving situations with respect to their suitability for lane changing using deep learning architecture based on a Bidirectional Recurrent Neural Network.

Killing et al. (2021) studied situations in which there are no well-defined traffic rules available and introduce a “high-conflict driving scenario requiring negotiations between agents of equal rights and priorities”. The proposed scenario is modeled as a decentralized partially observable Markov Decision Process (dec-POMDP) (F.A. Oliehoek 2015). In our work, we encountered a similar problem, which led to the agent oscillation (see Sec. 4.2.2).

2.3 Future Prediction and Collision Avoidance

In this chapter, we review papers concerning motion trajectory prediction, collision avoidance, and social behavior aspects of robots and humans moving in a common environment. As far as in the considered indoor environments moving robots interact with humans, we reviewed the papers in the field of human movement patterns and robot-human social behavior.

Bennewitz et al. (2005) propose a technique for studying the typical motion patterns of people. The information about such motion patterns is used for improving mobile robot navigation and path planning. A Hidden Markov Model was applied to estimate the current and future positions of persons based on sensory input.

Ziebart et al. (2009) offered an approach for determining robot movements that efficiently accomplish the robot’s tasks while not hindering the movements of people within the environment. Authors model the sequence of actions that lead to a person’s future position using a deterministic Markov Decision Process over a grid representing the environment.

Kirby (2010) studied human-human interaction and applied the discovered principles to human-robot interaction. It is argued that “robots should behave according to human social principles”.

The author developed mathematical models of human behavior, in such a way as to allow a robot to follow similar conventions when navigating through hallways. The human social conventions for movement are represented in the discussed models as a set of mathematical cost functions: “Robots that navigate according to these cost functions are interpreted by people as being socially correct”. Paper demonstrates how the resulting behaviors follow human social norms and further describes “how the behaviors can be altered to produce different social “personalities”, such as extremely deferential (always moving to the right out of a person’s way) or more aggressive (continuing to face a person while passing)”.

The prediction of the future motion of dynamic actors is well studied in the literature. Lefevre et al. (2014) review classical methods, including Physics-based, Maneuver-based, and Interaction-aware motion models and highlight algorithms, which focus on the most relevant trajectories to speed up the computation.

Paden et al. (2016) survey the current state-of-the-art planning and control algorithms with particular regard to the urban setting and review the effectiveness of the proposed techniques. The paper overviews the decision-making hierarchy of self-driving vehicles, modeling for planning and control, motion planning, and vehicle control.

Chen et al. (2018) studied robot navigation in pedestrian-rich environments and emphasized the importance of modeling human behaviors and navigation rules (e.g., passing on the right). Using deep reinforcement learning the authors developed a time-efficient navigation policy that respects common social norms and presented the socially aware multiagent collision avoidance algorithm with deep reinforcement learning (SACADRL). The authors first described a strategy for shaping normative behaviors for a two-agent system in the RL framework and then generalized the method to multiagent scenarios.

Tang and Salakhutdinov (2019) introduce a probabilistic framework that efficiently learns latent variables to jointly model the multi-step future motions of agents in a scene. Presented Multiple Futures Predictor (MFP) is a probabilistic latent variable model that learns directly from multi-agent trajectory data. The effectiveness of MFP was checked using CARLA simulator (Dosovitskiy et al. 2017) and confirmed by an experiment on a standard dataset of real vehicle trajectories, the NGSIM dataset (Colyar and Halkias 2007).

Rudenko et al. (2020) provide a detailed survey in the field of human motion trajectory prediction. The authors also summarize the state-of-the-art and discuss the inherent strengths and limitations of different classes of approaches.

2.4 ROS

We used the open-source robot operating system (ROS) as a middleware for communication between sub-modules (tracking node, ROS controller, and Mapsim) and also for managing the coordinate frame transforms. Therefore, we are interested in studies connected to the usage of ROS.

Quigley et al. (2009) give an overview of the developed Robot Operating System (ROS) which provides a communication layer above the host operating system. The authors compare ROS with existing robot software frameworks and overview some application software that uses ROS.

Allgeuer et al. (2018) introduce the NimbRo-OP - an open humanoid platform developed by team NimbRo of the University of Bonn. They describe a software framework for the NimbRo-OP that is based on the Robot Operating System (ROS) middleware. The software provides functionality for hardware abstraction, visual perception, and behavior generation, and has been used to implement basic soccer skills. The NimbRo-OP robot was tested and demonstrated in the setting of humanoid soccer (i.e. RoboCup).

Tsardoulias and Mitkas (2017) describe ROS as a framework targeted for writing robot software. ROS includes a large number of tools, libraries, and conventions that aim for complexity reduction in writing complex and robust robotic software.

ROS includes a large number of tools for software development, robot management, and communication. Software development tools include powerful visualization programs as `rviz` and `rqt`. Robot management modules include localization, mapping, navigation, and other algorithms. Communication infrastructure is a low-level message passing interface. Apart from the message-passing functionality, it supports the recording and playback of messages via the `rosbag` tool, remote procedure calls, and a distributed parameter system.

3 Method

This chapter consists of two parts. In the first part, we describe the hardware prototype of the navigation system. The second one is dedicated to the planner and the use of lane division and future prediction.

3.1 Robomaster

In this section, we will describe all the components of our hardware prototype along with their specifications. We will show the data flow and describe, how the components are interconnected.

3.1.1 Setup

The setup of the hardware prototype is composed of a robot, which has to navigate through an environment, obstacles that the robot has to avoid, a tracking system that tracks the positions of both the robot and the target, and a Mapsim controller that will be described in Sec. 3.2 of the thesis.

The hardware components used to realize this prototype are the DJI Robomaster S1, a SteamVR tracking system, a Linux system with ROS, a Windows system with DJI native controller, and a router connecting the subsystems together (see Fig. 3.1).

The command pipeline works as follows:

- The simulator that is running on a Linux machine analyzes the robot's surroundings and decides where the robot should go (see Sec. 3.2.1). The simulator generates the world frame coordinates of a waypoint the robot should navigate to and publishes the coordinates to ROS.
- ROS node `controller` compares the robot and the waypoint coordinates to generate a movement vector (see Sec. 3.1.4).
- By communicating with the Windows machine via websocket, the vector is sent to the DJI Robomaster application, which sends the data to the robot (see Sec. 3.1.5).

(a) DJI Robomaster S1.

(b) Steam VR tracking system.

(c) Windows machine with DJI controller.

(d) Linux machine with ROS and Mapsim.

(e) Smartphone with WiFi Hotspot.

Figure 3.1: Components of our system.

- After receiving the motion command, the robot moves in the direction of the waypoint.
- This movement is tracked by the SteamVR tracking system (see Sec. 3.1.3). Tracking data is collected by the tracker and sent to the Linux machine via a Bluetooth dongle, where the SteamVR application manages the incoming data.
- The data is then accessed and published to ROS by an `openvr_tracking` ROS node.
- The published odometry is then used by the simulator for replanning.

3.1.2 Robot

For the purpose of testing the proposed solution we chose a DJI Robomaster S1 robot. The robot is based on an omnidirectional platform, equipped with a camera and WiFi adapter.

The Robomaster S1 is a tank-like wheeled rover remotely controlled via Wi-Fi from an app on Microsoft Windows, Apple iOS, and Google Android mobile

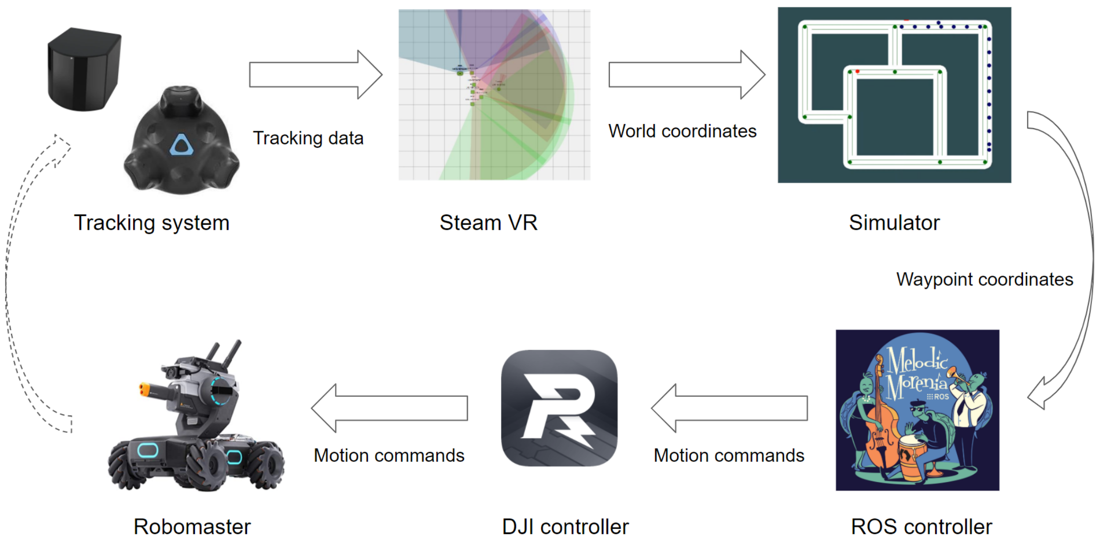


Figure 3.2: System setup with data flow

Table 3.1: Robomaster S1 specifications.

Dimensions (L×W×H)	320×240×270 mm
Weight	Approx. 3.3 kg
Chassis Speed Range	0–3.5 m/s (forward) 0–2.5 m/s (backward) 0–2.8 m/s (sideward)
Max Chassis Rotational Speed	600 °/s
Max Motor Output Power	19 W
Battery Capacity	2400 mAh
Battery Life in Use	35 minutes

devices. It was designed to be an advanced educational robot, so the user has to assemble it from loose parts out of the box.

The robot can only be controlled by a DJI Robomaster application (the problem is described in 3.1.5) that is available on a smartphone or a Windows PC. There are two ways of connecting the robot: direct connection or connection via a router. Since the Windows machine has to also be connected to the Linux machine, a router was used to connect the robot.

3.1.3 Tracking

To successfully apply the method to the prototype we need to know the location of the robot and the obstacle. Tracking is realized by the means of SteamVR. For the prototype testing, we use two HTC Vive trackers 2.0 and four SteamVR Base Stations 2.0 mounted around the testing area.

The HTC Vive Tracker is a small circular device that has 18 IR sensors that monitor its orientation and position along the X, Y, and Z axes. These sensors are located around the top surface of the tracker to maximize tracking accuracy. There are three protruding parts on the top of the tracker that have the sensors at different angles. The prototype setup includes one robot and one obstacle, so we have a tracker mounted on top of each of them.

According to the producer's specification, four base stations cover the area of $10 \times 10\,\mathrm{m}$. The base stations have a field of view of $150° \times 110°$. They are mounted in the corners of the testing area, directed into the center of the room, overlapping at the region of interest, where the robot is tested. Overlapping is required for the tracking to stay consistent if some of the base stations get obstructed.

SteamVR software is running on the Linux system and receives data from both trackers via Bluetooth. Then `openvr_tracking` module takes the tracking data from SteamVR and sends it into ROS as an Odometry message.

The received messages are in a base station frame, which is initialized inside one of the base stations, so we never can reliably say where the frame origin is located and where the base axes are pointed. To solve this we wrote a calibration module, which is run after the start of the SteamVR. We drive the robot around while recording the coordinates. After that, we fit a plane to the data and we can arbitrarily pick the origin point and X axis direction. Finally, we calculate the transform from the base station frame to the new world frame.

After calibration is done we use the calculated transform to transform all the incoming data to the world frame before processing. The odometry is then sent to Mapsim for path calculation and to the ROS controller node for control commands computation.

3.1.4 ROS controller

The controller node is responsible for transforming the waypoint coordinates into motion commands and sending them to the DJI controller. The node listens to tracking and simulator ROS topics. The simulator callback resets the target to new waypoint coordinates and a new target yaw angle. The tracking callback triggers a recalculation of the motion and rotation commands and sends the data to the DJI controller.

First, to calculate the motion command, the angle from the robot to the target Θ is calculated relative to robot's heading direction. V_{robot} is a vector that describes robot's heading direction in the world frame. V_{target} is a vector from the robot to the target in the world frame:

$$\Theta = \operatorname{sign}(V_{robot} \times V_{target}) \frac{\arccos(V_{robot} * V_{target})}{||V_{robot}|| \cdot ||V_{target}||} \tag{3.1}$$

Then we take a unit vector that describes the angle Θ and multiply it with the PD control. The PD control (C) is calculated as follows, where G stands for gain, T is target coordinates, R is robot coordinates, and S is the robot's velocity:

$$C = G_p \cdot ||T - R|| + G_d \cdot \operatorname{abs}(S) \tag{3.2}$$

The robot's velocity is given by odometry from the tracker. The PD controller is used here to slow down the robot in advance so that it does not overshoot its target. Since the vector of length 1 describes the full throttle, it is then capped at 1.

The rotation command (R) is given by the difference between the robot's angle (Θ_{robot}) and the target angle (Θ_{target}) and a control coefficient (C):

$$R = (\Theta_{robot} - \Theta_{target}) \cdot C \tag{3.3}$$

Lastly, the websocket message is constructed. The message consists of 4 values separated by a space: two values for each of two virtual control sticks. The first two values are sagittal and lateral movements, the third is looking up and down, so it is always set to 0, and the fourth is left and right rotation.

The websocket is now used to send the message to the DJI controller.

3.1.5 DJI controller

There is no API for the robot control, it is encrypted and is only available via DJI 'Robomaster' application. The application is available on smartphones and Windows. We used a Windows application and modified it to enable robot control from ROS.

The application is written in the Unity game engine on C#. We used a decompiler 'dnspy' to decompile the controller and add our own code to it. The main .dll that contains the core code is called Assembly-CSharp.dll. We merged it with a WebSocketServer.dll (Mazlov 2019) which realizes a websocket server on C#.

We added a class WebSocket to the {-} namespace, where we implemented the use of the websocket: we receive messages from a client and parse them as control commands, and send confirmations back.

In the code, we located two important places. One being the creation of the Controller class, which is called only once when the program starts. We create an instance of our WebSocket class there. The other is where the Controller gets input from the keyboard and sends it to the robot. We intercept the keyboard data and switch it to the data from the websocket.

The movement function is still called only if there is any keyboard input, but we use it as a failsafe. We hold a button while the robot is moving and if something goes wrong, it is easy and fast to release a button to make the robot stop.

We use a python websocket module in a ROS node to connect to and communicate with the DJI Robomaster app. The server is running on the Windows machine, so we can connect and reconnect as a client without having to restart the DJI app. From this ROS node, we send the velocity information to the DJI controller. The information stays relevant until the next message arrives or until one second passes. If for some reason a connection with ROS is broken, the robot stops in one second.

3.2 Mapsim

In this section, we describe the planning part of our method. We describe the use and benefits of the lane division and future prediction techniques.

3.2.1 Simulator

This software is used by the robot to determine its movement, so the robot will have to intelligently react to its surroundings. So the first thing to do in creating this software is creating a simulator that will represent the map, the robot, and the obstacles.

To implement the software in python we used two libraries: pygame and networkx.

Pygame is a library created for the purpose of game development, but it perfectly suits our needs. It includes a drawing module, which displays the simulated map in a window. The simulator runs as a game with continuous change in its parts and pygame provides the looping for this constant iteration.

Networkx is a module that provides graph functionality. We define nodes, edges, and connections and assign them properties. Nodes have just one property –

coordinates. Edges have 12 properties, describing the connected nodes, width, length, amount of lanes, lane width, and corridor orientation.

Networkx is also providing functionality for finding the shortest routes on the graph, for which the weight of each edge is specified by the 'length' parameter.

Corridor orientation is described by two angles, which are calculated in the equations below. "Start" and "end" are nodes of the edge, here their sequence is important, so directionality of the edge is stored as a parameter.

$$\Theta_{across} = atan2(X_{start} - X_{end}, Y_{end} - Y_{start}) \tag{3.4}$$

$$\Theta_{along} = atan2(Y_{start} - Y_{end}, X_{start} - X_{end}) \tag{3.5}$$

The sin and cos of these angles are changed only during the creation of the map but are frequently used at runtime, so it makes sense to calculate those only when the map is changed and store as edge parameters.

The angle parameters are used at runtime to convert the coordinates between the world and corridor coordinate frames. Each corridor has its own frame that is described by distances across and along the corridor. The origin is located in the corridor corner to the left hand side of the start node, facing towards the end node (see Fig. 3.3).

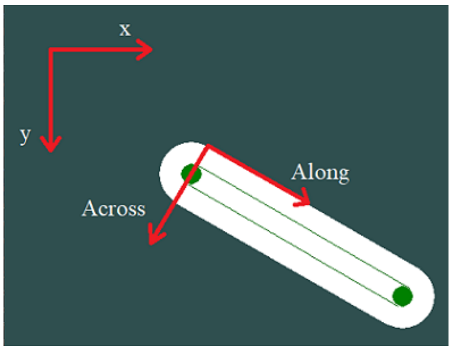


Figure 3.3: Coordinate frames. World coordinate frame described with X and Y, corridor coordinate frame described with Along and Across.

To represent the map we use a graph structure, where each corridor is represented by a graph edge and a node is placed in each corridor intersection. An example of such a graph is shown in Fig. 3.4a. The program loads the graph from a file, where all the coordinates of nodes and all the connections are specified.

For user convenience, we added some functionality to change the graph numerically (via loadable file representation of a graph) and by hand (at runtime). The graph is saved as a .txt file, in which every line is a new object. Lines start with a keyword that is followed by the parameters of a specific object. If a line starts

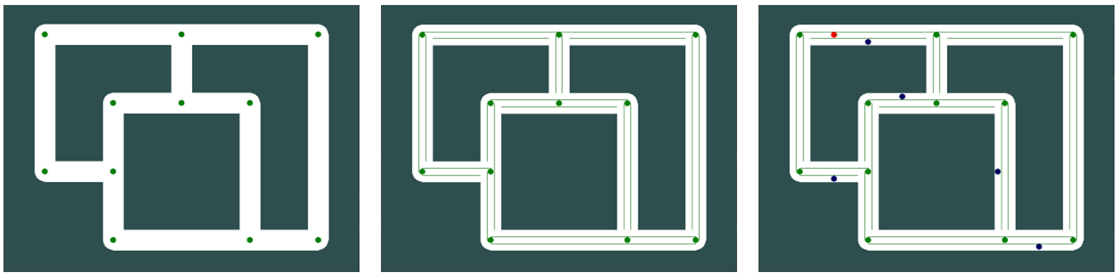

(a) Map layout. (b) Lanes drawn. (c) Robot (red) and agents (blue) drawn.

Figure 3.4: Simulator.

with any other word or symbol than a keyword, it is considered a comment and is skipped.

A line with the keyword ‘node’ adds a node to the graph. This keyword should be followed by 3 integers: node id and x and y coordinates. Anything after these parameters is considered a comment. Keyword ‘edge’ describes an edge and requires two node ids and corridor width. Rooms are added by keyword ‘room’ with the specified id, a corridor id (given by two node ids), and a distance along the corridor at which the room is located.

The chosen file format is convenient when it is necessary to precisely specify the required distances and angles. It may also be convenient to move the nodes to their proper place by hand. That is why it is possible to change the nodes’ coordinates by dragging the nodes or the corridors around in the simulator at runtime.

In the case of a large map, it is required for the map to be scalable and movable. We defined our own functions for drawing primitives that are built on top of the pygame draw functions, that incorporate the scale and shift variables. These variables are also defined in the graph file using keywords ‘scale’ and ‘shift’ respectively. The scale and shift are changed by scrolling or dragging the mouse over the background.

All performed changes may be saved or discarded by reloading the previous version of the graph by pressing a respective button. Saving and reloading are crucial for the runtime adjustments to be possible. Otherwise, to make a small change to the graph the user would have to tune the numerical parameters in the file and restart the program to test the new configuration.

The data format was chosen to be as simple and as readable as possible so that it is easy to understand what and where to change if necessary. It is simple and compact enough to work well with large graph sizes.

A drawback to this configuration is that it is impossible to apply a shift to a group of nodes. This leads to problems in the case of merging maps. The user will have to manually make sure that all the node and room ids are unique and apply a shift by hand in the graph file.

The simulator includes a time progression. There is a movement model for the robot and a simple human motion model. The simulated robot can be controlled in 3 ways: manually from the keyboard, automatically by the simulator, or by a tracker, moved by the real robot.

The human motion model requires start and goal nodes to navigate between. An agent in the simulation can be controlled by the motion model or by a tracker. The robot and the agents are shown in Fig. 3.4c. With the highest likelihood, the model will move the agent in a straight line, but there is a chance that the agent will move sideways, change its speed, stop for a moment or will randomly change its goal (see Eq. 3.6). To simulate abrupt behavior like a person remembering something and changing plans, the goal change happens after a stop for a random period of time with a probability of 4%.

$$\text{Agent's behavior}: \begin{cases} \text{Change speed, } 3\% \\ \text{Change lane, } 3\% \\ \text{Stop for a random period of time, } 3\% \\ \text{Continue without changes, } 91.3\% \end{cases} \tag{3.6}$$

The percentages indicate probability per second. Continue without changes has a probability of 91.3% because other possibilities are independent and do not exclude each other.

3.2.2 Lane division

To describe the movement across corridors and to make robot navigation easier the corridors are divided into lanes. The robot navigates along a lane as long as the lane is empty and will switch to another one if anything appears on the way.

The lanes are drawn in the simulator by thin green lines inside the corridors (see Fig. 3.4b). All lanes of one corridor have the same width and the number of lanes depends on the width of the corridor.

3.2.3 Path Planning

Path planning starts with calculating the shortest path between every pair of nodes on the graph. This function is called at the start and whenever the layout of the map is changed. The result is a double dictionary that lists the nodes of the shortest path given start and goal nodes. The calculation is done by the `shortest_path` method from the networkx library using the Dijkstra method with the weight of the edges being corridor length.

At every iteration of the program's main loop we look up the node path from the robot to the goal, but there may be a problem to solve. The robot may not start exactly at a node but from an arbitrary point inside a corridor. In this case, there are two nodes that are close to the robot and both of them may be the starting nodes. The goal of the robot (which is a door to a room) is also located in the middle of a corridor, so we have two possible goal nodes (see Fig. 3.5a).

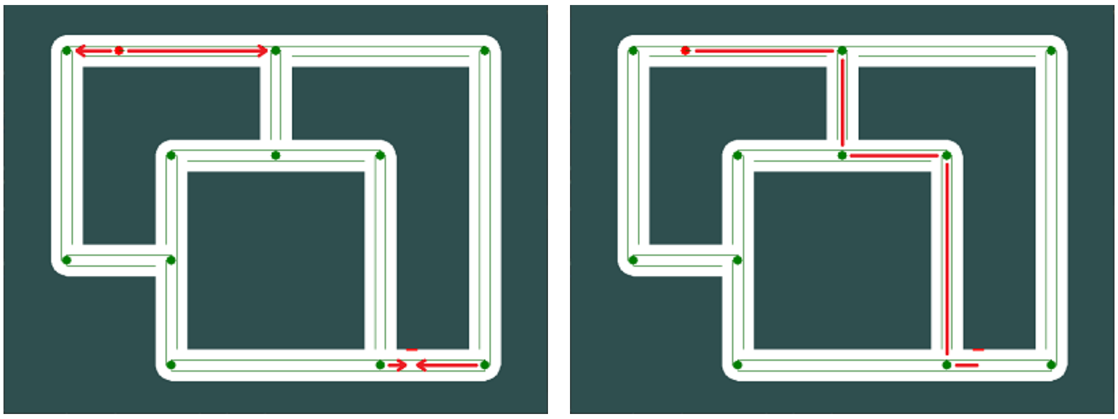

(a) Start and goal ambiguity. (b) Chosen path.

Figure 3.5: Path choice.

To solve this issue we consider all four possible combinations of start and goal nodes, we look up the shortest distances between the nodes and sum them with the respective distances from nodes to the robot and the goal. We then select the path with the shortest distance (see Fig 3.5b).

If a robot is initially located close enough to a node, the start ambiguity is resolved. We now have to consider only two paths to resolve the goal ambiguity.

3.2.4 Future Prediction

The future prediction mechanism is realized with the simulated movement of the robot and the agents. As this function is the most computationally heavy, it is reasonable to only consider agents in proximity to the robot. The Euclidean distance between the robot and the agents is not applicable here because an agent can be spatially close but in a separate corridor. So we initially decided to consider only agents located in the same corridor with the robot and the next one on the robot's path if the path contains at least one more corridor. In order to account for the agents, coming into the intersection from adjacent corridors, we now consider all the corridors connected to the next node on the robot's path. After selecting the corridors, Mapsim loops over all the agents present on the map and looks up in which corridors they are located.

To run the future prediction we use the automatic movement functions of the robot and the relevant agents. This leads to a position change of the agents in the simulation. After the future prediction is finished, the initial agents' positions have to be restored. So before running the prediction all the relevant agents are copied and only the copies are propagated into the future. These copies are deleted afterwards and all the agents are in their proper places for the program to continue.

Given the dynamic office environment, exact future prediction is impossible due to the prediction uncertainty, so it is only feasible to predict the agents' movement for several seconds. Since the robot is very agile, a short-term prediction is sufficient to ensure a safe path. To perform the future prediction, we take the robot's current speed and divide the corridor into pieces that the robot will traverse in one time step of the simulation. Starting with the first section of the corridor, all the agents are moved to the next time step, and the program checks if any appear inside the section. After this, Mapsim takes turns in considering the next section and moving the agents.

If an agent is located very close to the considered section, it is possible that on the next time step it will be bypassed without triggering a collision, so the considered section is always buffed with some margins.

If an agent is detected in the section, Mapsim looks up in which lane the agent is moving. In Fig. 3.6 the horizontal corridor is divided by the black vertical lines into sections. All the cells, resulting from the crossing of sections and lanes, which are colored green are safe to traverse and the ones in red are dangerous. One can see that dangerous are the cells in between the robot and the agent. The agent is slow enough not to endanger the robot in the left-most cells. The agent will also move from the cell that it occupies at the moment making the cell safe for the robot to traverse when it gets to it.

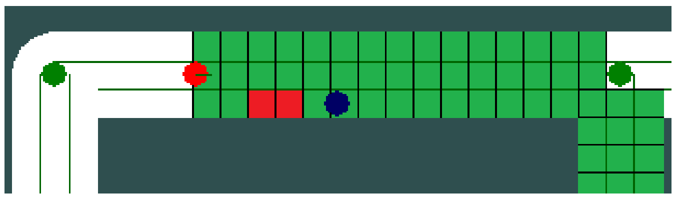


Figure 3.6: Future prediction corridor segmentation. Green cells are safe to traverse, red ones are dangerous.

After running the future prediction, the data is saved as a list of lists, where every inner list is an analyzed time cell, that contains numbers of lanes that will be occupied by the agents and are therefore dangerous. This data will be used later in the obstacle avoidance module.

3.2.5 Collision Avoidance

To perform collision avoidance we define waypoints on the robot's path. Each waypoint is located on one of the lanes at a certain distance along the corridor from a previous one. The obstacles are being avoided by choosing a proper lane for each waypoint.

We start calculating waypoint coordinates by determining to which prediction cell it belongs. Since the robot coordinates and all the distances between the waypoints are known, the waypoint distance along the corridor is also known. It is then compared to the prediction cell start and end distances. We then process the data from the future prediction module and see which lanes are occupied. Given the number of lanes in the corridor and the occupied lanes, free lanes are found. The right-most free lane is assigned to the waypoint.

After determining the lane, the waypoint coordinates are calculated. First, we calculate the coordinates of the waypoint in the corridor coordinate frame (described in 3.2.1). Since we already know the distance along, only distance across is to be calculated. I stands for lane index, W – for width, D – for distance:

$$D_{across} = (I + 0.5) * W_{lane} - \frac{W_{corridor}}{2} \tag{3.7}$$

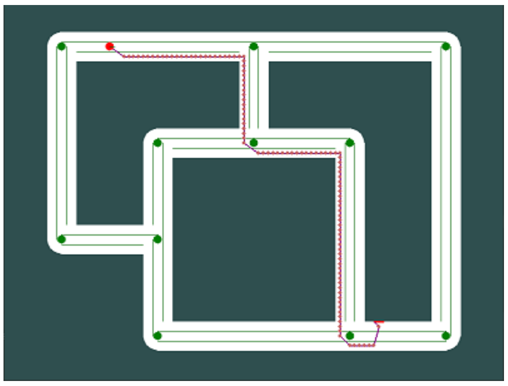
Now we transform the coordinates from the corridor frame (across, along) to the world frame (X, Y). x_{start} and y_{start} are the origin coordinates of the corridor frame, D stands for distance, and Θ_{across} and Θ_{along} angles that describe corridor orientation, sin and cos functions of which are looked up edge parameters.

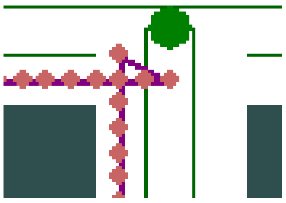
$$X = x_{start} - cos(\Theta_{across}) \cdot D_{across} - cos(\Theta_{along}) \cdot D_{along} \tag{3.8}$$

$$Y = y_{start} - sin(\Theta_{across}) \cdot D_{across} - sin(\Theta_{along}) \cdot D_{along} \tag{3.9}$$

We iterate over all waypoints from the robot to the goal, appending the coordinates (X, Y) to the waypoint path (see Fig. 3.7a). The future prediction module only returns data for the next several seconds, so if the goal is further than the predicted distance, all further waypoints will be assigned the default right-most lane. The coordinate calculation is fairly cheap, so it is not a problem to calculate them all the way until the goal.

There is one more problem to be solved regarding the waypoint path, which is the corridor overlap. We calculate the waypoints from the robot to the end of the


(a) Waypoint path.


(b) Overlap shown.

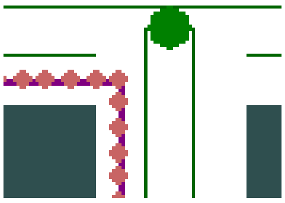

(c) Overlap removed.

Figure 3.7: Waypoint path.

corridor and then start again at the beginning of the next corridor. Since corridors are rectangles, they may overlap (see Fig. 3.7b). To solve this problem, we save the indices of waypoints at the end of each corridor and run a function after the path is calculated. The function `removeWaypointOverlap` looks at the distance between the ending and starting waypoints. If removing a waypoint shortens the distance, then the waypoint should be removed (see Fig. 3.7c).

The solution leads to a removal of the unnecessary loop in case of a right turn. Since the turn is sharp, the waypoints are also positioned sharply.

In case of a left turn, the rectangular corridors are spaced out, this leads to a corner being cut. This behavior is desired since there is no explicit need for sharp turns and these soft turns will lead to the robot being faster and more predictable for others.

3.2.6 ROS Bridge

To connect the Mapsim with the trackers it has to be able to communicate with ROS. The `RosBridge` module realizes this communication and manages the incoming data from the trackers and the outgoing waypoint data.

`RosBridge` consists of three main functions: robot callback, agent callback, and publish. Robot callback is called whenever there is new robot odometry and resets the robot's position, orientation and velocity. Agent callback resets agent's coordinates in the world and the corridor frames, and velocity. If the bridge is active, from the simulator's main loop the publish function is called and passed the next waypoint and the desired robot's direction. The function then forms a PoseStamped ROS message from the data and publishes it to ROS.

3.2.7 Multiple robots

The method proposed in this thesis is suitable to perform in a multiple-robot environment. There were two main possibilities for making it possible: centralized and distributed systems. Both solutions have some benefits and some drawbacks.

Centralized System

In a centralized case, all the robots are controlled by one machine. This enables the system to better handle situations where the robots are passing each other, or to make sure such situations never happen. Since the robots only need to listen for commands and not calculate anything on their own, they can also be simpler than in the distributed case. There is no need for an on-board computer.

On the other hand, this control option leads to a higher computation load on the machine it runs on. As future prediction is the most time-consuming task for the system, when scaled in size and numbers of robots and agents, it may become unfeasible to run on one machine. Although, since the agents mostly run independently, optimization with paralleling the computations on a GPU is possible. It is also vulnerable to failures. If something goes wrong and the system breaks, all the robots will stop working.

Distributed System

In a distributed system all the navigation software is running directly on the robot. This demands a robot to have an on-board computer. Although, for many possible applications it is necessary either way, and the computer has to be installed for other purposes. This may decrease route efficiency in the case of robot interaction, but it is expected for it to be minuscule since robots are far more predictable than humans.

On the positive side, if there is a problem and the system breaks, it will only stop one robot from working, all the others will continue as expected. Since the robots have on-board computing powers, the system can be extended with visual recognition software. The distributed system is also simpler to implement. All the interaction between robots is basically the same as between robots and humans, so to make this system support multiple robots one just has to add the robots.

Due to the benefits and simplicity of the distributed model, it was chosen for the thesis. The system can not currently be completely distributed, due to the necessity of one SteamVR application for tracking. So only the navigation part was distributed.

To create a prototype a SteamVR tracking system was chosen. It has to run on a single machine and then split the tracking data between the robots. So the distributed system data flow will have a tracking hub (see Fig. 3.8).

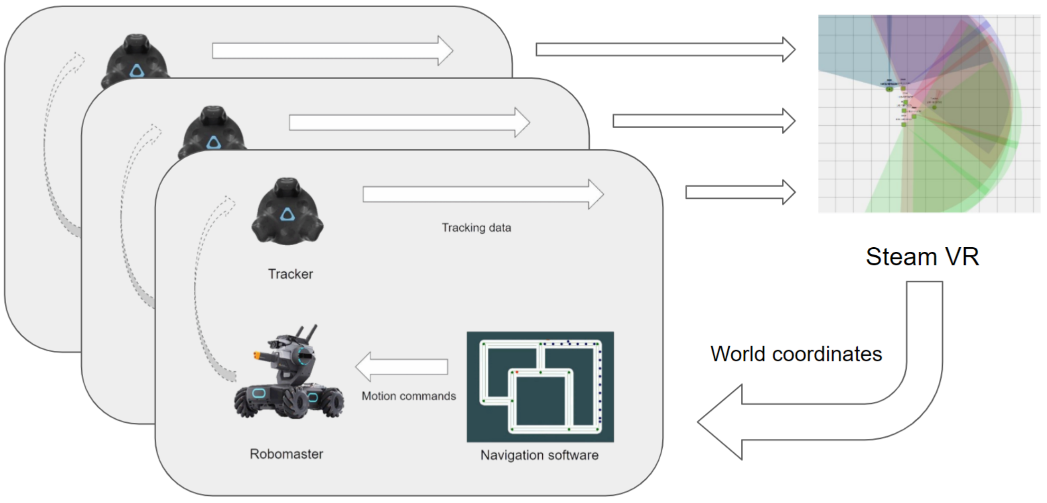


Figure 3.8: Multiple robots data flow.

4 Evaluation

In this chapter, we describe the experiments that we conducted with the robot and the navigation system. We show the test results and their evaluation. This chapter is divided into two sections: hardware and software evaluation.

4.1 Robomaster

In this section, we describe the prototype development testing and establish a bridge to the ROS ecosystem. The tests include the robot tests and the connection tests.

4.1.1 Tracking

Tracking system consists of the following components:

- 4 SteamVR Base Stations 2.0
- 2 HTC Vive Trackers 2.0
- 2 Bluetooth dongles
- SteamVR application.

Four StemVR base stations cover the area of $10 \times 10\,\mathrm{m}$. This area is sufficient for initial experiments but is insufficient for rolling out a practical system. Base stations are compact sized and only connected to a power supply. The work of the base stations during the tests was stable. To the naked eye, the tracking precision was not influenced by the number of base stations, however multiple base stations were necessary in case of obstructions. The tracker on top of the robot is always visible from any direction, but there is a chance of a person appearing between the robot and the base station. While the prototype was tested, a human agent was also equipped with a tracker, which was carried in a hand. This led to the tracker being always obstructed from some direction. This is why during testing we set up 4 base stations in the corners of the testing space.

The trackers determine their 6D coordinates using laser rays from the base stations, generate the tracking data, and then relay this data through Bluetooth to the dongles. Each tracker is paired with its own dongle. The dongles are connected to the Linux machine by wire. The trackers are powered by a battery, which lasts on average 4 hours. This is mostly enough for testing purposes, but when the battery died it was convenient that the trackers can also be powered directly by wire, connected to the computer instead of the dongle.

The Bluetooth connection has been tested and showed no problems in data transmission on distances up to 6 m.

SteamVR is an application developed by Valve Corporation to manage VR accessories. It manages the trackers in such a way that we can conveniently get the tracking data from it. But it has a couple of drawbacks. First, it only supports tracking with four base stations in parallel. This is not a problem for the prototype, since here, the environment is small and no significant occlusions occur. Nevertheless, this limitation can pose a problem for larger setups that cover a more complex environment. The second problem is that the data that the trackers send via Bluetooth is encrypted and we can not manage it ourselves without the SteamVR application.

The use of SteamVR also restricts our aim for a distributed system. All the robots with a Mapsim on board would have to be connected to a single SteamVR machine.

We first test the tracking and movement precision. The robot started at an arbitrary position in the testing area and moved to its target. We recorded the tracked distance from the target, which is presented in Tab. 4.1. To give a visual impression of the achievable accuracy, we show snapshots of the evaluation in Fig. 4.1.

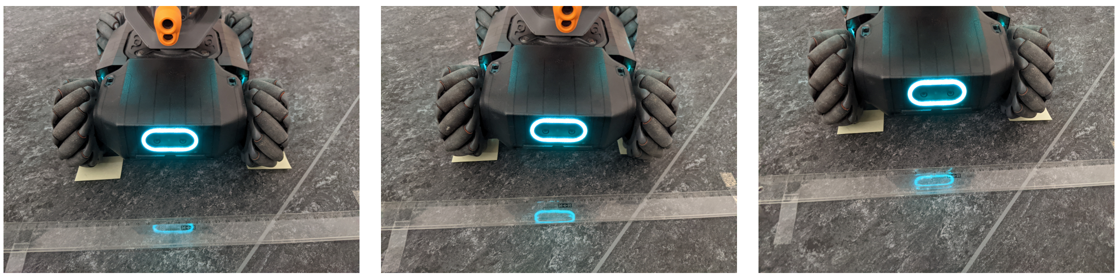


Figure 4.1: Precision test setup. We show three iterations of the test after the robot reached the target. The origin was marked with yellow papers directly under the front wheels.

The test results show that positioning precision is sufficient for the proposed application and the robot can stop at a point with a 2 cm tolerance. This tolerance may be lowered even further, but our goal was for the robot to go in a straight line and come to a full stop on the target without going too slow or overshooting the target.

Table 4.1: Movement precision. Shown as tracked coordinates in mm from the target point. The test was repeated 8 times.

i	x	y	z
1	−4	−4	1
2	1	19	0
3	9	9	0
4	−12	−4	1
5	1	7	1
6	−7	11	0
7	1	21	0
8	3	15	0

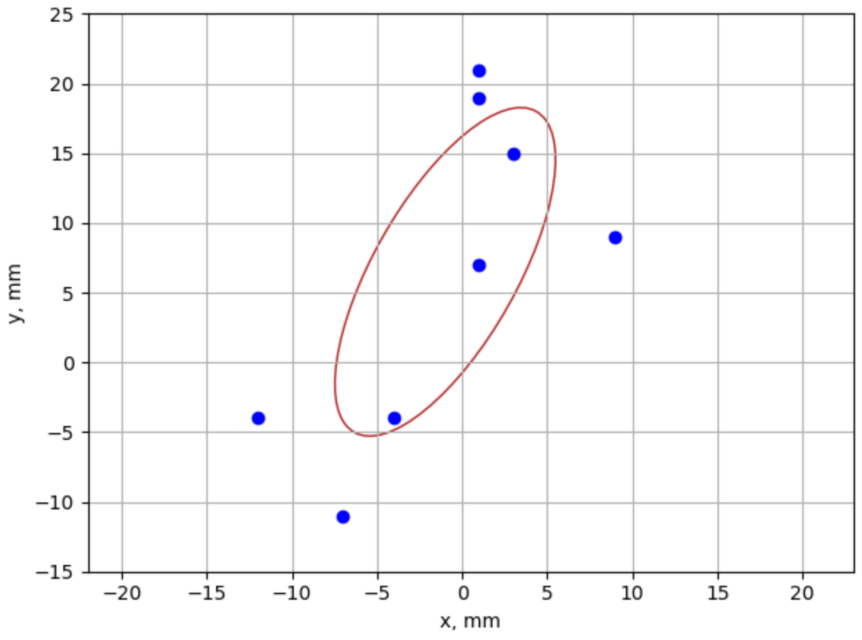


Figure 4.2: Precision test with 1 sigma confidence ellipse.

4.1.2 ROS Controller

To combine the tracking data from SteamVR with the navigation data from Mapsim and to compute the motion commands a `controller` ROS node was implemented.

During the testing of the initial version of the ROS controller, some issues with the rotation commands appeared. The formula 3.3 did not account for the angle difference larger than pi. So it was enhanced to properly calculate the direction and speed of rotation.

The testing also showed that the robot is very fast and often overshoots the target, so a PD control was implemented. After some testing, the coefficients were set for the robot to navigate around with high speed but approach a target point slowly.

The ROS controller also implements the websocket client to communicate with the DJI controller. During testing, we figured out that error messaging is necessary for the system. If an error is encountered in the DJI controller it is being sent to the ROS controller and displayed on the Linux machine.

4.1.3 DJI Robomaster

Here we evaluate the choice of DJI Robomaster S1 as a hardware robot platform for the prototype. There are many positive sides to this choice. The robot's omnidirectional platform is very agile and easy to control. The robot has a camera, data from which could be used to facilitate its environmental awareness.

The significant drawback of the S1 robot is that it is not ROS compatible. The robot can only be controlled by a Robomaster application. The control is encrypted, so we can not intercept it anywhere except from the Robomaster app itself. We had to manually fix this issue (see Sec. 3.1.5). The platform also does not support any extra modules to be placed on top of the robot, but for prototype purposes, this is not an issue.

Another issue is the automatic robot connection. We encountered some problems while trying to connect the robot to the Robomaster application. The connection should happen automatically, but if it doesn't, we can not easily fix it, as there is no way of troubleshooting.

Although there are some drawbacks, the robot's price is much smaller than any other educational robots' with similar capabilities.

4.1.4 Communication

In this section we describe the intra-system communication issues and solutions. The robot's internal WiFi router has one major drawback: only one device can be connected to it simultaneously. Since the Windows and Linux machines have to be connected together, we could not use the robot's internal router. The other possibility is to use an external WiFi router and connect both computers and the robot to it. This solution was realized and tested.

We tested multiple routers from different manufacturers and the testing showed, that the connection is difficult to set up. In some cases, the robot refused to connect to the router for an unknown reason.

The solution was found using an Android smartphone hotspot instead of a router. During testing, the system connected with a hotspot demonstrated fast and stable work.

4.1.5 Calibration

In this section we describe, how the calibration module (described in Sec. 3.1.3) was tested and evaluated. We have to define our own world frame to be able to shift it around without touching the base stations. This shifting is necessary to conveniently align the world frame with the simulated map.

To fit a plane to the data we need at least 3 data points. The tracking frequency is around 500 Hz, which is much more than necessary. For calibration, we lowered the tracking frequency to 1 Hz. During calibration testing, the dependency between the number of data points and the fitting precision was not confirmed.

After the calibration procedure is done, we calculate the transform between the abovementioned frames. Then we use a static tf broadcast function of ROS to conveniently transform the tracking data to the new frame and broadcast the data further to ROS controller and Mapsim. This ensures that every tracking subscriber gets the data in a proper frame and doesn't have to worry about frame conversion.

We now describe the precision test of our calibration method. First, we calibrate the robot inside an area of 1 m^2 (see Fig. 4.3). We then record the robot's reported height (see Tab. 4.2) at a distance from the calibration area with 1 m increments.

Table 4.2: Calibration precision. Measured as mm of height at a distance from calibration area.

Number of calibration points	33	52	43
Distance from calibration area in m	Error in mm		
0	0	0	0
1	−11	0	−1
2	−30	−2	−3
3	−48	−5	−6
4	−65	−7	−9

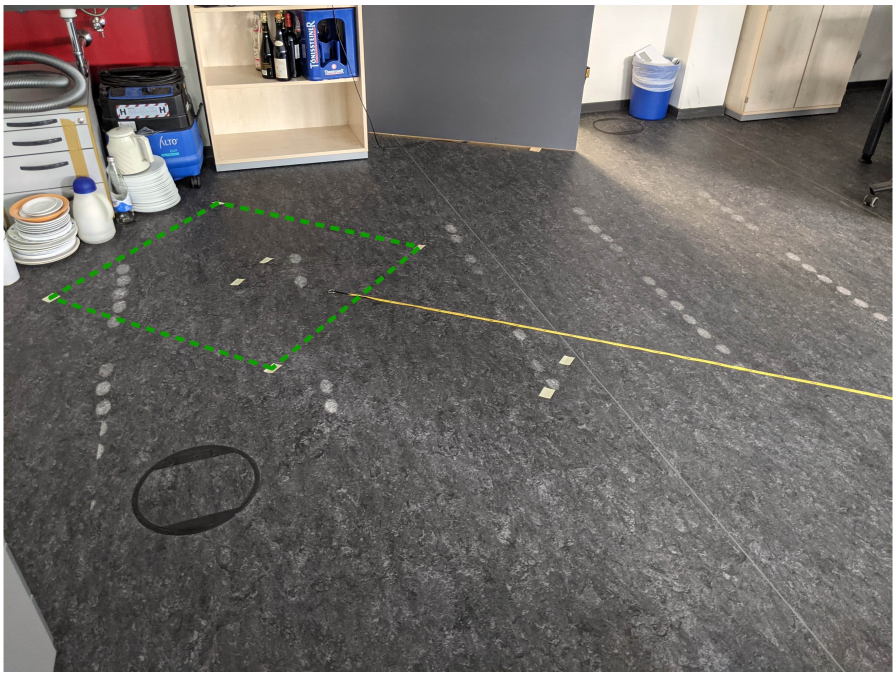


Figure 4.3: Calibration test setup. The calibration area of 1 m^2 is marked with green dashed line, the distance was measured with the yellow ruler.

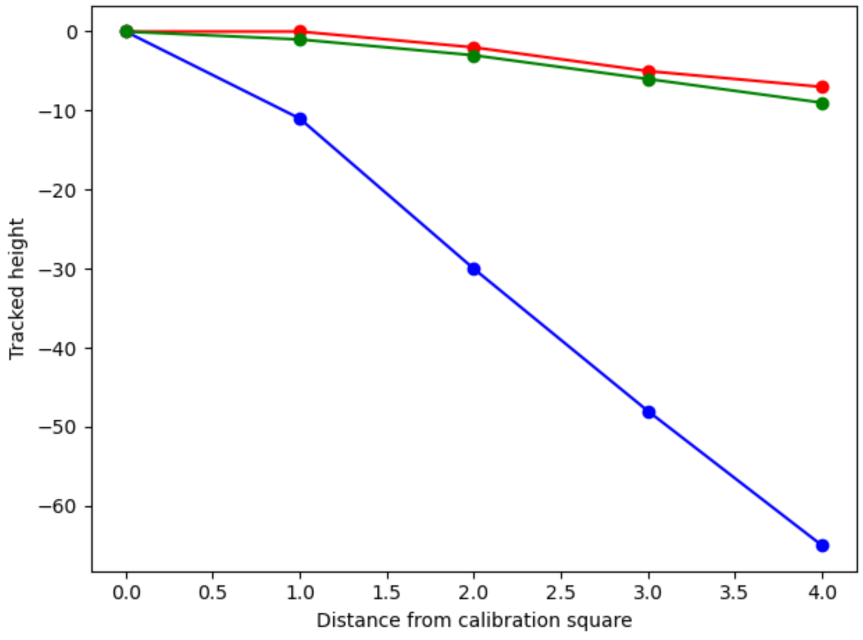


Figure 4.4: Calibration test. Robot's tracked height (0 expected, measured in mm) at a distance from calibration square (measured in m).

Then we plot the data to a graph (see Fig. 4.4). From this test, we concluded, that the calibration precision is sufficient for the targeted application, given the small region of calibration. The consistency of negative numbers on the y axis would be interesting to investigate. Since we only conducted this test in one setup, the possible reason for this trend could be a not completely straight floor. The blue line differs from the others but still shows acceptable precision. The possible problem with this test run could be a human factor or a tracking error during calibration.

4.1.6 Runtime

In this section, we describe runtime testing of the whole prototype assembled. We have tested the system in a $3 \times 3\,\mathrm{m}$ space in the middle of a room, the testing space is shown in Fig. 3.1c from one of its corners captured by the robot's camera and displayed on the Windows machine. The testing setup was described in Sec. 3.1.1.

For the first test, we have created a simple square map. The map consists of four orthogonal corridors, each divided into three lanes. We forbid the robot to move directly to the target, so the robot had to go around.

Initially, we did not use any obstacles, and the robot navigated around the map flawlessly. During the second test, a tracker was placed on the robot's path as a stationary obstacle. The robot successfully avoided the collision by switching the lane and continued on its path to the goal.

The robot's progression is shown on Fig. 4.5:

- The robot anticipated the obstacle movement, so initially, it built a path straight through the obstacle.
- After the start of the movement, it detected the idleness of the obstacle and changed its path around it.
- On the third set of images it is shown, how the robot continues on its path.

To perform the test with a moving obstacle, we created another map with one straight corridor. The corridor was divided into two lanes, one of which was occupied by a person, who moved in the direction of the robot at various speeds.

The test results are shown on Fig. 4.6:

- The robot is standing still at the beginning of the corridor. The obstacle is not yet moving, so the robot plans a path with a lane switch directly in front of the obstacle.

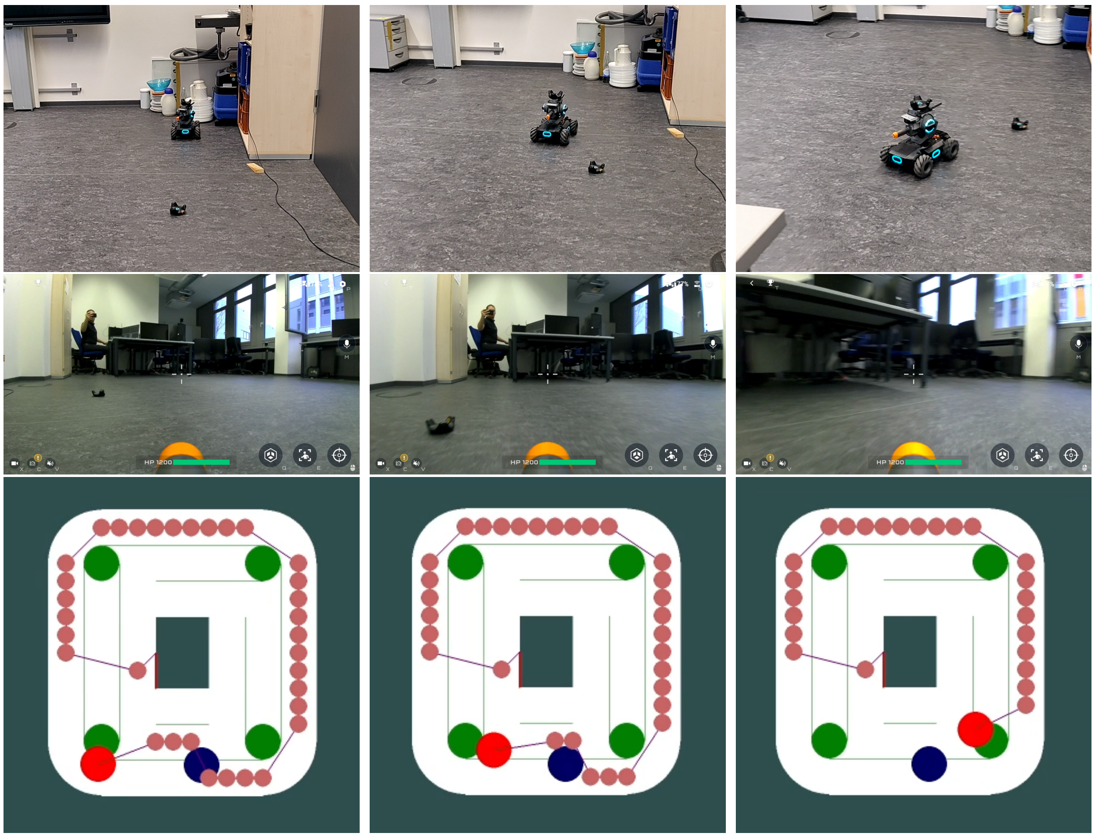


Figure 4.5: Static obstacle test. In three rows three different perspectives depicted, top to bottom: external camera, robot camera, Mapsim view. In three columns three time steps shown: before movement, moment of obstacle passing, movement continuation after obstacle passing. In Mapsim the robot is shown in red, the obstacle is shown in blue.

- Both the robot and the obstacle start moving simultaneously, the robot starts moving on the right lane (in Mapsim). Then the robot predicts the collision and switches to the safe lane. Given relatively high obstacle speed in the shown test, the robot switches to the left lane (in Mapsim) almost immediately.

- After passing the obstacle, the robot returns to the desired lane and continues on its path.

In this test, the rviz view shows the testing area from above. The world frame origin's axes are shown in green and red. During its movement, the robot leaves a blue trail. We flipped the rviz view upside down so that it matches the Mapsim view.

After testing, several minor mistakes were found, e.g. mirroring. The Mapsim

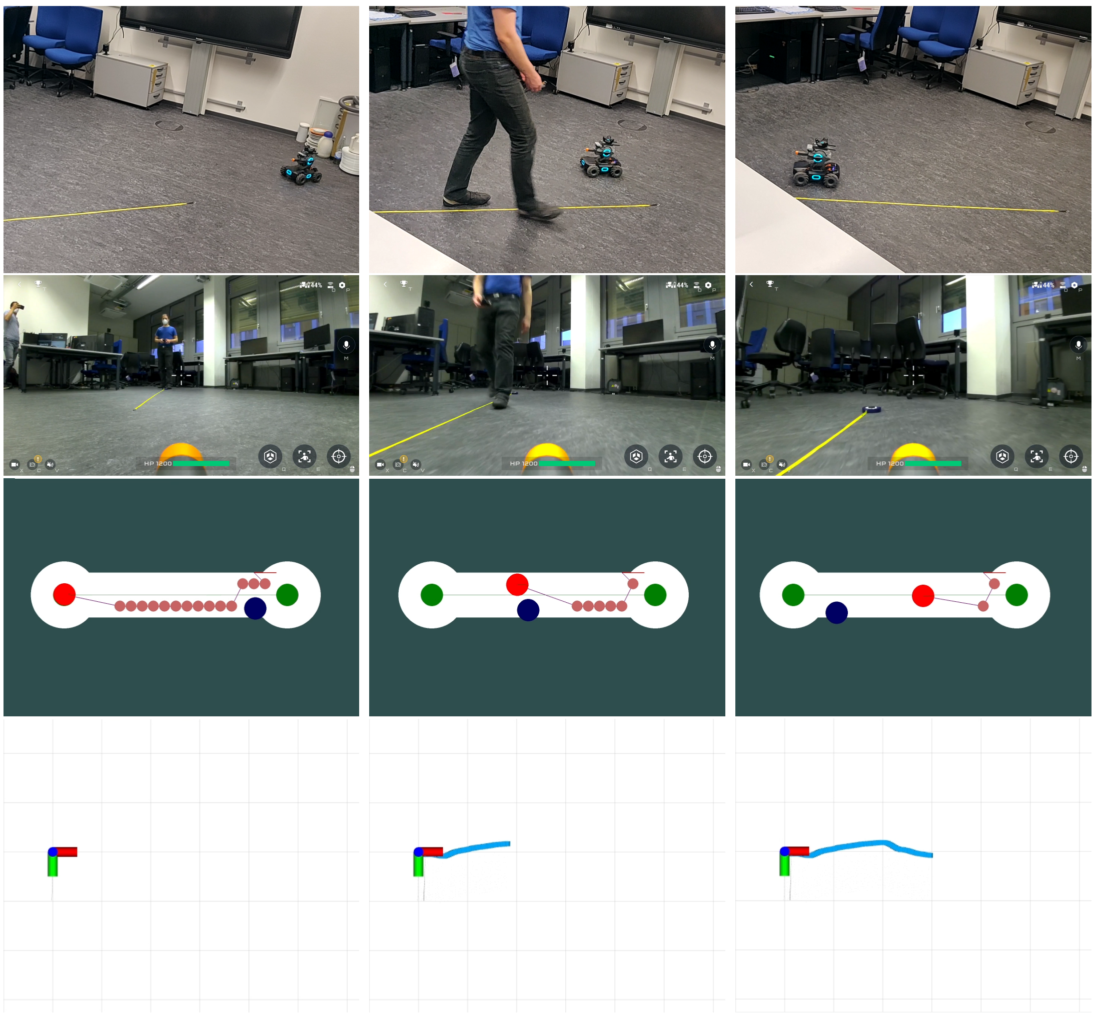


Figure 4.6: Dynamic obstacle test. In four rows four different perspectives depicted, top to bottom: external camera, robot camera, Mapsim view, rviz view. In three columns three time steps shown: before movement, moment of obstacle passing, movement continuation after obstacle passing. In Mapsim the robot is shown in red, the obstacle is shown in blue.

coordinate system was inherited from the pygame window pixel coordinates, so the y axis is pointed down. This led to the map being mirrored. This was easily fixed.

The test was repeated with different obstacle speeds: from very slow to fast walking. Running was not tested as there was not enough space in the testing area. The robot avoided all collisions successfully. It also showed different behavior depending on the obstacle speed: the slower the obstacle was moving the later the robot changed its lane. Given slow enough obstacle, the robot did not return to the right (in Mapsim) lane but continued directly to the goal.

4.2 Mapsim

This section contains the evaluation of the simulator along with all the techniques and algorithms that are used in it. It also contains the descriptions of all the performed tests and their results.

4.2.1 Simulator

External modules

For the creation of Mapsim, we used a pygame module. Its main tasks are time management, visualization, and event handling. Pygame was designed for video games. Since a simulator is itself a system very similar to a video game, a game engine is perfectly suitable for it. Initially, we considered the possibility of using an existing simulator, e.g. Gazebo. Gazebo is a complicated instrument, its size is 120 Mb, in comparison, Mapsim weighs 1.5 Mb. Our simulator is specialized for our tasks and therefore is much more simple to use.

Pygame is equipped with draw functionality. We initially used these functions for drawing the map. This limited the map size by the window size and did not provide any zooming functionality. To solve these issues we defined our own functions for drawing primitives that are built on top of the pygame draw functions, that incorporate the scale and shift parameters. This extension allowed us to create maps of unrestricted size and conveniently zoom to regions of interest (see Fig. 4.7).

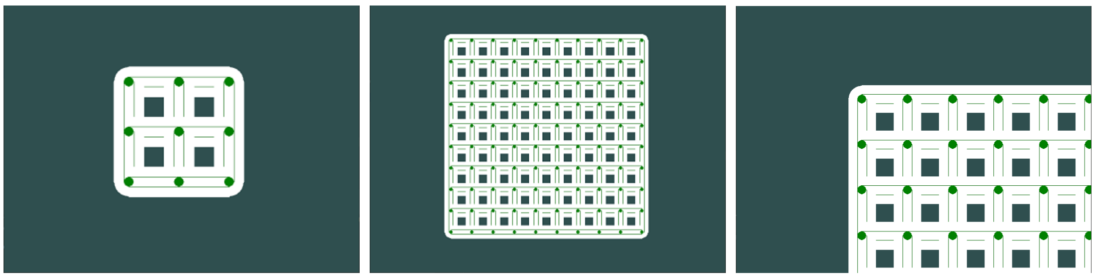

(a) Restricted map size. (b) Unrestricted map size. (c) Unrestricted sized map zoomed in.

Figure 4.7: Incorporating scale and shift into the simulator.

Complex corridor structures are easily and conveniently represented by graphs. We chose the networkx module to provide us graph functionality. We studied the possibility of creation of our own graph implementation. After testing the networkx module, it showed all the necessary functionality along with ease of use and convenient path planning features.

Optimization

Although the simulator is easy and functional, some features of it are not optimized. The simulator was tested on large-sized maps and it was found to significantly drop in performance with an increase in the number of corridors. For the performance test, we generated maps of various sizes (see Fig. 4.7). We ran the program for exactly 10 seconds and used a profiler to record the drawing time of the corridors (see Tab. 4.3, Fig. 4.8). The node distance is measured in the Mapsim units. Their scale to meters is a variable parameter, in the prototype we used a 40:1 scale.

Table 4.3: Corridors drawing time in ms depending on distance between nodes.

	Number of corridors			
Node distance	12	60	112	180
100	716	1149	2469	3881
600	933	1799	3289	5214
2000	1360	2971	5331	7468

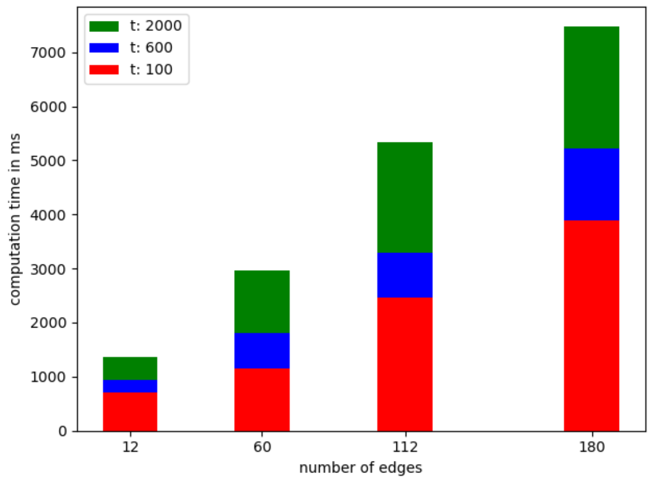


Figure 4.8: Drawing time is shown against the number of corridors. (t) denotes the distance between the nodes in Mapsim units.

Optimization possibilities were researched and we discovered that a large amount of data was being recalculated at each iteration of the main loop. This issue was only partially fixed. All the trigonometric functions are now only calculated once, after any change in the map layout, but a lot of pixel coordinates are still recalculated. We neglected these calculations initially, but it turned out to be a problem given a large enough amount of objects to draw.

We are unsure about the dependence of the drawing time on the corridor lengths since it does not affect the amount of calculations in Mapsim. This is probably related to how drawing is realized in pygame.

The excessive calculations may be a problem for a large environment, in case constant human monitoring is necessary. Otherwise, this is not a problem, since all the UI may be easily turned off without any consequences for the path planning. It also may be a problem for the testing, but currently, it is not, because all of our testing environments are small enough.

Corridors

We chose a rectangle as a corridor representation on the map. A rectangle is a natural choice since typical real world corridors have this form. It is easy to work with rectangles because they are easy to draw, it is easy to figure out if a point is inside the rectangle or not, they are easily scalable and easily dividable in lanes.

During development, we figured out that this approach has some drawbacks. Graph nodes are located in the center of opposite rectangle sides, so the corridors overlap at intersections. This problem was solved for path planning (see Sec 4.2.6), but given another corridor representation, we probably would not have to deal with it at all.

We researched the possibilities of changing the representation and came to a conclusion that it is possible to improve this representation without changing it too much. The nodes that bound the corridors may be located not exactly at the rectangle's side, but have some padding. This would remove the problem of corridor overlaps, but we would have to work out the robot's and agents' movement inside the nodes. Such representation would be more logical and would look nicer, but the current representation works good enough not to worry about it.

Map File

The graph representation of the map is stored in a .txt file. The file structure along with the used keywords is described in Sec. 3.2. The representation is very convenient to understand and use in the case of a simple map. The drawback of more complex maps is that it is impossible to group the nodes and apply certain changes only to those groups. This also makes it quite hard to merge different maps, since the user will probably have to change all the coordinates of one of the maps by hand.

A solution that groups the nodes and allows shift application only to certain groups is expected to be easy to implement, so we consider it an expedient extension.

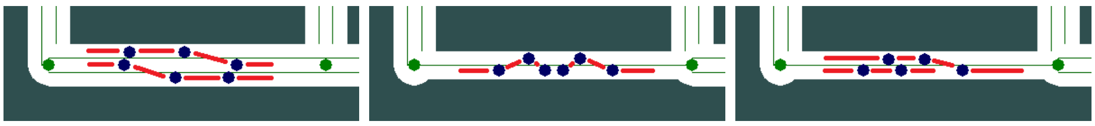
4.2.2 Agents

For the purpose of people simulation and behavior prediction, a simple human motion model (HMM) was created and applied to agents that navigate around the map. HMM contains a set of behavioral patterns that resemble a real human navigating a corridor. The modular concept of HMM allows a possibility of replacing it with another model, that contains different patterns. As long as this module has a proper interface, it can be connected to the simulator without issues.

Initial testing discovered several problems. The first problem is related to how agents avoid hitting each other. Agents are navigating along corridors looking at other agents nearby. If an agent nearby is detected and it is moving on the same lane, this lane is then considered dangerous and the first agent tries to move from this lane. The passing-on-the-right rule is implemented here, so given enough empty lanes no problem occurs (see Fig. 4.9a).

The problem arises when there is only one empty lane. The agents detect each other at the same time and each sees their lane as occupied and the other as empty. This leads to them both trying to switch lanes only to discover that after the lane switch nothing has changed (see Fig. 4.9b).

We have solved this problem by forbidding the passing-on-the-left for the agents (see Fig. 4.9c). This leads to a slightly unrealistic movement of the agents but solves the oscillation problem.



(a) Everything is fine given enough empty lanes.

(b) Oscillation happening.

(c) Oscillation solved.

Figure 4.9: Agent oscillation problem.

The second problem is also related to a drawback of the HMM. If an agent comes close enough to another agent and there is no empty lane to go to, the agent will stop and wait for other agents to move. In case the corridor gets initially obstructed in some way, other agents that are trying to go through this corridor will come close to the obstruction from both sides and stop moving.

In some testing scenarios, we ran a large number of agents, and sometimes these situations occurred. These situations can happen both in the middle of a corridor (see Fig. 4.10a) and at an intersection (see Fig. 4.10b).

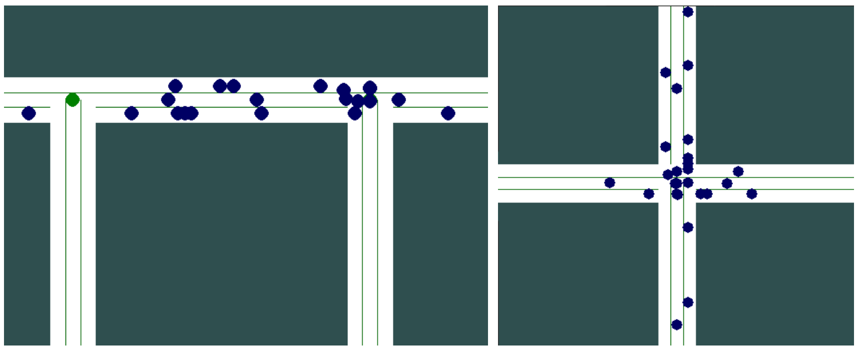

(a) Jam in the middle of a corridor. (b) Jam at a corridor intersection.

Figure 4.10: Corridor jam problem.

This is an unsolved problem so far, but it is not a crucial one. During testing, this problem occurred only when excessive amounts of agents were released, which is an unrealistic situation. In other tests this situation never took place.

Table 4.4: Agent move time in ms depending on the number of agents.

Number of agents	10	50	100	500
time in ms	484	1502	2936	9539
move call count	5375	27 940	55 387	77 625
max move call count	6000	30 000	60 000	300 000

We tested, how much time does the program need to simulate the movement of the agents. We ran the program for 10 seconds at 60 fps, so if we multiply both numbers with the number of agents, we get the maximum number of calls to the `move` function. In the first 3 tests (see Tab. 4.4) the call count was slightly less than maximum, as the program also does some extra calculations at the start. During the last test, the number of agents was too high, so the program throttled. This is shown by the call count being much lower than expected.

From this test, we can conclude that the program is capable of running dozens of simulated agents without problems. At runtime, the program is able to handle even more agents, as they are not simulated.

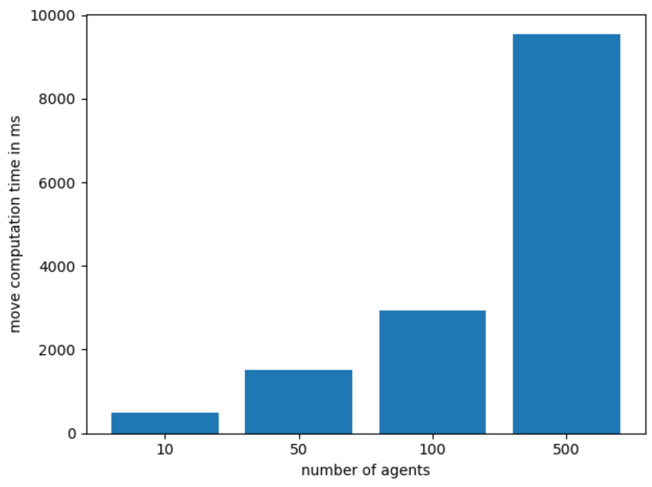


Figure 4.11: Agent move time is shown against the number of agents.

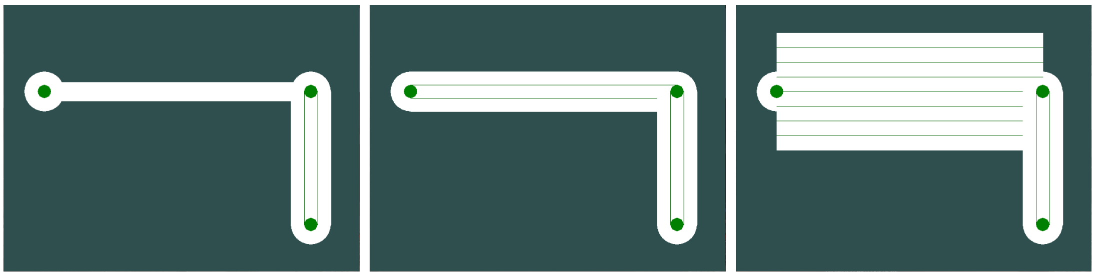
4.2.3 Lane Division

Initially, we chose the corridors to be divided into three lanes, which allows the ease of navigation around obstacles. This division showed an acceptable result but is too restricted for real use. A couple of hyperparameters were added to ensure the flexibility of the solution. Real corridors significantly differ in width, so many more people may simultaneously use them. In such a case all three lanes would be permanently blocked, so we need to calculate the number of lanes depending on the width of the corridor. Another aspect of the variable lane width is the ease of obstacle avoidance: we can assume the size of the obstacle to be the same as the lane width, this assures a safety margin between the robot and the obstacle.

It is possible to create corridors of different widths in Mapsim (see Fig. 4.12). We have tested corridors from very narrow to very wide. Such wide corridors may be useful to describe large halls, so we did not put any boundaries on corridor widths. In a corridor with a single lane the robot can not avoid any obstacles and will always stop if someone enters the corridor from the other side.

4.2.4 Path Planning

The path planning calculation is realized by the `shortest_path` method from the networkx library. The library provides two methods to solve the problem: Dijkstra and Bellman-Ford algorithms. We run both algorithms on the graph and calculate the shortest path between each pair of nodes. The planning of the path at runtime comes down to a simple look-up. We evaluated the computation time of this look-up table initially depending on the number of nodes and on the used algorithm



(a) Narrow corridor with 1 lane. (b) Regular corridor with 3 lanes. (c) Wide corridor with 8 lanes.

Figure 4.12: Corridors of different widths.

(see Tab. 4.5, Fig. 4.13).

Table 4.5: Path calculation time in ms against the number of nodes.

	Number of nodes		
Algorithm	9	36	100
Dijkstra	0.35	2.76	21.08
Bellman-Ford	0.43	7.97	69.22

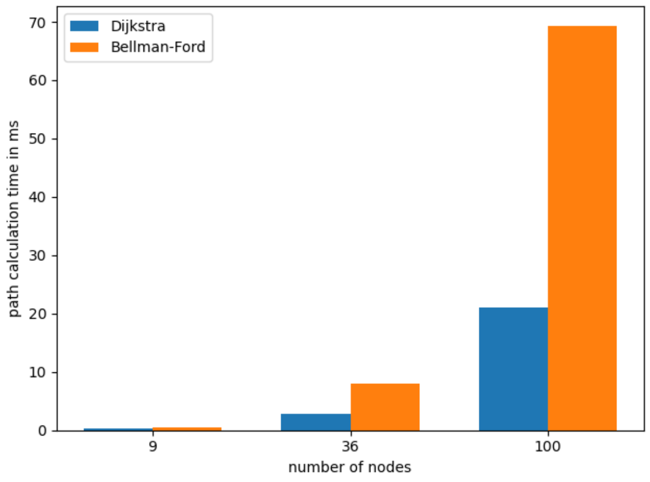


Figure 4.13: Path calculation time against the number of nodes. Two algorithms shown.

The tests show that for our problem the computation time is minuscule. It is notable, how much faster the Dijkstra algorithm is. Unlike Dijkstra, the Bellman-Ford algorithm is capable of handling negative weights, but since we are dealing with distances, we don't need it.

4.2.5 Future Prediction

For the purpose of facilitating obstacle avoidance, a future prediction technique was used. It uses the human motion model to predict, where the agents will move and which lanes should the robot avoid. As described in Sec. 3.2.4, we divide the corridor into time sections. Section size (S) is defined by multiplication of the distance, that the robot covers in one time step and a fastforward (F) parameter (will be described later). The time step is calculated, given simulator's frames per second (f). R stands for robot's current speed:

$$S = \frac{R}{f} \cdot F \tag{4.1}$$

In the initial version of the future prediction subsystem, the number of time sections to consider was equal to the number of time steps in the prediction horizon (will be described later). This led to excessive amounts of computations since the prediction precision is insufficient to support such small time sections. We then introduced a fastforward parameter, which allowed us to increase the size of these time sections, which led to much faster computations without loss in precision.

Table 4.6: Future prediction computation time in ms depending on the number of agents that are close to the robot and the fastforward parameter. In each data cell the call count is reported at the top and the time at the bottom.

	Number of agents				
Fastforward	1	3	5	10	20
1	589	341	–	–	–
	3850	7925	–	–	–
5	586	588	589	426	–
	1007	3044	4536	7257	–
10	594	590	589	593	372
	580	1708	2711	4868	7411
30	589	588	588	588	590
	230	777	1278	2309	4606

We conducted a test to see the performance of the future prediction module. We ran the program for 10 seconds and with the help of a profiler recorded the time used by this module (see Tab. 4.6). The call count of this module only depends on the run time and the fps. So the expected call count is around 590. The gathered data shows that given a low fastforward value, the supported agent count is also very low, e.g. with a fastforward of 1, a maximum of 2 agents is supported without throttling.

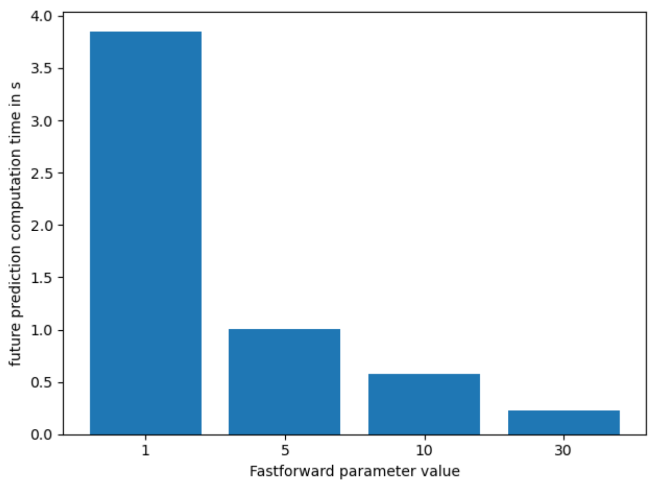


Figure 4.14: Future prediction computation time is shown against the fastforward parameter. Only a single agent is predicted.

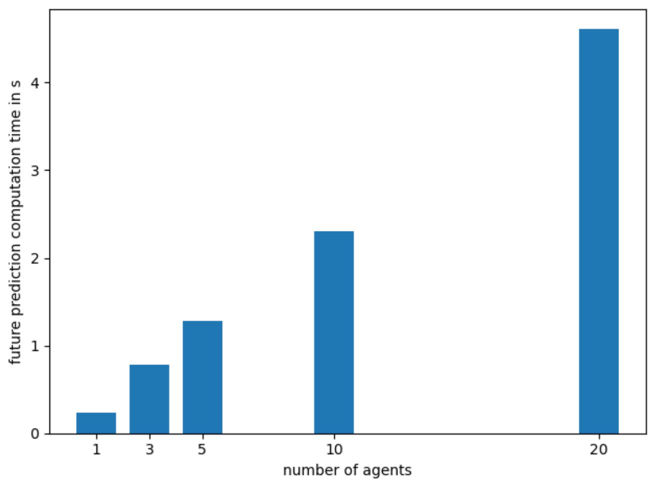
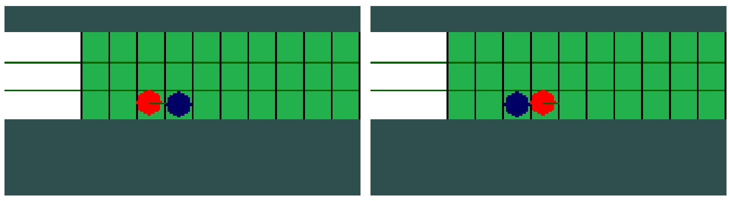


Figure 4.15: Future prediction computation time is shown against the number of agents. Fastforward parameter is set to 30.

During tests, we also looked at the resulting paths of the program. With a



(a) Current prediction step. (b) Next prediction step.

Figure 4.16: Neighboring time sections problem. Robot is shown in red, obstacle is shown in blue.

fastforward parameter of 30, the paths looked very similar to ones that used lower values.

Developing the future prediction module we found two major issues. The first one is encountered when the fastforward parameter is low enough so that the time section size is less than the robot's size. This leads to such a path that robot's center avoids hitting the obstacle's center, but of course, in the real world, a collision would happen. The second one happens when the robot and the agent appear in the neighboring time sections and switch places in the next prediction step. This is demonstrated in Fig. 4.16.

To solve both these issues we introduced the prediction margin parameter. During prediction, when the section coordinates are calculated, we extend the section size in both directions by a safety margin. For the first problem, it leads to the section size never being less than the minimum distance between the robot and the obstacle.

As for the second problem, it is solved, given a reasonable agent speed. If an agent moves fast enough, currently, it is impossible to predict the collision, since the infinite margins are not feasible. On the other hand, this is not a problem, since a human is far better at solving dangerous situations than the robot, and if a person is running along a corridor, the robot has to stay as predictable as it can be. This is achieved exactly by ignoring the running person.

We now introduce the concept of the prediction horizon. As mentioned in Sec. 3.2.4 it is only feasible to perform the prediction in the nearest future. The prediction boundary, which we call the prediction horizon is defined as the time (in seconds) that we predict into the future.

4.2.6 Collision Avoidance

The collision avoidance module builds a waypoint path that leads the robot around the obstacles, using the input data from the future prediction module.

We decided that the suitable format for the prediction data transfer would be the list of lists, where every inner list is a time section, that contains ids of lanes that will be occupied by the agents when the robot will reach that time section.

We equip the robot with a set of behavioral rules that allows it to avoid moving and standing obstacles efficiently and in a way predictable for humans. For example, we implemented a passing-on-the-right rule, that makes the robot to prefer a socially more expected action. In case passing-on-the-right is impossible, the robot then chooses to pass on the left.

In case all the lanes will be occupied by agents at some point in time, the robot stops and stands still waiting for the agents to move. The drawback of such an approach is that in case the corridor is blocked by a group of obstacles, that do not wish to move, the robot will be stuck indefinitely. A valuable extension, solving this issue would be a ban of movement in this corridor after a timeout. This will lead to the robot recalculating its path using other corridors. It would be then important to set a checking function for unbanning the corridor whenever the blockage is gone.

Testing demonstrated that there is a drawback in the robot's rule set. Whenever the robot encounters the situation displayed in Fig. 4.17, it falls into a trap. While calculating the path, the robot decides not to switch lanes avoiding the first obstacle, because it does not consider the second obstacle yet. Then the second obstacle is encountered and it can't be avoided, because the first obstacle prevents the robot from changing the lane through it. So the robot sees this obstacle formation as unavoidable, even though there is an empty lane on the left.

This problem can be solved using retrodiction. After calculating the prediction data, we should look through the data in reverse order and in case of such trap formations extend the obstruction of the trap lane.

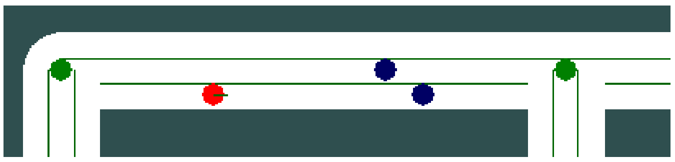


Figure 4.17: The trap that prevents the robot from avoiding the collision is shown.

We have tested the Mapsim on dozens of complex maps and have visually examined the calculated paths. We came to a conclusion that the paths are close to the shortest ones possible. There is one parameter that defines the distance between the consequent waypoints. Since we define a lane change as an assignment of different lanes to different waypoints, this distance defines the smoothness of the lane change. The higher is the distance, the smoother is the change, but the more time the robot needs to avoid danger.

As an extension, it is feasible to change the constant waypoint distance to a variable one depending on the distance to the closest agent. To have more maneuverability in proximity to an agent and to ensure a smoother and more predictable path at a distance.

The initial version of the collision avoidance module was called at every time step of the simulation to recalculate the waypoint path. This has been proven to be excessive, as the same result could be achieved with planning at a lower frequency with a major performance improvement.

A throttling test has been carried out, where the throttling parameter shows the number of simulation updates between planning recalculation. The recorded time (see Tab. 4.7) is given by the profiler after 10 seconds of program runtime with 5 agents near the robot and a fastforward parameter of 30. It is shown that it is possible to reduce the planning time by a lot, but it affects how quickly the robot can respond to a sudden situation. In our further tests, the throttling parameter was set to 1.

Table 4.7: Path planning time in ms depending on throttling parameter.

Throttling	1	3	5	10	30
Time is ms	4427	1622	1035	497	160

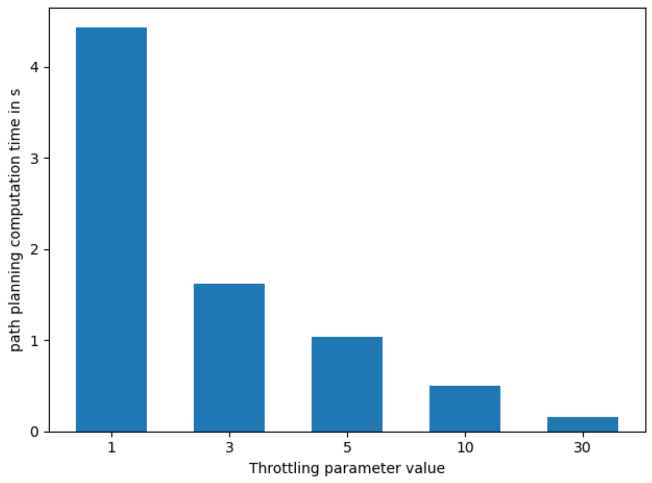


Figure 4.18: Dependency of planning performance on plan throttling.

4.2.7 RosBridge

RosBridge is a part of Mapsim that is responsible for communication with ROS. This module realizes the subscription to the odometry topics and updates the coordinates of the robot and the obstacle.

The odometry message is 3-dimensional and represents the rotation as a quaternion. Mapsim is 2-dimensional, so the rotation should be converted to a euclidean angle. The conversion is done with the following formulas, where Q stands for quaternion and Θ is the sought angle:

$$A = 2(Q_w \cdot Q_z + Q_x \cdot Q_y) \tag{4.2}$$

$$B = 1 - 2(Q_y^2 + Q_z^2) \tag{4.3}$$

$$\Theta = \arctan2(A, B) \tag{4.4}$$

The bridge is then used at each Mapsim iteration to publish the coordinates of the next waypoint to ROS.

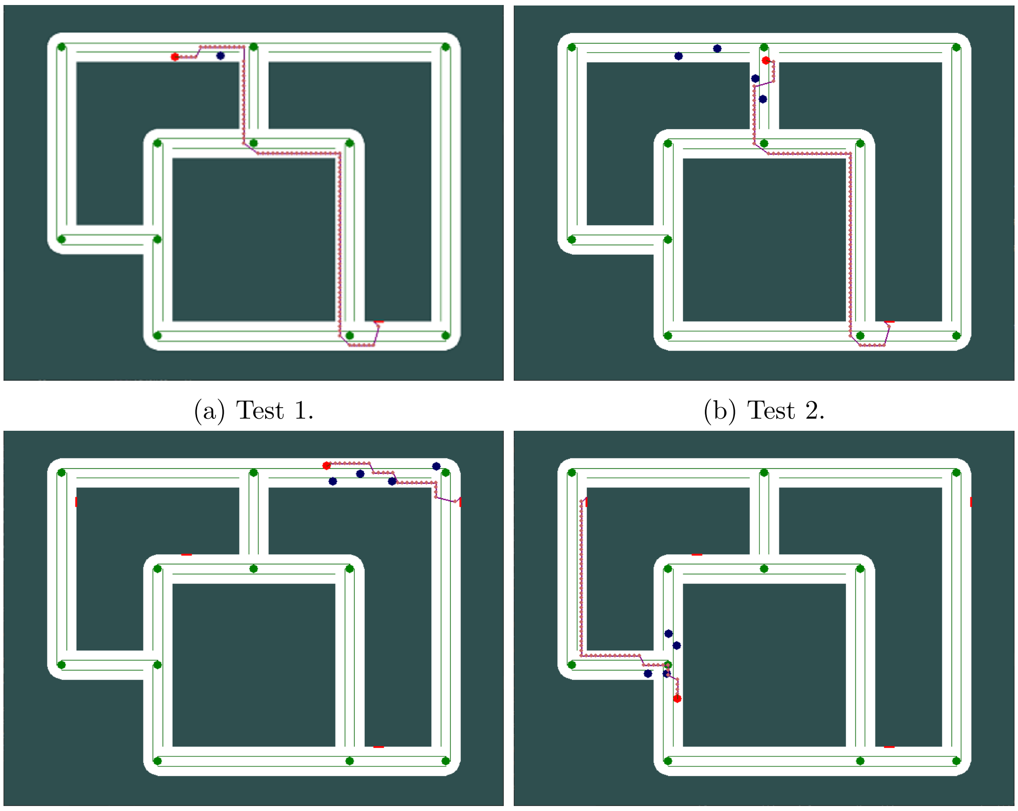
The big issue that RosBridge solves is the consistency of units of measurement. The odometry is measured in meters, but the Mapsim operates in its own units of measurement. Those units were initially the pixels, but since scaling was introduced, we can not call them pixels any more. After some tests, we determined a coefficient that is used for coordinate scaling in RosBridge.

4.2.8 Runtime Evaluation

We have tested the Mapsim in various situations, some of which are shown in the images below. As navigation without obstacles does not pose any challenges, only tests with multiple agents are shown (see Fig. 4.19).

Tests that are shown in Fig. 4.19a and Fig. 4.19b depict how the robot avoids collision in easy and complex situations. The directions used in the description are the directions on the images. Test in Fig. 4.19c shows how the robot goes around the agents, that move in the opposite direction. It may seem that the margin between the robot's path and the agents is too small, but the agents will move further left when the robot will reach that point in space. So there is no risk of collision.

The last test (see Fig. 4.19d) shows a complex scenario when multiple agents are coming from the left and top corridors and go down directly into the robot. The test shows that the robot is capable of predicting all the incoming traffic and build a safe path to the target. One of the agents is located directly on the robot's path and this may be confusing, but in the simulation, it moved at a relatively high speed and went down before the collision.



(a) Test 1. (b) Test 2.

(c) Test 3. (d) Test 4.

Figure 4.19: Mapsim test.

A scaling test was conducted with 70 agents running on a large map. it is shown in Fig. 4.20 that the robot is able to avoid collision and build the path directly to the target. It is actually possible for the robot to go to its target in a straight line, which would make the path shorter. But this is the case only in such a testing environment, in a real world situation, such shortcuts would not be possible.

Mapsim was designed as a distributed system. Each robot in an environment should run its own instance of the navigation software. Mapsim is able to simulate the robots the same way as with the humans by using another behavioral rule set. We ran a test with 1 robot and 1 agent that represented a second robot and looked at the robot's behavior (see Fig. 4.21). We also set the robots to prefer the middle lane for navigation. Then we reversed this scenario and looked at it from the perspective of the second robot. In this case, the collision was easily avoided by the robots moving from each other's way.

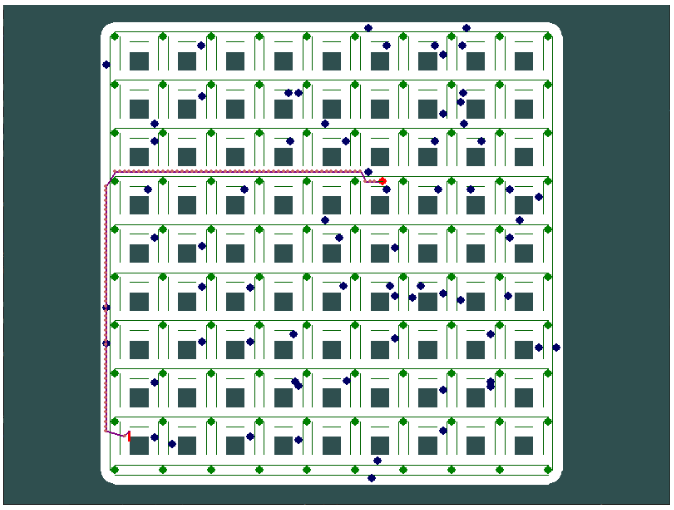
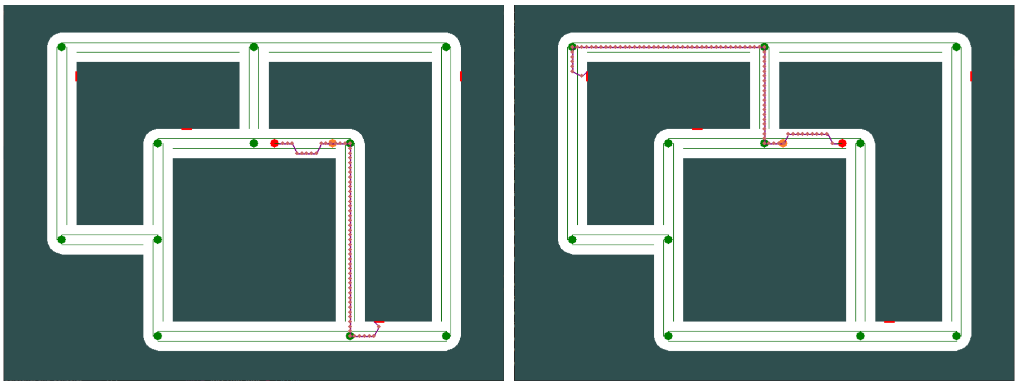


Figure 4.20: Mapsim scaling test.



(a) Point of view of robot 1. (b) Point of view of robot 2.

Figure 4.21: Test with two robots. Robot that is navigated by Mapsim in red, the other robot in blue.

5 Conclusion

Achievement

In this work, we developed a Mapsim method for robot navigation and collision avoidance. Mapsim offers a simple and computationally low-cost path planning solution with the ability to predict the behavior of other robot and human agents.

Mapsim sequentially implements four tasks: lane division, path planning, future prediction, and collision avoidance.

Mapsim is optimized for environments consisting of complex corridors. The simulator divides the corridor into a finite number of imaginary lanes. Such a concept simplifies the path planning and makes it possible to apply a wide variety of existing navigation and path planning methods created for self-driving vehicles, such as lane changing policies, following another object, solving a conflicting scenario with an approaching agent moving in the opposite direction, etc.

Path planning in a complex corridor environment can be easily solved using a Dijkstra algorithm. Such type of environment is not prone to change and even in the most complex map, all the paths can be precomputed. The path planning problem remains easily solvable until other actors are added to the scene. In such a case, precomputed path planning can still be used, but it has to be enhanced by a collision avoidance module.

In a dynamic social environment, the robots are expected to navigate safely and predictably in proximity of people. For the purpose of safety, we predict the movement of other agents in the vicinity of the robot. This leads to plausible behavior and a sufficient safety margin.

Prediction of other actors in the scene facilitates our collision avoidance algorithm. We use the lane changing technique with a set of defined rules in order to efficiently perform collision avoidance. This set of rules allows to achieve robot predictability.

During software simulations in Mapsim, the robot showed its ability to navigate safely in complex environments with dozens of agents. The robot efficiently changes lanes to avoid static and dynamic obstacles utilizing the rules of social behavior. The system has a big number of parameters that can be changed to account for different situations. Depending on corridor width and robot and obstacle size, the number of lanes can be easily changed.

We have designed a safe system, using inexpensive and widely available hardware components, easily adaptable for different tasks. The system was tested as a hardware prototype, using DJI Robomaster S1 and SteamVR. Testing confirmed the ability of the system to be applicable to real hardware. Our method showed the ability to avoid static and dynamic obstacles.

We achieved our goal of creating a safe, effective, and computationally cheap navigation software and applied it as a real world prototype.

Extension

The developed system demonstrated safe and reliable behavior and can be further developed with a perspective for practical applications.

Mapsim can be extended in multiple ways. The simulator lacks some functionality, like grouping nodes and simultaneously shifting a group of nodes. The future prediction module can be enhanced by using a more profound human motion model. The collision avoidance module could be extended by more sophisticated lane changing maneuvers. The robot behavior can be enhanced by handling special situations at intersections.

As a hardware extension, other hardware platforms can be used, as far as both Robomaster S1 and SteamVR showed some limitations. For example, the tracking system can be enhanced or even replaced by a camera with a computer vision module that would detect and track obstacles.

The prototype can be extended to include several robots moving in a common environment.

List of Figures

List of Tables

Bibliography

Allgeuer, P., M. Schwarz, J. Pastrana, S. Schueller, M. Missura, and S. Behnke (2018). "A ros-based software framework for the nimbro-op humanoid open platform." In: arXiv: `1809.11051 [cs.RO]`.

Behnke, S. (2004). "Local multiresolution path planning." In: *Robocup 2003: robot soccer world cup vii.* Ed. by D. Polani, B. Browning, A. Bonarini, and K. Yoshida. Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 332–343. ISBN: 978-3-540-25940-4.

Bennewitz, M., W. Burgard, G. Cielniak, and S. Thrun (2005). "Learning motion patterns of people for compliant robot motion." In: *The international journal of robotics research (ijrr)* 24.1, pp. 31–48.

Brechtel, S., T. Gindele, and R. Dillmann (Oct. 2011). "Probabilistic mdp-behavior planning for cars." In: DOI: `10.1109/ITSC.2011.6082928`.

Broggi, A., A. Zelinsky, M. Parent, and C. Thorpe (Jan. 2008). "Intelligent vehicles." In: pp. 1175–1198. DOI: `10.1007/978-3-540-30301-5_52`.

Chakraborty, M. (2021). *Top 5 self driving car companies to watch out in 2021.* URL: `https://www.analyticsinsight.net/top-5-self-driving-car-companies-to-watch-out-in-2021`.

Chen, Y. F., M. Everett, M. Liu, and J. P. How (2018). "Socially aware motion planning with deep reinforcement learning." In: arXiv: `1703.08862 [cs.RO]`.

Colyar, J. and J. Halkias (2007). *Us highway 101 dataset. fhwa-hrt-07-030.*

Dijkstra, E. (1959). "A note on two problems in connexion with graphs." In: *Numerische mathematik.*

Dosovitskiy, A., G. Ros, F. Codevilla, A. Lopez, and V. Koltun (2017). "Carla: an open urban driving simulator." In: arXiv: `1711.03938 [cs.LG]`.

F.A. Oliehoek, C. A. (2015). *A concise introduction to decentralized pomdps.*

Hart, P. E., N. J. Nilsson, and B. Raphael (1968). "A formal basis for the heuristic determination of minimum cost paths." In: *Ieee transactions on systems science and cybernetics* 4.2, pp. 100–107. DOI: `10.1109/TSSC.1968.300136`.

Karaman, S. and E. Frazzoli (2010). *Incremental sampling-based algorithms for optimal motion planning.* arXiv: `1005.0416 [cs.RO]`.

Kavraki, L., P. Svestka, J. Latombe, and M. Overmars (Sept. 1996). "Probabilistic roadmaps for path planning in high-dimensional configuration spaces." In: *Ieee transactions on robotics and automation* 12, pp. 566 –580. DOI: `10.1109/70.508439`.

Khatib, O. (1985). "Real-time obstacle avoidance for manipulators and mobile robots." In: *Proceedings. 1985 ieee international conference on robotics and automation.* Vol. 2, pp. 500–505. DOI: 10.1109/ROBOT.1985.1087247.

Killing, C., A. Villaflor, and J. M. Dolan (2021). *Learning to robustly negotiate bi-directional lane usage in high-conflict driving scenarios.* arXiv: 2103.12070 [cs.LG].

Kirby, R. (2010). "Social robot navigation." PhD thesis. Pittsburgh, PA: Carnegie Mellon University.

LaValle, S. (2006). *Planning algorithms.*

LaValle, S. M. (1998). *Rapidly-exploring random trees: a new tool for path planning.* Tech. rep.

Lefevre, S., D. Vasquez, and C. Laugier (July 2014). "A survey on motion prediction and risk assessment for intelligent vehicles." In: *Robomech journal* 1. DOI: 10.1186/s40648-014-0001-z.

Mazlov, I. (2019). *Websocketserver c#.* URL: https://github.com/wi1k1n/WebSocketServer.

Paden, B., M. Cap, S. Z. Yong, D. Yershov, and E. Frazzoli (2016). "A survey of motion planning and control techniques for self-driving urban vehicles." In: arXiv: 1604.07446 [cs.RO].

Quigley, M., B. Gerkey, K. Conley, J. Faust, T. Foote, J. Leibs, E. Berger, R. Wheeler, and A. Ng (2009). "Ros: an open-source robot operating system." In:

Robotics 2020. Multi-Annual Roadmap for Robotics in Europe. Horizon 2020 (2015).

Rudenko, A., L. Palmieri, M. Herman, K. M. Kitani, D. M. Gavrila, and K. O. Arras (2020). "Human motion trajectory prediction: a survey." In: *The international journal of robotics research* 39.8, pp. 895–935. ISSN: 1741-3176. DOI: 10.1177/0278364920917446. URL: http://dx.doi.org/10.1177/0278364920917446.

Scheel, O., L. Schwarz, N. Navab, and F. Tombari (2018). *Situation assessment for planning lane changes: combining recurrent models and prediction.* arXiv: 1805.06776 [cs.CV].

Tang, Y. C. and R. Salakhutdinov (2019). "Multiple futures prediction." In: arXiv: 1911.00997 [cs.LG].

Tsardoulias, E. and P. Mitkas (2017). *Robotic frameworks, architectures and middleware comparison.* arXiv: 1711.06842 [cs.RO].

Ulbrich, S. and M. Maurer (Sept. 2015). "Towards tactical lane change behavior planning for automated vehicles." In: pp. 989–995. DOI: 10.1109/ITSC.2015.165.

Xu, G., L. Liu, Y. Ou, and Z. Song (Sept. 2012). "Dynamic modeling of driver control strategy of lane-change behavior and trajectory planning for collision prediction." In: *Intelligent transportation systems, ieee transactions on* 13, pp. 1138–1155. DOI: 10.1109/TITS.2012.2187447.

Ziebart, B., N. Ratliff, G. Gallagher, C. Mertz, K. Peterson, J. Bagnell, M. Hebert, A. Dey, and S. Srinivasa (Dec. 2009). "Planning-based prediction for pedestrians." In: pp. 3931–3936. DOI: 10.1109/IROS.2009.5354147.