Hochschule für Angewandte Wissenschaften Hamburg
Hamburg University of Applied Sciences

Bachelorthesis

Marco Casagrande

Robust setup of an Autonomous Mobile Robot research platform with multi-sensor integration in ROS

Fakultät Technik und Informatik
Department Informations- und Elektrotechnik

Faculty of Engineering and Computer Science
Department of Information and Electrical Engineering

Marco Casagrande

Robust setup of an Autonomous Mobile Robot research platform with multi-sensor integration in ROS

Bachelorthesisbased on the study regulations
for the Bachelor of Engineering degree programme
Information Engineering
at the Department of Information and Electrical Engineering
of the Faculty of Engineering and Computer Science
of the Hamburg University of Aplied Sciences

Supervising examiner : Prof. Dr.Ing. Stephan Pareigis
Second Examiner : Prof. Dr.Ing. Ulrich Sauvagerd

Day of delivery 12. April 2018

Marco Casagrande

Title of the Bachelorthesis

Robust setup of an Autonomous Mobile Robot research platform with multi-sensor integration in ROS

Keywords

Autonomous Mobile Robot, ROS, Autonomous Navigation, Mapping, Simulation, SLAM, LiDAR, Odometry

Abstract

The paper describes the implementation of an Autonomous Mobile Robot able to navigate the environment by combining range and odometry data from LiDAR and wheel encoders sensors in the Robot Operating System (ROS) framework. The SLAM algorithm uses this sensory information to produce a static map of the environment. This is then relied upon by the navigation stack of the framework to navigate the environment, where the sensor data is used to localize the robot in the map and to calculate an optimal trajectory towards a set destination that avoids static and dynamic obstacles. The system is tested in simulated and real scenarios and the main challenges of mapping and navigation are surveyed. The different approaches are then discussed with a particular focus on their robustness, by studying their shortcomings and advantages. This paper ultimately aims to guide the reader through the steps needed to implement the described system and outlines the best practices that lead to a sound solution.

Marco Casagrande

Titel der Arbeit

Setup und Programmierung eines autonomen Roboters als Forschungsplattform mittels Integration von Sensoren in ROS

Stichworte

Autonomous Mobile Robot, ROS, Autonomous Navigation, Mapping, Simulation, SLAM, LiDAR, Odometry

Kurzzusammenfassung

Diese Thesis beschreibt die Programmierung eines Autonomen Mobilen Roboters, welcher durch Auswertung von Entfernungs und Odometriedaten eines LiDAR sowie eines Rad-Encoder Sensors einen Pfad frei von mobilen und stationären Hindernissen berechnet. Der SLAM Algorithmus nutzt die Sensorinformationen um in echtzeit eine Umgebungskarte zu erstellen, auf welcher die Position des Roboters sowie der Hindernisse festgehalten werden. Das System ist durch Simulationen sowie reale Szenarien getestet worden, in welchen die Probleme und Komplikationen in der Navigation und Objekterkennung observiert wurden. Die verschiedenen Lösungsansätze und deren Vor- sowie Nachteile sind in dieser Thesis in besonderer Hinsicht auf ihre Stabilität gegenübergestellt. Ziel ist es dem Leser die Schritte zur Implementierung dieses Roboters aufzuzeigen, sowie die Dokumentation von Herangehensweisen welche die besten Resultate ergeben haben.

Acknowledgment

My gratitude goes to my mother and brother, for inspiring me to find my own way and supporting me all throughout this journey, and to my friends, for getting me through good and difficult times and for bearing with me all this while. You are my strength.

My respect and appreciation are for Professor Stephan Pareigis and Professor Ulrich Sauvagerd, to whom I owe what I was able to learn and achieve, and everyone that made an engineer out of me.

Contents

List of Tables

List of Figures

1 Introduction

At a time when robotics is increasingly present in our everyday environment and autonomous navigation is revolutionizing entire infrastructures, from urban mobility to industrial scenarios, it is of fundamental importance to gain a deeper understanding on how to implement a robust robotic system. Although the know-how and the technology to develop such systems have been the subject of numerous studies and researches, the complexity of the topic so far required extensive knowledge in fields such as Mechatronics, Electrical Engineering and Computer Science in order to develop the most rudimentary robot with features not remotely comparable to those of current systems. Moreover, the prohibitive cost of hardware and software alike did not allow developers to experiment outside of specifically equipped laboratories, limiting the user base and consequentially the rate of technological innovation.

The availability of robotics software frameworks is increased in recent years and, in a study highlighting the different characteristics of the most established solutions, ROS (Robot Operating System) proved to be among the open source frameworks with a compelling score in terms of high-level abstraction of complex routines, documentation and re-usability. [1] Its extensive pool of features and drivers provides the means necessary to control the numerous actuators and joints of a robot and visualize the data acquired through different types of sensors. Furthermore, complex algorithms used to interface a robot with its surroundings have been incorporated in ready-to-use navigation and manipulation stacks. All the while, configurable simulation tools made it possible to model complex robots and sensors and mimic their behaviour in ad-hoc created virtual environments to decrease the development time and cost, before seamlessly switching to the real scenario. [2] This allows users with a fundamental background in software development to deploy fairly complex systems, even without having access to expensive hardware.

An Autonomous Mobile Robot (AMR) is a fitting example, as it encompasses a drive system with multiple independent moving parts and inputs from different sensors. While different configurations are possible, a combination of LiDAR laser range and odometry data to effectively scan the surrounding environment and accurately localize the robot in it was successfully implemented to generate a 2-D map of an indoor environment and achieve autonomous navigation capabilities. [3][4] Furthermore, visual navigation with the aid of a camera has been proven effective in reacting to moving obstacles, common in a dynamic scenario, by accounting for their velocity when calculating the optimal trajectory. [5]

The core component of the system is however the software that interprets the inputs, creates a map of the surroundings, estimates the position of the robot and dynamically calculates the optimal route toward a defined destination. Despite the overwhelming complexity of the underlying components, by relying on robotic frameworks it is now feasible to implement this platform in a relatively short amount of time. The implementation and deployment of this system are therefore the focus of this project, as it is an excellent mean to explore different tools and methodologies available at the moment of this writing.

This paper outlines the development of an Autonomous Mobile Robot for indoor navigation using the ROS framework. It also describes best practices for developing the system with high degrees of robustness, efficiency and scalability, as well as the limitations and drawbacks of the approaches, while aiming at minimizing developments costs and time. The chapter *Related Work* highlights the state of the art of the research in the field, which may offer the reader some interesting development ideas. A more detailed description of the project objectives is given in the *Requirements* chapter, where the milestones to reach them are listed. Particular care is taken to outline each implementation step in the *Hardware and Software integration* chapter, giving a brief introduction on the fundamentals of ROS and detailing some technical aspects where needed, in order to give the reader the insight that could be helpful when deploying a similar system or extending the functionalities of the described platform. The *Approaches assessment* chapter lists different implementation choices, identifies the parameters that have been used for their evaluation and argues on the reasons why one approach was deemed the most suitable in the scope of the project. The *Evaluation* chapter details how the implemented functionalities were evaluated and describes measures that can be put in place to improve the performance. Moreover, the chapter lists the issues that remained open and may need to be addressed in further developments. Finally the *Consclusion* chapter summarizes the outcomes of the project, which are most relevant to the robustness of the AMR.

2 Related work

In order to frame the discussion on the current state of the art, it is a good idea to state the mission of the system, more thoroughly explained in the chapter 3. The robot shall be able to navigate in a dynamic indoor environment, namely inside the University building. A LiDAR sensor provides range data on the environment while rotary encoders track angular velocity of the wheels and provide odometry data. In order for the navigation feature to be implemented efficiently, a map of the environment shall be built.

In a paper published by M. Köseoğlu, O. M. Çelik and Ö. Pektaş is explained how an Autonomous Mobile Robot can be implemented. The system integrates a LiDAR, quadrature encoders, an inertial measurement unit (IMU) and an ultrasonic range finder. The latter is used in a collision avoidance algorithm, while the IMU is used to increase the reliability of the odometry data obtained from the encoders. In order to fuse the data obtained from the quadrature encoder and the IMU, an Extended Kalman Filter is used. Also known as linear quadratic estimation, it aims to estimate the current state of a system, which in this case is represented by its position, based on the previous states and current observation. An accurate estimate of the location of the device helps improving the overall precision in both mapping and navigation. [3]

In another publication, Q. Xu, J. Zhao, C. Zhang and F. He describe the implementation of a similar system and offer an insight on two possible mapping algorithms available in ROS. While both *gmapping* and *hector_slam* are mainly used in dealing with the SLAM problem and rely on laser scan data, the former additionally requires odometry data to locate the robot in the map. Instead, *hector_slam* estimates its location based solely on observations of its surroundings and is proven to be less effective in scenarios with scarcity of landmarks, like long corridors or large empty spaces. [4]

On the other hand, *gmapping* is an effective way to tackle the SLAM problem, according to A. Huletski and D. Kartashov. In their paper the algorithm is explained in depth, arguing that its prediction-correction loop offers reliable performance in a wide array of scenarios. In each loop iteration the measurements provided by an interoceptive sensor (e.g. odometry from wheel encoders) are refined by matching data obtained with an exteroceptive sensor (e.g. a camera or laser scanner) and already accumulated information about the environment. [6]

Autonomous navigation is also achievable using visual data from camera sensors and this approach is outlined by F. Chaumette. The authors state that visual navigation can provide more robust performance in a dynamic environment with moving obstacles. This is a typical scenario where the robot operates in a space crowded with people. An image processing algorithm tracks the obstacles in the field of view of the camera and calculates their speed and trajectories. The robot computes viable paths or *tentacles* based on the estimated future position of the obstacles, evaluates their cost with respect to efficiency and risk of colliding and travels the one scoring lowest. [5]

Further research in the field of robot-human interaction has been done by A. T. Angonese and P. F. Ferreira Rosa by implementing a Deep Convolutional Neural Network to be able to identify different persons based on the image stream of a RGBD camera. An application relying on people recognition is the so-called *follower* use-case, where the robot follows a specific person while keeping a set distance. A further integration with a SLAM algorithm allows overlaying the informations on the detected persons on the map built with the latter. [7]

3 Requirements

Two are the main objectives of the Autonomous Mobile Robot identified and described in this paper. First, it shall be able to produce an accurate static map of an indoor environment. The measurement error of the dimensions represented in the map, as well as the incorrect curvature of sections of it is assessed. Section 6.1 outlines the evaluation of the mapping feature.

Next, the robot shall safely navigate the environment towards a set destination. In order to achieve this, the robot shall avoid obstacles that were not previously mapped, recalculate a new feasible trajectory if an obstruction blocks the original path and reach the goal, stopping within a minimal distance from it. In Section 6.2 these features and their success rates are evaluated.

In order to progress systematically towards the end goals, several milestones listed in Table 3.1 were identified. It is worth noting that it would be feasible to implement the AMR neglecting the simulation step. This step however offers the possibility of evaluating the behaviour of each singular component as well as the soundness of the routines of the robot more quickly and with less effort than it would require when doing the same in a real scenario, while avoiding any accidental damage to the hardware.

Milestone	**Description**
Development environment	The development environment is configured
Sensor integration	The LiDAR, wheel encoders and the camera are configured in the real and simulated robot
Remote controller	A device is configured to remotely control the robot
Mapping	The robot produces a static map of the real and simulated environments
Navigation	The robot navigates autonomously towards a set destination in the real and simulated environments

Table 3.1: Requirements - Milestones overview

4 Hardware and Software integration

This chapter outlines the development steps to interface the *Pioneer P3-DX* and the different sensors with ROS and simulate their behaviour in *Gazebo*, ROS-compatible simulation engine. First, a brief introduction is given on the fundamentals of ROS in order to ease the reader in the framework and the vast subject of Autonomous Systems. An IDE is then introduced, which may be used to simplify the development in ROS. Next, from the configuration of the bare-boned robot, each sensor is subsequently integrated both in the hardware and in the simulation configuration. Finally, the software subsystem that relies on the integration of the different components is described. More thorough explanations are given when required to guide the reader in the implementation of the system without an extensive prior knowledge of ROS. It is however outside the scope of this paper to explain its underlying structure components, knowledge essential in order to gain a better understanding on the operation of the framework and for which there is abundant material online. The reader is therefore encouraged to refer to the official documentation for in-depth information as new concepts are introduced throughout the development of the system and to the resources listed in the respective sections.

4.1 Foundations of Autonomous Systems and ROS

This paper is meant to give a reader with little to no experience in Autonomous Systems a broad overview on how to implement an Autonomous Mobile Robot using ROS. It is therefore unavoidable to refer to hardware related concepts and components which are specific to the framework, which may be incomprehensible without a further look into the documentation. This section gives an overview of these basic concepts in order to improve the readability of this paper.

The ROS framework operates by initiating independent processes called *nodes* that, not unlike threads, execute in parallel and communicate with one another by subscribing to or publishing data structured in *messages* at a certain frequency via defined interfaces called *topics* in a polling approach. *Nodes* may also provide so-called *services*, procedures that can be requested remotely from other *nodes*. A defined set of *nodes* can be executed by configuring a *launch file*, without the need to initiate each one manually. The *messages*

published on the *topics* can be recorded in *bag files* in order to replay the data at later times.

Different tools are integrated in the ROS framework, most notably the *Gazebo* simulation environment and the *rviz* visualization tool. *Gazebo* allows to evaluate the performance of the system in a virtual scenario, drastically reducing development costs and time. *rviz* provides the means to graphically visualize output from sensors and is useful to see what the robot perceives.

SLAM or Simultaneous Localization and Mapping is a technique used to produce a static map of the environment using input from a variety of sensors. The *Navigation Stack* on the other hand encompasses the systems needed to calculate a viable trajectory to a set destination, such as *local* and *global planners* for short and long distance plans respectively.

Robust SLAM and navigation approaches use a combination of range data from a laser scanner or LiDAR and odometry data from rotary encoders. A 2-D or 3-D LiDAR sensor emits one or multiple laser beams respectively to probe the distance to the obstacles in the environment, while the odometry data represents the change in position of the robot over time and is obtained via rotary encoders tracking the angular velocity of the wheels of the robot. The same data is used by navigation algorithms as well, which allow the robot to drive towards a set destination on the map while avoiding static and dynamic obstacles alike. Listing 4.1 and 4.2 show the message definition for the 2-D and 3-D LiDAR data while Listing 4.3 represents the odometry data message.

The size of the messages can vary from sensor to sensor, depending on the number of points probed by the laser scanner for example, while the frequency at which the messages are published can be set manually according to the need. The *rostopic bw* command allows to monitor the bandwidth occupied by a specific topic the data is published to in any given circumstance.

```
# # # sensor_msgs/LaserScan Raw Message Definition # # #
# Single scan from a planar laser range-finder
#
# If you have another ranging device with different behavior
# (e.g. a sonar array), please find or create a different message,
# since applications will make fairly laser-specific assumptions
# about this data

Header header            # timestamp in the header is the
# acquisition time of the first ray in the scan.
#
# in frame frame_id, angles are measured around
# the positive Z axis (counterclockwise, if Z is up)
# with zero angle being forward along the x axis
```

```
float32 angle_min        # start angle of the scan [rad]
float32 angle_max        # end angle of the scan [rad]
float32 angle_increment  # angular distance between measurements [rad]

float32 time_increment   # time between measurements [seconds];
# if your scanner is moving, this will be used in interpolating
# position of 3d points
float32 scan_time        # time between scans [seconds]

float32 range_min        # minimum range value [m]
float32 range_max        # maximum range value [m]

float32[] ranges         # range data [m]
# (Note: values < range_min or > range_max should be discarded)
float32[] intensities    # intensity data [device-specific units].
# If your device does not provide intensities, please leave
# the array empty.
```

Listing 4.1: LaserScan - Raw Message Definition

```
# # # sensor_msgs/PointCloud2 Raw Message Definition # # #
# This message holds a collection of N-dimensional points, which may
# contain additional information such as normals, intensity, etc. The
# point data is stored as a binary blob, its layout described by the
# contents of the "fields" array.

# The point cloud data may be organized 2d (image-like) or 1d
# (unordered). Point clouds organized as 2d images may be produced by
# camera depth sensors such as stereo or time-of-flight.

# Time of sensor data acquisition, and the coordinate frame ID (for 3d
# points).
Header header

# 2D structure of the point cloud. If the cloud is unordered, height is
# 1 and width is the length of the point cloud.
uint32 height
uint32 width

# Describes the channels and their layout in the binary data blob.
PointField[] fields

bool    is_bigendian # Is this data bigendian?
uint32  point_step   # Length of a point in bytes
```

```
uint32 row_step     # Length of a row in bytes
uint8[] data        # Actual point data, size is (row_step*height)

bool is_dense       # True if there are no invalid points
```

Listing 4.2: PointCloud2 - Raw Message Definition

```
### nav_msgs/Odometry Raw Message Definition ###
# This represents an estimate of a position and velocity in free space.
# The pose in this message should be specified in the coordinate frame
# given by header.frame_id.
# The twist in this message should be specified in the coordinate frame
# given by the child_frame_id
Header header
string child_frame_id
geometry_msgs/PoseWithCovariance pose
geometry_msgs/TwistWithCovariance twist
```

Listing 4.3: Odometry - Raw Message Definition

Finally, the following brief glossary can be referred to while progressing through the following sections of the paper:

- *LiDAR*: Also called laser scanner, it measures distance to obstacles by emitting a laser beam and measuring the time of flight in which the beam is reflected, before rotating to scan at different angles.
- *Odometry*: Estimate of changes in position of the robot obtained by tracking the angular velocity of the wheels with rotary or wheel encoders.
- *Node*: A node is a process that performs computation. Nodes are combined together into a graph and communicate with one another using streaming topics. A robot control system will usually comprise of many nodes. For example, one node controls a laser range-finder, another the robot's wheels motors, one performs localization, one path planning, one provides a graphical view of the system and so on.
- *launch file*: Allows to initiate multiple nodes at once and to set and retrieve runtime parameters.
- *Message*: ROS uses a simplified description language for describing data that nodes publish or subscribe to. Messages define the data structure and act as data holders as nodes exchange messages through topics.
- *Topic*: Named channel over which nodes exchange messages. Topics use anonymous publish and subscribe semantics. In general, nodes are not aware of who they are communicating with. Instead, nodes that are interested in data subscribe to the

relevant topic, while nodes that generate data publish to the relevant topic. There can be multiple publishers and subscribers to a topic.

- *bag files*: A bag is a file format in ROS for storing message data. A variety of tools have been developed to allow storing, processing, analysing, and visualizing recorded data. Bags are typically created by the *rosbag* tool, which subscribe to one or more ROS topics and stores the serialized message data in a file as it is received, which can be played back at later times.
- *Gazebo*: 3-D simulator environment for ROS. With this tool it is possible to create a 3-D scenario with robots, obstacles and many other objects. *Gazebo* also uses a physical engine for illumination, gravity and inertia. The robot can be tested in difficult or dangerous scenarios without any harm. In most cases it is faster to run a simulation instead of performing the test in the real environment.
- *rviz*: A 3-D visualizer for the ROS framework. It is capable of visually representing data streams from different sensor sources, data from the mapping and navigation stack as well as robot models described in URDF files.
- *URDF*: The Universal Robotic Description Format is an XML file format used in ROS to describe all elements of a robot, including physical and inertial properties and joints between different robot components (or frames), used to calculate the transform between different frame coordinate systems.
- *xacro*: XML Macros are the alternative mean to the URDF to describe the elements of the robot, improving readability and reducing duplication in the robot description files.
- *gmapping*: A SLAM mapping algorithm using Rao-Blackwellized particle filters that relies on a combination of odometry and LiDAR data.
- *hector_slam*: A SLAM mapping algorithm that relies on LiDAR data and uses a scan matching technique to perform localization and mapping.
- *Navigation stack*: A set of software components that takes in information from odometry, sensor streams and a destination, sending velocity commands to a mobile base to reach it. As a pre-requisite to use the navigation stack, the robot must be running ROS, have a transform (or tf in ROS) tree in place, and publish sensor data using the correct message types.

4.2 Environment setup

The latest LTS Ubuntu distribution at the time of this writing is version 16.04.3 and is the recommended since it is compatible with ROS Kinetic, also the latest LTS framework distribution.

It is strongly recommended to install the 64-bit version on the hard drive of a performing machine with a dedicated GPU or, if another operating system is already installed, in one of its partitions. This is due to the fact that many tools necessary to simulate and visualize the data are demanding in terms of computation power. On the other hand, while developing on a Live USB version of Ubuntu is not recommended due to extreme drops in performance, installing the system on a virtual machine satisfying the requirements is a viable option.

The distribution of ROS used in the scope of the project is Kinetic Kame, specifically the Desktop-Full package. The installation process is straightforward and explained in detail in the official documentation. This would be sufficient in order to start developing with the framework, by manually navigating through the file-system in order to modify the project files and running the ROS commands on the console. Developing in an IDE might however be preferable to speed up the process and help keeping track of projects, regardless of their complexity. An extended section on the official ROS documentation lists numerous Integrated Development Environments that work with the framework.

The approach chosen for this project is the use of *RoboWare Studio*, mostly because of its straightforward integration with ROS and for the tools it offers. Here is a short list of features that the IDE provides:

- Code completion
- Syntax highlighting
- Diagnostic tools integration
- C++ and Python debugger
- Git built-in

In the following chapters of this paper there will be no further mention of *RoboWare Studio*, since the choice of the IDE is a subjective matter and not essential to the actual implementation of the system. The goal of this small introduction was rather to present the reader with the possibility of using this particular software to setup a working environment.

4.3 Pioneer P3-DX

The robot of choice for this project is the *Pioneer P3-DX* from *Omron Adept Mobile Robots*. It implements a differential drive system with rotary encoders to track the angular velocity of the wheels for reliable odometry data and an on-board microcontroller to provide an interface with the robot components. Additionally, a SONAR system is integrated on the front of the chassis. Its main advantages are the proven compatibility with ROS and the availability of numerous resources that accelerate the development on the platform. These include

ROS drivers, tutorials and URDF (Unified Robot Description Format) models describing the geometry and joints of the robot.

These are necessary to compute the transforms between the coordinate frames of the robot components to determine the relative position or pose of these frames with respect to one another. Figure 4.1 shows graphically the relationships between the different component frames of the robot and their common base link, described in the URDF model. More information on this topic is available in the official transform *tf* package documentation. [8]

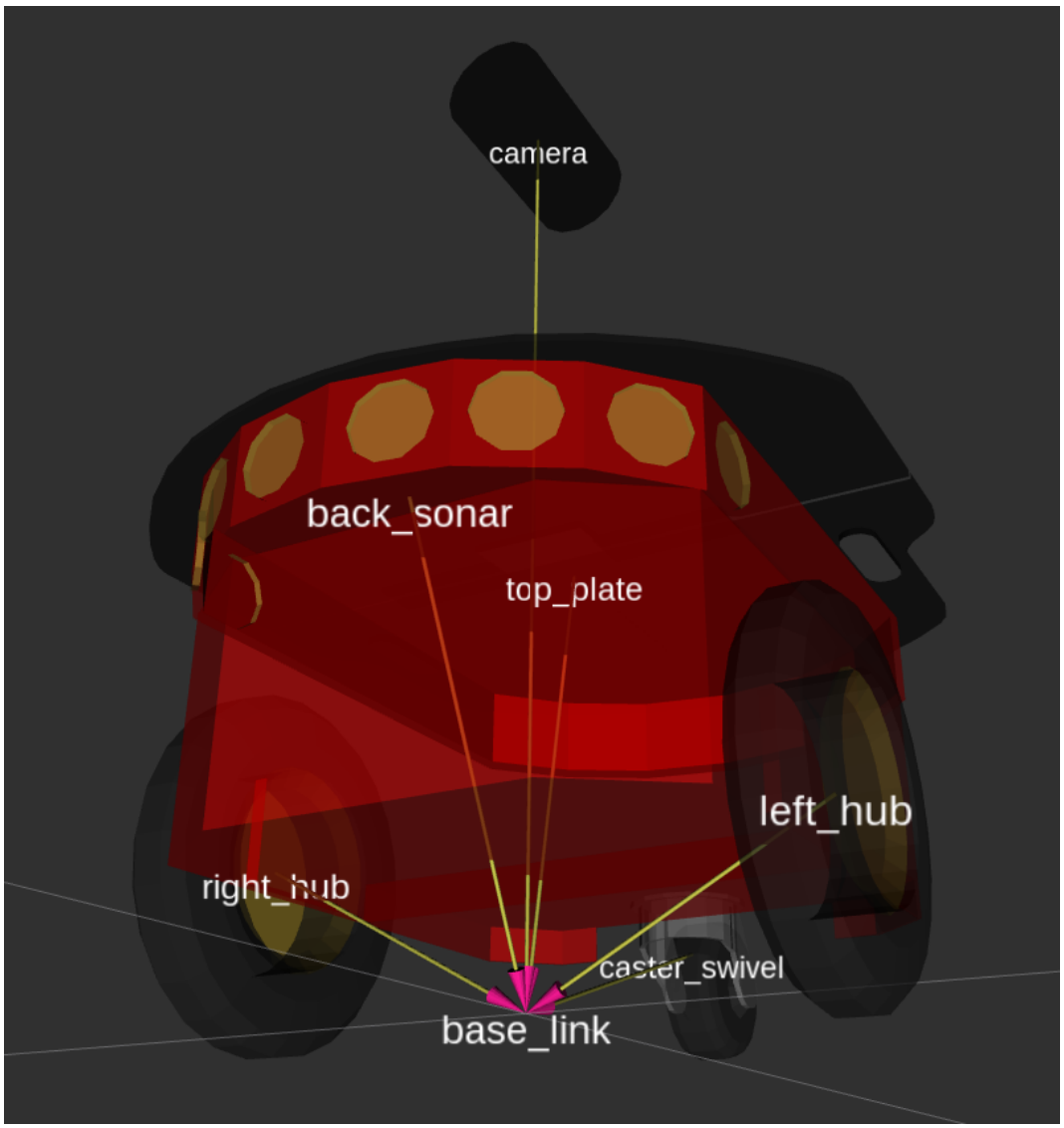

Figure 4.1: Robot - Pioneer P3-DX transform relations between the components

4.3.1 Simulation

According to the official MobileRobots ROS package, the latter does not provide working URDF models for *Gazebo*. [9] Luckily, ROS Kinetic comes bundled with demo packages which include the models needed. The complete and accurate instructions are described in the two-parts guide listed. [10][11] Note that these models work only in the simulation environment and are not suited for the operation of the actual robot.

4.3.2 Hardware

To deploy the robot in the real environment, the drivers for the microcontroller, the ROS interface and the robot description URDF files need to be configured. This requires the *ARIA* SDK (Advanced Robot Interface for Applications) to be installed and the *ROSARIA* package to be put in the workspace, as thoroughly explained in the official ROS documentation. [12] This package includes the launch files that initiate the communication between ROS and the on-board microcontroller of the robot.

Next the *amr-ros-config* package needs to be downloaded and placed in the ROS workspace. [9] This package includes the official URDF models of the robot, as well as launch files that start the *rviz* tool to visualize the model, as shown in Figure 4.1. *rviz* is ROS graphical tool that allow to visualize data from a wide range of sensors and is an essential diagnostic tool as it allows to interpret the robot perception. It may also be used to interact with the navigation stack to set specific destinations on a map where the robot should travel to.

4.3.3 Odometry calibration

At this point the robot should be configured and it should be possible to control its movements with ROS. A noteworthy issue is however that of odometry calibration. It may very well be that, due to different pressure of the tires and load on the chassis, the wheel encoders map their rotation to the wrong distance travelled, which may degrade the performance of both mapping and navigation operations. In order to verify whether this is the case, the following tests need to be performed:

1. Drive the robot forward and verify it keeps following a straight line.
2. Measure the actual distance travelled and compare to the one reported.
3. Rotate the robot in place, measure the angle and compare to the one reported.

One way to do so is using a teleoperation *teleop* tool in ROS to command the robot with the keyboard of the laptop and monitor the odometry data published. Both *rqt_topic* and *rviz* tools can help reading and visualizing the data respectively. If during any of the previous tests an unacceptable error is observed, the odometry is not calibrated and the respective parameter needs to be adjusted. Following are the *Pioneer P3-DX* robot-specific parameters:

1. *DriftFactor*: measures the horizontal error vs. forward motion and is used to correct the translation and rotation drift; should be adjusted until the robot drives in a straight line.
2. *TicksMM*: is the number of encoder ticks per millimeter; should be adjusted until the travelled distance matches the one reported.
3. *RevCount*: is the differential number of encoder ticks for a 180-degree rotation of the robot; should be adjusted until the rotation matches the one reported.

It is recommended to do so dynamically while the robot is operating, using the *rqt_reconfigure* tool, to avoid restarting the robot every time a parameter is modified. Once the parameters have been properly adjusted, it is possible to save them in an appropriate configuration file that can be then loaded when starting the robot.

The calibration method above relies on visual observations and manual adjustments of the values until the perceived error is negligible. There are other approaches to perform the calibration, and a fairly precise one that relies on laser scan readings is explained in the listed resource. [13]

4.4 LiDAR

Range data is crucial in the implementation of an AMR with mapping and navigation capabilities. In this project, a *SICK TiM310 S01* LiDAR is used as it reliably provides this data in an indoor environment. Its field of view has a 270° horizontal aperture angle with an angular resolution of 1°, and a maximum range of 4m. Navigation and mapping algorithms rely on readings from a laser beam oriented parallel to the ground to keep track of the position of the robot in the map. This however does not allow to detect objects higher or lower than the emitted laser beam. A solution to overcome this problem is to use a 3-D LiDAR, such as the *Velodyne VLP-16*, which emits more beams at different angles at the same time, or mount a 2-D LiDAR on a tilting joint controlled by a servo that adjust the angle. In the scope of this project, the 2-D laser scanner is placed in front of the chassis and can only detect obstacles at a height of approximately 15cm.

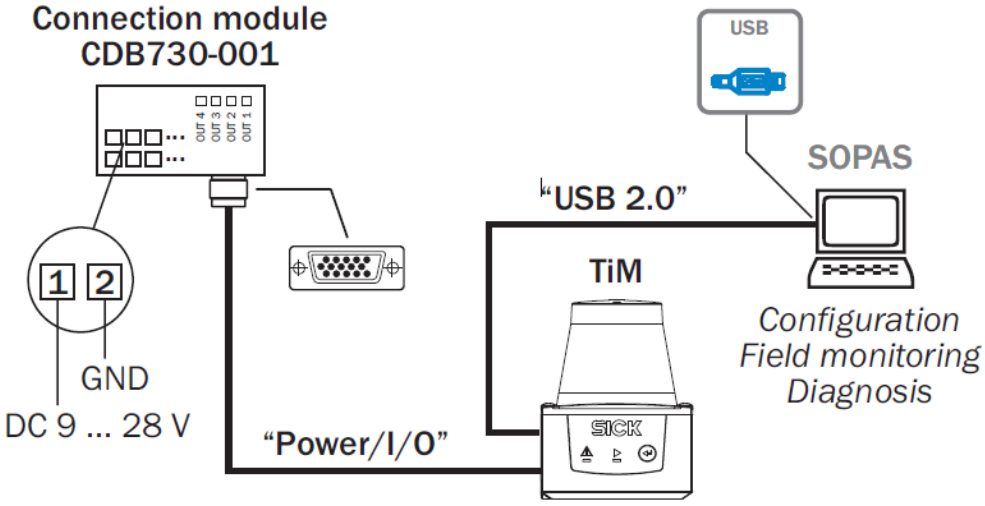

Figure 4.2: LiDAR - TiM310 Connection Diagram

source: TiM31X Operating instructions manual

As shown in Figure 4.2, the device is connected via a USB interface to the computer, where in this case ROS instead of the proprietary software SOPAS is installed, and with a 15-pin D-sub-HD plug to the supplementary Connection module *CDB730-001*, to which the power is supplied. No further detail on the module above is relevant to the scope of this project, however more on the topic can be found in the listed technical documentation. [14]

The next step is to download the device driver for ROS. The official ROS wiki has a page dedicated to drivers for a variety of sensors and the package *sick_tim* includes the one for the *SICK TiM310 S01* model as well the URDF models and the files needed to simulate the sensor in *Gazebo*. [15] After cloning the repository in a ROS workspace, the instructions to access the device via the USB interface are provided in the */udev/README* file. The package comes with ROS launch files for each of the supported sensors, which create a ROS node interfaced with the sensor that publishes *LaserScan* messages. These contain notably the array of ranges or distances to the obstacles detected and the respective intensities.

Rviz can be used to visualize a graphical representation the data by adding the *LaserScan* display and setting the topic. By adding the *RobotDescription* display, it is possible to see the model of the LiDAR as well. To test the reliability of the data and study the behaviour of the device, the following test scenario was arranged so that the following elements were located in the field of vision in front and at the sides of the sensor:

- Square object with one side perpendicular to the beams.

- Square object with one edge facing the beams.
- Spheric object.
- Plane with an inclination of approx. 30° and a peak height of 7cm.
- Pole with a square base behind the inclined plane.

The simplified setup could be observed in Figure 4.3, where only the meaningful elements were extracted from the surroundings. The height of the inclined plane was chosen so to test the behaviour of the device when objects are present in its field of vision which are barely interrupting the laser beam. Anything below this height is not detected by the sensor. The *rviz* graphical representation of the laser scan data is shown in Figure 4.3, where the dark blue points represent the detections with a high reflection intensity. Figure 4.4 shows the overlay of the test scenario and the sensor output.

It appears that, in the case of the inclined plane, the sensor noise is moderately higher as the points are not perfectly aligned and with a low intensity value due to a non-optimal reflection angle between the laser beam and the plane surface. Moreover, the distortion is greater at the point where the pole is located behind the plane. The presence of an object behind a barely detectable one seem to cause this odd behaviour. This is nonetheless a very specific corner case and therefore is not further investigated.

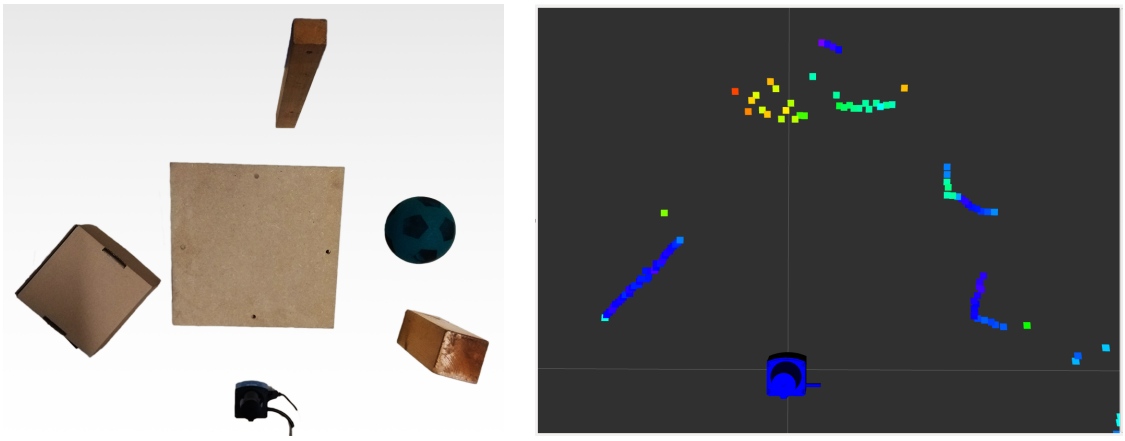

Figure 4.3: LiDAR - Test setup (left) and output visualization (right)

4.4.1 Simulation

Assuming the instructions outlined in Section 4.3.1 were followed, it should be possible to simulate the robot, which integrates a simulated camera and a laser scanner. This would

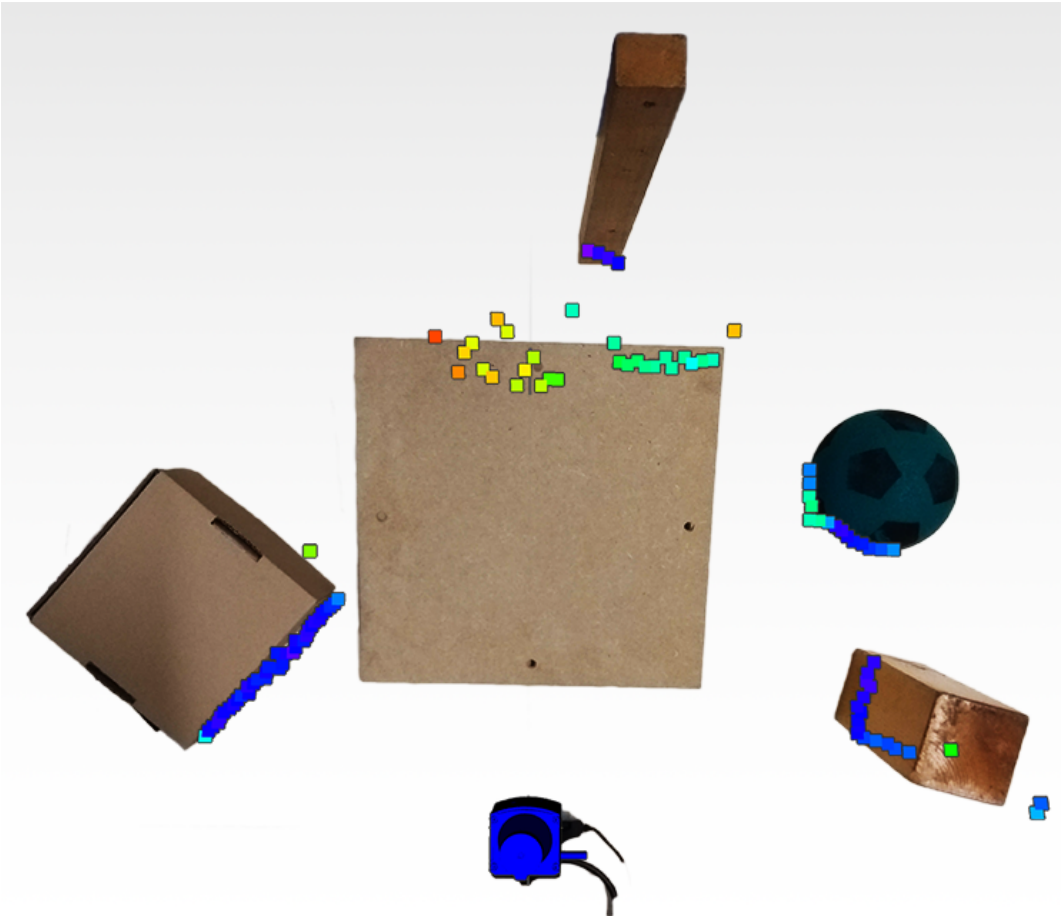

Figure 4.4: LiDAR - Overlay of laser scan data with real scenario

be sufficient to start testing the mapping and navigation algorithm, however it is desirable to replace the laser scanner with the *SICK TiM* LiDAR used in this project to rely on simulation data closer to the real life scenario. To do so, the *sick_tim* package mentioned in Section 4.4 needs to be included in the workspace. Next, the model of the robot needs to be modified to include the sensor.

While it is outside the scope of this paper to explain in detail how to create and edit the model of a robot, a brief explanation of the subject is in order. To describe the geometric and physical properties of the different parts of a robot, ROS makes use of either Xacro (or XML Macros) or URDF files. The main benefit of using Xacro instead of URDF is the possibility to import defined component macros simplifying the description of the robot, reducing the complexity and improving the maintainability and readability of the files. In the code, also included in the Appendix, the macro of the *SICK TiM* laser scanner is loaded and a joint is defined to connect the laser to the front of the chassis of the robot. By launching the simulation, the robot in *Gazebo* should appear as shown in Figure 4.5.

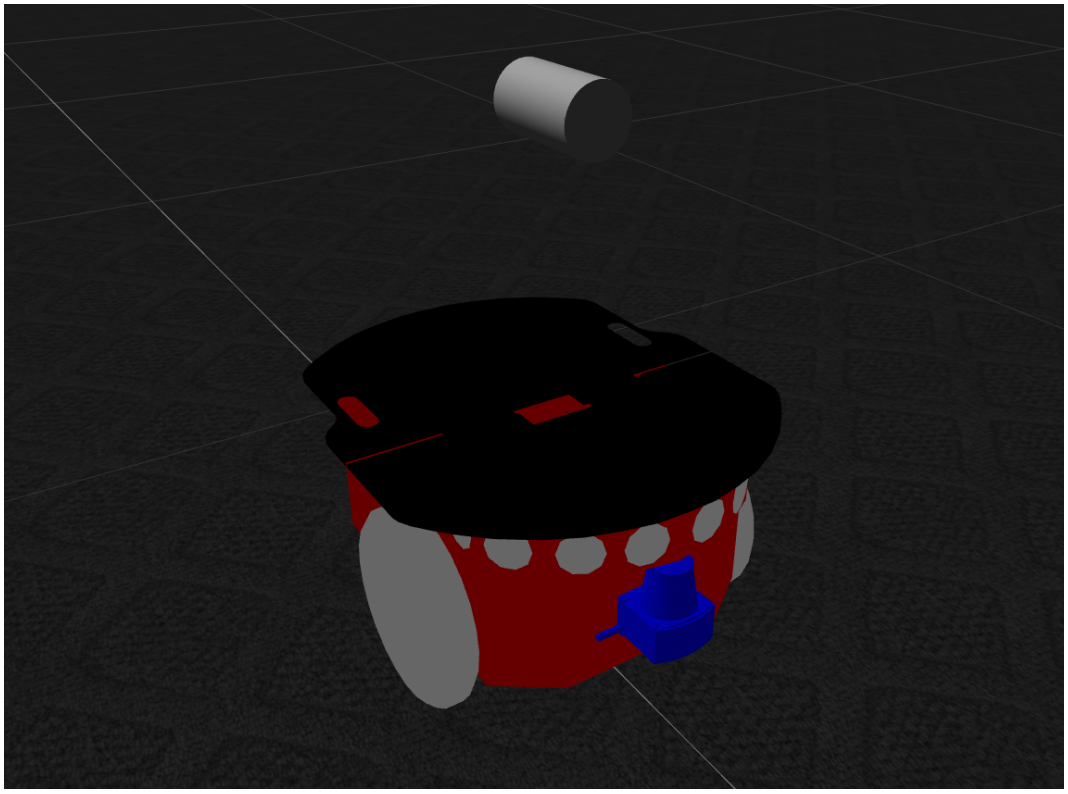

Figure 4.5: LiDAR - Simulation with SICK TiM laser mounted in front and camera on top

The beams of the sensor can also be made visible in *Gazebo* by editing the respective line in the xacro model. A cube, a cylinder and a sphere are all placed in the field of view of the sensor and the reading are visualized in *Gazebo*, as shown in Figure 4.6. This proves that the sensor is simulated correctly and the data is reliable.

4.4.2 Hardware

The LiDAR is mounted on a support on the front of the chassis, and the Connection module is connected to the 12V un-switched auxiliary power output. The data USB cable is then connected directly to the laptop where ROS is installed. The models require some additional configuration in order for the laser scanner to be added. First, the *sick_tim* and the *amr-ros-config* package need to be in the same workspace. Next, the Xacro file of the robot needs to be modified to include the laser scanner. Refer to the Appendix, where the models are included.

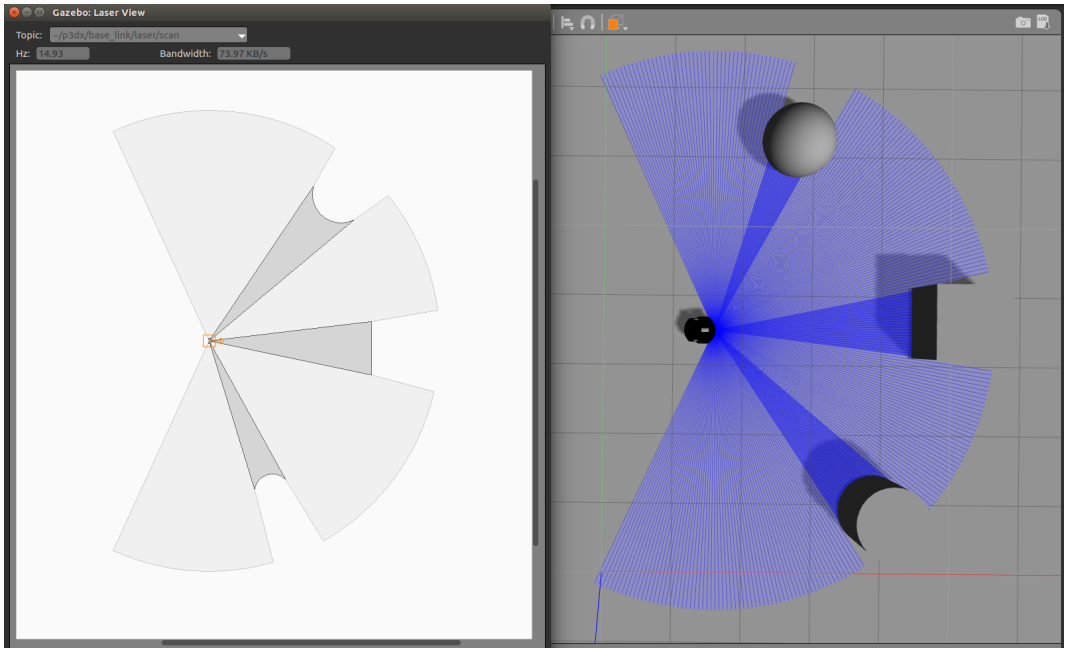

Figure 4.6: LiDAR - Simulation with SICK TiM laser mounted in front and camera on top

4.5 Camera

The camera sensor provides visual information on the surrounding environment and offers the potential that laser scanners lack to interpret image data. Moreover, ToF (Time of Flight) cameras are capable of providing additional 3-D range data on the environment in their field of view. In the scope of this project however a basic *Microsoft Lifecam* webcam is integrated to record the experiments in order to provide a visual feedback from the point of view of the robot.

4.5.1 Simulation

Assuming that the instructions listed in Section 4.3.1 have at this point been completed, a simulation model of the robot which includes a generic camera is already available. To test its operation, it is sufficient to spawn the robot in *Gazebo*, open *rviz* and add a *Camera* layer. A video stream from the simulated camera should appear visible in the graphical tool.

4.5.2 Hardware

In order to interface the camera with ROS and read its data with it, a ROS driver must be correctly configured and the sensor calibrated. The *cv_camera* OpenCV ROS driver provides a node that can process the camera data stream. Additional information on the package is given in the listed resource. [16] Assuming the device is correctly configured in Ubuntu and the driver is installed, by executing in the terminal the line in Listing 4.4 it is possible to test whether the device is interfaced with ROS.

```
rosrun cv_camera cv_camera_node
```

Listing 4.4: Bash command to start the cv_camera_node

This command should start the node streaming the data from the camera, even if no configuration has been already carried out. If this is not the case or if the wrong camera sensor is started, it may be that the *device_id* parameter needs to be set correctly. The graphical tool *rqt_image_view* should be able to visualize the video stream with the command in Listing 4.5.

```
rosrun rqt_image_view rqt_image_view
```

Listing 4.5: Bash command to start the rqt_image_view tool

Trying now to visualize the data in *rviz* or to implement image processing ROS applications before calibrating the camera would not work. In order to perform the calibration, the *cv_camera_node* must be publishing data in ROS. The procedure is detailed in the listed resource. [17] Finally, the calibration data is output in a compressed file that contains a *ost.txt* and a *ost.yaml* file. Next, a launch file to start the *cv_camera_node* and adjust the calibration is shown in Listing 4.6. At this point, the camera should be correctly configured and it should be possible to visualize the camera stream in *rviz*.

```
<launch>
  <node pkg="cv_camera" type="cv_camera_node" name="[node_name]" >
    <param name="device_id" type="int" value="[device_number]" />
    <param name="camera_info_url" type="string"
        value="[path_to_folder/ost.yaml]" />
  </node>
</launch>
```

Listing 4.6: cv_camera - Launch file example

As was the case for the LiDAR, the camera sensor needs to be placed on the model of the robot in the correct place in the xacro file. Refer to Appendix for the model.

4.6 Virtual Joystick

It is useful to be able to control the robot remotely in order to manually drive it as it collects and records data on the environment, which can then be fed to a SLAM algorithm to create a map of the environment, or to give the user the ability to take over control during autonomous operations. For this purpose, a virtual joystick application can be installed on an Android device that communicates with the ROS server running on the laptop controlling the robot.

4.6.1 Network

In order to enable the communication between the devices it is advisable to setup an WiFi hotspot interface on the laptop which the Android device can connect to. Although there may be different more robust ways to do so, the scope of this project does not entail particular concerns in terms of security or performance. The only constraints are given by the nature of the network policies in force, which are quite strict in the university infrastructure. An external WLAN-USB-Adapter is therefore used to provide a direct access point to the Android device. Setting up a hotspot in Ubuntu 16.04 does not present particular challenges. After the interface has been set up, the environment variable has to be set according to Listing 4.7 to allow ROS to communicate using that network interface.

```
export ROS_IP=[interface_IP_address]
```

Listing 4.7: Configure ROS to communicate via a network interface

4.6.2 Android libraries

At the moment of this writing, no working Android application for ROS Kinetic is available in the Google Play store. Source code developed for Android in collaboration with Google is however available. [18] The instructions are provided to build from the source and install the application with Android Studio on a device. Attention must be paid to download in Android Studio the correct API for the device of choice. The latter must be also properly configured: first the development settings must be enabled and the USB Debugging Mode must be activated.

4.6.3 Teleoperation

The *android_core* stack provides a collection of components and examples that are useful for developing ROS applications on Android. Among the different pre-packaged applications

in the stack, the *android_tutorial_teleop* offers a mean to remotely control the robot with an interface resembling that of an actual joystick. After connecting to the ROS master on the laptop, this application sends velocity messages composed of linear and angular velocities, that ROS interprets to move and rotate the robot. The official documentation thoroughly describes how to set up the *Android Studio* development environment, compile and install the applications in the stack. [18] There are however some additional steps to interface the virtual joystick with the robot.

After having installed the *android_tutorial_teleop* application on a *Moto G4* Android device, during the tests the application crashed regularly. The error seemed to occur when *LaserScan* data is published on the ROS server while the Android device is connected, when the amount of memory requested for allocation quickly exceeds the available size. A closer look at the code in the *src/org/ros/android/android_tutorial_teleop/MainActivity.java* file shows that many layers are initialized, which are unnecessary for a basic remote controller application. The code in Listing 4.8 was therefore edited to the code in Listing 4.9.

```
@Override
public void onCreate(Bundle savedInstanceState) {
  super.onCreate(savedInstanceState);
  setContentView(R.layout.main);
  virtualJoystickView = (VirtualJoystickView) findViewById(R.id.
      ↪ virtual_joystick);
  visualizationView =(VisualizationView)findViewById(R.id.visualization);
  visualizationView.getCamera().jumpToFrame("map");
  visualizationView.onCreate(Lists.<Layer>newArrayList(new
      ↪ CameraControlLayer(),
  new OccupancyGridLayer("map"), new PathLayer("move_base/NavfnROS/plan")
      ↪ , new PathLayer("move_base_dynamic/NavfnROS/plan"), new
      ↪ LaserScanLayer("base_scan"),
  new PoseSubscriberLayer("simple_waypoints_server/goal_pose"), new
      ↪ PosePublisherLayer(
  "simple_waypoints_server/goal_pose"), new RobotLayer("base_footprint"))
      ↪ );
}
```

Listing 4.8: Remote controller - Java code with unnecessary layers

```
@Override
public void onCreate(Bundle savedInstanceState) {
  super.onCreate(savedInstanceState);
  setContentView(R.layout.main);
  virtualJoystickView = (VirtualJoystickView) findViewById(R.id.
      ↪ virtual_joystick);
  visualizationView =(VisualizationView)findViewById(R.id.visualization);
  visualizationView.onCreate(Lists.<Layer>newArrayList());
}
```

Listing 4.9: Remote controller - Java code with no unnecessary layers

While the remote controller publishes messages on the */virtual_joystick/cmd_vel* topic, the robot is controlled by the ROSARIA library which subscribes to the */rosaria/cmd_vel* topic. One solution is to forward the messages from the joystick to the right topic. The *topic_tools/relay* node does exactly that. In order to launch the node automatically, the line in Listing 4.10 must be added in the launch file where it is desired to control the robot with the remote controller.

```
<node pkg="topic_tools" type="relay" name="[node_name]"
      args="/virtual_joystick/cmd_vel /rosaria/cmd_vel">
```

Listing 4.10: Remote controller - Relay node to forward the velocity commands from the remote controller to the robot

4.7 Mapping

An Autonomous Mobile Robot could be theoretically implemented to navigate an unknown environment, by setting a goal at a distance relative to the robot starting position. In this scenario, the robot would try and reach its destination by driving towards it in the shortest path and, as it finds an obstacle, guessing another trajectory to reach it. This trial-and-error approach is evidently not the most efficient when the geography of the environment does not change in time. To optimize the navigation of the robot in such a scenario it is desirable to obtain a map, which the robot can then use to calculate a viable path towards its set destination while avoiding travelling in dead-ends along its way. In case the geography of the environment is known, it is possible and recommended to create a map based on this information which is correctly scaled and compatible with the navigation stack, while avoiding to include in it objects that may be located temporarily in the environment and that the SLAM algorithm would interpret as static obstacles. When the map is not available, ROS implements SLAM algorithms that can create one with the aid of sensors, such as laser scan data from a LiDAR and odometry data from rotary encoders.

4.7.1 gmapping

The *gmapping* algorithm available in ROS relies on *LaserScan* data from a LiDAR that is oriented so that its beams are parallel to the ground. This is because the SLAM approach compares incoming sensor data to the previously scanned environment and tries to calculate the shift in the robot's position, to localize the device within the map by matching the readings. If the LiDAR would be inclined to scan the ground in front of the robot, the algorithm would not be able to directly perform the matching, as incoming sensor data cannot be compared to existing map information. The algorithm relies on odometry data as well, obtained from the wheel encoders.

The information on the movements of the robot is conveyed through a coordinate frames transform tree used to compute the relative position of the robot with respect to its starting position. Additionally, the transform information between the laser scanner and the chassis of the robot is required. Figure 4.7 shows a correct *tf* transform tree that provides the necessary information to the *gmapping* algorithm, where the arrows indicate transform relations between frames. It is therefore crucial to make sure that these are correctly published before proceeding in mapping the environment. For that to happen, it is necessary that the URDF model of the robot correctly defines the link between the laser scanner and the chassis. The Appendix provide these models for the real and simulated *Pioneer P3-DX* robot. Additionally, the *joint_state_publisher* and *robot_state_publisher* nodes need to be started to publish the transform information based on the model, as shown in the example launch configuration in Listing 4.11.

```
<launch>
  <param name="robot_description" textfile="[path_to_urdf_file]" />
  <node pkg="joint_state_publisher"
        type="joint_state_publisher" name="[node_name]" />
  <node pkg="robot_state_publisher"
        type="state_publisher" name="[node_name]" />
<launch />
```

Listing 4.11: Launch configuration publishing tf transform information

Next, the launch file needs to be configured to initiate the following operations:

- Start the robot with the *rosaria* interface.
- Load the odometry calbration parameters, if available.
- Publish *tf* transform data based on the URDF robot model.
- Start publishing LiDAR LaserScan data.
- Start recording data with the *rosbag* tool.

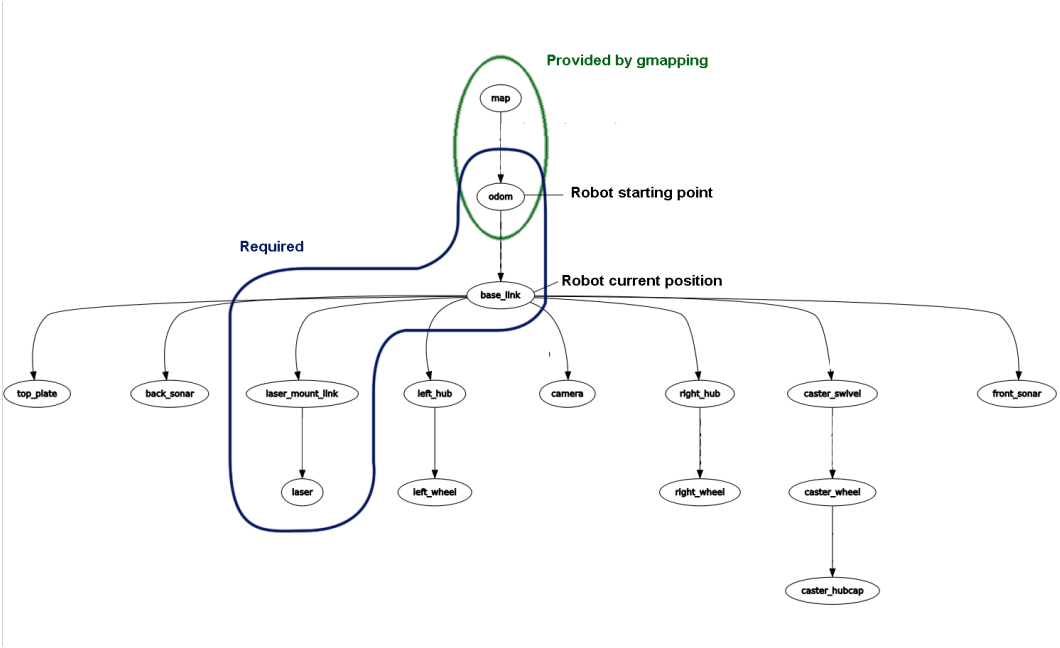

Figure 4.7: Mapping - tf tree configuration for gmapping

The robot can be then driven with the remote controller in the environment while the data is being recorded from the LiDAR and the *tf* transforms providing odometry information. To minimize the cumulative odometry error that may still be present after calibration it advised to drive the robot at a steady velocity and trying to steer as little as possible. Furthermore it is recommended to start and end the exploration at the same point in order to close the loop, to allow the algorithm to try and adjust the map by accounting for eventual drifts in the localization. Neglecting these steps may result in oddities during the mapping phase. It is also worth mentioning that, although it is possible to map the environment in real time while the robot is navigating in it, this is not recommended. This is because both navigation and mapping processes are considerably expensive in terms of required computation power and a sudden slowdown in the performance on a machine with limited resources could deteriorate the quality of the resulting map. Moreover, having a recording of the exploration allows to adjust the *gmapping* parameters and test the effects on consistent data and without the need to repeat the exploration each time.

The *gmapping* is, according to the documentation, a SLAM approach that uses a particle filter to obtain static grid maps. [19][20] The location or pose of the robot in the map is estimated by a probability distribution obtained by computing a set of hypothetical poses, the so-called particles. This process uses a combination of laser range and odometry data to decrease the uncertainty of the prediction. The region in the map with the highest particle density is then assumed to be where the robot is located. Simultaneously to the localization, the map is incrementally updated. Describing the algorithm in depth is outside the scope

of this paper. Instead an explanation is given on the parameters that can be adjusted to improve the quality of the map. Although the algorithm relies on complex probabilistic models and its stochastic behaviour can produce seemingly inconsistent results between iterations, even with the same input and configuration, some parameters influence the outcome in a predictable way. By replaying the same data set, or bag file, and singularly adjusting these parameters, it is possible to compare the results and identify the pattern of how they affect the outcome. Below are the parameters which appear to have a direct impact when mapping the hallway of the university, where the effect is evaluated in Section6.1.

- maxRange: in meters, this depends on the laser scanner used.
- *maxUrange*: in meters, this value must be less than *maxRange* and tells the algorithms to consider the area up until this distance as free from obstacles
- delta: sets the resolution of the map, with lower values resulting in a larger map with a higher pixel/meter ratio
- lstep and astep: controls the linear and angular adjustment step of the map based on the estimated position from the laser scan matching; it should be minimized if the odometry information is reliable

Moreover, slowing the rate at which the recorded data is replayed seem to positively affect the quality of the map, as the computer is not over burdened with having to hastily process the data and update the map. After the map is built, it is necessary to save it using the *map_saver* command of the *map_server* package. This produces two files, one map image like the one shown in Figure 4.8 and one text file that specifies the resolution of the map, its dimensions and the colour thresholds between free, unknown and busy space.

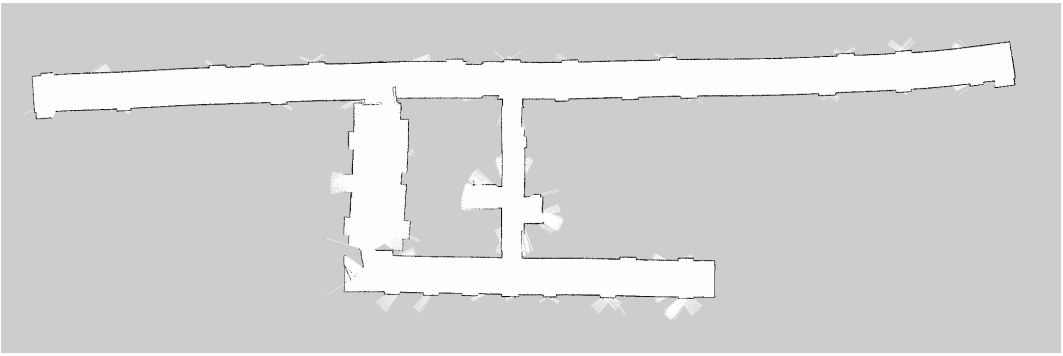

Figure 4.8: Mapping - Example of a map generated with gmapping

4.7.2 From actual map

As mentioned, a map which is compatible with ROS is simply an image and a text file that specifies a few parameters. Knowing this, it is straightforward to adapt an available map that is correctly scaled in any image editing software. The resulting map shown in Figure 4.9 represents the ground-truth map of the environment and can be then used to command the robot to autonomously drive in it. The procedure is the following:

1. Determine the desired resolution of the map in pixels per meter.
2. Compute the dimensions of the map based on its resolution and the dimension of the environment.
3. Create an empty image and fill it with grey colour, representing an unknown space.
4. Overlay the actual map on top and scale it according to the resolution.
5. Fill areas corresponding to free space in white.
6. Paint a black border around them, representing walls and obstacles.

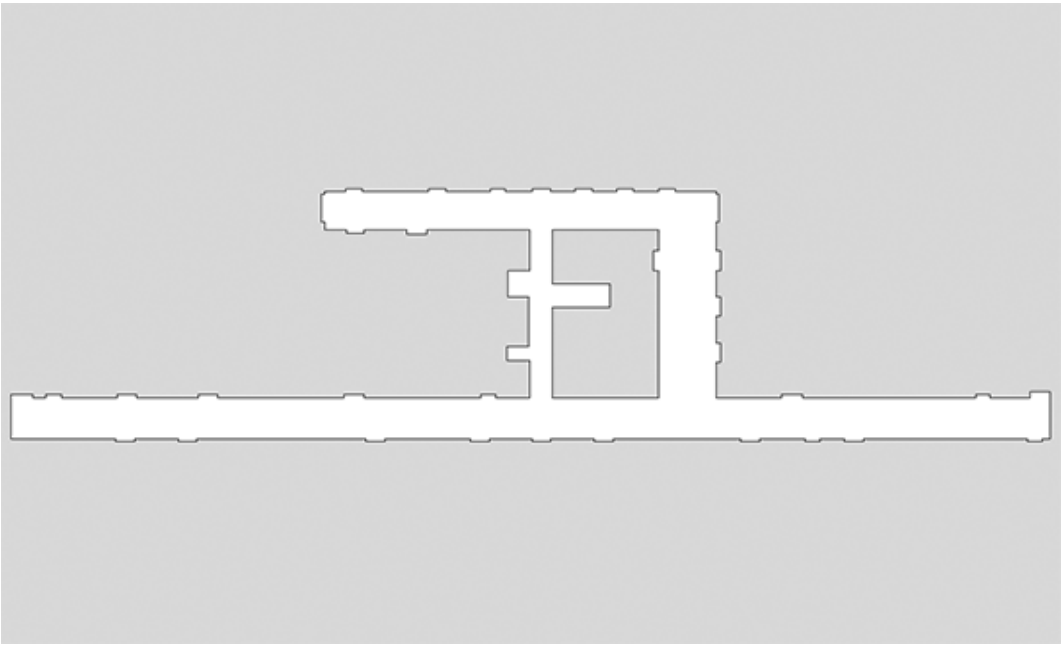

Figure 4.9: Mapping - Example of a map generated manually from ground truth data

4.8 Navigation

The goal of an Autonomous Mobile Robot is ultimately to navigate the environment towards a set goal as efficiently as possible while avoiding obstacles in its way, both static and dynamic. The former category includes obstacles such as walls and furniture which can be previously mapped and which are assumed not to change their position. Dynamic obstacles on the other hand change their position over time and it is not possible and even not desired to include them in a static map. These must therefore be detected and successfully avoided while the navigation is in process.

The literature and resources on this topic are extensive and diverse and ROS provides complete documentation on its mapping and navigation stacks in addition to many examples. [21] In order to achieve autonomous navigation capabilities, the ROS navigation stack must be integrated with the *Pioneer P3-DX* robot. Several sub-systems need to be interfaced according to the digram shown in Figure 4.10. This section describes each of these and outlines how they can be configured.

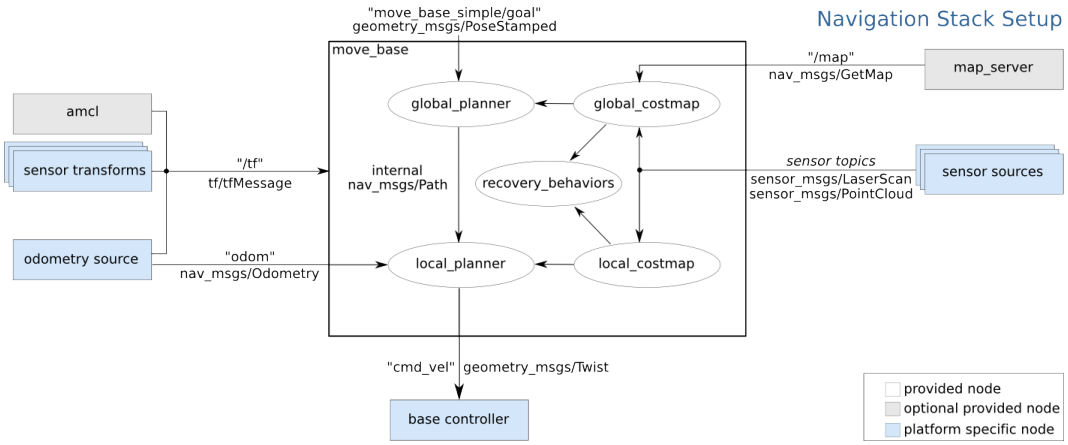

Figure 4.10: Navigation - Stack block diagram

http://wiki.ros.org/move_base

4.8.1 Interface

The *move_base* node is the high level core component in the stack, as it is the one that links global and local planner and acts as an interface with the external sensors data stream and the localization sub-system. It is also responsible of commanding the robot by publishing velocity messages. The *move_base* system allows to choose between an array of different

local and global planners, enable various recovery behaviours and adjust the frequency at which the global planner executes, among others listed in the official documentation. [22]

4.8.2 Costmaps

Regardless of the specific planners, these act based on their respective costmap, which encompass additional layers of informations on the immediate surrounding of the robot (*local_costmap*) or on the known map of the environment (*global_costmap*). While the *local_costmap* is created in a limited window centered on the robot which is populated with the obstacles that the sensors currently perceives, the *global_map* is either initialized based on a static map or empty, and is updated as the robot explores the environment. The additional layers of information mentioned above comprise most notably the following:

- *static_layer*: Contains information on the static map, if provided.
- *obstacle_layer*: Contains information on the obstacles that are or have been detected.
- *inflation_layer*: Inflates the obstacles to increase the cost of the cells which are adjacent to them.

It is worth mentioning how the *inflation_layer* is implemented. As shown in Figure 4.11, this layer increases the cost of the space adjacent to the obstacles. The space within the radius of the robot is considered to have a lethal cost, while the space further away has a decreasing cost. It is possible to configure the *inflation_radius* and the *cost_scaling_factor* affecting the rate at which the cost function decays, in order to bias the planners to prefer trajectories more distant to the obstacles. In order to visualize this additional information is important to set up a layered costmap configuration. The official documentation offers a valid reference on this process. [23]

4.8.3 Localization

The *amcl* or Adaptive Monte Carlo Localization subsystem estimates the pose of the robot based on a known map of the environment and laser scan data from the LiDAR, and its purpose is to improve the accuracy of the localization process, especially when the odometry data is not reliable. The process of localizing through odometry is called *Dead Reckoning*, where the current position is estimated based on the previous calculated location and the perceived movement of the device from there. As the information from the wheel encoders inevitably presents a measurement error, this leads to an incremental drift of the perceived location from the actual one that keeps increasing after each estimate. To overcome this problem, the *amcl* uses a statistical model that evaluates the likelihood of the location of

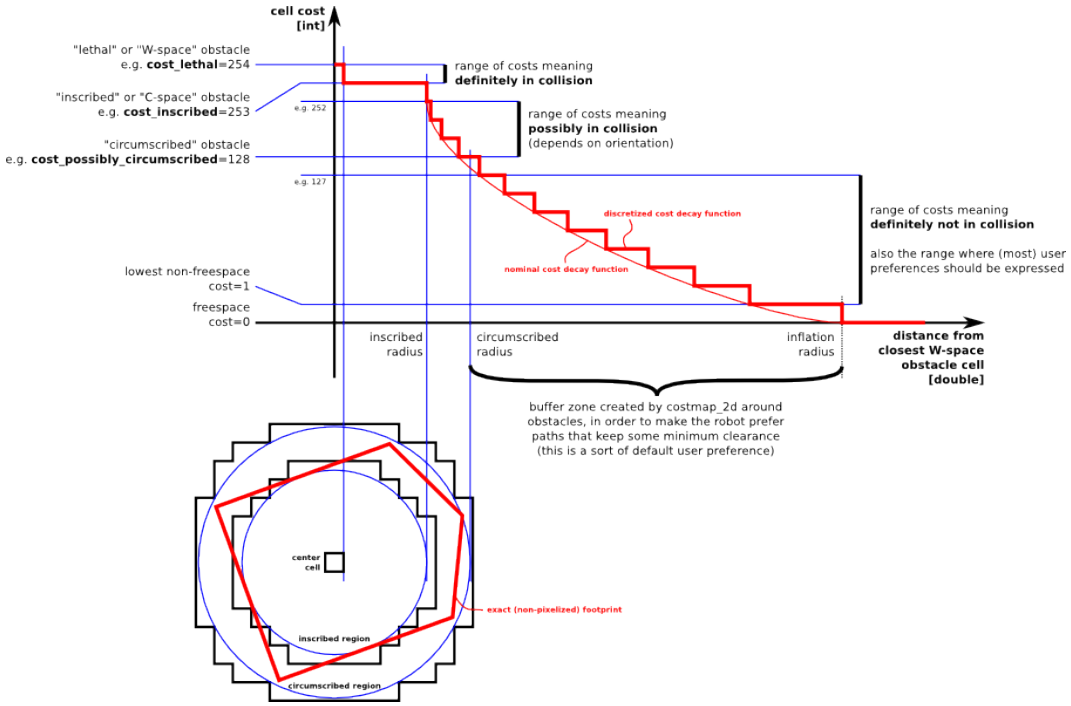

Figure 4.11: Navigation - Inflation layer cost characteristic

http://wiki.ros.org/costmap_2d/hydro/inflation

the robot based on its surroundings, performing scan-to-map matching and computing the transform between the *base_link* or chassis of the robot to the *map_frame*. As shown in Figure 4.12, this localization method effectively bypasses the odometry transform used in the *Dead Reckoning* approach.

It is recommended to integrate the *amcl* subsystem when deploying the robot in a real life scenario as explained in the official documentation. [24] As the *amcl* node is started, an initial pose estimate with respect to the map needs to be provided, either using the *initial_pose_x*, *initial_pose_y* and *initial_pose_a* parameters or graphically with *rviz*. Its parameters can be fine tuned to account for the expected noise of the laser scanner and the drift of the odometry and can be used to bias the localization more towards one source or the other. In Listing 4.12 is a basic launch configuration with the default parameters.

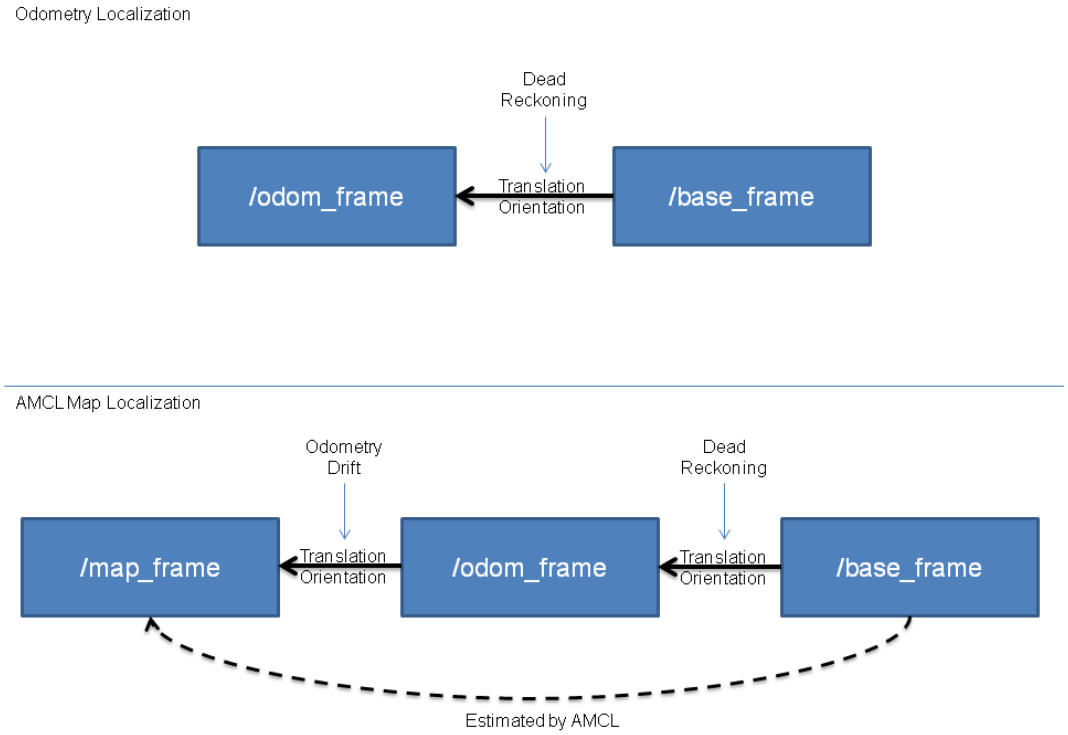

Figure 4.12: Navigation - amcl localization module

http://wiki.ros.org/amcl

```
<launch>
  <node pkg="amcl" type="amcl" name="[node_name]">
    <param name="initial_pose_x" value="$([initial_pose_x])"/>
    <param name="initial_pose_y" value="$([initial_pose_y])"/>
    <param name="initial_pose_a" value="$([initial_pose_yaw])"/>
  </node>
<launch />
```

Listing 4.12: Launch configuration using the amcl component

When testing in the simulation environment however, the *amcl* sub-system seemed to cause odd jumps in the reported location of the robot in order to simulate excessively inaccurate odometry data. In this case, if the behaviour of the *amcl* algorithm itself is not the focus of the experiment, the recommended approach is to use the *fake_localization* that converts error-less odometry data into pose, particle cloud, and transform data of the form published by *amcl*. The initial pose can be set via the *delta_x*, *delta_y* and *delta_yaw* parameters. This node requires however an additional transform between

the map and the odometry frame. Listing 4.13 provides an example launch configuration.

```
<launch>
  <!-- creates a tf transform between the odometry and the map frame
       so that they coincide -->
  <node pkg="tf" type="static_transform_publisher"
  name="map_odom_broadcaster"
  args="
       0 0 0 0 0 0     <!-- x y z yaw pitch roll -->
       /map            <!-- parent_frame_id -->
       /odom           <!-- child_frame_id -->
       100             <!-- publish_period_in_ms -->
  " />

  <node pkg="fake_localization" type="fake_localization"
       name="[node_name]">
    <param name="delta_x" value="$([initial_pose_x])" />
    <param name="delta_y" value="$([initial_pose_y])" />
    <param name="delta_yaw" value="$([initial_pose_yaw])" />
  </node>
<launch />
```

Listing 4.13: Launch configuration using the fake_localization component

4.8.4 Global planner

The *global_planner* sub-system is in charge of evaluating a long-term path from the location of the robot to the its final destination. This decision is made based on the global_costmap, which represents the knowledge of the environment acquired up until the current moment and is based on a static map, if available, and on previous exploration. The planner executes when a new goal is set or when the *local_planner* signal that the planned trajectory is obstructed. Alternatively, it can be configured to run in set time intervals. To configure a global planner, the *move_base* package *base_global_planner* parameter need to be set. Notable examples of global planners implemented in ROS, according to the official documentation, are:

- *navfn*: A grid-based global planner that uses a navigation function to compute a minimum cost plan from a start point to an end point in a grid. The navigation function is computed with Dijkstra's algorithm.
- *global_planner*: A fast, interpolated global planner built as a more flexible replacement to *navfn*.

- *carrot_planner*: A simple global planner that takes a user-specified goal point and attempts to move the robot as close to it as possible, even when that goal point is located inside an area occupied by an obstacle.

4.8.5 Local planner

The other essential sub-system of the navigation stack is the local planner, in charge of computing short-term trajectories to drive the robot close to the path towards the destination laid out by the global planner, while avoiding unexpected obstacles not previously detected, tracked in the *local_costmap*. Different implementations are available in ROS and Table 4.1 lists the features of two alternatives, the *Timed Elastic Band* and *Dynamic Window Approach* planners, evaluated in Section 6.2 of this paper.

The *Dynamic Window Approach* or DWA is thoroughly described in the listed resources, along with the procedure on how to configure it. [25][26] Quoting the listed papers, the goal of DWA is to produce a (v, w) pair of translational and rotational velocities respectively, which represents a circular trajectory that is optimal for the robot's local condition. DWA reaches this goal by searching the velocity space in the next time interval. The velocities in this space are restricted to the admissible set, which means the robot must be able to stop before reaching the closest obstacle on the circular trajectory dictated by these admissible velocities. Also, DWA will only consider velocities within a dynamic window, which is defined to be the set of velocity pairs that is reachable within the next time interval given the current translational and rotational velocities and accelerations. DWA maximizes an objective function that depends on the progress to the target, clearance from obstacles and forward velocity to produce the optimal velocity pair.

The *Timed Elastic Band* approach is described in the listed publication. [27] Quoting the latter, the algorithm implements an optimal local trajectory planner, where the initial trajectory generated by a global planner is optimized with respect to minimizing the trajectory execution time (time-optimal objective), separation from obstacles and compliance with kinodynamic constraints, such as satisfying maximum velocities and accelerations. The current implementation complies with the kinematics of non-holonomic robots (differential drive and car-like robots) and holonomic robots. The optimal trajectory is efficiently obtained by solving a multi-objective optimization problem. The user can provide weights to the optimization problem in order to specify the behaviour in case of conflicting objectives.

Configuring the planners above can be challenging due to the multitude of parameters that depend on a variety of factors, such as the robot characteristic, the deployment environment and the performance of the machine. Each algorithm has its own set of parameters and discussing their effect on the outcome is not in the scope of this paper. To achieve a balanced configuration it is however recommended to follow these general guidelines:

- *Planner frequency*: This parameter of the higher-level *move_base* component determines the rate at which the planner executes and transmits velocity commands to the robot. This parameter value should be high enough so that the response of the system is satisfactory and the movement of the robot is fluent, with no jitter. Further increasing this frequency may lead to a high computational load and a decay in the performance.
- *Trajectory optimization*: These parameters depend on the specific algorithms. These include how far in space and in time should the algorithm plan, the number of different trajectories to be evaluated and the amount of effort made during the optimization step to select the most efficient one. It is desired to adjust these parameters to lower the computational load while maintaining a satisfactory performance.

	TEB	**DWA**
Alias	Timed Elastic Band	Dynamic Window Approach
Strategy	Continuous trajectory optimization predictive controller	Sampling-based trajectory generation, predictive controller
Optimality	Time-optimal (or ref. path fidelity) with kinodynamic constraints (multiple local solutions, parallel optimization)	Time sub-optimal with kinodynamic constraints, samples of trajectories with constant curvature for prediction (multiple local solutions)
Kinematics	Omnidirectional, differential-drive and car-like robots	Omnidirectional and differential-drive robots
Computational burden	High	Low/Medium

Table 4.1: Navigation - Local planners comparison

Robot Operating System (ROS): The Complete Reference, Volume 2 [28]

Independently of the chosen local planner, the following robot-related parameters need to be set according to the specifications, in order for the planner not to misjudge the feasibility of a calculated trajectory:

- *maximum linear velocity* in m/s
- *maximum linear acceleration* m/s^2
- *maximum angular velocity* in rad/s
- *maximum angular acceleration* in rad/s^2

In order to determine the maximum linear velocity and acceleration when the robot documentation omits them, the robot is controlled to drive forward and, as soon as it reaches a constant velocity, the published odometry data provide the information. For the angular velocity and acceleration parameters the procedure is similar, but the robot rotates in place instead.

5 Approaches assessment

Throughout the development of the project, implementation issues arose that could be tackled using different approaches. In each case, a thorough evaluation was made in order to identify the most suitable solution. In this chapter, these evaluations are presented while explaining the reason why one approach was ultimately favoured and pursued.

5.1 Robotic Framework

In the very first phase of development, two frameworks were identified which were deemed suitable to develop the system described in this paper while satisfying the outlined implementation requirements. As this choice is going to define the whole development process, it is crucial to thoroughly evaluate the available solutions and carefully consider which one best suits the needs. Rushing into the development without doing so may lead to dead ends in further stages.

The ever growing open-source community involvement in the field of robotics software made many frameworks available and the quality of some of the solutions offered is remarkable, but it can be challenging to find one that can offer guarantees in terms of reliability, support and usability needed in order to deploy a robust and lasting system. While many different robotics frameworks are available at the moment of this writing, there is not one which is absolutely superior in every aspect. As a matter of fact, it is not unusual that a combination of different frameworks is implemented to meet diverse needs of some complex system.

It should be also mentioned that, given the current dynamism in robotics software, it is likely that the platforms currently in use and developed by a large community are going to be abandoned in favour of emerging new solutions within a short span of time.

Covering all the robotic frameworks currently available is not in the scope of this paper and the reader is encouraged to research whether new solutions might have emerged following this publication. Here two of the frameworks, ROS and ROCK, have been evaluated as both have been proven suitable to the development of Autonomous Mobile Robots, as systems with mapping and navigation functionalities were successfully deployed.

ROS

The Robot Operating System is an open-source framework that provides a diversified set of tools to model different robots and simulate their behaviour in a virtual environment, interfaces with sensors and actuators and libraries to implement advanced functionalities such as SLAM and dynamic trajectory calculation.

The architecture is based on a message-based peer-to-peer communication infrastructure between independent software units, or nodes, blocks of functional code and are implemented as classes that wrap robotic software libraries. This independence between nodes allows different specialists in robotics to develop and maintain single packages, either in Python or C++, that communicate with each other through well defined interfaces, or topics, by sending message asynchronously using a Publisher - Subscriber pattern. Of particular relevance in the scope of this project is the availability of a navigation stack of packages which implement complex SLAM and trajectory calculation algorithms, transparent to the user.

ROCK

Developed by Deutsche Forschungszentrum für Künstliche Intelligenz GmbH (DFKI), the Robot Construction Kit or ROCK is built upon the Orocos Real-Time Toolkit. It implements Kinematics, Dynamics and Bayesian Filtering Libraries that can be deployed in navigation and mapping application. The core component of the toolkit is however its toolchain that enables different software components to interact asynchronously in a hard real-time fashion.

Developed by KU Leuven University since 2001, Orocos is one of the oldest robotics frameworks available. A wide variety of applications have been implemented with this framework where Real Time capabilities are mission-critical, ranging from industrial applications to Autonomous Mobile Robots. Most notably the Berlin Racing Team used the Orocos RTT toolkit to develop the software components of an autonomous vehicle, which was selected by DARPA as semifinalist in the 2007 Urban Grand Challenge Competition. The project is active, although the latest stable version 2.6.0 of the toolkit dates back to Dec 2012.

Building upon the Orocos RTT, ROCK (Robot Construction Kit) offers additional features aimed to improve sustainability, with error detection, reporting, and handling capabilities, scalability and reusability, allowing the integration of ROCK framework-independent drivers and components using another framework of choice. Additionally, it provides data visualization with the integrated Vizkit tool. The components are developed in C++ while the integration in the framework is carried out by configuring with Ruby the oroGen tool that builds the RTT structure.

Development status

As the aim of the project is to implement a platform that could be used for researches for years to come, one of the key issues is choosing a framework that is up-to-date and is going to be maintained and developed while the platform is in use.

ROS latest long term stable release, Kinetic Kame, dates back to May 2016 followed by a new one, Lunar Loggerhead, dating back to May 2017. Furthermore, the activity on the official and repository web-pages is sustained and consistent, with an large community contributing to the steady development of the project.

On the other hand, while ROCK is officially maintained and significant updates are expected to be released in early 2018 on the development branch of the software, the latest stable official release dates back to February 2012. The activity on the official website is also sporadic, raising the concern of whether a platform implemented using ROCK could be kept up-to-date in the future.

Resource availability

Since the platform is likely to be used for short-term researches carried out by developers with little or no previous experience with the framework, it is crucial that sufficient resources and consistent community support are available.

A wide array of material on ROS is available online, ranging from tutorials and code templates to books and publications, both on official and third-party websites. Different IDEs can also be configured to develop with the framework, simplifying the development process.

The ROCK platform provide a moderate amount of tutorials of the core components of the framework, but there is virtually no resource available outside the official website. This leads to a particularly steep learning curve and to the risk of development dead ends.

Support

During the course of any project it may very well be that external support is necessary to overcome some issues. Therefore, it is helpful to be able to rely either on an active community or a dedicated official support team should the necessity arise.

The community behind ROS is extended and diverse, ranging from robotics specialists to involved users, which contribute to a vast knowledge base. It is therefore fairly common to promptly find an answer for any given topic in the forum.

The ROCK community is mostly comprised of researchers and developers of the DFKI. While the level of competence is undoubtedly high, receiving a prompt answer is on the other hand not a likely scenario, potentially slowing the development process significantly.

5.2 Robot

A fundamental choice that carries numerous implications in the development of the project is clearly the robot platform on which the system is implemented. While a fundamental characteristic of ROS is that it aims to be hardware-independent, offering standard interfaces of communication between the different nodes, a navigation stack that provides high-level abstraction and functionalities regardless of the components it relies upon, ultimately the framework needs to control the actuators and motors of the robot via compatible low-level drivers. Moreover, the simulation environment also requires accurate models of the hardware to be able to accurately recreate the behaviour of the robot in any given test scenario. These models contain information on the geometry of the robot, the accurate position and inertial characteristics of each of its components, on the connections between them and on the joints with the wheels that the robot uses to move. Finally, the availability of examples and demo applications is mostly robot-specific, with platforms widely used by the community offering

more so than experimental ones. Although both drivers and models could theoretically be realized when sufficient access to the specifications and technical documentation is provided, and implementing some basic application is feasible, given the relatively short project timeframe it is however preferable to choose a robot for which these resources are available.

Initially, the *Fraunhofer VolksBot XT* was identified as a suitable platform. The main reason for this choice was the rugged design and sturdiness of the chassis, which could theoretically withstand severe impacts and adverse meteorological conditions. Moreover, the peculiar topology of the wheels would allow it to drive through extreme terrain conditions (hence the name) and supposedly climb stairs. Eventually, the *Pioneer P3-DX* robot from Adept Technology was chosen for the reasons listed below:

ROS support: The main factor that ultimately determined the abandonment of the *VolksBot* platform is the availability of ROS compatible software and drivers. The robot uses two *Maxon DC* motors to drive the wheels but while in current models these are interfaced with a *EPOS 70/10* controller from the same brand for which a ROS driver is available, the legacy model available for this project implements a Fraunhofer Motor Controller built in-house and for which the support has been discontinued. In order to be able to interface the robot with ROS, this controller would have needed to be replaced for an addition cost. The *Pioneer P3-DX* comes with a fully ROS compatible software package and implementation examples, making it the most suitable solution.

Step climbing feature: The *Fraunhofer VolksBot XT* is marketed as capable of climbing stairs, which would have made the platform particularly suited to drive in outdoor environments. Previous experiments with this feature however reportedly failed, as the robot would capsize backwards while trying to drive up the steps, losing the advantage against the *Pioneer 3-DX* robot.

Weather resistance: The *VolksBot XT* chassis is robust but its open frame design make it so that any external components mounted on it would be exposed to weather conditions. Since the system needs to be implemented in a short time, the ROS framework is installed on a laptop instead of designing some more elaborate solution and placing the latter on the frame of the robot would not offer any protection against rain. In this respect, the *VolksBot XT* does not offer any additional benefit compared to the *Pioneer 3-DX* platform, which on the other hand provides flexibility in the installation of external protective casing on top of its flat surface.

5.3 Remote controller

Controlling the robot while it is moving on the ground is not possible using the laptop mounted on top of the chassis. It is then advisable to setup a communication interface between the

laptop and another device. While different devices could be used as remote controllers, an Android device is ultimately preferred to the other options evaluated, a radio transmitting joystick device and another laptop. The criteria this choice is based upon are customization capability and portability.

Customization capability: This indicates the possibility to implement new functionalities on the device, other than bare minimum navigation controls. While a legacy radio joystick could perhaps send some function signals that could be interpreted by ROS to enable different features during operation, this device is hardware limited as it is not possible to implement a Graphical User Interface. A second laptop on the other hand could make use of the whole range of ROS tools, to visualize every parameter and sensor data stream and control the robot and would therefore be the best candidate in this respect. An Android device is capable of implementing new functionalities and a few core libraries are already available implementing basic features, such as navigation control or teleoperation and image visualization from a remote camera, among many others. It is however more cumbersome to build and install these applications and it may be necessary to customize them to meet specific needs.

Portability: Considering that the mobile robot is expected to navigate in relatively large environments, it makes sense to be able to physically follow it while retaining the control over its operation. In this case, a radio transmitting device and an Android phone or tablet offer the same portability advantage on a larger laptop device, which is not suited for mobile operations.

6 Evaluation

This section describes the experiments conducted, which aim to evaluate the reliability of the different approaches that lead to an accurate map and to a safe and efficient navigation in the environment. While performing these experiments, new issues surfaced which were not foreseen at an earlier stage. Due to limited time availability, some of them remain however unsolved and are described, along with possible approaches on how to tackle them. These could offer the reader ideas on how to further refine the platform and improve its robustness.

6.1 Mapping

It is of interest to determine the most robust hardware and software configuration that can produce a 2-D map of the environment, which is as accurate as possible to the actual one in terms of geometry and dimensions. Specifically, the experiments are aimed to answer the following questions:

- Which SLAM algorithm can produce the most accurate map of the test environments?
- Which sensor configuration can produce the best results?
- What are the factors that influence the quality of the map?
- How can the accuracy of the map be improved?

The test scenario is the hallway of the 7th floor of the university building and is characterized by long corridors with narrows paths connecting the two main wings. Moreover, the corridors lack distinguishing landmarks, such as curves or furniture. In the experiment, the robot travels in the environment shown in Figure 6.1 following the path in red, starting and ending at the same point, in order to create a static map. The robot is controlled manually during the trip using an Android remote controller device, while the environment is sampled with the LiDAR and the data is recorded in a bag file. Once the exploration is concluded, the map is built using the *gmapping* algorithm, one implementation of SLAM in ROS, while replaying the recording.

Figure 6.2 shows how the following parameters are measured to evaluate the accuracy of the map, where the blue contour represents the boundaries of the map created using SLAM:

- Width and length of the south and north wing corridors.
- Drift in the south wing, measuring how severe is the bending phenomenon observed.

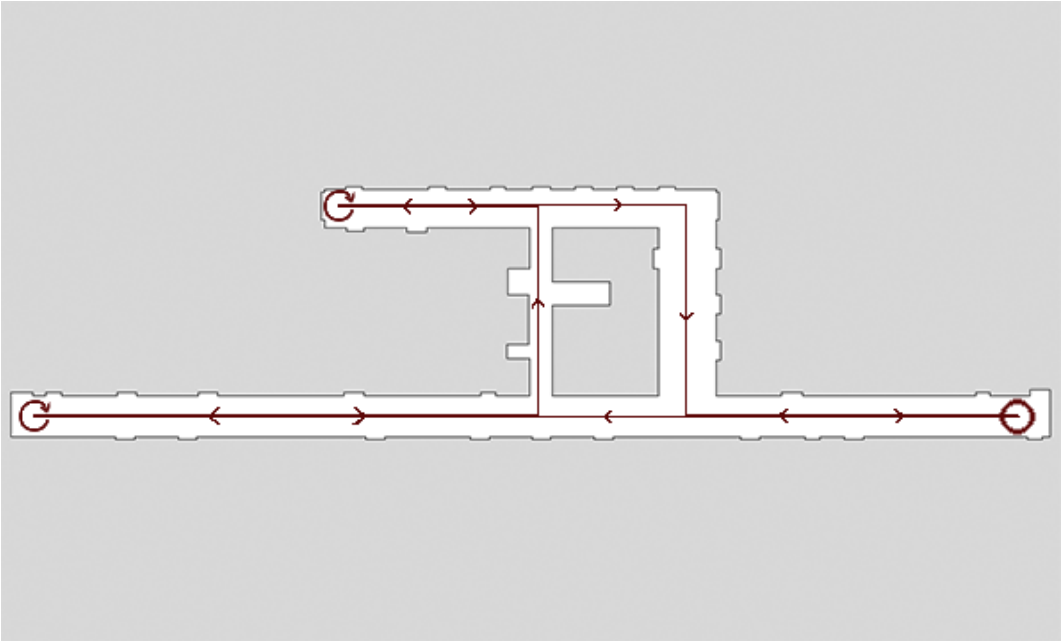

Figure 6.1: Map - Experiment execution

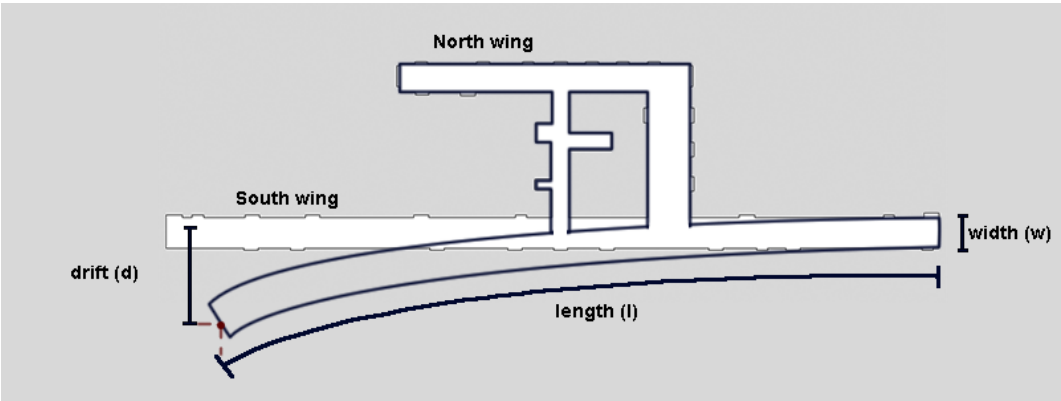

Figure 6.2: Map - Experiment evaluation

6.1.1 Simulation

The scenario is first created in the *Gazebo* simulation environment by modelling the actual map of the hallway. The setting, shown in Figure 6.3, is dimensioned according to the real one. The map resulting from the *gmapping* SLAM algorithm is shown in Figure 6.4 overlaid with the ground truth map. It displays some imperfections, as some false detections are scattered along the path of the robot and the corridor is slightly bent.

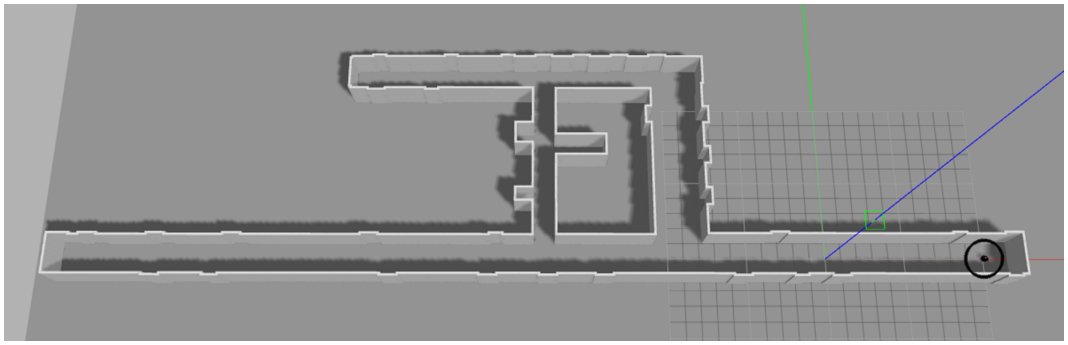

Figure 6.3: Map - Simulation of the hallway in Gazebo

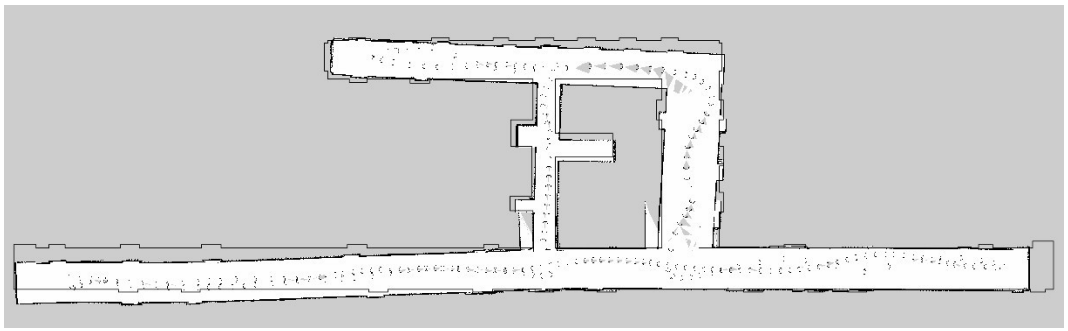

Figure 6.4: Map - Simulated environment

6.1.2 Uncalibrated odometry

Having assessed the SLAM configuration in the simulation, the next test is run in the real environment with the same settings. The robot is controlled manually to drive the same itinerary while recording the data. Figure 6.5 shows the resulting map overlaid with the odometry data and the ground truth map, while Table 6.1 lists the measurements. The trajectory does not remotely resemble the actual one of the real robot. This causes the east-end of the south

wing corridor to be mapped as two different locations as the robot travels through it the second time, closing the loop. The map is also scaled differently compared to the correct one, as the measured length of the south wing corridor has a negative error of 3m. The reason is that the odometry data coming from the wheel encoders is not accurate. Section 4.3.3 outlines how to evaluate the accuracy of the odometry data and to calibrate the encoders.

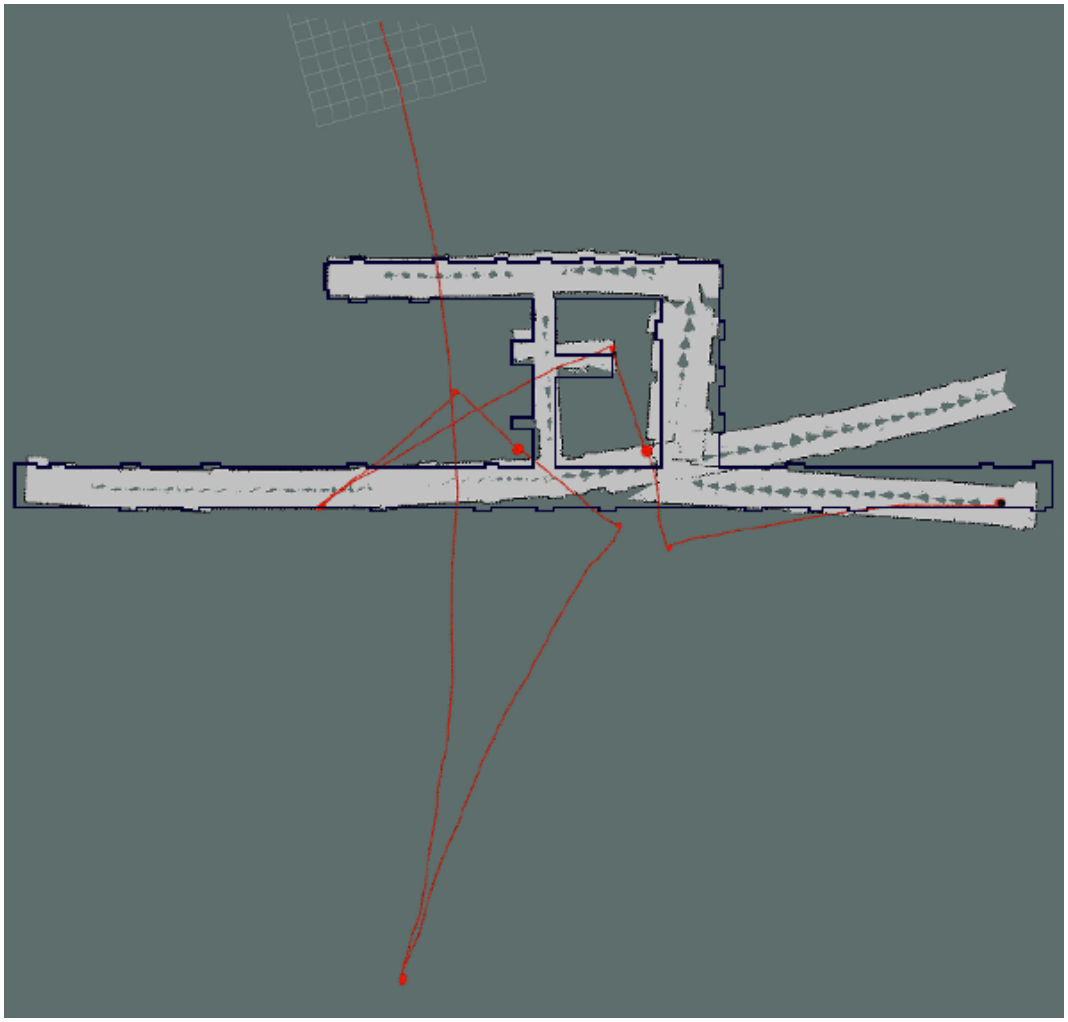

Figure 6.5: Map - Inaccurate odometry

6.1.3 Calibrated odometry

While testing the odometry accuracy and comparing the actual position with the one reported by the robot, the following approximate errors were measured:

- Horizontal drift of 3cm left every meter forward.
- Positive offset of 1cm every meter forward.
- Positive offset of 5° every 180° of clockwise rotation.

After calibrating the odometry, the same experiment is repeated, yielding the result shown in Figure 6.6 and measured in Table 6.1. The reported trajectory is much more faithful to the actual one, where the begin and end point are closer to one another. Consequently, the south wing corridor is now mapped correctly as a single location. Its scaling is more accurate, as the measurement error of its length decreases by 2m. It is however still bent, with a drift of 3.6m.

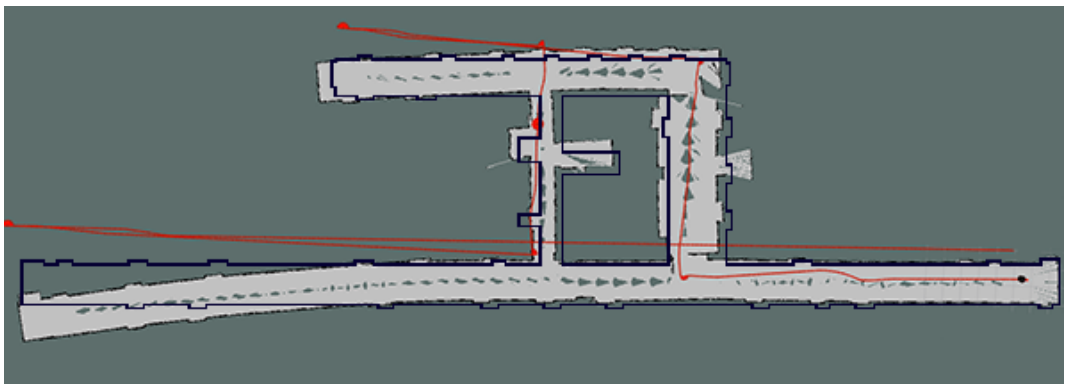

Figure 6.6: Map - Calibrated odometry

6.1.4 Parameters adjusted

The next step is to adjust the parameters of the *gmapping* algorithm in order to post-process the recorded data in the most effective way to produce a reliable map, as explained in Section 4.7.1. The result is shown in Figure 6.7 and the measurements are listed in Table 6.1. The corridors are straight and parallel with each other, with a drift of 0.8m, while the scaling and dimensions are accurate, with an error of 0.1m in the length of the south wing corridor. The only remaining flaw is the noise which appears in the proximity of the doorways, as beams of the laser seem to travel a longer distance due to reflection on the corners. On the other hand, the false detections that marked the trajectory of the robot are removed by restricting the valid field of vision of the LiDAR that previously included some sections of the chassis of the robot, which were perceived as obstacles.

A detailed explanation on the parameters of the *gmapping* algorithm is given in the official documentation. However, the focus here is to describe how adjusting the following parameters affected the result:

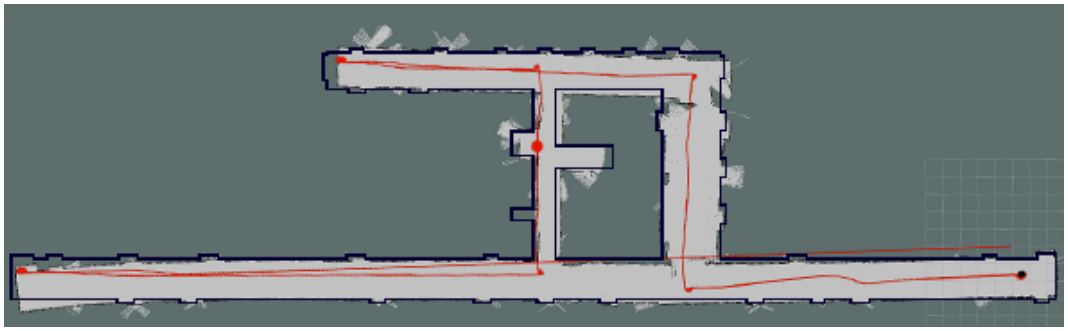

Figure 6.7: Map - Optimized settings

	North wing		South wing		
	Width / m	Length / m	Width / m	Length / m	Drift / m
Actual	2.2	22.7	2.2	60.0	0
Simulated	2.2	21.8	2.2	57.6	0.3
Uncalibrated	2.2	22.4	2.2	57.0	9.8
Calibrated	2.2	22.8	2.2	59.0	3.6
Optimized	2.2	22.6	2.2	59.9	0.8

Table 6.1: Map - Measurements

- maxRange = 4.0: in meters, eliminates the false detections throughout the path of the robot.
- *maxUrange = 3.0*: in meters, trims the laser scan ranges to this value, reducing the reflection on the corners.
- delta = 0.3: in metres per occupancy grid block. By increasing the resolution of the map the definition of corners and obstacles in general is improved.
- lstep = 0.01 and astep = 0.01: in meters, improve the scaling of the map and partially correct the drift.

The last two parameters, *lstep* and *astep*, control how much the algorithm relies on laser data to correct the map. These appear to be the most relevant in this specific scenario, where the environment is characterized by long corridors lacking specific landmarks. In this case, even if the robot moves constantly forward and the odometry data reflects the movements, a short ranged laser keeps sampling an almost identical local environment, with an even wall on each side and no front wall, since its range is not enough to reach it. The consequence is that, based on the laser data, the robot perceives the same surroundings and its position does not seem to change. The *gmapping* algorithm relies on a combination of odometry and laser range data to determine its position and, if the parameters mentioned above are not

reduced, it excessively corrects the position of the robot based on the laser scans, introducing measurement error in the map. One alternative approach would be to use a laser scanner that can reach the end of the corridor to provide a reliable reference point.

6.1.5 Hector SLAM

This issue is much more evident when using a SLAM approach that relies solely on laser data for both localization and mapping. *hector_slam* is one of these algorithms available in ROS that does not use odometry data and can therefore be considered a viable option for robots lacking wheel encoders or similar sensors. By building a map using the same data set, the result in Figure 6.8 shows a remarkable discrepancy between the trajectory of the robot and the perceived map of the hallway, leading to a much shorter length of the corridor and an overlap of sections of the map when the robot closes the loop, rendering the map completely inaccurate.

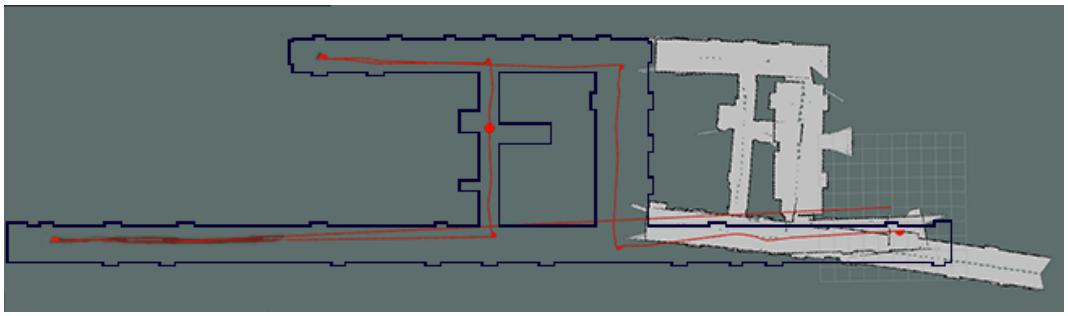

Figure 6.8: Map - Hector SLAM algorithm, only laser data

6.1.6 Sonar mapping

Using the sonar readings with *gmapping* did not produce any usable results, since the system comprises 8 sonars for 8 range data points each scan, not enough to localize the robot in the environment. Figure 6.9 shows a map completely distorted. Worth noting is that the odometry was not yet calibrated at the time the data was collected. It is nonetheless evident that the sonar alone is not sufficient to create a map using the *gmapping* SLAM approach.

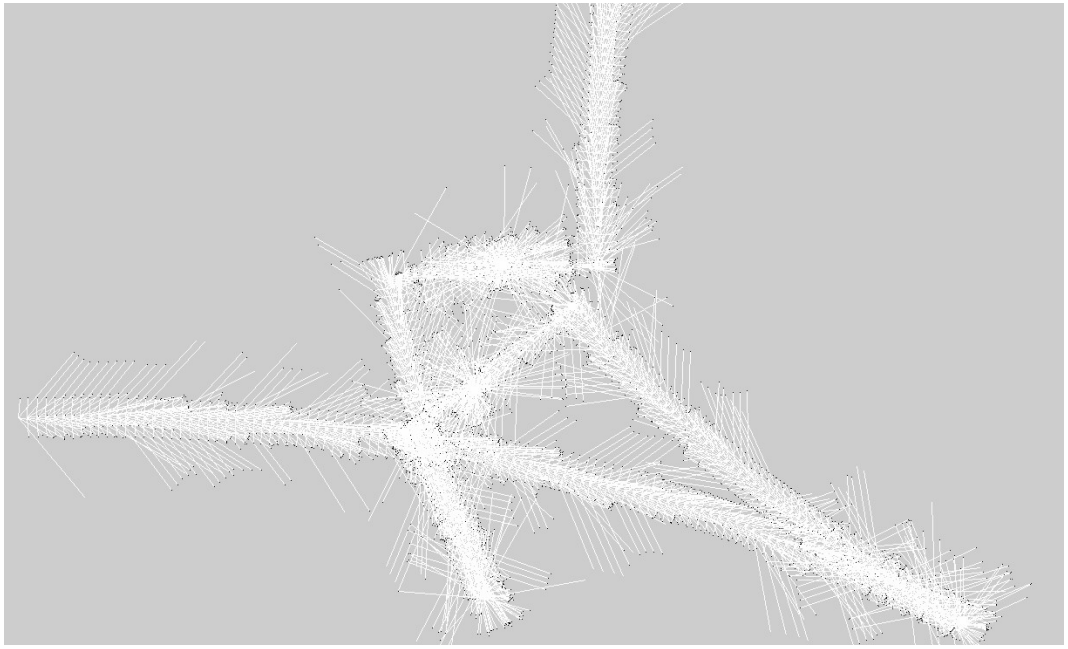

Figure 6.9: Map - Sonar data

6.1.7 Irregular trajectory

Next the robot is driven through the same path, but this time following an irregular trajectory, as it is manually steered from left to right with varying speed in a random fashion. The purpose of the experiment is to assess whether the accuracy of the map is affected by an odd odometry pattern. Two observations can be made about the result shown in Figure 6.10. It is evident that, by steering the robot throughout its itinerary, the odometry error increases drastically, as the coinciding start and end point are perceived by ROS as far apart from each other. This is because the framework relies on the rotary encoders to track the trajectory of the robot and, even after a manual calibration of the related parameters, the remaining rotation error accumulates every time the robot swerves. This causes a measurement error of

approximately 2m in the length of the south wing corridor and a drift of 4m. It can be assumed that, when the *gmapping* algorithm relies on calibrated odometry, the error caused by driving in an irregular trajectory is generally less severe than in the case where the odometry is not accurate. However, this map is generated from data collected in a different exploration than in the previous cases and the difference might not be entirely caused by the trajectory followed by the robot.

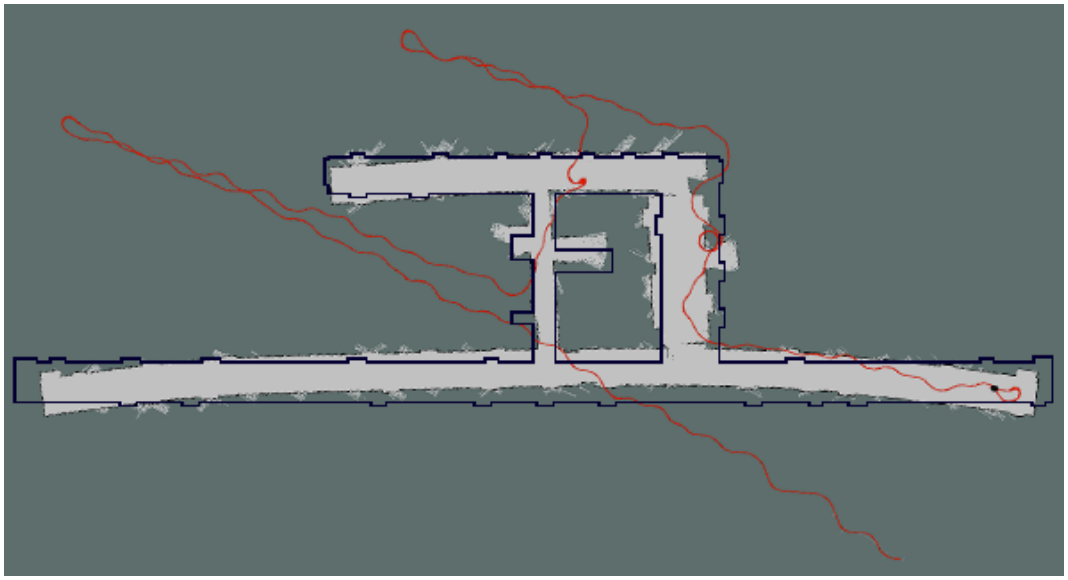

Figure 6.10: Map - Irregular trajectory

6.1.8 Dynamic disturbances

Lastly, the robot travelled once the south wing corridor back and forth as the author walked in a random fashion in front of the laser scanner. The goal of this experiment is to determine whether dynamic objects can influence the outcome of the mapping process and appear as static false detections in the map. Shown in Figure 6.11 the blue trace represents the laser readings as an obstacle travelled in the field of view of the laser scanner. The quality of the resulting map does not seem to be remarkably affected nor any false detections appear in it after the process. While a drift of approximately 1.7m is measured, comparing it with previous results from different explorations is hardly meaningful.

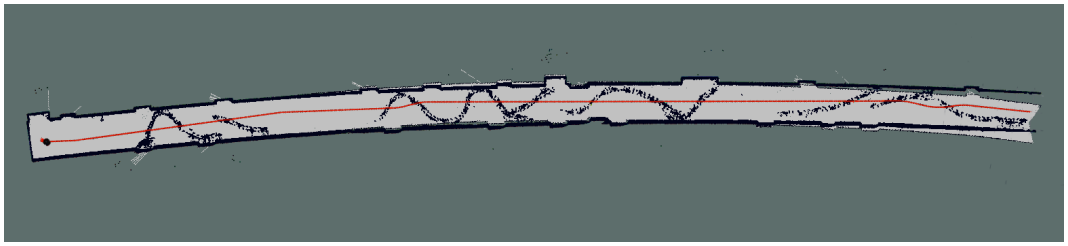

Figure 6.11: Map - Dynamic obstacles

6.1.9 Summary

The findings can be summarized as follows:

- The simulation environment is an ideal tool to test the SLAM configuration as it offers an inexpensive and quick mean to have some preliminary results.
- In case the SLAM algorithm of choice relies on odometry information, it is recommended to assess whether this data is accurate before attempting to map an environment and calibrate it if it is not, since the outcome of the process may be significantly affected by it.
- Mapping without odometry information is feasible and accomplished by relying on laser scan data, however the outcome largely depends on whether landmarks in the environment are in the range of the scanner to provide a reference.
- The *gmapping* algorithm does not appear remarkably affected by dynamic obstacles transiting in the field of view of the laser scanner nor by direction changes of the robot during the exploration.
- It is recommended to perform the mapping after having previously recorded the data during the exploration, in order to best adjust the algorithm parameters to produce an accurate map.

6.1.10 Open issues

Undetectable obstacles

In the scenarios described above, the robot was purposely kept clear of undetectable obstacles, such as furniture elevated from the ground and the descending staircase. The 2-D *SICK TiM* LiDAR mounted in front of the device and scanning the plane parallel to the ground is virtually blind in these circumstances. This is clearly a highly critical risk that is to be considered accordingly. The most straightforward but tedious approach is manually correcting

the map with an image editor, placing boundaries around the undetected obstacles. While tilting downwards a 2-D laser scanner so that its beams are pointed directly on the ground in front of the robot could solve the staircase issue, higher obstacles would still be invisible. Finally, a more sophisticated approach would be to use a 3-D laser scanner, or a 2-D model where its inclination is controlled by a servo motor, that could be used to create a 3-D map signalling the presence of these obstacles.

6.2 Navigation

The experiments described in this section aim to evaluate the robustness of the navigation stack included in ROS. Robustness refers to the ability to reach a set goal in different scenarios, with different conditions interfering in the process.The question these tests are meant to answer are the following:

- How close can the robot position itself from a set destination?
- What are the factors that influence this accuracy?
- How does the device react as dynamic obstacles are detected?
- Can the robot reach a set destination in an unknown environment?

6.2.1 Slalom

The goal of this experiment is to evaluate the behaviour of the robot as it needs to keep adjusting its trajectory and travel in a zig-zag pattern in order to avoid obstacles located on its ideal trajectory, at increasingly closer distance to one another.

As was the case for the mapping feature evaluation, the first test is performed in the simulation environment. The purpose is to assess the behaviour of the algorithm before the deployment in the real environment. The scenario is the hallway and its map is loaded to provide the information about the geography of the environment. The obstacles are in the shape of 1m wide cubes and are located at a distance of approximately 3m from one another, while the diameter of the robot is approximately 60cm. Figure 6.12 shows the scenario.

In the simulation the robot avoids the obstacles and travels successfully to its destination, proving the correct implementation of the navigation stack. As Figure 6.13 shows, the robot lingers on the marked spots while trying to pass the obstacles. This means that the navigation stack is not configured in an optimal way, as the robot travels too close to the obstacles and needs to stop and adjust its trajectory.

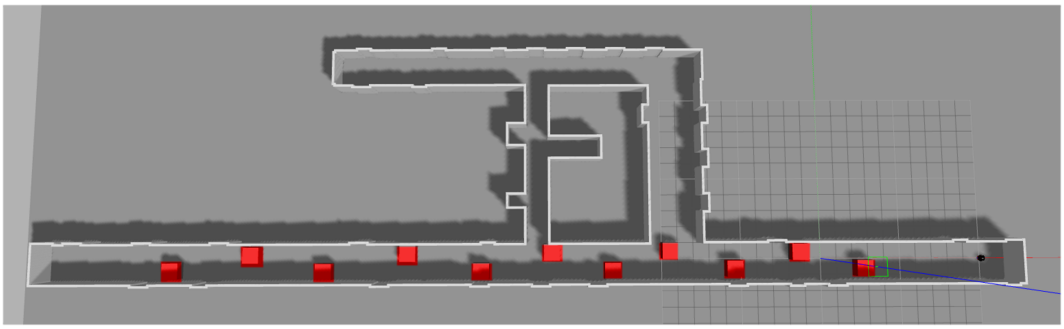

Figure 6.12: Slalom - Simulation of the hallway with a partially occluded corridor

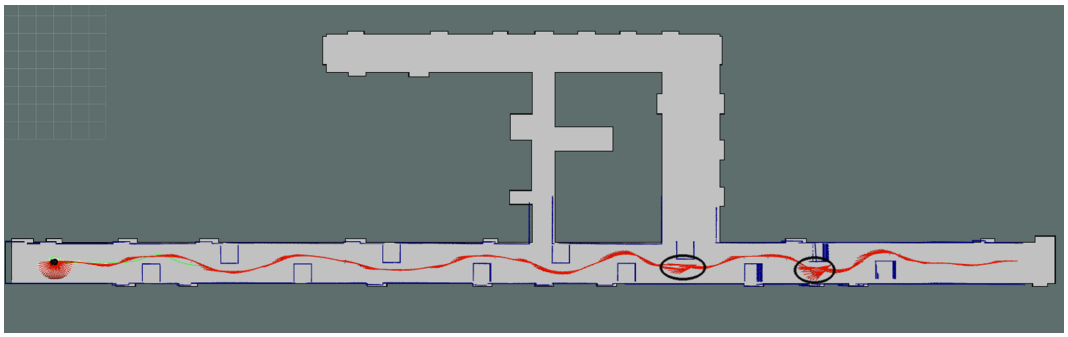

Figure 6.13: Slalom - Navigation in the simulated hallway with a partially occluded corridor

It is however hard to pinpoint the precise root cause of this behaviour with this basic configuration, where the information on the different layers of the costmap is not displayed. Setting up a *layered costmap*, as described in Section 4.8.2, helps in this regard by allowing fine tuning of individual layer parameter and a more informative graphical representation in *rviz*. This information includes the size of the local map, useful to visualize the range in which the device is able to detect and react to an obstacle in its path. The simulation is run once again with this configuration.

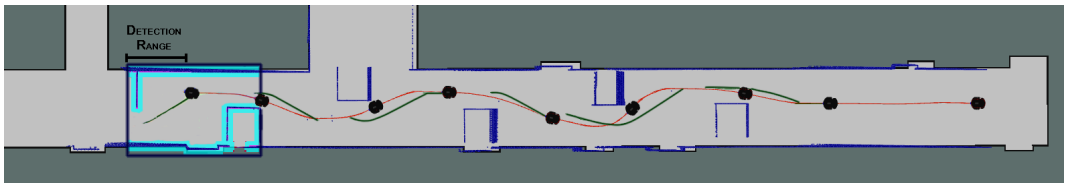

Figure 6.14: Slalom - Planned (green) and actual (red) trajectory

Figure 6.14 shows that a trajectory is calculated to avoid the obstacle as it enters the range of the LiDAR. The robot tries to steer in order to follow it but does not do so fast enough and ultimately approaches the turn as it is already too close to the obstacle. This leads to a

situation where the robot needs to slow down and engage in evasive actions in order to avoid colliding with the obstacle. This means that by the time when the obstacle is perceived, the robot is travelling too fast to react promptly. A straightforward solution to this is to limit the maximum speed of the robot and to increase the distance where the obstacle is acknowledged as such, according to the range of the LiDAR. After these adjustments, the navigation improved as the device followed the planned trajectory without drastically correcting it, as shown in Figure 6.15.

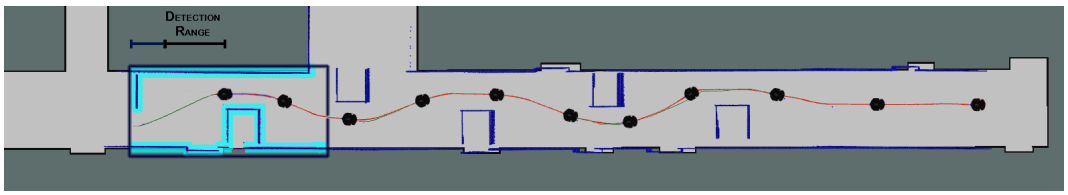

Figure 6.15: Slalom - Slow speed and increased detection range

It appears however that in a few occasions the robot cuts the corners when passing past the obstacles. Although it does not compromise the success of the mission in this particular scenario where the obstacles do not and can not move, it may be advisable to increase the distance that the robot should try to keep from the obstacles to account for the ones that may move, causing potential collisions. On the other hand, it is undesired to make the robot stop where the space is limited and this distance can not be maintained, such in narrow hallways.

This can be accomplished by *inflating* the obstacles to increase the cost of the trajectories closer to them, in order to bias the local planner to choose one further from them. More details are provided in Section 4.8. The result is shown in Figure 6.16, where the robot keeps a sensibly larger distance of approximately 20cm to the obstacles and does not cut the corners as much.

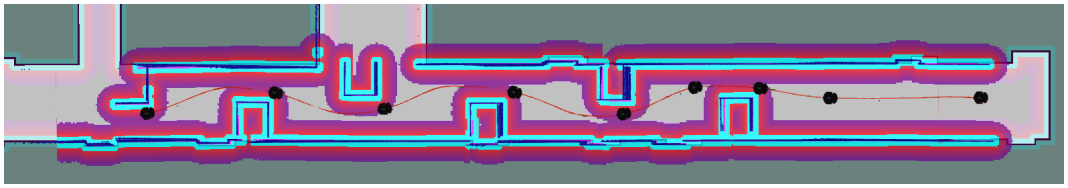

Figure 6.16: Slalom - Inflated obstacles

After having refined the configuration in the simulation environment, the experiment is then executed in real life. The obstacles, represented by cardboard barriers, are placed first at a distance of 2m. Figure 6.17 shows that the robot drives through the track without interruptions.

The distance between the obstacles is then incrementally decreased to 80cm where the robot has a set diameter of 60cm, providing a passage of 20cm at best, barely sufficient for it to drive through. The latter initiates the navigation and successfully proceeds while avoiding the obstacles, proving that the system can be configured to navigate in spaces 20cm larger than the diameter of the robot, as shown in Figure 6.18.

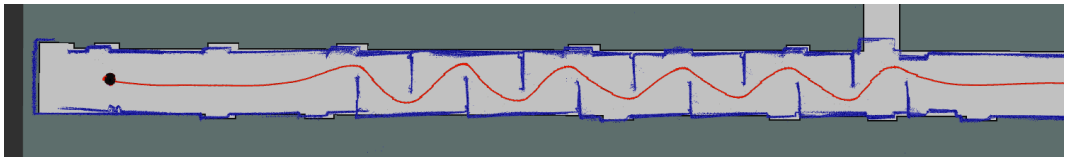

Figure 6.17: Slalom - Real scenario with obstacles at 2m distance

Figure 6.18: Slalom - Real scenario with obstacles at 80cm distance

6.2.2 Dynamic obstacle persistence

During the tests, it appeared that some odd behaviour manifested when an obstacle passed briefly through the field of view of the LiDAR, causing the robot to abruptly stop and engage in steering manoeuvres or abort the mission, even though the path had been cleared in the meantime. In this experiment, meant to investigate the issue, an obstacle is put in front of the laser scanner and removed shortly afterwards, while the navigation stack is started to enable the object detection feature and monitor its response on *rviz*. A camera is located in front of the LiDAR and records the scene, as shown in Figure 6.19. As the obstacle is lifted, the navigation algorithm does not clear it from the map and the obstruction persists, as Figure 6.20 shows. In the figures, the dark blue dots represent the actual readings from the laser, while the purple ones represent the space that the robot assumes is occupied by an obstacle.

This exposed a flaw in the configuration. The underlying problem is that, in order to clear an obstacle from the map, the algorithm must receive a laser reading that returns a range greater than the maximum detection range and less then the maximum range of the LiDAR. Furthermore, this must not exceed a set *raytracing* threshold parameter. What happens in practice is that some laser scanners return a range of zero or *NaN* when no object is detected within their range, as is the case with the *SICK TiM* LiDAR used in this project, effectively

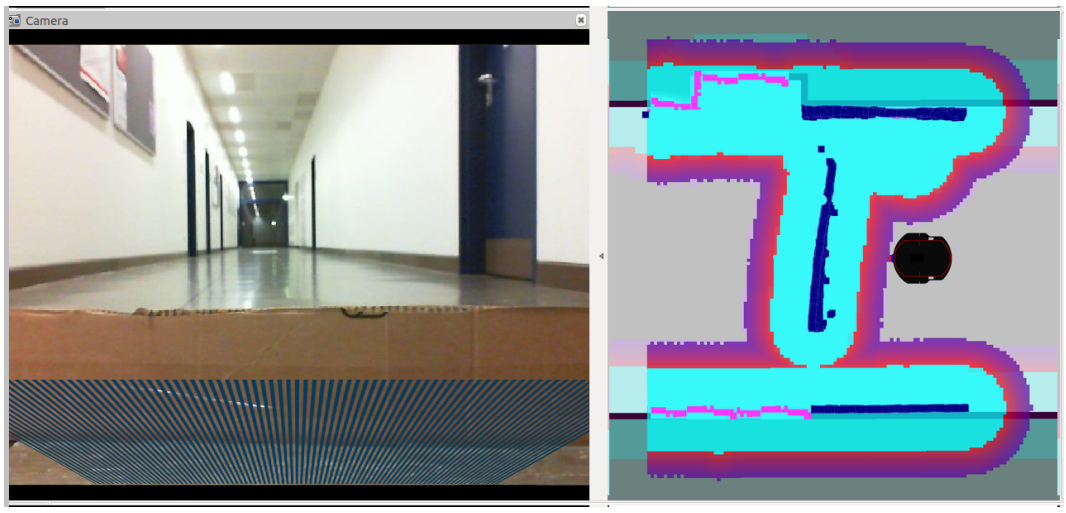

Figure 6.19: Obstacle persistence - Obstacle in front of sensor

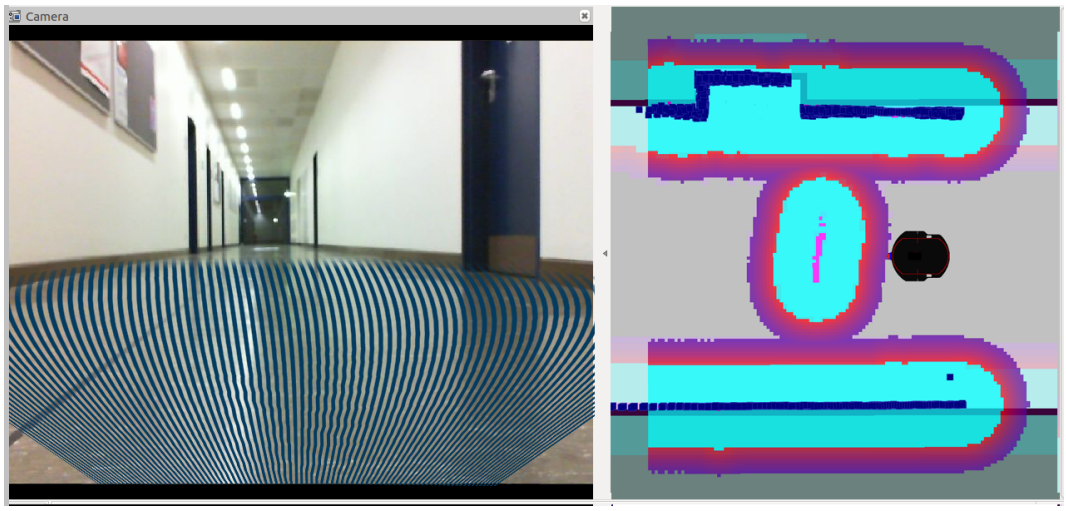

Figure 6.20: Obstacle persistence - No obstacle in front of sensor

hindering this obstacle clearing feature. To fix this problem, a laser scan range filter must be implemented, that converts the readings with a range of zero and *NaN* to a value sensibly lower than the *raytracing* threshold parameter and higher than the obstacle detection range, in order not to cause any false detections. When the test is then repeated, the obstacle is cleared as soon as it is not in the field of view of the LiDAR, as expected.

6.2.3 Dead-end

In the following test, the obstacles are completely obstructing the direct itinerary to the goal, effectively creating a dead-end, as shown in the simulated scenario in Figure 6.21. Due to the limited range of the simulated *Sick TiM* LIDAR, the robot is not aware of the obstruction and should recalculate its trajectory as soon as the obstacles are sensed, choosing to travel around through the connecting corridors instead.

The simulation proceeds as expected, as the robot reaches the occlusion, performs a U-turn and proceeds through the other corridor, eventually reaching its goal, as Figure 6.22 shows. The experiment was repeated in real life by leaving the door on the corridor closed and keeping the other ones open. The robot reaches the occlusion and recalculate its trajectory as expected.

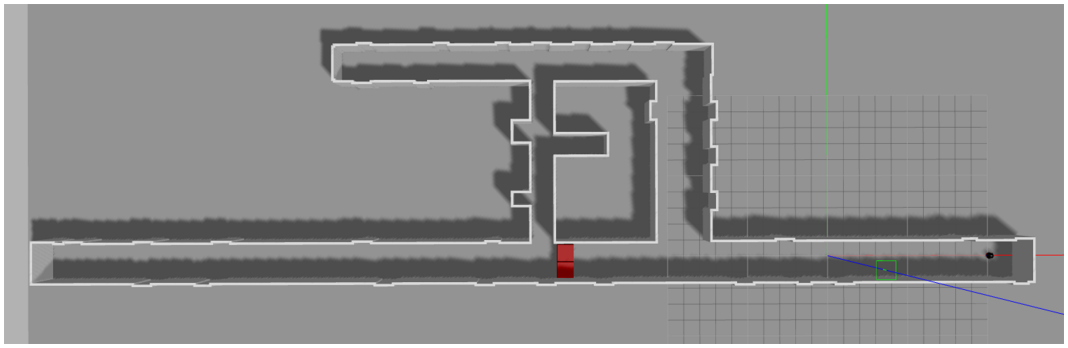

Figure 6.21: Dead-end - Simulation setup with one path occluded

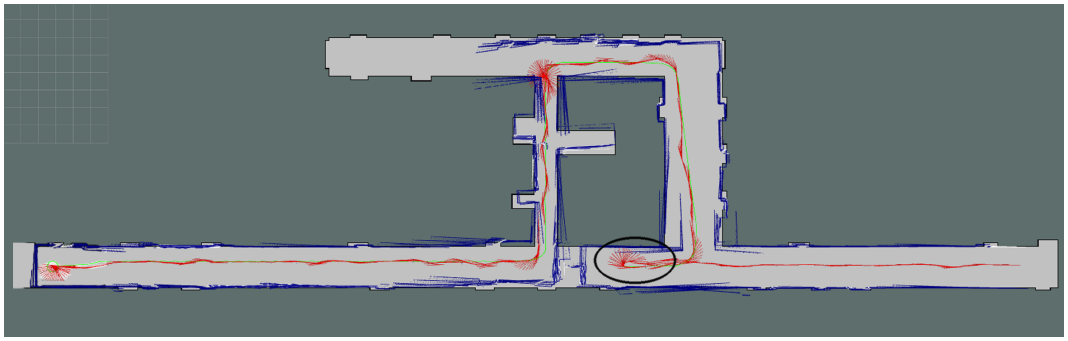

Figure 6.22: Dead-end - Simulation result with one path occluded

In the next test both the possible itineraries are occluded, preventing the robot from completing its mission, as shown in Figure 6.23. The purpose of this is to assess how the algorithm reacts when such a situation occurs. The test is successfully carried out in the simulation environment, as Figure 6.24 shows, where the robot reaches the second occlusion and aborts the mission, since no possible path to reach the destination is viable.

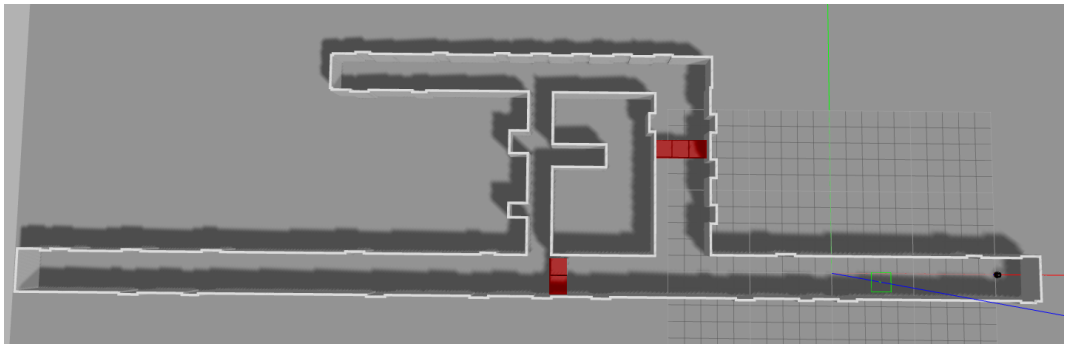

Figure 6.23: Dead-end - Simulation setup with both paths occluded

Figure 6.24: Dead-end - Simulation result with both paths occluded

6.2.4 Target

In this experiment, the robot is commanded to reach a precise destination, corresponding to a target located on the ground and rotate 180° in place. The aim is to evaluate the navigation accuracy in terms of offset from the desired target that the robot displays under different conditions. This accuracy may be significant in tasks involving high precision positioning. A remarkable application would be a device able to autonomously drive to an inductive charging station and position itself over the coil as its batteries reach a set minimum level.

The distances are measured as shown in Figure 6.25 and the measurements are displayed in Figure 6.26, where the red crosses represent the points where the robot located itself with respect to the center of the target. The center of the robot is defined in its models to be precisely in between the left and right wheels and when it lands outside the perimeter, defined as a circle with a 30cm radius, the test is considered failed.

First the target is located in the proximity of one end of the corridor and the robot starts from the other end. The robot must travel a distance of approximately 55m to reach its goal, located 3 meters from the wall at the end of the corridor. This is inside the range of the

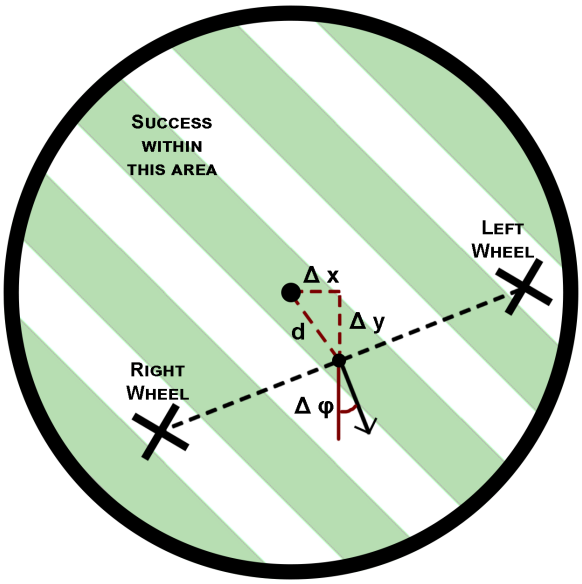

Figure 6.25: Target - Measurement mode

LiDAR of 4m and it should provide the navigation algorithm, which relies on both odometry and laser scan data, with a useful reference to correctly position the robot.

While the accuracy is overall satisfactory and in each test the robot positioned itself well within the target radius of 30cm, the results in Table 6.2 under the label "Long distance" show a consistent error in the x- and y-axix, with an average distance of 122mm from the center of the target and a standard deviation from the mean of 26mm, and a minor rotational delta of -0.9°. The assumption is that this may be due to the long distance travelled, throughout which the relatively small error in the odometry remaining after calibration increases incrementally.

Having a closer look at the recording of the experiment, it appears that the reported distance of the robot from the goal is considerably less than it really is. This seems to corroborate the previous assumption, as an error in the odometry could cause the robot to perceive its position incorrectly. To prove whether this is indeed the case, in the next test the robot starts from 16m from the target. The results, under the label "Short distance" in Table 6.2, show that the error is marginally reduced by 25% to a mean distance of 91mm from the center. It seems therefore that the remaining odometry error has an impact on how close the robot position itself to the center of the target.

The algorithm documentation revealed that the goal distance tolerance can be adjusted. The previous test is run once again with a tolerance reduced from the default value of 100mm

by a factor of 5 to 20mm. As shown in Table 6.2 under the label "Reduced goal tolerance", the error is reduced by a factor of approximately 2 to a mean distance of 40mm with a standard deviation of 14mm, not entirely matching the expectations of a mean value of less than 20mm. On the other hand, in order to reduce its distance, the robot engage in multiple rotation on the spots, gradually adjusting its position. By further decreasing the goal tolerance, the robot keeps adjusting in a seemingly endless loop, never reaching its destination.

Next the target was moved away from the front wall, to evaluate whether the lack of range information from this reference has any influence on the accuracy. The results in Table 6.2 under the label "No wall reference" show that this is not the case, as the mean distance of 36mm is just marginally reduced from the previous case with the reference wall in the range of the LiDAR, meaning that the navigation algorithm does not seem to rely significantly on the laser reading for localization purposes as much as it does on the odometry.

The main findings of this test can be summarized as follows:

- The localization process relies mostly on odometry data to estimate the pose of the root.
- This implies that the accuracy of the initial robot pose estimate has a direct impact on the outcome precision.
- The lack of references in the range of the laser scanner does not affect the accuracy remarkably.
- The goal tolerance can be lowered down to a set threshold at the expense of adjustment time and manoeuvres.

It is however not investigated how different settings would influence the outcomes. This test was performed with the *AMCL* localization algorithm with default settings and in a scenario characterized by the lack of unique features and landmarks, and a slight change in the setup could lead to a very different sets of results altogether. The *AMCL* algorithm for example could be configured to rely more on the laser scan readings than on the odometry data. The results outlined in Table 6.2 above however should give a broad impression on the degree of accuracy which is possible to achieve.

	Long distance				Short distance			
N	Δx/mm	Δy/mm	Δφ/°	d/mm	Δx/mm	Δy/mm	Δφ/°	d/mm
1	20	-88	-1	90	-18	-48	-3	51
2	20	-133	2	134	-23	-74	0	77
3	-33	-150	-3	154	-2	-70	1	70
4	-46	-85	-3	97	-19	-93	0	95
5	-65	-66	-2	93	-38	-21	-1	43
6	-31	-165	2	168	-36	-89	0	96
7	-74	-97	3	122	-29	-125	1	128
8	-32	-106	-3	111	-30	-111	-2	115
9	-34	-125	-3	130	-26	-144	-3	146
	$\mu\varphi$/°	μd/mm	σd/mm		$\mu\varphi$/°	μd/mm	σd/mm	
	-0.9	122	26		-0.8	91	33	
	Reduced goal tolerance				**No wall reference**			
N	Δx/mm	Δy/mm	Δφ/°	d/mm	Δx/mm	Δy/mm	Δφ/°	d/mm
1	-42	-29	-2	51	3	22	0	22
2	-28	-54	0	61	-13	10	1	16
3	2	-38	-1	38	-5	61	6	61
4	-16	-37	2	40	2	25	1	25
5	-9	-36	-2	37	-8	10	0	13
6	-1	-12	-5	12	-10	15	3	18
7	-47	-30	1	56	-11	-50	-1	51
8	22	-32	-2	39	1	-47	2	47
9	-26	-14	0	30	0	-70	-1	70
	$\mu\varphi$/°	μd/mm	σd/mm		$\mu\varphi$/°	μd/mm	σd/mm	
	-1	40	14		1.2	36	20	

Table 6.2: Target - Measurements

6.2.5 Elevated obstacles

The obstacles encountered up until this point were positioned on the ground with no section overhanging from the sides that could constitute an obstruction to the robot. The characteristic of a plausible scenario however is more complex, as there could be elevated objects causing potential collisions which can not be detected using a two dimensional LiDAR scanning the plane parallel to the ground. Among the possible approaches, a ToF (Time of Flight)

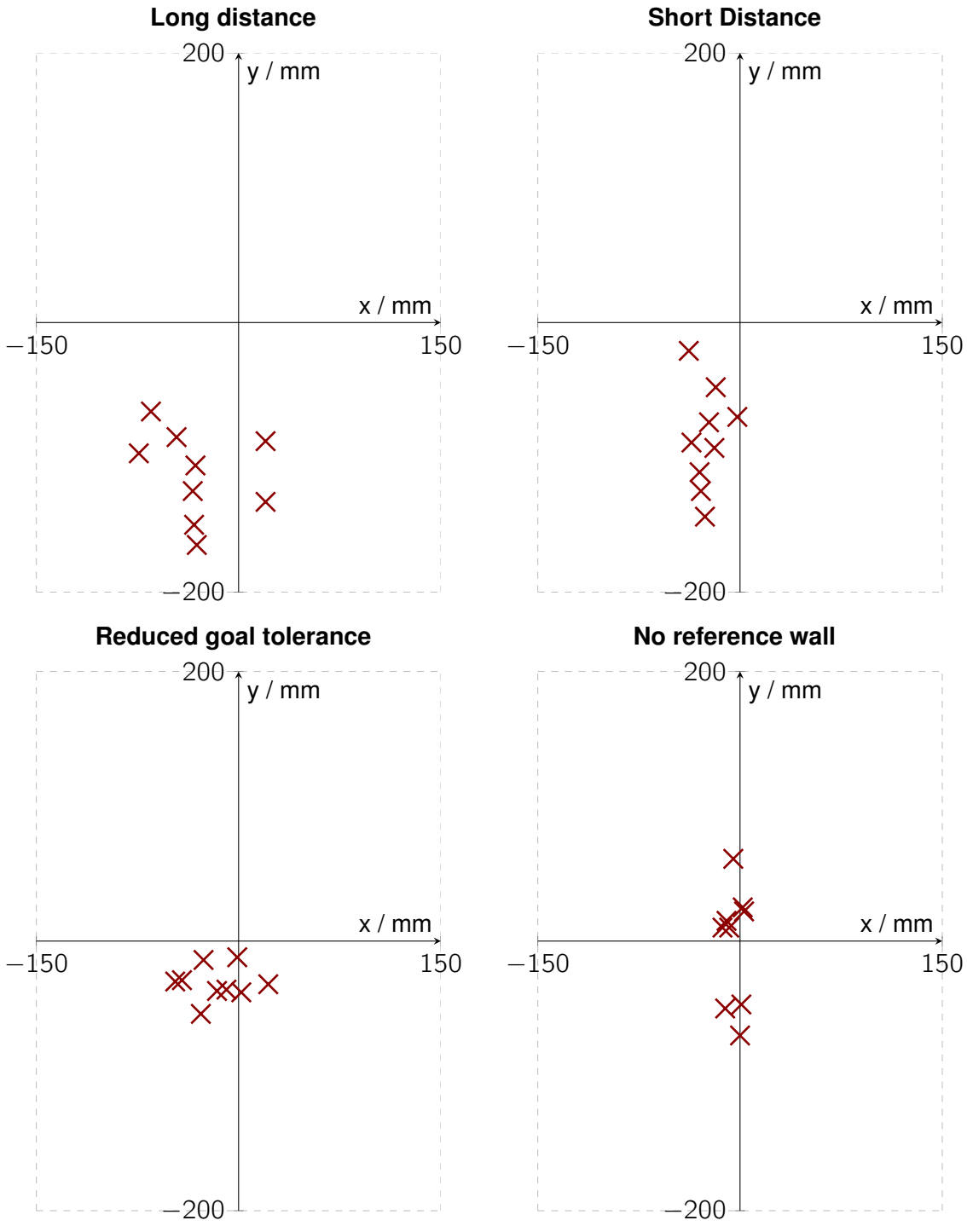

Figure 6.26: Target - Measurements

camera and a 3-D laser scanner sensor can provide range data with different resolutions on the environment within a horizontal and vertical range, determined by the field of view of the sensors and the number of laser beams emitted from the LiDAR. This range data is structured in the ROS *PointCloud* format, described in Listing 4.2 in Section 4 and visualized in Figure 6.27.

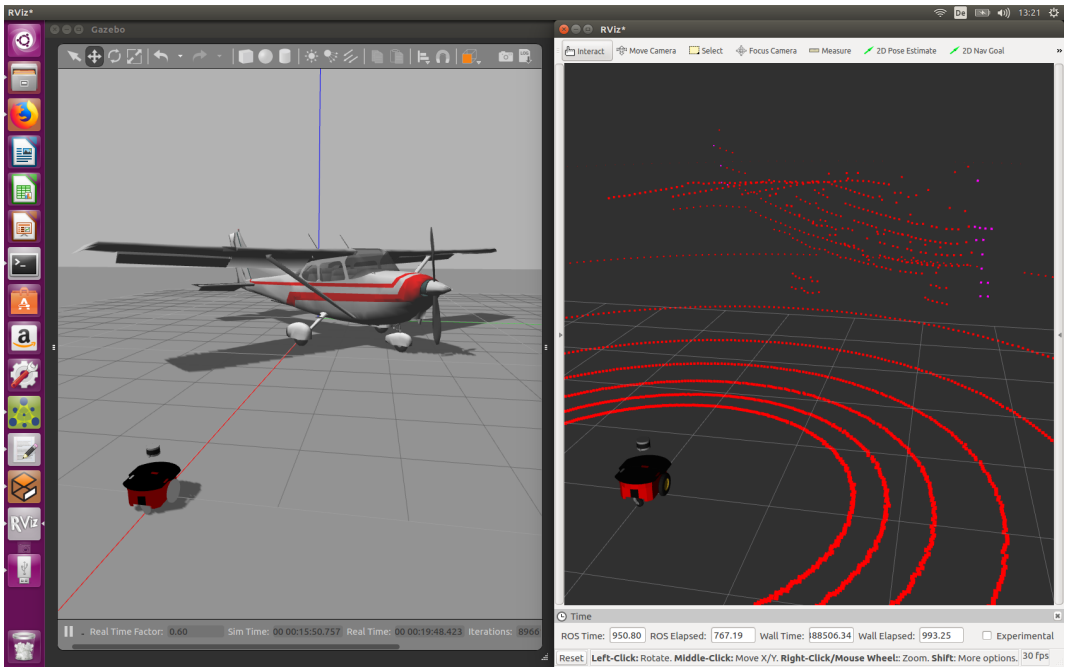

Figure 6.27: Elevated obstacles - PointCloud visualization in rviz

In this experiment, two objects are on the ideal trajectory between the robot and its destination. The first is at a height of 25cm from the ground and effectively represent an obstacle, as the height of the robot is approximately 60cm, while the second one stands at a height of 70cm from the ground, leaving enough space for the robot to pass underneath it. The 2-D *SICK TiM* LiDAR located at a height of 10cm from the ground can not perceive these obstacles, but the 3-D *Velodyne VLP-16* laser scanner located on top of the chassis can. By setting the parameters *min_obstacle_height* and *max_obstacle_height* in the navigation stack configuration to respectively 0cm and 60cm, it is possible to define a range in between which the detected objects are to be considered obstacles. During the successful test, the planner correctly perceives the object with a height of 25cm as an obstacle and calculates a trajectory around it and beneath the second object with a height of 70cm, to reach finally its destination, as shown in Figure 6.28

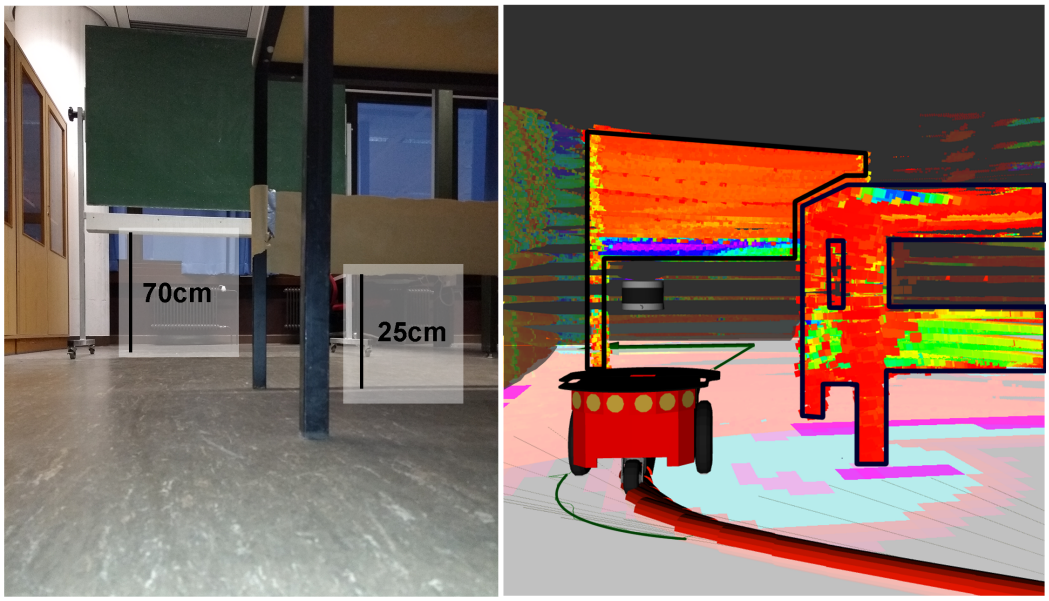

Figure 6.28: Elevated obstacles - Experiment

6.2.6 Labyrinth

In this experiment, the robot must reach a destination set in an unknown environment, represented by a maze. As the AMR advances through narrow corridors and tight corners and, as it stumbles upon dead ends, performs U-turns with little space for manoeuvres, the trajectory has to be planned meticulously. Moreover, the navigation algorithms should keep track of the previous routes and avoid travelling the second time to the same dead end. This is meant to test the soundness of different navigation algorithms as the robot travels through uncharted territory.

In this experiment the focus lies on two specific local planners, namely the *teb_local_planners* and the *dwa_local_planner*. More detail on these algorithms and on how they are integrated in the navigation stack is provided in Section 4.8. In this experiment, the following situations are relevant when evaluating the robustness of the local planners, as both directly influence the success ratio of the test:

- *Dead ends*: the robot should travel safely out of the dead end.
- *Corners*: the robot should turn the corners in a wide arc, avoiding driving too close to the edges and the walls.

The environment is first reproduced in the simulation environment. For this purpose, the maze shown in Figure 6.29 is used as the blueprint. The respective model is created in *Gazebo* and is shown in Figure 6.30.

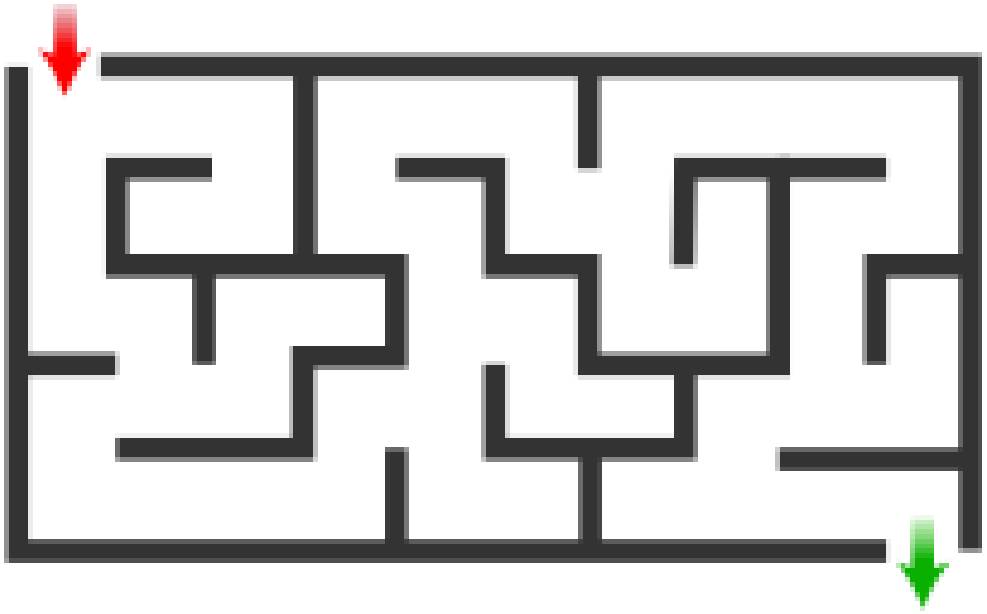

Figure 6.29: Labyrinth - Blueprint

By Jkwchui - Own work, CC0, https://commons.wikimedia.org/w/index.php?curid=15002205

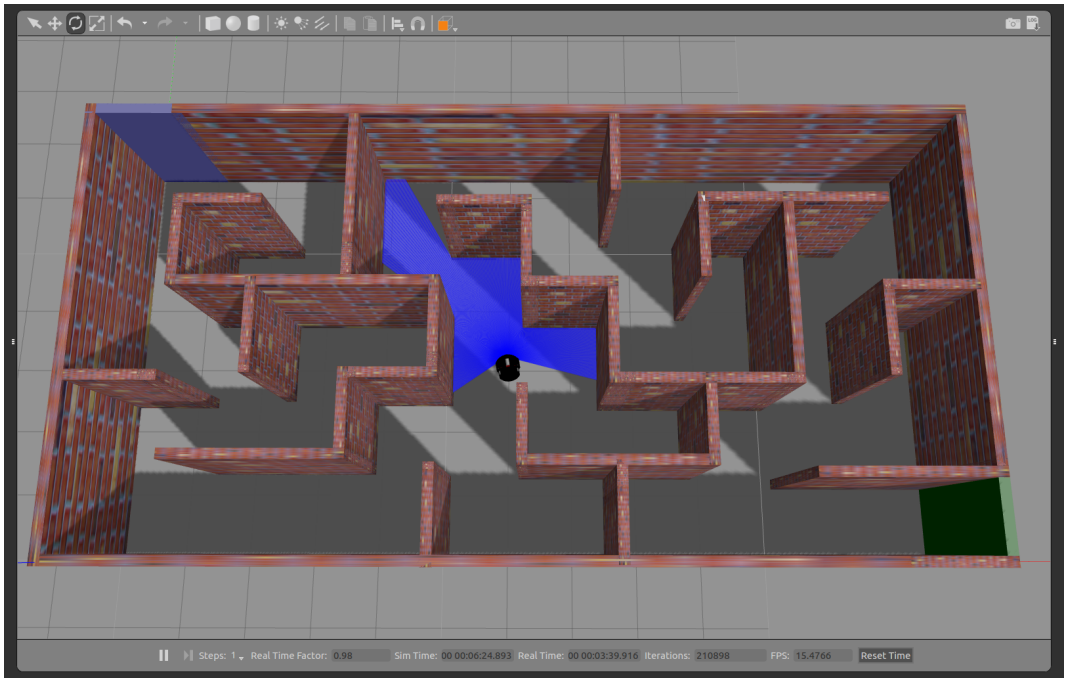

Figure 6.30: Labyrinth - Setup in the simulation

First the *dwa_local_planner* algorithm is tested. Also known as Dynamic Window Approach, this algorithm predicts the cost of a set of trajectories, based on the distance to obstacles, to the end goal and to the global plan, each weighted by a adjustable bias factor, and choose the least expensive one. The following behaviours are observed in the first runs:

- *Dead ends*: As the robot finds itself in a dead end, it often starts spinning in circles for a considerable amount of time, struggling to plan the right trajectory. When it eventually does and if the space allows it, it starts performing a U-turn, otherwise aborts the mission.
- *Turns*: When it turns corners, the robot does so barely avoiding touching edges and walls. This often results in situation where the robots stops centimetres short of walls and aborts the mission, since it perceives obstacles too close to perform any movement safely.

Two are the issues which appear to be causing the problem and prevent the robot from reaching its goal. First, the *dwa_local_planner* seems to favour shorter trajectories that cut corners instead of those that go safely around obstacles. This seems also to be causing the spinning behaviour, when the planner refuses to follow the correct path laid out by the global planner that, going backwards, seems to lead the robot further from the goal. To improve these behaviours, the biases need to be adjusted so that the trajectories are chosen with this order of priorities:

1. Close to the global plan.
2. Far from obstacles.
3. Closer to the goal.

The other issue is that the algorithm does not seem to perform in-place rotation and needs a minimum forward velocity in order to turn successfully. This is what causes the robot to abort the mission, when no space is available to perform the desired manoeuvre. Especially critical in this respect are U-turns. The partial solution in this case is to lower the minimum forward velocity that the robot is allowed to travel while turning. This shows however a critical shortcoming in the *dwa_local_planner* while navigating narrow spaces since it does not exploit the differential drive system of the robot, as the turning radius can be reduced but not zeroed.

After the adjustments above, the test is repeated multiple times and the robot eventually succeeds in reaching its goal as shown in Figure 6.31, with a success ratio of approximately 30% on average. Ultimately the ease at which the *dwa_local_planner* aborts the mission suggests that, as long as these issues are not resolved, this local planner might not be a sound choice in a robust AMR that needs to navigate narrow spaces.

Next, the test is repeated using the *teb_local_planner*, or Timed Elastic Band approach. According to the authors of the algorithm, this is a trajectory planner especially designed for differential and car-like robots (Ackermann steering) with limited turning radius, allowing for in-place rotations and manoeuvres while in reverse gear. The algorithms is thoroughly explained in the reference listed. [28] As a starting point, the default configuration is implemented and adjusted to match the basic robot parameters. In the resulting solution shown in Figure 6.32, the following behaviours are observed:

- *Dead ends*: As the robot perceives the dead end and a new trajectory is promptly recalculated, the robot safely executes a U-turn, by reversing the speed while steering to accelerate the manoeuvre. It does so accounting for the obstacles that may be laying behind it, avoiding collisions.
- *Turns*: When approaching the corners, the robot tends to keep a safe distance from the edge, but does so in an un-optimal way, stopping abruptly and performing small adjustments, before resuming driving.

A similar labyrinth was realized in the real environment, where the walls consisted of thick paper stretched between bottles representing the edges. In this way, it was possible to adjust the layout with ease. Figure 6.33 shows the setup, overlaid with start and end position and the solution of the labyrinth. It is worth noting that just before the end goal, the path presents a bottleneck where the legs of the table leave a space of approximately 60cm, barely enough for the robot to travel through.

First, the *dwa_local_planner* configuration that was successful in the simulation was implemented in the real environment. The performance proved however to be drastically different from the one experienced in the simulation, as the robot could not perform U-turns and was not nearly able to solve the maze. The issue could theoretically be addressed by carefully adjusting the settings of the planner to suit this specific scenario. In the opinion of the author however this particular algorithm is not suitable to navigate in narrow passages that may end in dead ends.

The *teb_local_planner* tested in the simulation was then used in the real environment. In the successful attempt shown in Figure 6.34, the circles represent the locations where the robot briefly stops to correct its trajectory when instead it should have been possible to continue in an uninterrupted movement. The motion of the robot seemed also to present some jitter. The main cause is that the frequency of 2Hz at which the algorithm is executed and at the which it sends the velocity commands to the base is not high enough.

Increasing this rate without lowering the trajectory optimization effort is however counterproductive, as the computational burden quickly exceeds the capacity of the system, leading to miscalculations and a delay in the response of the robot. On the other hand, excessively lowering the optimization decreases the chances of a feasible and efficient trajectory being

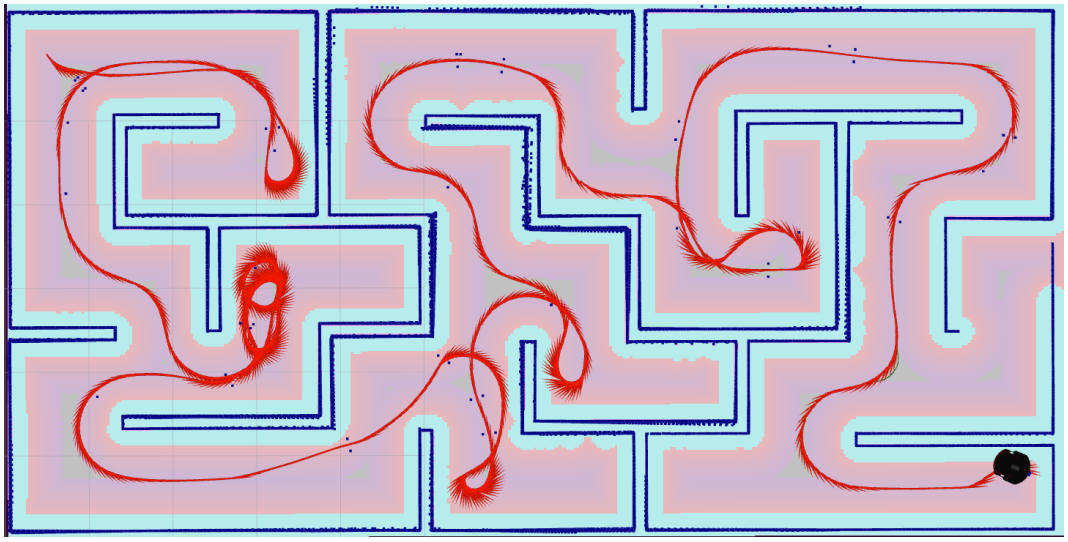

Figure 6.31: Labyrinth - DWA solution to the labyrinth

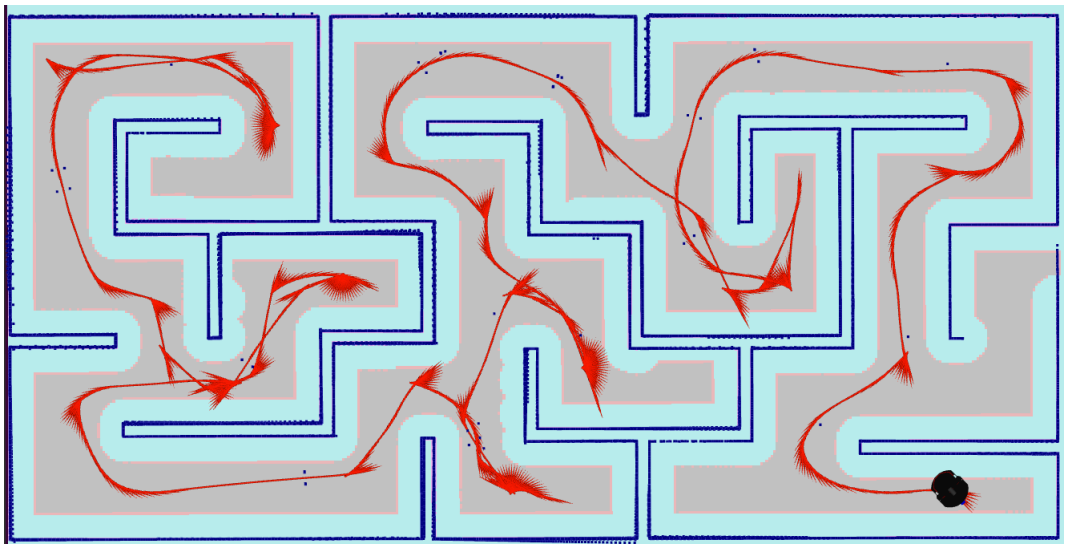

Figure 6.32: Labyrinth - TEB solution to the labyrinth

computed. The settings need to be balanced accurately, repeating the experiment multiple times until the performance is considered to be consistently satisfactory and the official package documentation describes how to best configure these settings. [29] Notably, it was observed that increasing the rate to 10Hz and decreasing the following parameters positively affected the response of the system:

- *max_number_classes*: Controls how many alternative trajectories are computed.
- *no_outer_iterations* and *no_inner_iterations*: Sets the number of times the trajectory computation and optimization loop iterations respectively are performed.

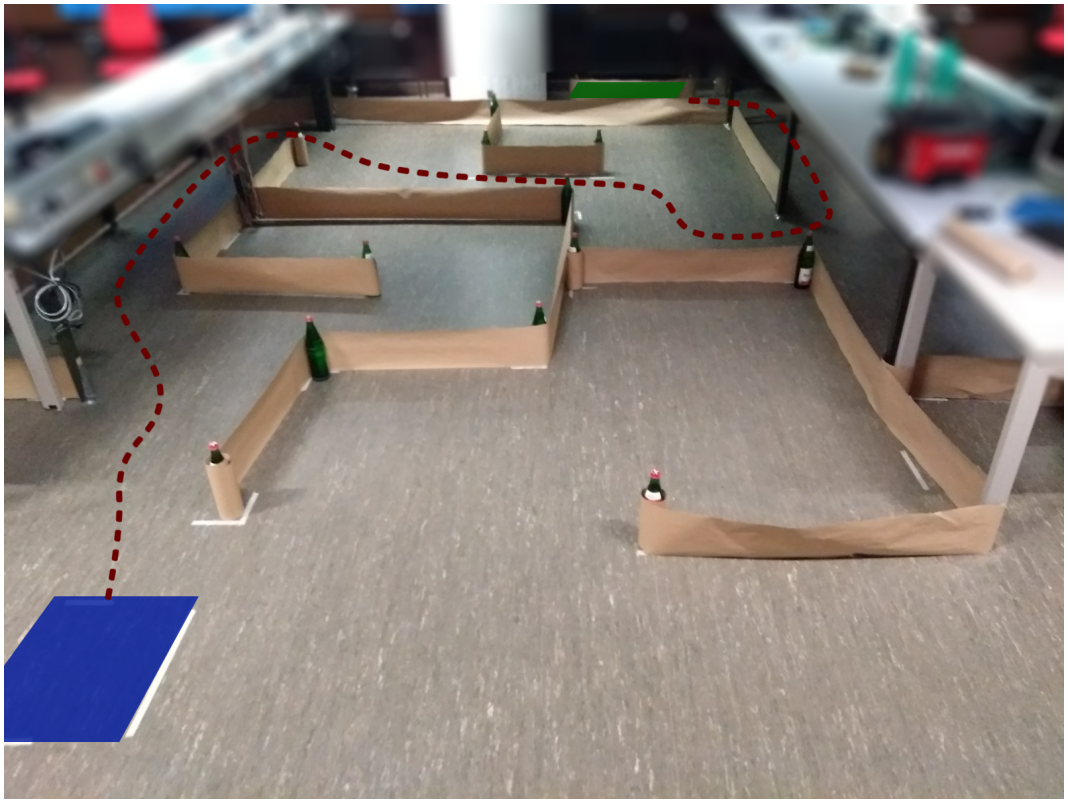

Figure 6.33: Labyrinth - Setup in the real environment

Figure 6.34 shows the result, where the number of corrections reduced drastically. The overall trajectory appears to be more efficient, as the turns are performed in a narrower trajectory.

To objectively evaluate the configuration, the following parameters are considered:

- *Success ratio*: How many times out of 10 tries the robot successfully solved the labyrinth.
- *Time elapsed*: The time in which the robot was successful.
- *Distance travelled*: The distance that the robot travelled from start to end, when successful.

The result shown in Table 6.3 indicates that in 20% of the cases the robot travelled successfully up until the bottleneck where the planner could not find a viable trajectory. This is despite the fact that the radius of the robot is set to 25cm, providing a theoretical 10cm large pathway to the objective. The root cause appears to be related to an unacceptably high cost of the trajectory that may travel too close to the obstacles and could perhaps be alleviated

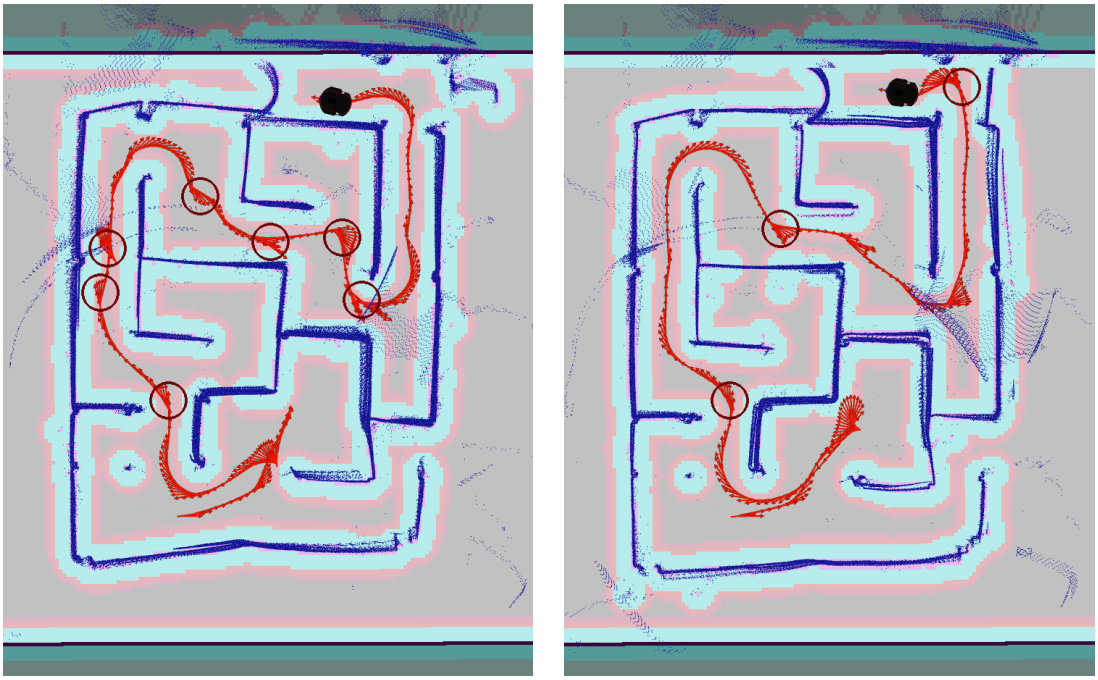

Figure 6.34: Labyrinth - teb_local_planner solution to the real labyrinth with standard (left) and with adjusted (right) settings

by further reducing the obstacle inflation radius and adjusting the biases of the planner to attribute a lower weight to the obstacle distance factor.

In another 20% of the tests, the robot stopped and aborted the mission as it encountered a dead-end and the planner failed to find an alternative path. The cause of this problem is not clear, however the issue seems to occur consistently when the goal lies within the boundaries of the local costmap highlighted in Figure 6.35.

The distances to the goal travelled in the successful 60% of the trials are in the range of 20m to 24m and the differences are due to different paths being chosen and small trajectories adjustments. Interesting however is that the time needed to complete the maze is on average 92s but ranges from 70s to 116s. This is due to an oscillatory behaviour of the planner that keeps switching between different possible trajectories, effectively stalling the robot as shown in Figure 6.35, where the trajectories that have been planned in a time of a few seconds are overlaid and shown in green. Eventually, an oscillation recovery behaviour is triggered and the robot chooses one, resuming its path. This is a known issue of the *teb_local_planner* and is also outlined in the official documentation.

In the previous experiments, the algorithms settings have been tuned in order to achieve a success ratio of 60%. It may very well be possible to achieve better results and even

overcome the critical issues outlined above by adjust some additional parameters. While the focus of this test was to evaluate the performance of the algorithms to determine their fundamental behaviour, the reader is encouraged to test new settings configurations.

teb_local_planner maze statistics			
N	Distance/m	Time/s	Note
1	-	-	Bottleneck reached
2	20.1	82.5	
3	23.0	112.5	
4	-	-	Bottleneck reached
5	20.2	73.0	
6	21.5	116.0	
7	24.0	114.5	
8	-	-	Bottleneck reached
9	21.0	96.0	
10	-	-	Lost
11	-	-	Bottleneck reached
12	20.6	88.5	
13	-	-	Lost
14	-	-	Lost
15	20.5	81.0	
16	20.0	90.0	
17	20.5	87.5	
18	19.7	70.0	
19	-	-	Lost
20	21.0	92.0	
	μ/m	μ/s	% to bottleneck % success
	21	92	80 60

Table 6.3: Labyrinth - Measurements

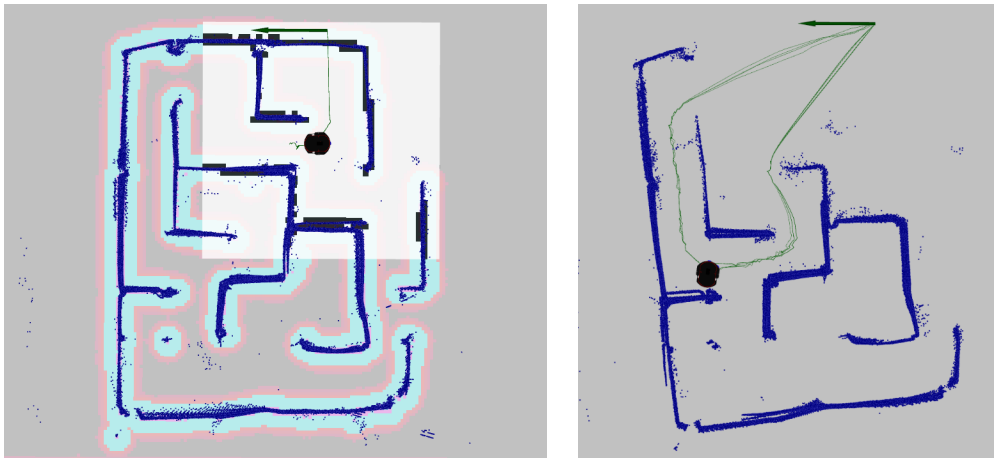

Figure 6.35: Labyrinth - teb_local_planner can not find a trajectory (left) and oscillates between two possible trajectories (right)

6.2.7 Summary

The observations from the previous experiments can be summed up in the following findings:

- While navigating in a location clustered with unmapped obstacles, the main challenge is to configure the navigation algorithms to perform in narrow spaces by lowering the maximum allowed speed and minimizing the minimum allowed distance to the obstacles.
- If the distance the robot needs to cover is significant and the obstacles are expected to be typically far apart from one another, it might be preferable to increase the maximum allowed speed, the minimum allowed distance to the obstacles, to make it safely go around them, and the obstacle detection range, to avoid rushed avoidance manoeuvres.
- The robot can reach an objective with an error measurable in less than ten centimeters from the set destination with reliable odometry information available, however excessively reducing the goal distance tolerance may lead to a situation where the robot keeps adjusting its position without ever reaching the goal.
- Laser scan data can be integrated in the localization process to compensate for cumulative odometry error, however its positive effect is limited in an environment lacking reference landmarks.

- The choice of the local planner for trajectory calculation should be carefully weighed based on the task since it has an impact on the feasibility of certain manoeuvres.
- Specifically, while the *Dynamic Window Approach* planner does not contemplate manoeuvres in reverse and struggles to drive the robot out of narrow dead-ends, the *Timed Elastic Band* planner is capable of doing so.

6.2.8 Open issues

Automatic pose recovery

In order to purposely navigate in the environment when a static map is used, the navigation algorithm *move_base* relies on a fairly accurate initial pose estimate. Without it, the robot may not be able to locate itself in the map or, if the initial pose is given but is not accurate, it may try to reach its set destination but will likely stop a considerable distance short from it. Setting the initial pose manually at the start of the operation can be a tedious process, especially in case the navigation starts from a point where no reference landmarks can be found nearby. Moreover, it may happen that, during prolonged navigation, the cumulative error from the odometry data might fool the robot into mistakenly estimating its position in the map. If the platform is to be deployed in a real-life scenario, it is unrealistic to assume someone would be able to adjust the position of the robot each time it loses track of its position. It is therefore a crucial issue to implement a pose recovery routine that is engaged whenever this situation happens. Judging from the experiments described in this paper, a 2-D LiDAR with limited range does not provide sufficient information to implement this feature. One possible solution would be to command the robot to start looking for a landmark which location is known and from where its position can be estimated. One option would be deploying radio beacons in the operations scenario, which the robot can sense and locate. Alternatively, a high resolution camera could spot QR codes or unique landmarks using Computer Vision features extraction algorithms. These algorithms have been proven effective to tackle this issue using several point cloud sensing devices such as 3-D LiDAR sensors or RGB-D cameras, as explained more in-depth in the listed resource. [30]

Autonomous mapping

As mentioned, to be able to determine a right trajectory that goes around static obstacles rather than blindly though them, the navigation stack needs to know the geography of the environment it is set to navigate in. This information is provided by the static map and section 4.7 explains how to produce one, either by adapting an available map or by sampling the environment while driving the robot in it and using a SLAM algorithm to process the data.

Both methods are performed manually and this can quickly become tedious, as this has to be repeated for each scenario where the robot is going to be deployed in. A semi-automatic approach is to implement an algorithm that command the robot to explore while avoiding obstacles and sampling the environment at the same time just in order to create a map. Another approach that combines mapping and navigation in a one step solution could be to let the robot increasingly build the map as it navigates towards specific destinations. In the very first run, the geography of the environment is completely unknown and the robot may take different wrong turns before finally reaching its goal, but as the number of runs or missions increases, the device keeps learning as it explores new parts of the maps, until eventually the whole map is completely known.

7 Conclusion and future developments

In this paper two fundamental tasks have been identified as requirements, which an Autonomous Mobile Robot should be able to perform, namely mapping and navigating autonomously an environment. Specifically, to efficiently and safely reach a set destination, an accurate map of the static obstacles should be available and the most adequate trajectories should be calculated, which promptly account for dynamic obstacles and overcome challenging sections in the planned path. These tasks represent the foundations upon which more advanced features may subsequently be implemented. In order to achieve the requirements, five milestones have been identified and reached.

The development environment was configured using ROS as the software framework of choice which, compared to other alternatives, proved to be the most suitable to robustly implement both the tasks mentioned above with relative ease. Particularly, the SLAM algorithms and navigation stack integrated in the framework have the characteristics to effectively perform mapping and navigation. Additionally, the framework can operate on multiple remote devices, including Android devices.

The sensors were integrated as intended in both a real and simulated environment. The *Pioneer P3-DX* was identified as a suitable robot platform for the purpose, as its integrated rotary encoders are able to provide reliable odometry information when the relative parameters are correctly calibrated. Moreover, the layout of its chassis allows to comfortably place a variety of sensors. In particular, a *SICK TiM310 S01* 2-D LiDAR placed in front can successfully map and navigate the environment, detecting obstacles at the exact height of the laser beam, while a better option is to place a 3-D LiDAR, like the *Velodyne VLP-16*, in an elevated position with respect to the base of the robot, in order to detect obstacles on a 360° horizontal radius and located at different heights.

An Android device was successfully used as remote controller by means of a virtual joystick application. This was used to manually drive the robot to explore the environment while recording the sensor output, to create maps from consistent data.

Mapping was achieved with the *gmapping* SLAM algorithm, which relies on both laser range and odometry data. After calibrating the odometry and adjusting the algorithms settings, the resulting 2-D map was accurate, with correct dimensions and geometry.

Autonomous navigation was implemented with the *Timed Elastic Band* approach. The robot was able to navigate an unknown environment characterized by narrow spaces and corners and dead ends, finally reaching its destination without colliding with obstacles located at different heights along its way.

Overall, the system successfully performed a variety of tests related to mapping and autonomous navigation and can thus be considered a robust research platform, one that can be further enhanced by integrating new sensors, like camera or microphones for video and audio related tasks, and manipulators. New applications can as well be developed and installed on a remote device to have more advanced and customizable interactions with the robot.

Appendices

This Bachelor Thesis contains an appendix of the following contents on a CD. This Appendix is deposited with Prof. Dr. Pareigis.

Appendix A: ROS packages

- *p3dx_gazebo*: configuration, models and launch files to operate the robot in the simulation environment
- *p3dx_real*: configuration and launch files to operate the robot in the real environment
- *amr-ros-config*: models of the real Pioneer 3-DX robot; developed by third parties [9]
- *rosaria*: driver to operate the Pioneer 3-DX robot; developed by third parties [12]
- *sick_tim*: drivers and models of SICK TiM laser scanners; developed by third parties [15]
- *velodyne*: drivers and models of Velodyne laser scanners; developed by third parties [31]

Appendix B: bag files

- *exploration.bag*: recording of the exploration mission described in Section 6.1
- *labyrinth.bag*: recording of the labyrinth experiment described in Section 6.2.6

Bibliography

[1] A. Hentout, A. Maoudj, and B. Bouzouia, “A survey of development frameworks for robotics,” in *2016 8th International Conference on Modelling, Identification and Control (ICMIC)*, Nov 2016, pp. 67–72.

[2] K. Takaya, T. Asai, V. Kroumov, and F. Smarandache, “Simulation environment for mobile robots testing using ROS and Gazebo,” in *2016 20th International Conference on System Theory, Control and Computing (ICSTCC)*, Oct 2016, pp. 96–101.

[3] M. Köseoğlu, O. M. Çelik, and O. Pektaş, “Design of an Autonomous Mobile Robot based on ROS,” in *2017 International Artificial Intelligence and Data Processing Symposium (IDAP)*, Sept 2017, pp. 1–5.

[4] Q. Xu, J. Zhao, C. Zhang, and F. He, “Design and implementation of an ROS based autonomous navigation system,” in *2015 IEEE International Conference on Mechatronics and Automation (ICMA)*, Aug 2015, pp. 2220–2225.

[5] A. Cherubini, F. Spindler, and F. Chaumette, “Autonomous Visual Navigation and Laser-Based Moving Obstacle Avoidance,” *IEEE Transactions on Intelligent Transportation Systems*, vol. 15, no. 5, pp. 2101–2110, Oct 2014.

[6] A. Huletski and D. Kartashov, “A SLAM Research Framework for ROS,” in *Proceedings of the 12th Central and Eastern European Software Engineering Conference in Russia*, ser. CEE-SECR ’16. New York, NY, USA: ACM, 2016, pp. 12:1–12:6. [Online]. Available: http://doi.acm.org/10.1145/3022211.3022223

[7] A. T. Angonese and P. F. F. Rosa, “Multiple people detection and identification system integrated with a dynamic simultaneous localization and mapping system for an autonomous mobile robotic platform,” in *2017 International Conference on Military Technologies (ICMT)*, May 2017, pp. 779–786.

[8] T. Foote, E. Marder-Eppstein, and W. Meeussen, “ROS tf package - Official documentation.” [Online]. Available: http://wiki.ros.org/tf

[9] OMRON Adept MobileRobots, “URDF, launch files, and other ROS configuration for AMR robots.” [Online]. Available: https://github.com/MobileRobots/amr-ros-config

[10] S. Amoako-Frimpong, “Running the Pioneer 3DX in Gazebo and ROS Kinetic - Part I.” [Online]. Available: https://afsyaw.wordpress.com/2017/01/12/running-the-pioneer-3dx-in-gazebo-and-ros-kinetic/

[11] S. Amoako-Frimpong, “Running the Pioneer 3DX in Gazebo and ROS Kinetic - Part II.” [Online]. Available: https://afsyaw.wordpress.com/2017/01/14/running-the-pioneer-3dx-in-gazebo-and-ros-kinetic-part-ii/

[12] S. Juric-Kavelj, “ROSARIA package - Official documentation.” [Online]. Available: http://wiki.ros.org/ROSARIA

[13] Z. Kaiyu, “ROS navigation tuning guide - Official documentation.” [Online]. Available: http://wiki.ros.org/navigation/Tutorials/Navigation%20Tuning%20Guide

[14] SICK AG, “TIM310-1030000S01 - Reference Page.” [Online]. Available: https://www.sick.com/de/en/detection-and-ranging-solutions/2d-lidar-sensors/tim3xx/tim310-1030000s01/p/p297950

[15] University of Osnabrück, “A ROS driver for the SICK TiM series of laser scanners.” [Online]. Available: https://github.com/uos/sick_tim

[16] T. Ogura, “ROS OpenCV camera driver.” [Online]. Available: http://wiki.ros.org/cv_camera

[17] J. Bowman and P. Mihelich, “ROS calibration of monocular or stereo cameras using a checkerboard calibration.” [Online]. Available: http://wiki.ros.org/camera_calibration

[18] J. Cerruti, “ROS android package - Official documentation.” [Online]. Available: http://wiki.ros.org/android

[19] G. Grisetti, C. Stachniss, and W. Burgard, “ROS GMapping - External documentation.” [Online]. Available: http://openslam.org/gmapping.html

[20] B. Gerkey, “ROS GMapping - Official documentation.” [Online]. Available: http://wiki.ros.org/gmapping

[21] E. Marder-Eppstein, “ROS 2D navigation stack - Official documentation.” [Online]. Available: http://wiki.ros.org/navigation

[22] E. Marder-Eppstein, “ROS move base - Official documentation.” [Online]. Available: http://wiki.ros.org/move_base

[23] D. V. Lu, “ROS layered costmaps configuration - Tutorial.” [Online]. Available: http://wiki.ros.org/costmap_2d/Tutorials/Configuring%20Layered%20Costmaps

[24] B. P. Gerkey, “ROS amcl - Official documentation.” [Online]. Available: http://wiki.ros.org/amcl

[25] D. Fox, W. Burgard, and S. Thrun, "The dynamic window approach to collision avoidance," *IEEE Robotics Automation Magazine*, vol. 4, no. 1, pp. 23–33, Mar 1997.

[26] K. Zheng, "ROS Navigation Tuning Guide." [Online]. Available: http://kaiyuzheng.me/documents/navguide.pdf

[27] C. Rösmann, F. Hoffmann, and T. Bertram, "Timed-Elastic-Bands for time-optimal point-to-point nonlinear model predictive control," in *2015 European Control Conference (ECC)*, July 2015, pp. 3352–3357.

[28] A. Koubaa, Ed., *Robot Operating System (ROS): The Complete Reference (Volume 2)*, ser. Studies in Computational Intelligence. Cham: Springer, 2017, vol. 707.

[29] C. Rösmann, "ROS teb local planner - Official documentation." [Online]. Available: http://wiki.ros.org/teb_local_planner

[30] C. Costa, "Robot Self-Localization in Dynamic Environments," 01 2015.

[31] J. O'Quin, "Basic ROS support for the Velodyne 3D LIDARs." [Online]. Available: https://github.com/ros-drivers/velodyne

Declaration

I declare within the meaning of section 25(4) of the Ex-amination and Study Regulations of the International De-gree Course Information Engineering that: this Bachelor report has been completed by myself inde-pendently without outside help and only the defined sources and study aids were used. Sections that reflect the thoughts or works of others are made known through the definition of sources.

Hamburg, April 12, 2018

City, Date

sign