Bachelorarbeit

ROS Navigation Stack On A Loomo Segway robot

Verfasser
Maximilian [redacted]

angestrebter akademischer Grad
Bachelor of Science (BSc)

Vienna/Munich, 2018

Studienkennzahl lt. Studienblatt:	[redacted]
Fachrichtung:	Informatik - Medieninformatik
Betreuer:	Univ.-Prof. Dipl.-Ing. Dr.Wolfgang Klas

Abstract

The navigation stack of the ROS framework was implemented for the Segway robot Loomo in the course of this thesis. The robot was connected to the ROS system with an Android app. Different setups of achieving navigation in a lively office space were explored. To reach this goal multiple components were used including a laser scanner and a Raspberry Pi. The stereo depth sensing vision of the robot is compared to the standard laser scanner. Crucial parts of the system setup and the theory behind the different concepts of path planning, simultaneous localization and mapping (SLAM) and Monte Carlo localization (MCL) that were used are discussed in this thesis. In the end fast and stable navigation including obstacle avoidance was reached.

Index terms— *ROS, navigation, SLAM, MCL, Segway robot, stereo vision, laser scan*

Contents

1 Introduction

The following section describes the motivation of this work. It contains the problem description and the goals of this work. The process of work and a summary of the main contributions are included as well.

1.1 Motivation

Working with robots is no easy task and the barrier before getting started can appear insurmountable.

> Writing software for robots is difficult, particularly as the scale and scope of robotics continues to grow. Different types of robots can have wildly varying hardware, making code reuse nontrivial. [1]

The purpose of ROS is to decouple and abstract the implementation of robotic software so one does not have to reinvent the wheel.

The motivation of this work is to demonstrate and document how the framework achieves its goals on the basis of implementing a set of use-cases on a robot.

For this work Loomo [2], a self-balancing Segway robot was used. With their robot, Segway Robotics [3] offered a relatively affordable fully functional robot companion. A more detailed description can be found in the *Design* section of this document (see section 3) . The robot was chosen because it is accesible and fully configured. This shall demonstrate the decoupling of ROS.

1.2 Problem Statement and Goals

The robot purpose focused on in this works is, to act as a personal assistant in an office environment. Its tasks involve recognising and greeting people, answering questions and show guests around.

For this work the third scenario was chosen: Navigation!

Loomo is a very compact but mobile robot. This provides some challenges for navigation especially in a dynamic environment like an office space.

Formulated in a user story a new person in the office should be guided to the required meeting room by the robot. Defined as use-cases this involves

- Remote control

  For safety reasons a remote control is included in the use cases. This enables to manually manoeuvre the robot out of potentially dangerous areas.

- Localization

  To safely and accurately navigate, the robot has to know its own position.

- Autonomous navigation

  The robot should be able to navigate by itself to a given goal.

- Obstacle avoidance

  The robot should avoid static and dynamic obstacles

To test if the defined goals were reached, they will be demonstrated with a user scenario as formulated in the user story. The measurement to determine if the requirements are fulfilled will be visual comparison of how the robot performed. The criteria and the outcome can be found in the *Evaluation* section (see section 5).

Usually robots are heavily loaded with instruments and sensors and navigate rather slowly. Loomo is a fast compact robot which opens new possibilities in the office space. With the help of ROS this work aims to achieve fast reliable navigation for a robotic office assistant.

1.3 Process of Work

In order to achieve those goals one has to look into the subject of ROS in general and how it works. Although the framework eases requirements of working with robots, without some basic knowledge one will get lost. This thesis will try to give a comprehensive summary of what ROS is and how to work with it and cover some basic robotic knowledge as well. The *Definition of Terms* section (see section 1.6) introduces some of the used vocabulary.

To achieve the goals of this work multiple components were used. On the one hand there is the robot which had to be set up and optimised to be connected with ROS. On the other hand we have the ROS system and its nodes. To guarantee the functionality and achieve comparable results, additional components had to be used, for example the laser scanner and the Raspberry Pi. Android apps were included to provide user interfaces. Their functionality is described in the *Design* section of this document (see section 3) and the process of setting them up can be found in the *Implementation* part (see section 4).

Programming languages that were used include Java for the Android apps on Loomo and on the smartphone for the user interface. The used nodes were written in Python or C++. But any coding of nodes for this work was done in C++. The rest of the work consisted in setting up the components, deciding the system architecture, mapping the according topics and tweaking the parameters.

1.4 Main Contribution

In the course of this work the Android app for Loomo which provides the connection to ROS via ROSJava was optimized.

A prototypical construction to mount a laser scanner on Loomo was built and the Raspberry Pi set up with an Ubuntu system to publish the laser scan data.

A combination of ROS nodes were used to implement the functionality of the navigational tasks. Some of those nodes are used unchanged as provided by the ROS community, some were altered and some were written specifically for this work. To enable the components of the system to work with each other they had to be set up to communicate over the according topics. The decisions made are described in the *Design* part of this Document (see section 3).

To demonstrate the possibilities of ROS in mobile environments two Android apps were provided to remotely control the robot and give it goals to navigate to.

The nodes that provide the main functionality for the navigation use a range of parameters to adjust the program to the specific attributes of the used robot.

To enable smooth navigation those parameters had to be tuned to the needs of the robot used in this work. In order to set them correctly one has to understand what exactly they do and then verify the outcome via testing. The meaning of the tuned parameters is discussed in the *Implementation* part of this Document (see section 5 and the outcome of the experiments can be found in the *Evaluation* (see section 5).

1.5 ROS

The following section will try to give a comprehensive summary of ROS.

1.5.1 Overview

ROS is supposed to make working with robots a little easier and more accessible.

As described by the contributors to ROS from Willow Garage, Stanford University and University of Southern California [1], ROS tries to fulfill a few philosophical goals that lead to its design and implementation:

- Peer-to-peer
- Toolsbased
- Multilingual
- Thin
- Free and Open-source

One big benefit of ROS is that it is free and open source. There is already a lot of material provided by the ROS community and ready to be used. Either on GitHub or on the ROS wiki [4] one can find several ROS packages to chose from. To each package there should be corresponding ROS wiki pages, a readme file or a link to an external source. This should provide everything one needs to know to get started. One can install those packages either over the package manager or from source by cloning the git repository.

1.5.2 The Framework

ROS described by Steve Cousins in his article about ROS topics:

> A ROS system is a computation graph consisting of a set of nodes communicating with one another over edges. The communication consists of messages that are organized by topics. [5]

Additionally for non-broadcast synchronous tasks that do not fit into this publish- subscribe pattern, services were added which can be called with 2 messages, a request and a response.

The nodes must have unique names and the messages are broadcasted over a declared topic with a defined message type. This is where the peer-to-peer aspect of ROS comes into play. Each node can be run on any device within the ROS environment. They just have to share the information on the ROS Master which is identified by name or IP address. This can save resources and facilitates decoupling and switching of components.

Each node can be switched out easily by turning it off and replacing it with another one that subscribes and publishes to the same topics. As in robotics a system often has many components this can be very useful for debugging and testing.

Additionally ROS is language independent. The nodes share information with strictly defined message types which can be wrapped in any programming language one desires.

To optimise working with ROS even more, a range of tools is provided by the framework. A few of them will be covered in the *Implementation* section (see section 4).

To demonstrate the power of ROS one best looks at an example. For this work the task of navigation was chosen. Lets imagine one is at the point when the logic is done and everything is set up to provide a path for the robot. But now one needs to substitute the robot for some reason to another one that has three wheels instead of four. The movement abilities of the robot change completely. But with ROS this does not provide a problem. The navigation system will work for any robot that accepts the ROS messages in form of a position and orientation. The robots job is to move in a way to achieve this position. How it achieves it does not matter for the system. Additionally it will work for any number of robots of any kind with just small adjustments. One can refer to the *Design* section of this work (see section 3) where the substitution of certain components in our system is described to get further proof of this concept.

1.6 Definition of Terms

The following section will cover the various terms used in this work and give further description.

- Nodes.

  The software parts of ROS where all the computation takes place. They are language independent but most commonly written in Python or C++.

- Topics

  Nodes communicate over topics with a specified name. Each topic has a specified message type. One can rename topics and assign them namespaces. There are some naming conventions and the topics are usually mapped to them by default. The default values can be found in the according ROS wiki page of the package.

- Remapping.

  Renaming topics or assigning them namespaces in ROS is called remapping. It enables powerful options that can affect the design of the system. Especially when the components get more complex or when working with multiple robots remapping becomes a valuable tool.

- Odometry.

  Odometry is used to determine the position of the robot [6]. There are two ways of doing that: relative odometry and absolute odometry. Usually robots have some form of relative odometry. Wheeled robots like Loomo have sensors that record the rotation of the motors and calculate the distance travelled. But this is prone to errors. If the wheels are not exactly the same size or the ground is slippery, the measurements can be off. Absolute odometry calculates the position on the base of external sources like landmarks, beacons, maps, or GPS.

- Tf.

  Tf is short for transformation frames. It is a tree of coordinate frames which describe the position of the robots most important parts. If a sensor, for example a camera, records data the system needs to know the position of the sensor relative to the robots base which is the very center of the robot to translate the data correctly. Therefore, a transformation tree is used. It is used to store the position but also the rotation of the different robot elements. If they can move independently say there are joints or hinges, the information of the transformed position should be saved here as well.

- Frame ID.

  The coordinate frames of the Tf are labelled with frame IDs to distinguish them. Base_frame or base_link are often referred to as the robots base, the very central point of the robot. All movement commands get sent to this point of the robot from which every coordinate in the transformation tree origins.

- Quaternion.

  To describe the orientation of a robot quaternions are often used [7] as they miss some quirks other methods have. For example opposed to the pitch-yaw-roll system that is also commonly used, it does not matter in which order one applies the quaternions rotation.

- Footprint.

  The robots footprint is a description of the outlines of the robot at ground level, the same level where base_link is located. It is either defined as a radius for a round robot or as a set of coordinates describing the polygon that outlines the robots base.

2 Related Work

In the following section related work and similar approaches to this one will be discussed. As ROS is open-source there are many solutions to the different problems in robotics. A few will be mentioned and compared to this work at this point.

Navigation is a common problem in robotics. The challenge of letting the robot navigate in the uncertain real world is described very well by Sebastian Thrun [8] in his article about probabilistic robotics. He summarizes a few of the practises most commonly used in robotics and navigation.

The navigation stack [9] of ROS is a collection of those practises and this thesis follows them closely. The approaches used in this work were chosen because they are well documented and commonly used. This enabled the robot to reach a state of basic navigation. From this point more experiments could be concluded and compared.

A standard work about navigation is the office marathon [10] by Willow Garage. They trained their robot to navigate through an office space while avoiding complex objects and fitting through narrow doorways. The paper gives a good overview of the basic implementation of navigation and they have interesting approaches to process their sensor data. For example the random sample consensus filter, RANSAC [11] they used might be applicable for this work as well and could be used for optimising the sensor data as mentioned in the *Future Work* section (see section 6). The difference to this work is that they used a tilting laser scanner to explore the 3 dimensional space. The robot for this thesis is equipped with a stereo depth sensing camera which provides 480x360 pixel of depth information at 30fps. This gives him a speed advantage as there is no tilting needed to scan the 3-dimensional space. In comparison the RP2 robot used by willowgarage needs to stop and scan difficult areas first. Other examples of navigation implementation on different robots that relate to this work include a Pioneer 3-DX robot by the University of Graz [12] and a TurtleBot [13] with a kinect camera similar to our stereo camera.

ROS provides tools like gazebo [14] and stage [15] which can simulate robots and generate virtual environments for them, which can be useful for research purposes when the purchase of a real robot is not possible. Before working with a real robot it can also be valuable to theoretically implement a use case first, like it was done in the work of Pajaziti and Avdullahu [16] that covers navigation with a simulated TurtleBot.

As a reference for the comparison of our robots sensor data in the *Evaluation* part (see section 5) the work of Pérez [17] was used that compares vision systems to laser scanners.

The work of Lin [18] provides a similar implementation of laser scans and depth images. The difference is that they fused the information of the depth camera and the laser scanner. The reasoning behind that is that laser scans provide exact mapping and localization but can miss complex objects as they only scan a 2D Plane, as discussed in the *Evaluation* part of this document (see section 5). So they fuse the depth data of the camera into their environment and create a more complex map. With the setup used in our work this is not necessary. The laser scanner is used for localization and mapping and the depth camera only for obstacle avoidance. What the camera detects has no impact on the navigational task except detecting obstacles. This is beneficial as the strength of both sensors are unhindered by the weaknesses of the other. Also the two sensors do not have to be rectified to each other which saves a little bit of effort.

3 Design

In the following sections the system architecture will be justified and the functionalities of each component explained.

3.1 Devices

The system consists of several components including hardware devices and software components. In the following section first the devices, then the components will be discussed.

3.1.1 The Robot

Loomo (cd. Figure 1) is a very mobile Segway robot. Robots are usually very expensive, and complicated to set up. There are a few entry level friendly models for example the TurtleBot [19] which is often used as an example for ROS programs. But as the name suggests the TurtleBot is not very mobile. Many other accessible robots have this in common. Loomo in comparison is self-balancing and moves very fast. It can reach up to 8 km/h in robot mode.

With its Segway technology Loomo can even recover from uneven terrain and bumps which is an ability most common robots lack. Its compact size benefit it to be a practical robot companion as well. It is equipped with various sensors and an Android operating system.

There is no need to do any previously configuration. As soon as one installs the provided app, Loomo is ready to be operated with ROS. Said app acts as a wrapper for ROS. It was adapted from the android_core package which bases on ROSJava. ROSJava does not yet provide the full range of the ROS framework but it serves the purpose of connecting the robot to the rest of the system.

In our case this is a beneficial system architecture as the load of the computing takes place on an external source and one is not dependant on the computing power of the robot as demonstrated by Barbosa [20]. Therefore, not the complete navigation stack is implemented on Loomo. The robot only publishes the required sensor topics to connect to the rest of the ROS system which is deployed on a different machine and listens for movement commands coming from

Figure 1: Loomo Segway robot

it. That also means that the time-frame where the sensor data can be previously processed and filtered is very small. As the data first has to travel over Wi-Fi there remains less time to filter it, because it would get outdated and unusable by the ROS nodes.

With the app from Segway Robotics and some adjustments the robot was set up to publish its sensor topics and listen on a separate topic for movement commands. In this work only the odometry data, the coordinate transformation tree and the depth images were used as sensor data from Loomo. The robot is eligible to provide more data but it remains unused in this context to avoid bottleneck problems when publishing too much because one has to communicate over Wi-Fi.

When the robot is set up the according ROS nodes can be started with the launch files provided. Separate approaches to reach the goal of navigation were explored in this work. The dividing factor were the sensor data from Loomo which is used to draw the map. Loomo is equipped with an Intel RealSense ZR300 camera which provides stereo depth images.

As discussed by Pérez [17] vision systems can obtain comparable results to laser scanners. The latter provide more exact and reliable data but are also more expensive especially when more than a 2D plane is to be scanned. When the robot is prone to bow forward when accelerating or backwards when braking the laser data can be distorted. Also objects lower or higher than the scanner can be overseen and , therefore, result in a collision. Depth images provide much more versatile data and can be useful for very mobile robots like Loomo.

In the following sections the benefits and disadvantages of those two systems are discussed. Both were included in the testing of this work and will be compared in the Evaluation section of this work.

3.1.2 The Stereoscopic Depth Sensing Camera

The Microsoft Kinect [21] is a popular device for robotic projects as it provides reasonable depth data and is a lot cheaper than a laser scanner. But as Microsoft has stopped their support for this product, Intel tries to fill this void. With their RealSense products they try to cater for the specific requirements one has for camera sensors in robotics.

According to its datasheet [22] the Intel RealSense ZR300 camera sends depth images with a resolution of 480x360 at 30hz. Additionally it provides 1080p RGB color stream, a Fisheye optical sensor and an inertial measurement unit(IMU). Those are all very useful additional sensor sources for robotic use cases. For this work only the depth and IMU information are used.

To calculate the depth values the ZR300 uses stereo vision with two infrared cameras and a laser projector. For each pixel a depth value is calculated. It can detect complex obstacles and humans are visible in the resulting images which is useful for obstacle avoidance in office areas. More information on the quality of the data and how it performed in our system can be found in the *Evaluation* part of this Document (see section 5).

3.1.3 The Laser Scanner

In comparison to the stereo camera a RPLIDAR A1 Laser scanner was mounted on Loomo. Laser scanners are commonly used for map drawing and localization in robotics. They provide very exact and stable data.

The scanner used provided depth information at 8hz up to 8 meters with a resolution of 1% of the scanned distance. Meaning that in a full rotation it provides 360 points of depth data.

The scanner is not naturally implemented in Loomo, therefore, a prototypical construction was used to mount the laser scanner on the robot.

3.1.4 Android Smartphone

To prove that ROS is applicable for an accessible mobile robot with minimal setup this work provides two Android apps that demonstrate how the robot can be controlled with a standard Android smartphone. The sample apps provide minimal graphical user interfaces that are not fully developed and only give a glimpse of what might be possible with ROSJava on Android.

The remote control app *Teleop* is provided by the ROSJava core and does not need any modification to work with Loomo. This is further proof of the decoupling of ROS. *PubsubTutorial* was modified for this work to provide an interface to publish goals for Loomo to navigate to.

3.1.5 Raspberry Pi 3

In case the sensor data from Loomo should not be enough for the navigational task, an RPLIDAR A1 laser scanner from Slamtec was used [23].

Loomo has a hardware extension bay and one could hook up the laser scanner there. But this would require more setup on a deeper hardware level and it would not compress the complexity as the data has to travel over Wi-Fi to the ROS Master in any case. So for this work another path was chosen.

The laser scanner was connected to a Raspberry Pi 3 [24] with a light Ubuntu system installed. On it ROS and the rplidar_ros node are installed. With this setup it publishes the laser scans over Wi-Fi. The hardware extension bay of Loomo does not provide enough power to supply both the scanner and the Raspberry Pi, therefore, an additional power source is required, in this case a power bank with at least 1 ampere output.

3.2 Components

In the following sections the functionality of the different software nodes of the ROS system and the reasoning of why they were chosen will be discussed.

3.2.1 Slam_gmapping

Before a robot can localize itself in a certain environment a map is needed. As the robot senses the world differently than humans do, a simple floor plan is not enough for a robot to navigate. Therefore, a map has to be drawn by the robots sensors or similar sources.

The problem of drawing the map and simultaneously knowing the position of the robot is often referred to as SLAM, Simultaneous Localization and Mapping. In this work the node gmapping as described on its ROS wiki page [25] is used to draw a map from laser scan data (cd. Figure 2) .

The SLAM problem is solved with Rao-Blackwellized particle filters [27]. The first provided scan is used as an initial map. Then with each timestep, the odometry data is compared with the scan sources. A particle cloud is created and the particle with the highest scan matching is the most probable location. Then the map is drawn according to this location.

The map is an occupancy grid where each cell has a value for occupied, free or unknown. The sensors can mark occupied cells as free when the obstacle has moved, resulting in only static obstacles to be drawn into the map. For this they have to find a sensor reading behind said obstacle to securely clear it.

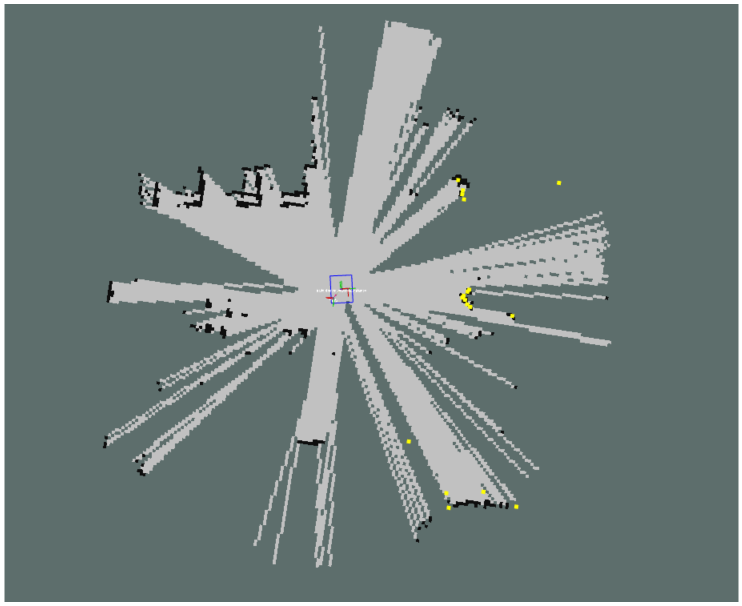

Figure 2: The world as seen by the robot. The dark grey areas are unknown, black dots are obstacles and light grey space is free. The yellow dots are the points recorded by the laser scanner. Visualised by RViz[26]

3.2.2 Depthimage_to_laserscan

One difference between laser scans and depth images is that laser scanners can provide a larger angle of data but only in a two dimensional plane. The stereo depth camera used in this work can create depth data of 480 x 360 pixels. Its observation angle is smaller but it can detect objects on different heights. Its recorded depth data is between 0,5 and 2,8 meters according to its datasheet.

For mapping in ROS by standard only two dimensional laser data is used. But without too much efforts other sensor data like point clouds or depth images can be converted to a laser scan array. The node depthimage_to_laserscan converts the two dimensional depth data from the stereo camera to a scan array. One or multiple rows can be chosen at a certain offset in the image and for each column the lowest value will be stored in the array.

Before converting the images several filters were implemented for this work. For efficient filtering the OpenCV bridge [28] for ROS was used which provides the functionality for image processing. To switch between different filters parameters were added to choose the filter type and kernel size. During run time one can select the different filters with the rqt_reconfigure tool provided by ROS and observe the resulting images and laser scan data. The results will be discussed in the *Evaluation* section (see section 5).

3.2.3 Amcl

To achieve the goal of navigation, localization as it is seen by [29] is the basic requirement of autonomous navigation.

Localization is an important part of the navigational tasks and robots usually rely on their odometry data which records the movement of its motors to calculate the distance travelled and the resulting location of the robot. But this data can be easily distorted. The more distance travelled the bigger the error due to unreliable terrain or sensor inaccuracy. Therefore, additional information is required to rectify the position.

For robotics the *Monte Carlo Localisation* approach has proven itself as most suitable for exact and computational localization [30]. In this work it is implemented by the amcl node [31]. With a given map, odometry and 2D laser scan sources it can approximate the robots position on the global map at any time. This is achieved with a generated particle cloud in which each particle represents a random possible position of the robot. From each position the scan values are compared with the received map. The particle with the highest score is seen as the most probable location and is accordingly weighted. As the robot moves around the position gets more exact and the particle cloud smaller. To prevent the cloud from being stuck in a wrong position due to weighting, the cloud is re-sampled when only a few particles with high weights remain (cd. Figure 3).

Amcl provides functionality for global localization, therefore, it was chosen for this work to operate with a previously drawn map, in this case by the slam_gmapping node.

For amcl it is crucial to receive a map as authentic as possible. It should be generated by the same or similar sensors the robot uses for navigation, as the algorithm matches the sensors data with the received map. This means that a simple floor plan will not suffice for reliable navigation.

3.2.4 Move_base

As soon as an initial map is recognised, provided either by slam or a static map by a map server, move_base takes over the navigational task. Its responsibilities are to find the optimal paths from point A, the robots position, to point B a chosen coordinate in the defined coordinate frame. This frame can be defined as the map which is useful if one wants to navigate to a certain room as this will always have the same coordinate. One could send a goal to the frame of the robot which would cause the robot to move to this position respective to its current coordinates. One provides point B as a goal in form of a stamped pose with position and orientation. Move_base then calculates the optimal path to this pose.

The path planner takes different parameters into consideration which decide if the path should be safe and conservative or faster and more risky. To set those parameters correctly one needs to know how the planners work. The individual parameters will be discussed in the *Implementation* section (see section 4).

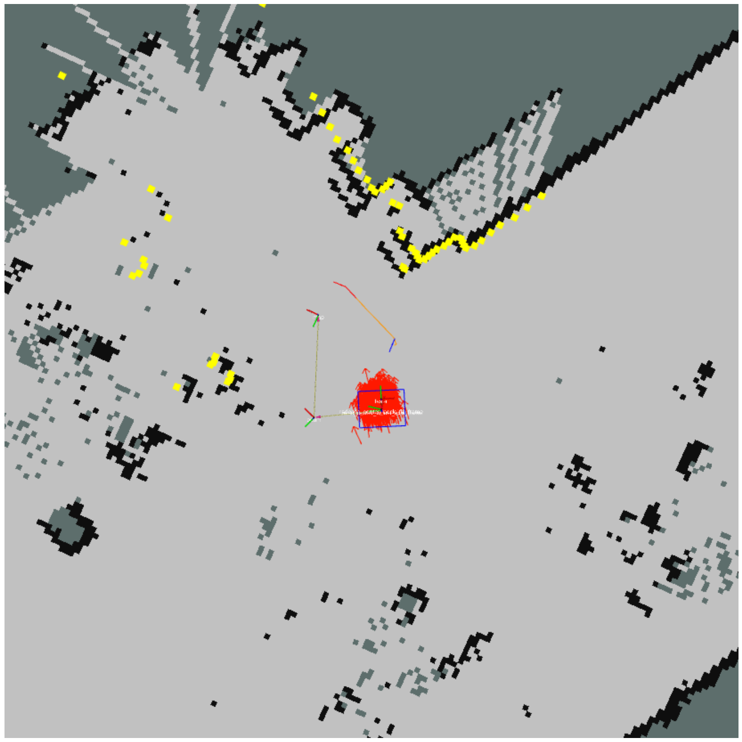

Figure 3: Amcl matches the sensor data with the map. The red arrows are the particle cloud of the weighted, possible positions of the robot. The smaller the cloud the more accurate the position is. Visualised by RViz [26]

The planning is divided into two costmaps. A global costmap takes the whole information of the map into account and draws with path finding algorithms, in our example the Dijkstra algorithm, as described by Dijkstra originally [32] and adjusted to robotic uses by Konolige [33], the optimal, least costly path (cd. Figure 4).

A local costmap (cd. Figure 5)is a smaller map, that in comparison to the global map, is not static. It moves along with the robot and its purpose is to avoid collisions with dynamic obstacles or find shortcuts when the global path was not optimal. In this case the dynamic window approach [34]was used.

As this robot is supposed to move around in a workspace environment it is practical to make some areas, for example areas with a lot of moving office chairs or hard to detect obstacles, forbidden. In real life environments one might not want the robot to move everywhere it wants to. The map used by move_base can differ from the map of amcl. One could map move_base to another map topic as it was done in this work. There a simple floor plan or a modified map can be provided for move_base to navigate in. Amcl still uses the original map for localization but the global path planning now takes place in the artificial map.

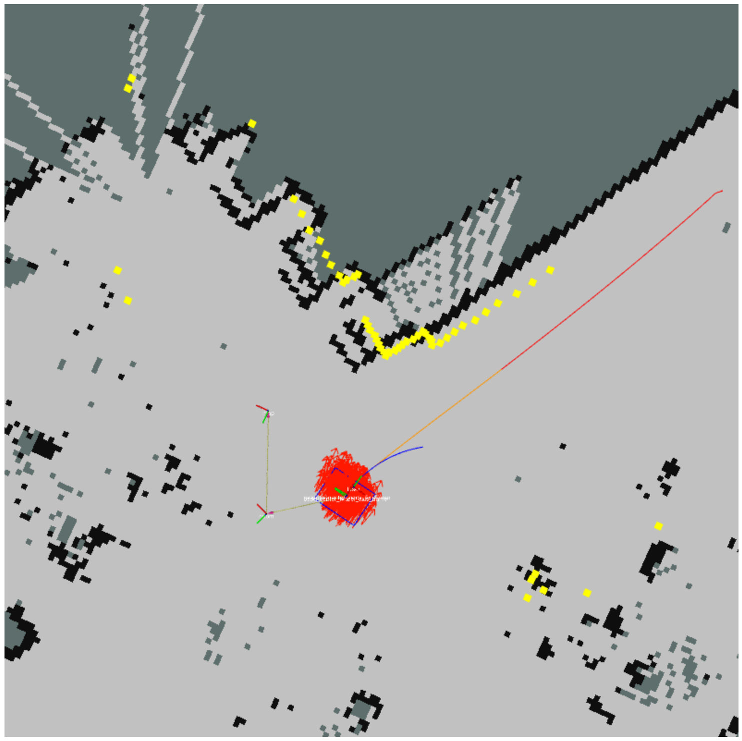

Figure 4: Visualisation of the different paths created by the planners. Red is the global path, orange is the local path and blue is the next movement of the robot calculated by the TrajectoryPlanner. Visualised by RViz [26]

With this method some dangerous areas can be restricted and the robot will not navigate there as the global planner will not provide a path. In this work the original map was edited and some parts were restricted by simply blackening the dangerous areas with an image editor.

3.2.5 Laser_filters

The task of obstacle avoidance is done by the local costmap. It accepts laser scans but also other sensor data like 3D point clouds from depth images. When an obstacle is found by a sensor, a new local path to avoid it can be generated. But if this obstacle is dynamic it has to get cleared out of the map if it moved away. For this the sensor needs to get a reading behind that object. Then it can raytrace it and remove it from the costmap. The problem with common lasers is that they often report out of range values as infinite or zero. This results in dynamic obstacles not getting cleared when there is a lot of free space behind them. To avoid this, laser filters were used in this work. The filter replaces any infinity value in the sensor data with the maximal possible distance the sensor

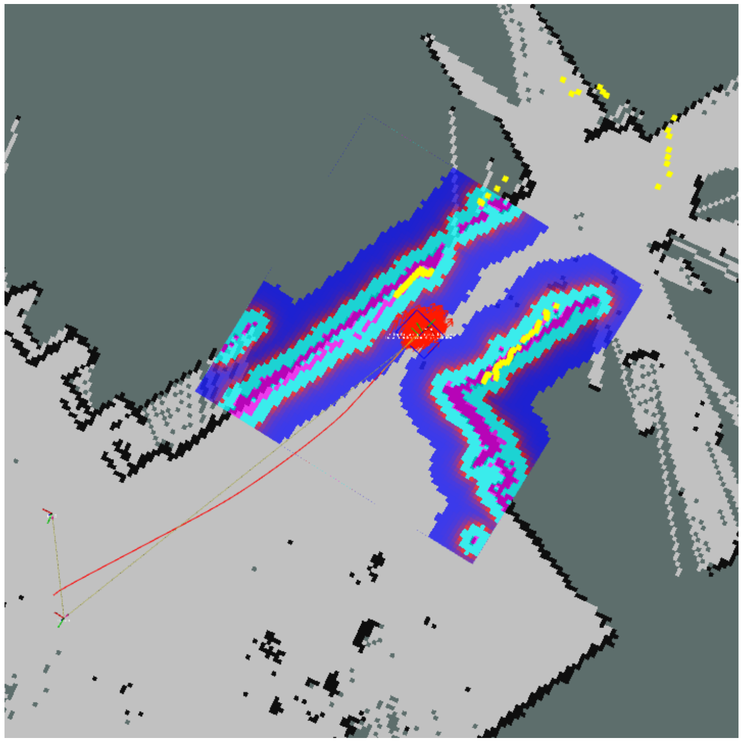

Figure 5: Visualisation of the local costmap where the dark blue areas are the inflated obstacles, that result in a more costly path so the robot keeps a safe distance to the wall. The local path planner takes care of the dynamic obstacle avoidance and uses only the information of the local costmap to minimise the complexity of this computationally heavy task. Visualised by RViz [26]

can read. It has to be registered as a new sensor source that is only allowed to clear obstacles. Amcl and slam still require the original laser data to properly localize and map the environment. Another laser filter is used to ignore the range of the sensor that would scan the robot itself.

3.2.6 Static_transform_publisher

This node is used to publish the position of the laser scanner so the data can be converted to the tf frame of the robot.

4 Implementation

In the following Section the Implementation aspect of this work is described. It will cover the important details of setting up the robot and the involved Android app. Additionally, the separate ROS nodes and how they are configured with the according topics and parameters will be covered.

4.1 Setting up the Robot

The used app for the robot is an app from the android_core package which is based on ROSJava. ROSJavas goal is, as the name suggests, to provide a way to use ROS with Java, in most cases for Android applications. The app was modified by Segway Robotics to provide Loomos base topics. It was set up to read out the sensor data over the API, publish it over the according topics as well as to subscribe to the cmd_vel topic for movement commands and forward them to the robot. At the time when this work begun, Segway Robotics stated they had stopped their ROS development and the app was still unfinished.

The following steps were taking to modify the app for this work.

- Adding odometry. A publisher for odometry data was added. The data is read in via the API of the robot.

- Deactivating unnecessary publishers. As only the depth images, odometry and the tf data are required by the rest of the system, the other sensor publishers were deactivated. The user interface on the app was also deactivated.

- Optimize publishing.

  In the app are two types of publishers:

  The odometry and tf publishers listen continuously on the API of Loomo for sensor data. They publish in a loop of a certain rate.

  The publishers for the images only publish when triggered by a FrameListener when it receives a full image. They were sourced out to separate classes and handled by a scheduled threadpool. This guaranteed better performance and no more crashing from the app.

- Filter data.

  Other uses of the app are to filter out corrupted data. The API of the robot kept sending faulty data of the tf with missing frame ids or wrong quaternions. The app was modified only to publish stable data.

- Connecting to the ROS system.

  Once the robot was set up and reliably publishing data it could be connected to the rest of the system. To implement autonomous navigation, several nodes were used in this work. The individual functions are explained in the Design part of this document (see section 3). A description of the process of setting them up follows.

Following are the topics the robot publishes when connected to the ROS Master via the provided app. Additionally the message types as specified by the ROS documentation are listed.

- /odom
  nav_msgs/Odometry Message
- /tf
  tf/tfMessage Message
- /loomo/realsense/depth/image_raw
  sensor_msgs/Image Message
- /loomo/realsense/depth/camera_info
  sensor_msgs/CameraInfo Message

4.2 Setting up the Raspberry Pi

In comparison to the depth images provided by Loomo a laser scanner was used in this work as well. The reason why it was chosen is described in the *Design* section (see section 3).

For the navigation to work the laser scanner has to be set up correctly. The installation process for the laser scanner node can be found in the appendix or at the git repository of Slamtec [23]. It contains the description of how to install the ROS node of the RPLIDAR. For this work this was done on a Raspberry Pi 3 with a light Ubuntu installation. Important steps were to synchronise the time of the Raspberry Pi and set up the network accordingly so it could publish the laser data over Wi-Fi for the ROS system.

For this work a construction was build to fixate the laser scanner. A stable position of the scanner improves the navigational behaviour.

The location of the scanner has to be measured and given to the transformation tree so the program can accordingly interpret the data. This is done by mentioned static_transform_publisher and as described in the section *How to Reproduce the Work* (see section 4.5).

4.3 Configuring the ROS System

This section will cover the setup of the ROS nodes and the reasons behind the assignment of topics as well as the most important parameters that have to be configured.

To implement any application in ROS it is recommended to make sure that the robot is providing the required information over the according topics. The process of setting up Loomo is described in the previous section. For the rest of the system to work the topics have to be assigned accordingly. The required topics of each node can be found in their corresponding ROS wiki page.

Another important aspect is to keep every component synchronised. When using multiple components make sure the system time is set accordingly.

ROS provides a range of useful tools for troubleshooting and visualisation. Rqt_graph provides a graph visualisation of the running nodes and topics. In the appendix graphs of the running system are provided and there one can observe how the nodes in our system communicate. This tool can be very useful to make sure that the nodes are actually connected. The tool *rostopic* provides a list of all open topics. With the command *echo* one can review the content and with *hz* the publishing rate of each topic. This is very useful to make sure that the nodes are actually publishing. Another important tool is *rosparam*. It provides a list of all the parameters that are required by the nodes. With the command *dump* one can save all the parameters in a document and check if they are set correctly. Rqt_reconfigure provides functionality to change those parameters during run time which is very useful for testing. This way one does not have to restart the whole system after changing a single parameter. The parameters have to be configured for rqt which is not always the case. Most of the parameters are described on the according ROS wiki page of the ROS nodes. How the parameters are set for this work and comments on it can be found in the config files in the appendix. A few important ones will be highlighted in the Parameter section.

4.4 Topic Mapping and Parameter Tuning

A big part of working with ROS is mapping the topics correctly and setting the parameters accordingly. With smart remapping of topics, quick design changes of the system can be implemented without having to modify any code. To enable ROS nodes to work with different robots parameters can be changed so the program can adjust to the robots properties.

In the following section some of those decisions will be explained and the basics highlighted.

4.4.1 Topics

Most ROS nodes need to subscribe to certain topics to work accordingly. The topics with the required message types are defined on the ROS wiki page or in the readme file of the package. For navigation to work the ROS system has some basic requirements of messages it needs to receive over certain topics that have to be mapped accordingly.

Typically the robot is required to publish its odometry data, its transformation tree and additional sensor data in form of laser scans. The first thing one needs to do is to check if the required topics are actually publishing and if they are mapped correctly. Most nodes assume that odometry data is published on the */odom* topic. But if the robot is for some reason publishing on */robotX/odom* and one did not notify the system, problems can occur. Thankfully ROS provides remapping and one can tell the system to listen to the according topics. This can be done via the console with additional commands in the form of

```
rosrun somepackage somenode /odom:=/robotX/odom
```

or one defines it in the launch or config file with the according XML markup.

```
<node name="somenode" package="somepackage"
type="somenode">
```

```
<remap from="/odom" to="/robotX/odom"/>
</node>
```

There are a few helpful tools provided by ROS for topic mapping. With

```
rostopic echo /odom
rostopic hz /odom
```

one can check the content of the messages or at what rate the topic publishes. The tool rqt_graph provides a helpful visualisation of the running nodes and the connected topics.

In addition to the required base topics some remappings worth mentioning were done for this work.

With a simple image editor, black lines were drawn on the generated map to restrict some areas for the robot. But the amcl node still needed the original map to properly localize the robot. So the original map was still published with a map_server on the */map* topic for amcl. Additionally another map_server publishes the edited map over */map_edited* and move_base was remapped to this topic. Now the path planning takes place on the edited map and the robot avoids the restricted area. Amcl still matches the sensor data with the original map and can properly localize.

Obstacles get put in the costmap by the different sensor sources of the robot. But if they move away they also have to get cleared out. To safely clear an obstacle the sensor needs to read a value behind that object. Laser scanners often report infinity values when they measure out of range. This can result in objects not being cleared out when there is a large open area or a corridor behind them. To work around this, laser filters were used in this work as described in the *Design* section (see section 3). The filter has two jobs:

To restrict the range of the sensor so it does not scan the robot itself.

To solve mentioned clearing.

The solution is to duplicate the scan and replace every infinity value with the max range of the laser. With only the altered scan slam would draw obstacles in a circle of the maximal range around the robot and amcl would look for circular walls in the costmap. To resolve this the original laser data was used for marking and clearing. The modified laser data only clears objects out of the map. This resolves in objects being cleared out reliably and the mapping and localization being unaffected.

The deptimage_to_laserscan node requires the depth images to be synchronised with the camera info topic. A node was modified to republish and synchronise images and camera. It listens on the topic *loomo/realsense/depth* for image_raw and camera_info which is the standard naming convention. It publishes the synchronised data on the */synchronized* topic. The node that converts the images has to be mapped to the synchronised topic and the output is mapped to */scan_depthimage* so there are no conflicts with the other scan topics.

4.4.2 Parameters

While using different ROS packages one comes across a range of parameters. The function and description are not always clear and as ROS is continuously developed and updated they can change as well. During research compromising information was found. This section is included to summarise what could be found out about the different parameters. The commented parameter files used by this work can be found in the appendix. A few relevant ones are to be mentioned here.

To run the navigation stack one uses a range of parameter files. There is a standard of what files are included one can follow. It can be found at the ROS wiki page of the ROS navigation stack. An overview of the used parameter files follows at this point to give an impression of what is necessary to include. It is advised to start with the most basic configuration and default values for the navigation to work and then add additional parameters, one by one. The usual way to structure the packages is to have XML files which start the separate nodes and load the according parameter YAML files. Those XML files can be loaded by launch files. Say one has an XML file that loads all the nodes for sensor processing, another XML file loads the map drawing node slam and another to load amcl. Then one would include the first and the second XML file for map drawing. To navigate in a finished map one would include the first and the last XML file. So one can arrange the nodes according to the demands of the use cases.

The individual parameters can be found on the ROS wiki pages [9] of the according packages. In the appendix the commented parameters used by this work can be found in the according files as described below. Their functionality and what to consider when tuning them is documented in the comments.

An example of the standard structure for the navigation stack follows:

- A launch file to start the initial nodes and the Master if required. Ideally only nodes which are the basic requirements for the robot are started in the initial launch file. It includes various XML files for other nodes.
- In the XML file one starts the different nodes for the required use case and loads the required parameter files. In our case it is either slam and move_base or amcl and move_base.
- Costmap common parameters is the place to store parameters that concern the global and the local costmap.
- Global costmap parameters stores only parameters for the global costmap
- Local costmap parameters stores only parameters for the local costmap
- Planner parameters stores parameters for the local path planner

For completeness a list of the most important parameters that are worth spending time to tune correctly follows:

- slam : *delta*

  This defines the resolution of the map and should be set according to the resolution of the sensor used to draw the map. There is no use in setting it higher than the sensors recorded resolution but one could set it lower for higher performance at the cost of precision.

- slam: *maxRange and maxUrange*

  Those parameters should be set accordingly to clear free space as described in the ROS wiki[35]:

```
maxUrange < maximum range of the real sensor <= maxRange
```

- amcl: min and max particles

  For localization it can be useful to modify the range of the position samples in the point cloud from amcl. For this work the size of the cloud was turned down for performance reason.

- costmap common: *footprint*

  It is essential to describe a correct footprint of the robot so the navigation can take the robots size in consideration. This can be a simple radius or a more complex polygon described by coordinate points relative to the robots centre.

- costmap common: *inflation_radius and cost_scaling_factor*

  Setting the inflation radius to a higher value will inflate the obstacles in the costmap. This means the robot will keep a bigger distance to them. The cost scaling factor influences the distance. This can be observed in the costmap (cd. Figure 5) where the dark blue areas are the inflated obstacles. The cost of the paths in those areas will be increased according to the cost scaling factor. The lower the factor the further the distance to the obstacles. With tuning those two parameters one can influence the smoothness of the robots turns around edges and the distance to walls along his path.

- costmap common: *observation_sources*

  It is crucial to define the sensor sources here because without defining the correct frame and topic for each sensor used, the system will not work as expected.

- costmap common: *obstacle_range and raytrace_range*

  Obstacle_range and raytrace_range have to be set accordingly or dynamic obstacles will not get cleared out of the costmap correctly. The raytrace range should be lower than the sensor max range and the obstacle range has to be lower than the raytrace range!

The local planner is another crucial part of the system. One can choose any local planner from the ROS community or write a new one. For this work the TrajectoryPlannerROS, also called base_local_planner was used. It uses the Dynamic Window Approach as described in the *Design* section (see section 3). Another option is the DWA planner which is written specifically for this approach and provides very smooth path planning. It is specialised on holonomic robots though and did not provide functionality for driving backwards to escape certain situations.

- base local planner: parameters for the robots movement

  It is advised to define the actual maximal acceleration values, *acc_lim_x, acc_lim_y and acc_lim_theta* of the robot. They are only for the planner to project the trajectories and decide which movement is safe. The speed of the robot can be limited in the maximum velocity parameters *max_vel_x, max_vel_theta*.

- base local planner: *holonomic*

  It is important to set this correctly as it will influence the trajectories tried by the planner. A holonomic robot can move in every direction no matter his orientation. Robots with differential drive like the one used for this work can not do that.

- base local planner: *pdist_scale, gdist_scale, occdist_scale and stop_time_buffer*

  Those parameters are the most important ones for the navigational behaviour of the robot. The stop time buffer defines how soon the robot will stop when an obstacle is discovered in its path. With the three scaling parameters one can tune the behaviour of the robot according to how much it should try to reach its goal, how close it should stay to the path and how far away it should stay from obstacles. This is very useful to tune in the different environments.

A few steps to troubleshoot the system shall be added at this point as well. They base on the experiences and discoveries during writing this thesis.

When configuring a new node always check the frame_id parameters. Often the nodes need the name of the base_frame or the frame of the robots sensor sources. This is essential information and many systems will not work without them. A good practice is to try to get the system running with as few parameters as possible. Usually there are standard values defined which can be found on the according ROS wiki pages. Those standard values are great orientation points from where one can start tuning. If some default values are missing or one feels the list of described parameters is incomplete, one can print all the parameters with the command

```
rosparam dump somefilename
```

A file with all the parameters of the running system and their values will be created. There one can see all the parameters used and the standard values that were chosen if the parameters were left undefined.

4.5 How to Reproduce the Work

To start ROS nodes one first has to install ROS. In this work we used a laptop with Ubuntu 16.04 and a ROS kinetic distribution.

The installation process is described at www.ros.org. Additionally one can refer to the work of F. Enriquez [36] who discusses getting started with ROS and how to work with it. Mentioned work concerns an older release of ROS but is still applicable to the newer ones. The additional tutorials on ros.org are also strongly recommended as they give a deeper understanding on how ROS works. The robot also has to be set up with ROS. One can install the source code of the Android app for Loomo with the tutorial which can be found in the readme file of the app in the appendix.

The ROSJava and Android tutorials at ros.org are also recommended.

When the app is installed one has to start the ROS Master. It does not matter on which machine it is started as long as the other nodes know the IP address. The ROS Master IP has to be set up as described in the tutorials in the .bashrc file. Once a ROS Master is started any ROS node in the same submask can communicate with it. Now one can connect the app of the robot after entering the correct IP address. As soon as this is done Loomo publishes its depth images, odometry and tf.

To start the different applications of this work one has to clone the git repository in the appendix into the src folder of a created catkin workspace as described in the ROS tutorials. When succesfully cloned, one has to compile the workspace and then the launch files which start the separate applications can be used. To compile, change to the upper directory, the one where the src folder is, and execute

```
catkin_make
```

This should compile the code and the dependencies. Then one has to source the workspace again with

```
source devel/setup.bash
```

Now one can start the according launch files. The launch files are located in the package pba_msteiner_loomo. The command to launch them is

```
roslaunch pba_msteiner_loomo launchfile.launch
```

Replace launchfile.launch with the according launch file.

In this work several launch files are provided.

1. slam with depth images, *synchronized_depth_to_laser_slam.launch*
2. slam with laser scanner, *slam_laser.launch*
3. amcl with laser scanner, *dwa_amcl_laser.launch*
4. amcl with depth and laser data, *depth_and_laser.launch*

Along with the launch files, RViz as a visualisation tool is started. It can also be used as a user interface to publish navigation goals. Additionally two apps for smartphones are provided. One can remote control the robot manually which is applicable if one wants to draw the map and does not want to depend on the autonomous navigation. The other one provides an interface to publish coordinates as navigation goals.

4.5.1 Summary of the steps

1. Install ROS as described in the official tutorials
2. Set up Loomo as described in section *Setting up the Robot*
3. Set up laser scanner and Raspberry Pi as described in section *Setting up the Raspberry Pi*
4. Set up the workspace as described in *Configuring the ROS System*
5. Start the ROS Master
6. Start the launch file for the laser scanner on the Raspberry Pi with SSH.
7. launch the provided launch files

The provided Android apps can be installed via Android Studio onto any Android smartphone. With *TeleopTutorial* one can remotely control the robot with a joystick. With *PubSubTutorial* one can give the robot goals in forms of coordinates. The coordinates are found with the help of the RViz tool.

5 Evaluation and Discussion

5.1 Evaluation

The stated goal of this work was to implement a solution with ROS which enables the Segway robot Loomo to guide new people in an office to a respective meeting room. To guarantee that this will work in any environment, autonomous navigation provided by ROS is supposed to be used. In the use cases it was defined that Loomo should be able to draw a map of the office, navigate in it and avoid obstacles on the way. To test if the robot fulfills those requirements it was continuously driven around in the SLAB of Iteratec Munich. As it is supposed to be a human companion it was compared to how it behaves in a lively office space and interacts with humans. The methods were mainly visual observation of its behaviour. Observed was if the robot travels at acceptable walking speed so people could actually follow it and if it can avoid dynamic obstacles, mostly in form of people or office chairs, getting in its way and still reach its goal. Following checkpoints were included:

- Did the robot reach its goal?

- Did it take a somewhat optimal path?
- Did it avoid obstacles on the way?

Subtasks of this which were also observed are:

- Quality of drawn map.
- Localization accuracy.
- Computational Efficiency.

This was either observed in the real world or in RViz, the visualization tool provided by ROS.

We used different approaches to reach those goals. Those successful enough to provide comparable navigation are represented by the separate launch files. In the end the robot was reliably navigating at acceptable walking speed while avoiding obstacles as it was demonstrated in the presentation. In the following section we compare the most relevant influences to the mentioned criteria.

5.1.1 Quality of Map

The quality of the map is essential to smooth navigation as the robot will stop often to orient itself if it cannot find matches to its sensors. The first approach was to let the robot navigate only with its native sensors. To draw the map, depth images provided by the RealSense camera as described in the *Design* section (see section 3) were used. Those had to be converted to a laser scan by the node depthimage_to_laserscan. The raw images did not provide the quality needed (see figure 6). So a few filters were used to fortify the images.

With certain filter setups, in the end eroding and dilating provided the highest quality and best performance, states were reached that a map was able to be drawn. The resulting map enabled the robot to navigate but due to the unstable data exact localization was difficult (see figure 7).

In comparison a laser scanner was used which provided much more exact data and enabled stable localization but failed to detect more complex objects.

5.1.2 Navigational Behaviour

To navigate safely in an office environment a robot needs to plan its path smartly and avoid obstacles smoothly. The path planning especially on global level works great with the standard global planner by move_base and no modifications had to be done for this work. For obstacle avoidance the local map and the local planner come into play. With the drawn laser map (see figure 7) stable localization was possible but more complex obstacles like office chairs remained unnoticed by the sensor and did not get put into the local costmap. The laser scanner only gets depth data of a two dimensional plane. The plane happened to be exactly at the level where office chairs are the thinnest. One could raise the laser but this would only temporarily solve the problem. There are other obstacles like tables or flip charts that can have extending features that are unnoticed by the laser.

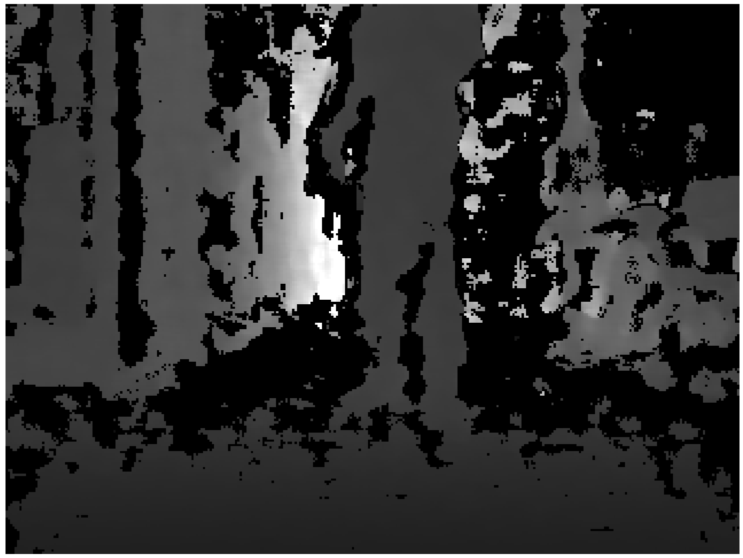

Figure 6: This is the raw depth image. On it one can vaguely distinguish a person.

One solution was to restrict dangerous areas by simply blackening them on the map (see figure 8).

The path planner will avoid those areas now but it restricts the robots movement and dynamic objects could move into the allowed path of the robot. So it still needs a way to detect those complex objects. The final solution was to combine the two sensors and use the laser scanner for mapping and the depth images for additional obstacle detection. This resulted in stable localization and detection of complex objects. But as the depth images still have to be previously processed it slows down the system a bit. With better filtering or a different component distribution as mentioned in the *Future Work* section (see section 6) this could be enhanced.

The path the robot takes through its map is also an important factor of navigation. As discussed in the *Parameters* section (see section 4) there are a range of parameters to be tuned. With the mentioned parameters, a setting was reached where the robot travels at a safe distance of obstacles approximately in the centre of the corridor and avoids dynamic obstacles at a safe distance. Glass walls still have to be blackened out manually on the map as they provide a different challenge to the robots sensors.

5.2 Lessons Learned and Unexpected Findings

Robotics is a very broad field and even in a simple use case a lot of challenging problems can be found. Navigation is a great example for exploring the possibilities of ROS and working with robots as many of the core problems of robotics

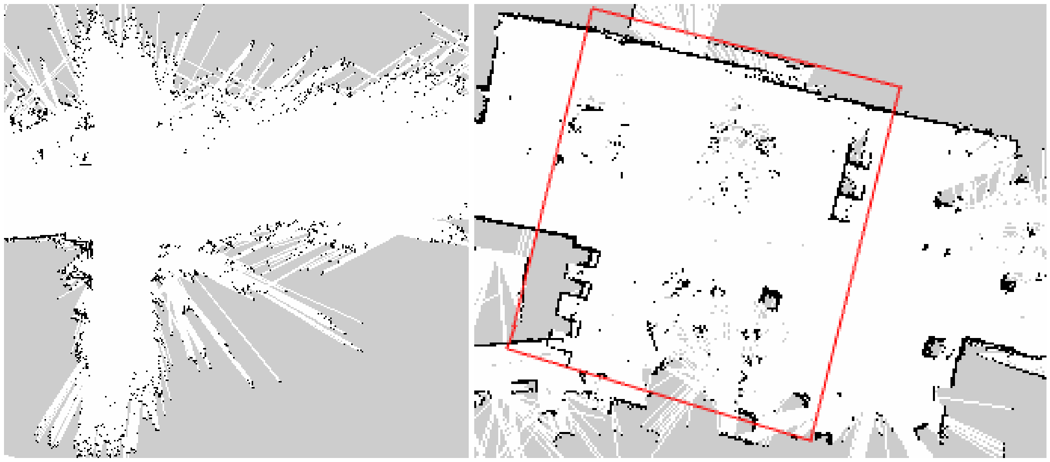

Figure 7: Left is the map drawn with the depth images as sensor source. In comparison on the right the map drawn by the laser. The red rectangle is the area covered by the left map.

are part of it.

In the course of this work we were able to document the process of working with ROS and what important parts are not to be overlooked to make a robotic system work. We can now provide a checklist of TODOs and troubleshooting which can come in handy when working with robots.

Additionally we learned about ROSJava and how the Android operating system can be included in working with ROS. The beginning of working with ROS can be very slow and just making sure that every part of the system is working as it is supposed to takes up a lot of time. But we discovered as one learns more about ROS and gets involved deeper, the development can speed up exponentially. This is the moment when the full functionality of ROS begins to shine. In the end we were able to quickly substitute different parts of our system and compare the results.

But opposed to the plug and play capabilities of ROS it is crucial to understand how each part of the system works. As the robot has to interact with the real world troubleshooting can be difficult. To figure out what part of the system is causing the problem is non trivial. And often one discovers that the network connection was the culprit.

6 Conclusion and Future Work

6.1 Summary

In the course of this work we set up the robot assistant Loomo, a very mobile Segway robot equipped with an Android operating system to work with the

Figure 8: With an image editor black lines or areas can be drawn on the map. They will be seen as obstacles by the path planner.

ROS navigation stack. The goal was to make the robot an office assistant that can guide new people around with the autonomous navigation provided by ROS. In the process of implementing we have documented the most important parts of working within the framework as well as with mobile robots. We have discussed the advantages of working with depth cameras as a sensor source versus a standard laser scanner. The use case of navigation is a core problem of robotics and we had to tackle a lot of the typical challenges that come with it for example localization, path planning and obstacle avoidance. To achieve the task of navigation several components were used. While handling those components we also got to explore the peer to peer aspect and the plug and play ability of ROS.

6.2 Assumptions and Limitations

We used Loomo because it is a fully functional, already setup robot which is relatively affordable. The goal of this work was to explore the possibilities of ROS and its provided packages for an easy obtainable office assistant. Therefore, the configuration on a deeper hardware level was not covered in this work. We assumed that Loomo is correctly set up and the data it publishes is correct. As this was not always the case some results could be improved by configuration on the hardware level of the robot. The limitations of the Intel RealSense camera were a challenge as well. Therefore, we had to use a different sensor source, the laser scanner. Opposed to the camera which is neatly implemented next to the robots interface, the setup of the laser scanner was more prototypical. This also gave some movement limitations to the robot as it could not accelerate as fast as before.

The approach to teach the robot navigation used in this work follows more or less the navigation stack of ROS. It includes general solutions to robot navigation. This served our purpose to proof the code-re-usability of ROS. On the other hand the used ROS packages were not catered to our robots specific needs. With more time and coding one could rewrite the nodes or write new ones to address Loomos requirements.

Next steps in the robots development could be to optimise the use of the different sensors simultaneously as it was demonstrated. Including beacons to improve navigation is a possibility as well, as we have already set up some in the workplace. In addition visual odometry might be worth looking into [37]. As Loomo is equipped with a relatively powerful camera and has already some image processing capabilities this might be an interesting path.

In conclusion the goal of navigating from point A to point B was reached. Robots do not see the world as we do and this mirrors in the outcome of this work. We have tried navigating with a stereo depth camera and a laser scanner. Both have shown advantages and disadvantages. The path planning of robots is an interesting aspect and there is a lot of potential for optimisation. It was shown that ROS can make robotics more accessible but it also shows that there is a lot of work in every aspect to be found.

7 Acknowledgement

I want to thank my girlfriend for supporting me during writing this thesis and keeping my spirits up when dark clouds shaded my mind. I want to thank her as well for proofreading my thesis that is so different from her field.

I wish to thank my colleagues from Iteratec Munich who allowed me to write my thesis in their SLAB and lent me Loomo to work with. Also many thanks to the colleagues from Iteratec Vienna who welcomed me in their office and let me present my thesis there.

My sincere thanks to my advisor Univ.-Prof. Dipl.-Ing. Dr. Wolfgang Klas for continuous support and allowing me to write my thesis in Munich and present it in the office in Vienna.

References

[1] M. Quigley, K. Conley, B. Gerkey, J. Faust, T. Foote, J. Leibs, R. Wheeler, and A. Y. Ng, "ROS: an open-source Robot Operating System", in *ICRA workshop on open source software*, Kobe, Japan, vol. 3, 2009, p. 5.

[2] (2018). What Can Loomo Do? Accessed: 2018-07-2, [Online]. Available: `https://loomo.com/`.

[3] (2018). Segway Robotics. Accessed: 2018-07-2, [Online]. Available: `http://www.segwayrobotics.com/`.

[4] (2018). ROS. Accessed: 2018-07-2, [Online]. Available: `http://wiki.ros.org//`.

[5] S. Cousins, "Welcome to ros topics [ros topics]", *IEEE Robotics & Automation Magazine*, vol. 17, no. 1, pp. 13–14, 2010.

[6] J. Borenstein and L. Feng, "Measurement and correction of systematic odometry errors in mobile robots", *IEEE Transactions on robotics and automation*, vol. 12, no. 6, pp. 869–880, 1996.

[7] K. Shoemake, "Animating rotation with quaternion curves", in *ACM SIGGRAPH computer graphics*, ACM, vol. 19, 1985, pp. 245–254.

[8] S. Thrun, "Probabilistic robotics", *Communications of the ACM*, vol. 45, no. 3, pp. 52–57, 2002.

[9] (2018). Navigation. Accessed: 2018-07-2, [Online]. Available: `http://wiki.ros.org/navigation`.

[10] E. Marder-Eppstein, E. Berger, T. Foote, B. Gerkey, and K. Konolige, "The office marathon: Robust navigation in an indoor office environment", in *Robotics and Automation (ICRA), 2010 IEEE International Conference on*, IEEE, 2010, pp. 300–307.

[11] M. A. Fischler and R. C. Bolles, "Random sample consensus: a paradigm for model fitting with applications to image analysis and automated cartography", *Communications of the ACM*, vol. 24, no. 6, pp. 381–395, 1981.

[12] S. Zaman, W. Slany, and G. Steinbauer, "ROS-based mapping, localization and autonomous navigation using a Pioneer 3-DX robot and their relevant issues", in *Electronics, Communications and Photonics Conference (SIECPC), 2011 Saudi International*, IEEE, 2011, pp. 1–5.

[13] H. I. M. A. Omara and K. S. M. Sahari, "Indoor mapping using kinect and ROS", in *Agents, Multi-Agent Systems and Robotics (ISAMSR), 2015 International Symposium on*, IEEE, 2015, pp. 110–116.

[14] N. P. Koenig and A. Howard, "Design and use paradigms for Gazebo, an open-source multi-robot simulator.", in *IROS*, Citeseer, vol. 4, 2004, pp. 2149–2154.

[15] R. Vaughan, "Massively multi-robot simulation in stage", *Swarm intelligence*, vol. 2, no. 2-4, pp. 189–208, 2008.

[16] A. Pajaziti, "SLAM–map building and navigation via ROS", *International Journal of Intelligent Systems and Applications in Engineering*, vol. 2, no. 4, pp. 71–75, 2014.

[17] J. Pérez, J. A. Castellanos, J. Montiel, J. Neira, and J. D. Tardos, "Continuous mobile robot localization: Vision vs. laser", in *Robotics and Automation, 1999. Proceedings. 1999 IEEE International Conference on*, IEEE, vol. 4, 1999, pp. 2917–2923.

[18] K.-H. Lin, C.-H. Chang, A. Dopfer, and C.-C. Wang, "Mapping and Localization in 3D Environments Using a 2D Laser Scanner and a Stereo Camera.", *Journal of information science and engineering*, vol. 28, no. 1, pp. 131–144, 2012.

[19] (2018). What is a TurtleBot? Accessed: 2018-07-2, [Online]. Available: `https://www.turtlebot.com/`.

[20] J. P. d. A. Barbosa, F. d. P. d. C. Lima, L. d. S. Coutinho, J. P. R. R. Leite, J. B. Machado, C. H. Valerio, and G. S. Bastos, "ROS, Android and cloud robotics: How to make a powerful low cost robot", in *Advanced Robotics (ICAR), 2015 International Conference on*, IEEE, 2015, pp. 158–163.

[21] (2018). Kinect für Windows. Accessed: 2018-07-2, [Online]. Available: `https://developer.microsoft.com/de-de/windows/kinect`.

[22] (2018). Intel Realsense Camera ZR300. Accessed: 2018-07-2, [Online]. Available: `https://software.intel.com/en-us/realsense/zr300`.

[23] (2018). RPLIDAR A1 - Slamtec - Leading Service Robot Localization and Navigation Solution Provider. Accessed: 2018-07-2, [Online]. Available: `http://www.slamtec.com/en/lidar/a1`.

[24] (2018). Raspberry Pi 3 Model B. Accessed: 2018-07-2, [Online]. Available: `https://www.raspberrypi.org/products/raspberry-pi-3-model-b/`.

[25] G. Brian. (2018). gmapping. Accessed: 2018-07-2, [Online]. Available: `http://wiki.ros.org/gmapping`.

[26] H. R. Kam, S.-H. Lee, T. Park, and C.-H. Kim, "Rviz: a toolkit for real domain data visualization", *Telecommunication Systems*, vol. 60, no. 2, pp. 337–345, 2015.

[27] G. Grisetti, C. Stachniss, and W. Burgard, "Improved techniques for grid mapping with rao-blackwellized particle filters", *IEEE transactions on Robotics*, vol. 23, no. 1, pp. 34–46, 2007.

[28] (2018). OpenCV library. Accessed: 2018-07-2, [Online]. Available: `https://opencv.org/`.

[29] B. Zhang, J. Liu, and H. Chen, "Amcl based map fusion for multi-robot slam with heterogenous sensors", in *Information and Automation (ICIA), 2013 IEEE International Conference on*, IEEE, 2013, pp. 822–827.

[30] F. Dellaert, D. Fox, W. Burgard, and S. Thrun, "Monte carlo localization for mobile robots", in *Robotics and Automation, 1999. Proceedings. 1999 IEEE International Conference on*, IEEE, vol. 2, 1999, pp. 1322–1328.

[31] G. Brian. (2018). amcl. Accessed: 2018-07-2, [Online]. Available: `http://wiki.ros.org/amcl`.

[32] E. W. Dijkstra, "A note on two problems in connexion with graphs", *Numerische mathematik*, vol. 1, no. 1, pp. 269–271, 1959.

[33] K. Konolige, "A gradient method for realtime robot control", in *Intelligent Robots and Systems, 2000.(IROS 2000). Proceedings. 2000 IEEE/RSJ International Conference on*, IEEE, vol. 1, 2000, pp. 639–646.

[34] 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, 1997.

[35] (2018). ROS. Accessed: 2018-07-2, [Online]. Available: `http://www.ros.org`.

[36] F. Enrique, L. S. Crespo, A. Mahtani, and A. Martinez, *Learning ROS for robotics programming: your one-stop guide to the Robot Operating System.* Packt Publishing, 2015.

[37] D. Nistér, O. Naroditsky, and J. Bergen, "Visual odometry", in *Computer Vision and Pattern Recognition, 2004. CVPR 2004. Proceedings of the 2004 IEEE Computer Society Conference on*, IEEE, vol. 1, 2004, I–I.