UPPSALA
UNIVERSITET

UPTEC F 23043
Examensarbete 30 hp
Juni 2023

ROS-based implementation of a model car with a LiDAR and camera setup

Marcus Nises

Civilingenjörsprogrammet i teknisk fysik

UPPSALA UNIVERSITET

ROS-based implementation of a model car with a LiDAR and camera setup

Marcus Nises

Abstract

The aim of this project is to implement a Radio Controlled (RC) car with a Light Detection and Ranging (LiDAR) sensor and a stereoscopic camera setup based on the Robot Operating System (ROS) to conduct Simultaneous Localization and Mapping (SLAM). The LiDAR sensor used is a 2D LiDAR, RPlidar A1, and the stereoscopic camera setup is made of two monocular cameras, Raspberry Pi Camera v2. The sensors were mounted on the RC car and connected using two Raspberry Pi microcomputers. The 2D LiDAR sensor was used for two-dimensional mapping and the stereo vision from the camera setup for three-dimensional mapping. RC car movement information, odometry, necessary for SLAM was derived using either the LiDAR data or the data from the stereoscopic camera setup. Two means of SLAM were implemented both separately and together for mapping an office space. The SLAM algorithms adopted the Real Time Appearance Based Mapping (RTAB-map) package in the open-source ROS.

The results of the mapping indicated that the RPlidar A1 was able to provide a precise mapping, but showed difficulty when mapping in large circular patterns as the odometry drift resulted in the mismatch of the current mapping with the earlier mapping of the same positions and secondly in localization when turning quickly. The camera setup derived more information about surrounding and showed more robust odometry. However, the setup performed poorly for the mapping of visual loop closures, i.e., the current mapping did not match the earlier mapping of earlier visited positions.

Teknisk-naturvetenskapliga fakulteten

Uppsala universitet, Utgivningsort Uppsala

Handledare: Mats Wiklander Ämnesgranskare: Ping Wu

Examinator: Tomas Nyberg

Populärvetenskaplig Sammanfattning

Analys av omgivning och miljö används inom ett flertal tekniska områden. Ett av dessa är inom kartläggning av omgivning vilket kan användas bland annat för automatisk navigation. Omgivningen kan analyseras med hjälp av sensorer. Sensorer kan ha olika uppgifter som att bedöma avstånd, att analysera objekt eller se färg. Två populära typer av sensorer som använts i detta projekt är Light Detection and Ranging (LiDAR) och kameror.

LiDAR är en ljusbaserad detektions och avståndsmätningsmetod. En LiDAR sensor mäter distans genom att emmitera ljus i form av en laserpuls från och mäta tiden för ljuset att reflekteras på ett objekt och färdas tillbaka till en detektor på sensorn. LiDAR teknologin används inom kartläggning tack vare dess pålitlighet och precision [1]. LiDAR teknologin finns tillgänglig för både 2D och 3D sensorer. 3D LiDAR används för att exempelvis kartlägga omgivningen eller större markytor i tre dimensioner. 2D LiDAR utför distansmätningar i två dimensioner och har användning för bland annat kartläggning och navigation inomhus. De kan därför användas i exempelvis hjälprobotar eller robotdammsugare [2].

Kameror används på ett flertal sätt för avkänning, analys och mätning av omvärlden. En populär användning inom automisering är i Advanced Driver Assistance Systems (ADAS) på moderna bilar. Kameror är då placerade runtom en bil och pekar i olika riktningar för olika syften inom bildanalys och distansmätning [3]. I de områden två kameror iaktar samtidigt från olika perspektiv, ofta på framsidan av bilarna, uppstår en stereoskopisk effekt och en distanskarta kan beräknas. En vanlig stereoskopisk kamera är konstruerad genom att två enkla kameror är placerade med en upmätt distans emellan och kopplade att ta bilder exakt samtidigt. Med två bilder på samma objekt från olika perspektiv kan objektets position i tre dimensioner beräknas genom att först matcha korresponderande punkter i de två bilderna och sedan beräkna hur långt punkterna skiljer sig i pixlar [4].

Simultaneous Localisation and Mapping (SLAM) är det beräkningstekniska problemet av att kartlägga ett område och samtidigt spåra positionen för objektet som utför kartläggningen [5], ofta en robot eller bil. För detta krävs information av omgivningen samt information om objektets rörelse. Informationen av omgivningen ges av sensorer som LiDAR eller kameror. Rörelseinformationen kan exempelvis utvinnas genom att spåra omgivningens rörelse i bilder, från LiDAR data, av en Internal Measurement Unit eller genom att läsa av hur många varv däcken har snurrat på en bil. Beräkningarna i grundproblemet är probabilistiska och ett flertal verktyg finns som kan utföra SLAM beräkningar. Verktyget som används i detta projekt är Real Time Appearance Based Mapping (RTAB-map).

List of Abbreviations

Notation	Description
ADAS	Advanced Driver Assistance Systems.
IMU	Internal Measurement Unit.
LiDAR	Light Detection and Ranging.
LTM	Long Term Memory.
OS	Operating System.
RC	Radio Controlled.
RGB	Red Green Blue.
RGB-D	Red Green Blue Depth.
ROS	Robot Operating System.
RP3	Raspberry Pi 3 model b.
RP4	Raspberry Pi 4 model b.
RTAB-map	Real Time Appearance Based Mapping.
SDK	software development kit.
SLAM	Simultaneous Localisation and Mapping.
TF	Transform coordinate Frames.
WM	Working Memory.

Table of Contents

1 Introduction

This degree project aims to perform an implementation of Simultaneous Localisation and Mapping (SLAM) on an Radio Controlled (RC) car using a 2D Light Detection and Ranging (LiDAR) sensor and a stereoscopic camera setup consisting of two monocular camera modules. On a theoretical level and conceptual level the SLAM problem is solved but difficulties remain in realising general solutions to the SLAM for mobile robots as the mapping of larger areas with increasingly difficult terrain is performed [5]. A good implementation of SLAM is often a prerequisite for automation of mobile robots as autonomous navigation requires accurate knowledge of the surroundings.

1.1 Background

Automation in robotics is a field that may become more prominent in the future. With applications for autonomous navigation such as in industrial material transfer, vacuum cleaners, lawnmowers and help robots, the usage of autonomous navigation may increase in years to come [6]. For a mobile robot to perform autonomous navigation information about surroundings is required and a means to obtain this is through SLAM.

The SLAM problem is the challenge of placing a mobile robot in an unknown location in an unknown environment and for the mobile robot to build an incremental and consistent map of its surroundings while simultaneously determining its location in that map [5]. Theoretically there are ways in which SLAM has been solved, but issues remain in realising SLAM implementation as no real life scenario is ideal. At the same time improvements will continue as improvements to hardware allow for more complex computation making SLAM work in more challenging environments and on larger scale.

Cameras have a multitude of possible applications including object recognition and analysis. They have been used to great effect in modern cars where Advanced Driver Assistance Systems employ a set of cameras around a vehicle for different purposes [3]. If a point is seen from two cameras simultaneously and the position and orientation of the cameras relative to each other are known, the distance and 3D position of that point can be calculated. In this thesis, two cameras are placed on a mobile robot and synchronised. This allows for points that can be recognised by both cameras to be mapped in three dimensions and used in SLAM.

Light Detection and Ranging (LiDAR) is a sensor type used for measuring distance. The LiDAR sensor emits a laser beam and measures the time for the laser beam to reflect on an object and return to a receiver on the sensor. It is commonly used for high-resolution distance maps, with applications in surveying, geodesy, forestry and navigation for autonomous cars [7]. The LiDAR sensor used in this theses is a 2D LiDAR sensor, which in SLAM has great usage in mapping indoor locations.

To perform SLAM for this project, Real Time Appearance Based Mapping (RTAB-map) was chosen. This was due to its availability in Robot Operating System (ROS) and as of 2019 being the only ROS SLAM library to provide compatibility with both stereo camera and LiDAR sensors [8]. This allows for SLAM to be performed by both sensors individually, as well as simultaneously. Projects using similar sensors have achieved good

results using RTAB-map [9]. To perform the SLAM RTAB-map needs information about surroundings and information about the mobile robot position and movement in relation to its surroundings.

1.2 Aim and goals

This project aims to implement a RC car with a LiDAR sensor and a stereoscopic camera setup to conduct SLAM. The goals of the project are that the SLAM is implemented successfully using the LiDAR and the camera setup so that the car can successfully perform mapping its surroundings.

1.3 Tasks

The tasks of this thesis are set as follows.

- Implement SLAM using a LiDAR sensor and Camera setup separately.
- Implement SLAM using a LiDAR sensor and Camera setup simultaneously.
- Test, evaluate and compare stereo camera and LiDAR SLAM using odometry obtained from stereo camera and LiDAR sensor separately.

1.4 Outline

Section 1 provides a short introduction, background and goal for this thesis. Section 2 describes theory behind the SLAM problem, and theory behind sensors and the means used to implement SLAM. Section 3 presents the method and implementation, including a system overview, hardware components used, the process of assembling the robot, an overview of the software used, and the installation process. The results are presented in chapter 4 and then discussed in chapter 5 in which challenges and possible improvements are also discussed. Chapter 6 finally reiterates the conclusions drawn from the results and provides suggestion for further improvement of the work done in this thesis.

2 Theory

2.1 Simultaneous Localisation and Mapping

The Simultaneous Localisation and Mapping (SLAM) problem is the idea of placing a mobile robot in a unknown environment and for it to incrementally build a consistent map of its surroundings while simultaneously monitoring its position in relation to its surroundings [5].

Given the quantities.

- x_k: State vector describing the location and orientation of the vehicle.
- u_k: Control vector, applied at time $k-1$ to derive the vehicle to a state x_k at time k.
- m_i: Vector describing the location of the i^{th} landmark whose true location is assumed time invariant.
- z_{ik}: An observation taken from the vehicle of the location of the i^{th} landmark at time k. At a time with multiple relevant landmarks or no specific landmark is relevant, the quantity is written as z_k.

The sets are defined as follows.

- $X_{0:k} = \{x_0, x_1, ..., x_k\} = \{X_{0:k-1}, x_k\}$: History of vehicle locations.
- $U_{0:k} = \{u_1, u_2, ..., u_k\} = \{U_{0:k-1}, u_k\}$: History of control inputs.
- $m = \{m_1, m_2, ..., m_n\}$: Set of all landmarks.
- $Z_{0:k} = \{z_1, z_2, ..., z_k\} = \{Z_{0:k-1}, z_k\}$: Set of all landmark observations.

The base Simultaneous Localisation and Mapping (SLAM) problem is a probabilistic problem and requires that the probability distribution

$$P(x_k, m|Z_{0:k}, U_{0:k}, x_0) \tag{2.1}$$

is computed for all times k. This distribution describes the joint posterior density of the landmark locations and vehicle location and orientation at time k given the set of landmark observations, history of control inputs and initial state of the vehicle.

Given an estimation of the distribution $P(x_k, m|Z_{0:k}, U_{0:k}, x_0)$ at time $k-1$ the joint posterior density at time k can be computed, with an observation u_k and observation z_k, using the Bayes theorem. For this a state transition model and an observation model is required.

The observation model describes probability of making an observation z_k, given joint posterior density x_k and landmarks m and is generally describes on the form

$$P(z_k|x_k, m) \tag{2.2}$$

The motion model describes joint posterior at time k x_k and is assumed to only require proceeding joint posterior x_{k-1} and control vector u_k

$$P(x_k|x_{k-1}, u_k) \tag{2.3}$$

The standard SLAM algorithm is now implemented as a standard recursive prediction of time-update and measurement update [5].

Time update:

$$P(x_k, m|Z_{0:k-1}, U_{0:k}, x_0) = \int P(x_k|x_{k-1}, u_k) \times P(x_{k-1}, m|Z_{0:k-1}, U_{0:k-1}, x_0)dx_{k-1} \quad (2.4)$$

Measurement update:

$$P(x_k, m|Z_{0:k}, U_{0:k}, x_0) = \frac{P(z_k|x_k, m)P(x_k, m|Z_{0:k-1}, U_{0:k}, x_0)}{P(z_k|Z_{0:k-1}, U_{0:k})} \quad (2.5)$$

2.2 Robot Operating System

Robot Operating System (ROS) is an open source software development kit (SDK) that provides building blocks for applications in robotics. A crucial component of ROS is the message-passing system, allowing interaction between software systems and hardware. With the ability to build and reuse code, the vast ROS community has standardised ROS message formats. This includes communicating sensor data , such as LiDAR and cameras, in standardised formats, as well as communication between software programs. The ROS ecosystem includes drivers, algorithms and user interfaces as well as tools such as launch, introspection, debugging, visualisation, plotting, logging and playback [10]. The ROS environment communication is structured using "Nodes", "Topics", and "Master", as shown in schematics in Figure 2.1.

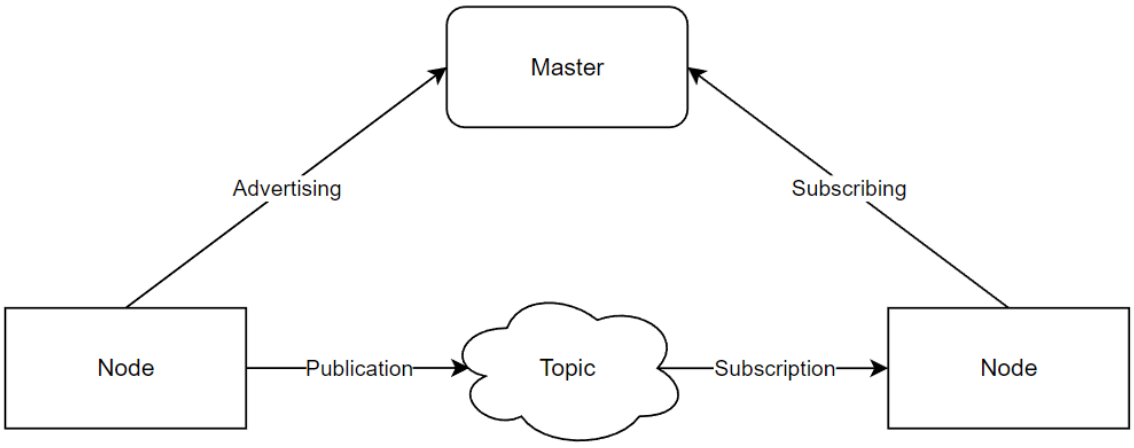


Figure 2.1: ROS environment schematic. A node "publishes" a message to a topic. Another node "subscribes" to that same topic. The nodes are registered by the system master. Created in "draw.io".

Nodes are executable files within a ROS package that use a ROS client library to communicate with other nodes. Nodes can publish or subscribe to messages within the ROS environment, they can also provide or use a Service [11]. Nodes perform actions within a ROS system, including hardware drivers, algorithms and user interfaces [10].

Topics are gatherings of information that Nodes publish or subscribe to. The Topic type is defined by the message type published to it. All communication between Nodes in a ROS environment occur using Topics. Multiple Nodes can subscribe to or publish to the same Topic [12].

Master provides a naming and registration service for the nodes in a system. The Master tracks publishers and subscribers to topics and services within the ROS system. This enables Nodes within the system to locate each other and allows for communication within the system [13].

The *Launch* tool allows for easily starting and managing multiple nodes simultaneously, including changing node parameters. This is handled in text files with the `.launch` ending. launch files can be run from other launch files or using the terminal command `roslaunch` [14].

2.3 LiDAR

Light Detection and Ranging (LiDAR) is based on laser triangulation ranging principle. A LiDAR sensor consists of an emitter and a receiver as shown in Figure 2.2. The emitter emits a laser pulse wave. The pulse wave propagates, then reflects on an object which it meets with and returns to the receiver. The time for the laser pulse wave to travel forth and back to the LiDAR sensor, Δt, can be measured with the sensor and the distance to that object can be calculated using the following equation

$$d = \frac{1}{2n} c\Delta t \tag{2.6}$$

where c denotes the speed of light, n the index of refraction for the propagation medium ($n \approx 1$ in air). The result is divided by two as the laser wave has travelled that distance twice for it to return [15].

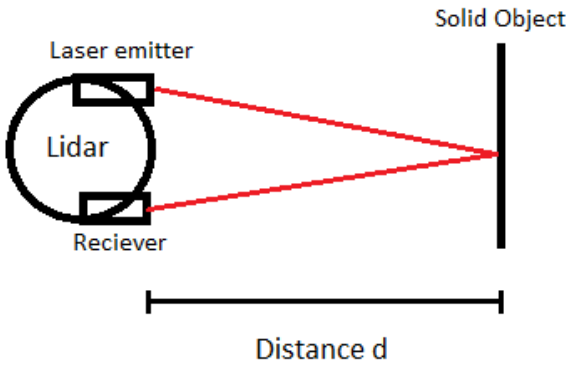


Figure 2.2: LiDAR sensor distance measurement using laser emitter and receiver.

2.4 Stereo Vision

A stereoscopic camera consists of two or more image sensors, calibrated and capturing images simultaneously. By capturing an image from two viewpoints, 3D vision and depth can be extracted. To calculate depth, the system need to examine points and pixels of the two or more images, determine which pixels correspond to each other, and then compute 3D coordinates using the geometrical relation between cameras obtained by calibration. The relative 3D coordinates of a recognised point is displayed in Figure 2.3.

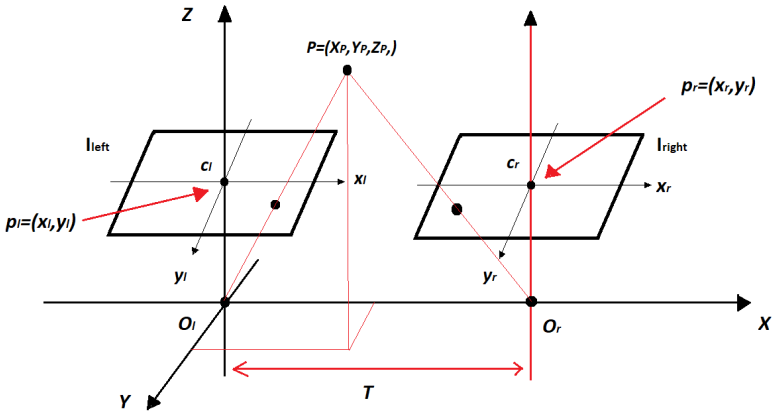


Figure 2.3: I_{left} and I_{right} represent left and right image planes respectively. O_l and O_r represents centres of projection. The position of point $P = (X_P, Y_P, Z_P)$ is determined by comparing left and right projections p_l and p_r in respective image plane.

Calculations for the coordinates of the spatial point $P = (X_P, Y_P, Z_P)$ represented in Figure 2.3, using trigonometry, are as follows.

$$x_l = \frac{X_P f}{Z_P}; \quad x_r = \frac{(X_P - T)f}{Z_P}; \quad y_l = y_r = \frac{Y_P f}{Z_P} \tag{2.7}$$

$$Z_P = f\frac{T}{x_l - x_r} = f\frac{T}{d}; \quad X_P = x_l\frac{T}{d}; \quad Y_P = y_l\frac{T}{d} \tag{2.8}$$

The position of point P is calculated using its projections in left and right camera lens p_l and p_r, using the known distance T between centres of projection O_l and O_r and using the focal length f of the cameras. Assuming the 3D coordinate frame has origin in left camera O_l. Disparity $d = x_l - x_r$ is calculated using the difference distance in amount of pixels of the points projection in the two lenses [4].

The disparity is inversely proportional to the distance, a point closer to the cameras will have higher disparity than one far away. An error of one pixel proves far more faulty at low disparity and a simple stereo system is thus likely to be inaccurate at far distances. Accuracy is proportionate to image resolution and proportionate to distance between cameras T [3].

2.5 Odometry

Odometry is the use of data from motion sensors to determine the mobile robots position and change in positioning relative to previous known position. Odometry information is commonly derived using wheel encoders or an Internal Measurement Unit (IMU). Odometry can also be derived using sensors such as cameras or LiDAR sensors by matching the current time step data to that of earlier time steps and match features of structural motion, thus determining movement [16]. In ROS the odometry information message contains the mobile robot position and velocity, represented as position with covariance and twist with covariance [17].

2.6 Real Time Appearance Based Mapping (RTAB-map)

Real Time Appearance Based Mapping (RTAB-map) is an open source SLAM library. RTAB-map implements loop closure detection with a memory management approach to satisfy requirements for long term and large size environment mapping. RTAB-map was created by Mathieu Labbé and Francois Michaud from the University of Quebec and released in year 2013 as a purely visual based mapping library. RTAB-map grew to implement graph based SLAM in 2017, and then to implement 2D and 3D LiDAR compatibility to allow implementation and comparison of a variety of 2D and 3D solutions for a wide range of applications. RTAB-map was also evolved to a ROS package for further robotic accessibility and possibility to implement and compare SLAM approaches [8].

The RTAB-map ROS package is named `rtabmap_ros` with the main node named `rtabmap`.

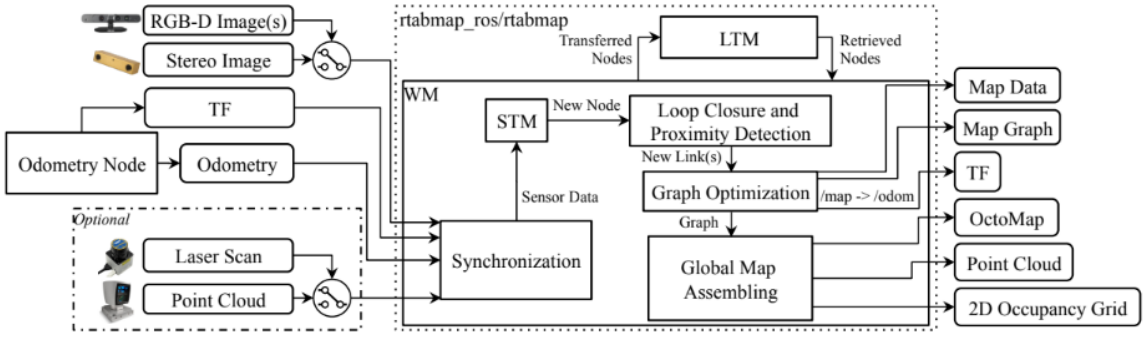


Figure 2.4: Block diagram of `rtabmap` ROS node structure [8].

The required inputs are one odometry source providing movement and Transform coordinate Frames (TF) information, as well as one sensor source, preferably an RGB-D or stereo image. It is also possible to work only with a Laser scan and odometry source. The inputs are synchronised and forwarded to the graph-SLAM algorithm using both Working Memory (WM) and Long Term Memory (LTM) to match current map features with existing ones to find loop closures. The final outputs are various mapping Data, point cloud, occupancy grid, and odometry correction published as TF [8].

Loop closure detection signifies a key feature in real life SLAM. Loop closure detection is the ability for the mobile robot performing SLAM to detect when the current

observations correspond to those of a previously visited location. Such a detection signifies that the robot is on a previously mapped position and the current position should be the same as that previous position. The mobile robot movement information can drift over mapping sessions and thus detecting loop closures and correcting movement drift can be crucial for obtaining coherent maps. The loop closure detection in RTAB-map is visual based and not implemented using 2D LiDAR sensor data [8, 18]. An example of loop closure detection and correction is displayed in Figure 2.5.

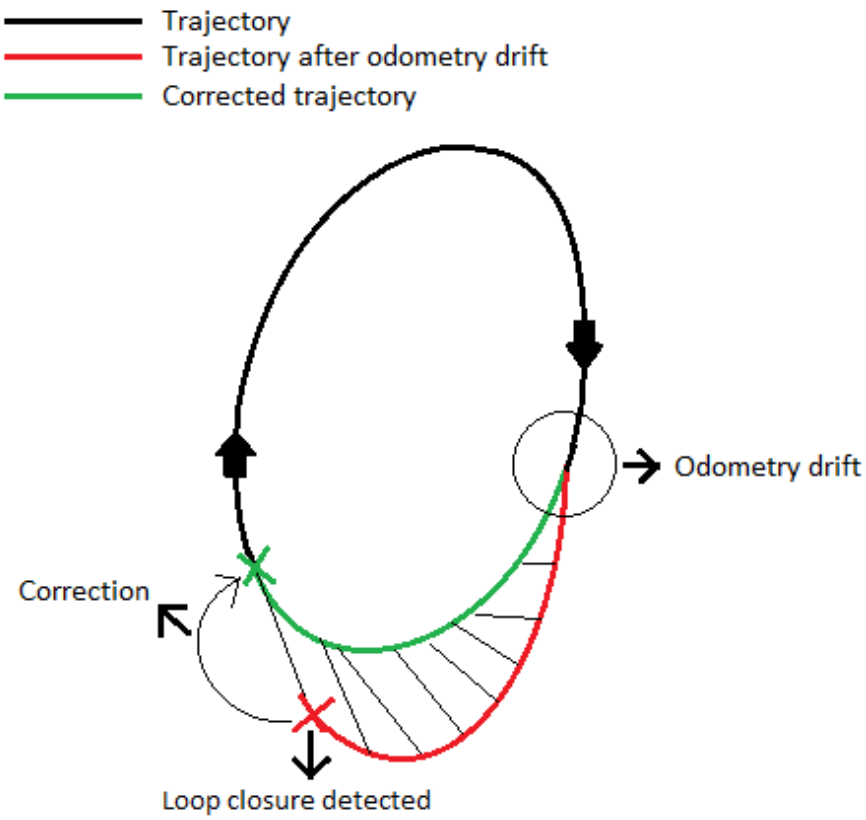


Figure 2.5: Example of loop closure detection and correction. The loop closure is detected at position marked by the red X and the faulty trajectory is corrected to correspond to the detected position marked by the green X.

3 Method and Implementation

3.1 Overview of the system

The SLAM was performed using a *remote mapping* setup. With the remote mapping setup, all SLAM computation and visualisation was performed on the laptop. The role of the RC car with sensors and minicomputers was to gather information about the surroundings and be controllable from the client.

On the RC car the RP4 was the communication hub and was the only unit communicating with the laptop. It was connected to the RP3 by Ethernet cable and to Arduino via HDMI cable. The movement instructions were sent to the Arduino from the RP4. The RP4 powered one camera module and the LiDAR. The RP3 only powered one camera module and broadcasted that information to the RP4. The Arduino constantly ran one Arduino ROS program which allows for varied input from the RP4. Movement instructions were communicated from the laptop to the RP4 and then relayed to the Arduino Uno. The Arduino Uno controlled the DC motors using analogue output. The schematic for the RC car hardware is displayed in Figure 3.1.

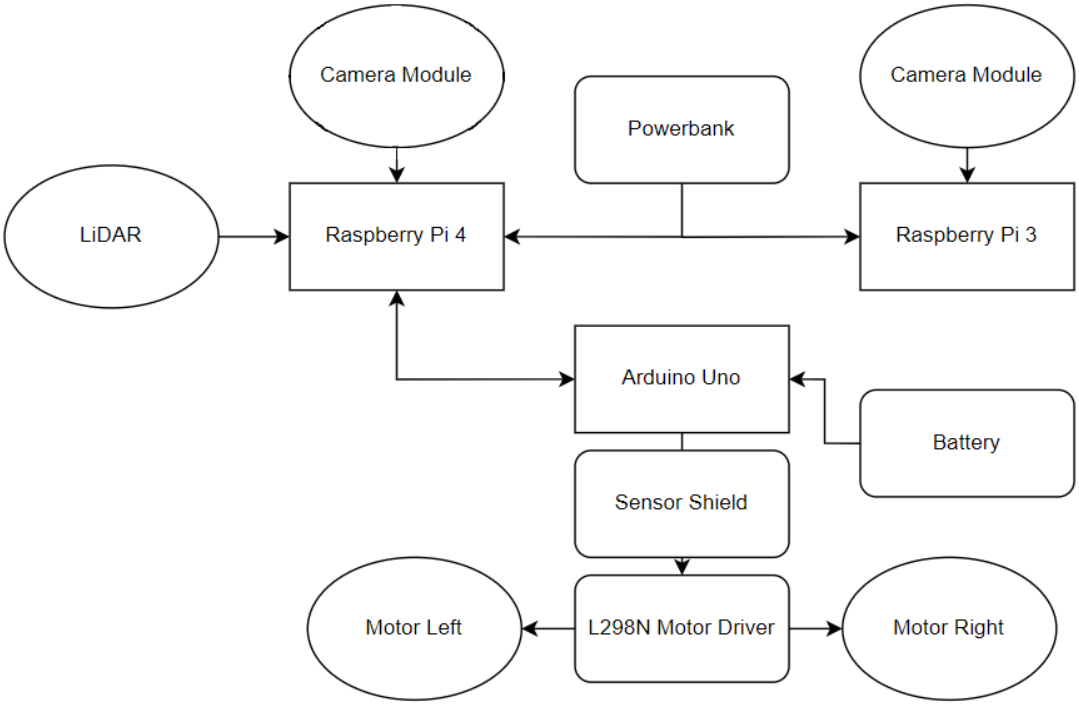


Figure 3.1: RC car hardware setup. The RP4 runs the left camera module, powers the RPlidar A1, and relays movement instructions to the Arduino Uno. The RP3 runs the right camera module and sends this data to the RP4. Created in "draw.io".

The remote mapping setup with base image synchronisation was implemented by communicating all information to and from the RC car via the RP4. On the RP3, the right camera `raspicam` node gathered compressed RGB images from the right camera module. This image stream was sent to the RP4 through Ethernet. The RP4 ran four ROS nodes. Firstly, the left camera `raspicam` node gathered compressed RGB images

from the left camera module. Then a custom node combined left and right image streams using a custom ROS message before sending this message to the laptop. This ensured that the left and right camera images arrived simultaneously to the laptop, satisfying base image synchronisation. The third node on the RP4 was the `rplidar_ros` node, gathering information from the LiDAR sensor and sending this to the laptop. Finally, a `rosserial_arduino` node was run, allowing the RP4 to relay movement instructions from the laptop keyboard to the Arduino Uno. The Arduino Uno continuously ran a `rosserial_arduino` node, requesting movement instructions from the RP4 and converting these to motor DC output. The system setup, with basic ROS overview, can now be represented by the following Figure 3.2.

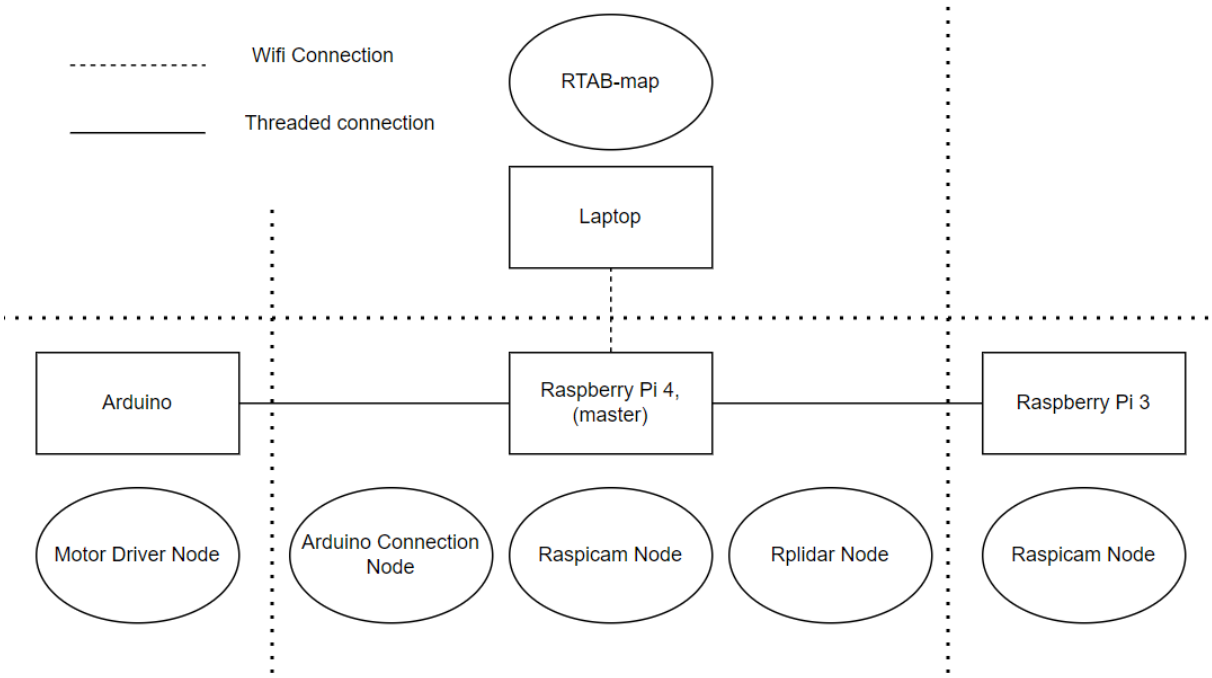


Figure 3.2: System setup with basic ROS overview. The RP4 communicates with the Arduino Uno using an HDMI cable, to the RP3 using an Ethernet cable and to the laptop via WiFi. Created in "draw.io".

The motivation for using the remote mapping setup was that the system had to satisfy a number of requirements on communication and computation. Firstly the ROS computation on odometry and SLAM proved too costly to continuously run on the RP4. Thus the bulk of the computation was chosen to be processed on a laptop connected to the RP4 by WiFi. This ROS setup was based on Labbé remote mapping work and is visualised by Figure 3.3 [20]. In this thesis, the RGB-D was exchanged to a compressed stereo image message and an external odometry source was not used as odometry was derived from stereo images or laser scan.

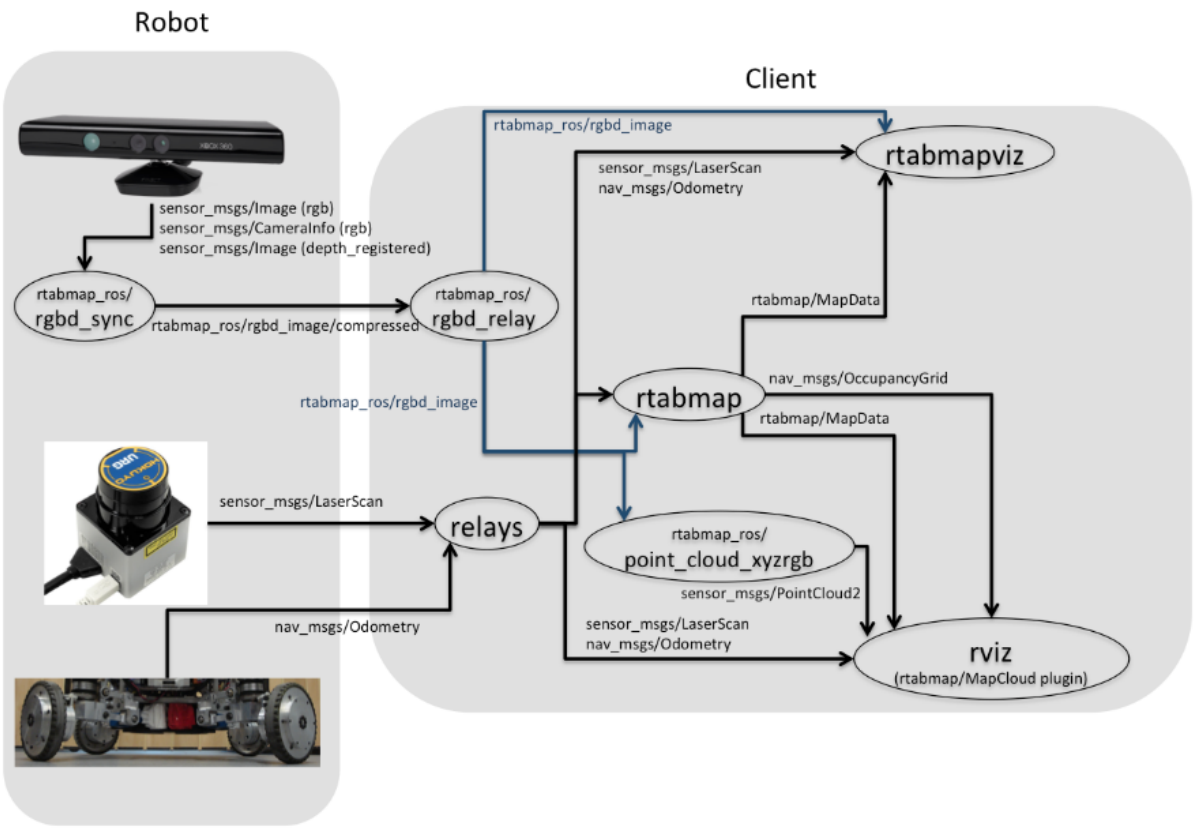


Figure 3.3: Remote mapping with RTAB-map [20].

3.2 Hardware and Components

The following list are the key components of the project.

- Raspberry Pi 4 model b
- Raspberry Pi 3 model b
- Arduino Uno
- Arduino Sensor Shield v5.0
- L298N Motor Driver Board
- RPlidar A1
- Raspberry Pi Camera Module 2, 2x
- RC car (basic components, such as wheels, battery, DC motor)
- Powerbank, 18W 10000mAh
- Cables (USB:s, HDMI, power cables, electrical wires)
- Laptop

3.2.1 Raspberry Pi 4 model b

One of the minicomputers used is a Raspberry Pi 4 model b (RP4) with a 64 GB SD card for memory. Quad core Cortex-A72 (ARM v8) 64-bit SoC @ 1.5GHz processor, RAM memory of 4 GB, two USB 3.0 and two USB 2.0 ports, USB-C Power Supply, two Micro HDMI ports and CSI camera port [21].

3.2.2 Raspberry Pi 3 model b

The second minicomputer used is a Raspberry Pi 3 model b (RP3) with a 64 GB SD card for memory. Quad Core 1.2GHz Broadcom BCM2837 64bit CPU, 1GB RAM, BCM43438 wireless LAN and Bluetooth Low Energy (BLE) on board, 100 Base Ethernet, 40-pin extended GPIO, 4 USB 2.0 ports, 4 Pole stereo output and composite video port, Full size HDMI, CSI camera port [22].

3.2.3 Arduino Uno

Arduino Uno is a microcontroller board based on the ATmega328P. It has 14 digital input/output pins (of which 6 can be used as PWM outputs), 6 analog inputs, a 16 MHz ceramic resonator (CSTCE16M0V53-R0), a USB connection, a power jack, an ICSP header and a reset button [23].

3.2.4 RPlidar A1

In this thesis the LiDAR sensor used was an RPlidar A1. The RPlidar A1 performs a 2D 360 degree scan in the plane it is mounted. The sensor consists of a range scanner system mounted in the dome like shape on its top, a motor system spinning the top dome in 360 degrees, and a customizable platform allowing mounting by screws. The top dome, with emitter and receiver, spins at a frequency of 5.5 Hz and collects approximate 2000 data point per second, this equates to approximately one point at every degree. The distance range is 0.15-6 m and the resolution error less than 1% of the distance. The sensor can be connected with USB cable [24]. A top view of the RPlidar A1 is displayed in Figure 3.4.

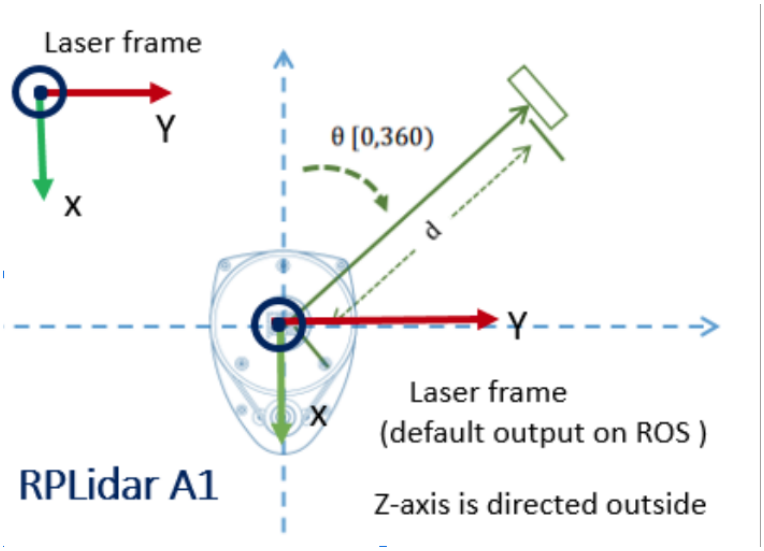


Figure 3.4: RPlidar A1 from top view

3.2.5 Raspberry Pi Camera Module v2

The Raspberry Pi Camera Module 2 is a camera module for Raspberry Pi with a Sony IMX219 8-megapixel image sensor. Capable of 3290x2464p images or 1080p 30fps video [25].

3.2.6 RC car

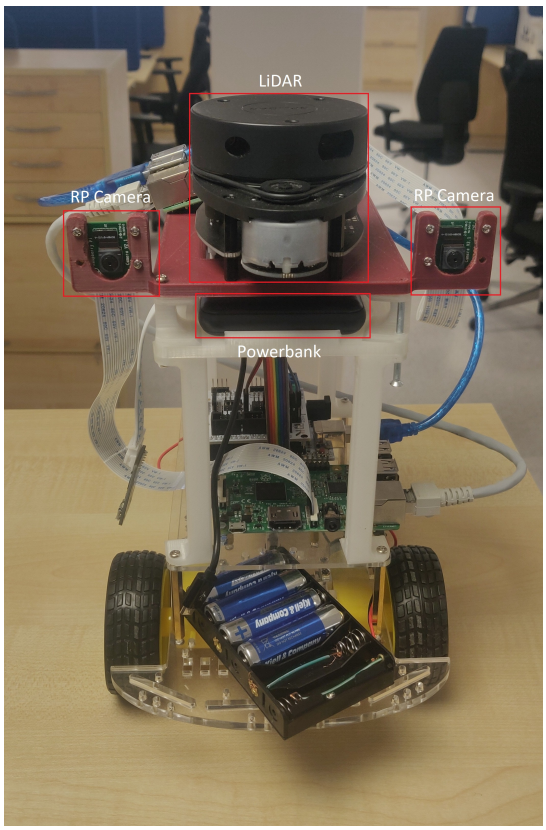
The RC Car based on the Playknowlogy car frame from Kjell & Company. It contained two DC motors, an Arduino Uno and a L298 Motor driver [26].

3.2.7 Laptop

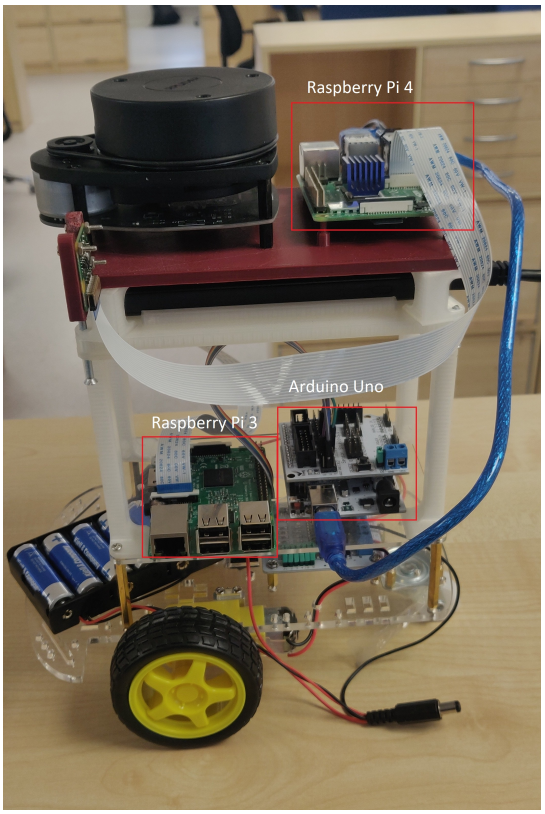
The Laptop used to perform computation and visualization is a Lenovo ideapad 720S-14IKB. It has an Intel i7-8550U 1.80GHz processor and 8GB RAM.

3.3 Assembling the robot

The Playknowlogy RC car was assembled according to contained instructions. The car frame with motors was mounted and the Arduino was connected to the L298 motor driver. Holes were drilled in the top layer allowing for the RP3 to be mounted. A taller frame of 10cm was 3D printed and mounted on top of this base frame. This taller frame contained holes for the RP4, a pocket for the powerbank, holes to mount the RPlidar A1 and two slots for the Raspberry Pi camera modules.



(a) Car from front



(b) Car from side

Figure 3.5: Pictures of RC car. Left image marks the camera modules, LiDAR and powerbank. Right image marks the location of RP3, RP4 and Arduino.

The LiDAR was mounted on top of the car to allow for full 360 degree range. The camera modules were mounted 13 cm apart by each side and 24 cm above ground. On the middle layer the RP3 and the Arduino Uno were placed. The bottom layer contained the L298 motor driver and an external battery pack for the motors. This resulted in the base connections of the hardware as shown in Figure 3.1.

3.4 Software and development tools

The following software and packages were used in the project.

- Ubuntu 20.04 server 64 bit (installed on both Raspberry Pi:s)
- ROS Noetic
- rplidar_ros, ROS library
- rosserial_arduino, ROS library
- raspicam_node, ROS library
- rtabmap_ros, ROS library
- Rviz, 3D visualization program for ROS
- OpenCV, Open source computer vision library

3.4.1 Ubuntu 20.04

The Operating System (OS) used on the RP4, RP3 and laptop was Ubuntu 20.04 server. This was chosen due to being a lightweight OS compatible with all required software and with a large knowledge pool

3.4.2 ROS Noetic

The ROS version "Noetic Ninjemys" was used on all units on this project. It is the latest version of ROS 1, with lots of documentation. It fulfills all requirements to operate RTAB-map for the purposes of this project.

3.4.3 OpenCV

Open Source Computer Vision Library (OpenCV) is an open source computer vision and machine learning library. The library contains both classic and state of the art computer vision and machine learning algorithms and is used by a multitude of established companies such as Google and Microsoft. Leaning mostly towards real-time vision applications it is a good fit for use in robotics and is supported in ROS as well as used for most ROS computer vision applications [19].

3.5 Software installation and configuration

3.5.1 Operating System

The chosen OS for the Raspberry Pi:s was Ubuntu Server 20.04. This OS is lightweight and compatible with the latest ROS 1 version ROS Noetic. The OS was installed following the Ubuntu tutorial "How to install Ubuntu Server on your Raspberry Pi" [27] and using the "Raspberry Pi Imager for Ubuntu" to flash the *Ubuntu Server 20.04 64 bit* OS to a 64GB microSD card. Initial WiFi configuration was created using the Raspberry Pi Imager Advanced Options, automatically connecting to same WiFi as laptop on boot and allowing for a headless setup using ssh Linux command from laptop. This was flashed to the microSD card which was then slotted into the Raspberry Pi and booted. After boot and login, the packages was updated to the latest version and the unit was rebooted using terminal commands.

```
$ sudo apt update
$ sudo apt upgrade
$ sudo reboot now
```

This concluded the initial Ubuntu Server installation.

3.5.2 ROS

On all units ROS Noetic was installed using the ROS wiki installation "Ubuntu install of ROS Noetic" [28]. On the laptop the *ros noetic Desktop-Full version* was installed, containing various extra packages. On the the Raspberry Pi:s the *ros noetic ROS-Base* was installed, containing base packaging, build and communication.

After installation the environment was setup by sourcing the ROS setup file in the .bashrc file, making sure the script was sourced in every terminal.

```
$ echo "source␣/opt/ros/noetic/setup.bash" >> ~/.bashrc
$ source ~/.bashrc
```

To create and build custom ROS packages, necessary packages were installed and the *rosdep* package was initialised.

```
$ sudo apt install python3-rosdep python3-rosinstall python3-
   rosinstall-generator python3-wstool build-essential
$ sudo rosdep init
$ rosdep update
```

Now ROS Noetic was fully installed and initialised. A workspace was now created in the home directory and sourced.

```
$ mkdir -p ~/catkin_ws/src
$ cd ~/catkin_ws
$ catkin_make
$ source devel/setup.bash
$ echo "source␣~/catkin_ws/devel/setup.bash" >> ~/.bashrc
```

Packages could now be created and used within this workspace.

For this thesis, the custom package containing all custom programs and launch files was named "stereo_formatting" and can be found on the thesis GitHub page referred to in Appendix 5.2.

3.5.3 ROS IP connection

For information to be communicated correctly between units in ROS the RP4 and laptop needs to communicate over WiFi and the RP4 and RP3 needs to communicate over Ethernet. Static WiFi IP address was set on the laptop by using the regular WiFi settings, the laptop was assigned WiFi IP 192.168.1.120. On the RP3 and RP4 static IP addresses were configured by editing system files. The RP4 was assigned WiFi IP 192.168.1.121 and Ethernet IP 10.0.0.1. The RP3 was assigned Ethernet IP 10.0.0.2 and WiFi IP 192.168.1.122. The RP3 WiFi was not used in ROS, only using ssh from laptop.

For ROS to recognise units in the system and correctly transfer information, the IP addresses had to be specified in ROS. This was done using the three parameters *ROS_MASTER_IP*, *ROS_HOSTNAME* and *ROS_IP*. These parameters had to be set in every terminal to prevent ROS from using the local unit as master. The parameters were added to the .bashrc file for simplicity. *ROS_MASTER_IP* specifies the master in the system, in this case it was the RP4 with IP 192.168.1.121. *ROS_HOSTNAME* declares the address of nodes run on that unit. The full ROS IP setup is displayed in Table 3.1.

Table 3.1: Unit IP addresses, and corresponding ROS configuration IP addresses.

Unit	IP Adress	ROS_IP	ROS_HOSTNAME	ROS_MASTER_IP
RP4	192.168.1.121, 10.0.0.1	192.168.1.121	192.168.1.121	192.168.1.121
RP3	10.0.0.2	10.0.0.2	10.0.0.2	10.0.0.1
laptop	192.168.1.120	192.168.1.120	192.168.1.120	192.168.1.121

Finally, the IP addresses of all units were added to each unit `/etc/hosts` file. This allowed for the ROS messages to be trusted and transferred correctly.

3.5.4 Manually controlling the car

The motors of the car were controlled from the Arduino which in turn was controlled from the RP4 in accordance with Figure 3.1. To manually control the car the *rosserial_arduino* library was installed as described in ROS wiki [29]. Binaries were installed on RP4 using terminal commands

```
$ sudo apt-get install ros-noetic-rosserial-arduino
$ sudo apt-get install ros-noetic-rosserial
```

A C++ program for controlling the car named was written in a package within the ROS project workspace. The script is named "varied_control.cpp" 5.2 and works as follows.

The packages *ros.h*, *geometry_msgs/Twist.h* and *Arduino.h* were included. The topic message controlling the car to be sent from the laptop is on the ROS "Twist" form, enabling car control from the keyboard. Variables were declared specifying motor output pins and for later use. Functions were declared.

The `void setup()` function runs once to initialize the Arduino pins as outputs and setting 0 current to start with motors idle. The node is initialized as a subscriber.

The `void loop()` function waits for a new message from the topic "turtle1/cmd_vel", before entering the `void drive()` function.

The `void drive()` function reads values from the message and calls to either the `void accelerate()` or `void turn()` function. The message will display one of the values $x = 2$, $x = -2$, $z = 2$, $z = -2$ and this value is passed along. The laptop keyboard input corresponds as the following. $x = 2$ is forward arrow, $x = -2$ is backwards arrow, $z = 2$ is right arrow and $z = -2$ is left arrow.

The `void accelerate()` and `void turn()` functions changes right and left speed variable values depending on the direction passed along in the argument. They then finally call to the `void drive()` function which writes analogue values to motors. The maximum value to write is $2^8 - 1 = 255$. This C++ program can be found in the Appendix 5.2.

3.5.5 Raspberry Pi Cameras

To use the Raspberry Pi camera modules in ROS, the Raspberry Pi:s first had to be configured. Since the OS used was Ubuntu server 64 bit using ARM64 architecture and not the Raspbian OS, the cameras did not work by default. To enable compatibility with the ARM64 OS, advice from the ROS discourse thread "Raspicam_node on ARM64 Raspberry pi" by user "anfederman" was used [30]. After basic install and network configuration of the OS, the following configuration was performed to enable Raspberry Pi camera compatibility.

Camera support libraries were installed using terminal command:

```
$ sudo apt install libraspberrypi-bin libraspberrypi-dev
```

`/boot/firmware/config.txt` file was opened using terminal command

```
$ sudo nano /boot/firmware/config.txt
```

The two following lines were added

```
start_x=1
gpu_mem=128
```

Camera was tested by capturing an image as test.jpg

```
$ raspistill -o test.jpg
```

To use in ROS the source version of the node named "raspicam_node" from Ubiquity Robotics was installed using instructions in the official Github page [31].

The node was run using a copy of a launch file included in the *raspicam_node* package, with base settings changed to disable exposure mode, using a 640×480 resolution with 10Hz frame rate and a shutter speed of $7000\mu s = 0.007s$.

```
<param name="width" value="640"/>
<param name="height" value="480"/>

<param name="exposure\_mode" value="off"/>
<param name="framerate" value="10"/>
<param name="shutter\_speed" value="7000"/>
```

3.5.6 RTAB-map

The RTAB-map ROS package was built on the laptop by installing ROS binaries according to the RTAB-map ROS GitHub page [32]. RTAB-map was used on the laptop. The RTAB-map binary install included launch files used to start the processes. For this project the custom launch file test.launch was used.

3.5.7 Stereo Camera setup

The RP4 was set as the ROS master unit and was connected to the RP3 using an Ethernet cable. Static Ethernet IP addresses was configured on both Raspberry Pi units, displayed in Table 3.1. The IP addresses were configured to allow for WiFi connection between laptop and RP4, and Ethernet connection between RP3 and RP4.

A custom message containing left and right camera info and compressed images was created. On the RP4 a ROS node was written collecting the information and packing as this custom message before publishing that custom message. In this stage all message header times were set to the RP4 message time. This was done to trick RTAB-map that the images were fully synchronised as well as being able to disregard wrong system time on the RP3, note that this setup is not optimal. The C++ program for this node was named `synchronizer.cpp` and can be found in the Project GitHub. On the laptop a corresponding script was created, unpacking that information and converting to was data using OpenCV in ROS before publishing using ROS image_transport. The C++ program for that node was named `unpack.cpp` and can be found in the Project GitHub 5.2.

The resulting image data flowchart is displayed in Figure 3.6.

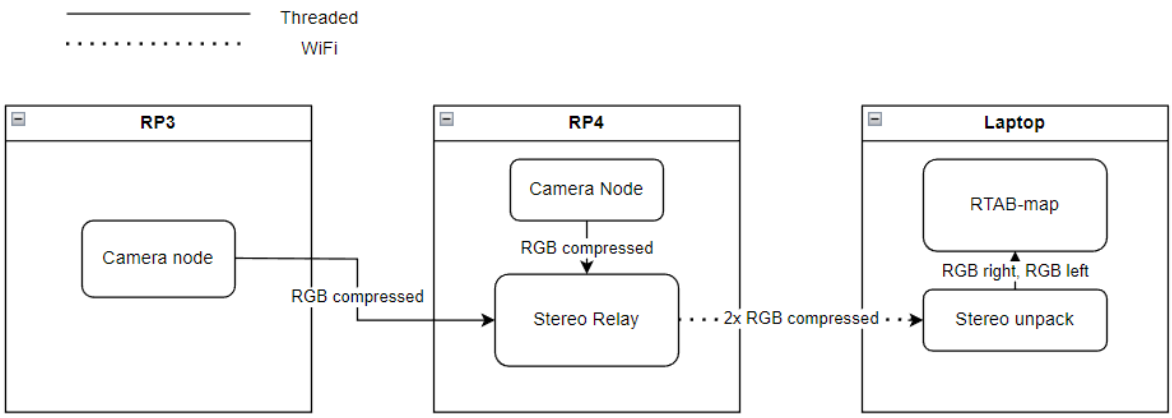


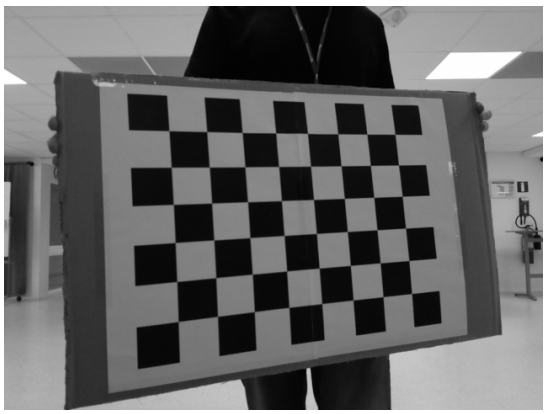
Figure 3.6: Image data flowchart. RP3 sends the right camera compressed RGB image stream over Ethernet to the RP4. The RP4 packages this message with the left camera compressed RGB image stream and sends to the laptop as one message. The Laptop unpacks and converts to raw RGB images. Created in "draw.io".

3.5.8 Calibration

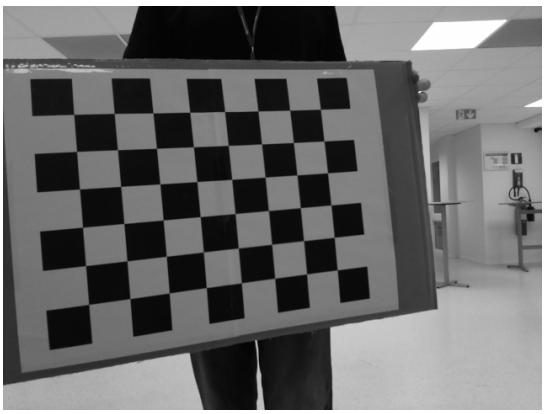
After stereo implementation the camera was calibrated using the ROS *camera_ calibration* package with the stereo calibration option. After synchronised raw images were sent from the RP4 and unpacked on the laptop, the following command was run in a laptop terminal.

```
$ rosrun camera_calibration cameracalibrator.py --size 6x8 --
   square 0.073 right:=/stereo/right/image_raw left:=/stereo/
   left/image_raw right_camera:=/stereo/right left_camera:=/
   stereo/left --no-service-check
```

This program calibrates the images based on different orientations and angles of the easy to find square edges of the chessboard pattern. This gave proper calibration files including distance between cameras. Using those calibration files the depth could be recognized in the images.



(a) Left camera calibration image



(b) Right camera calibration image

Figure 3.7: One of many left and right image pairs used in calibration of stereo cameras.

3.6 The SLAM configuration

The SLAM computation and visualisation, performed on the laptop, was launched using the launch file `test.launch` which can be found in the thesis GitHub page or in the Appendix. This launch file was created based on launch files from the RTAB-map "Setup RTAB-Map on Your Robot!" page [33]. The most important parts of the launch file are explained in text.

```
<launch>

<include file="$(find␣stereo_formatting)/launch/laptop.launch"
   />

<node name="scan_sync" pkg="stereo_formatting" type="scan_sync
   " />
```

The `<launch>` line signifies the start of the launch file. Another laptop launch file `laptop.launch` containing the node for unpacking the compressed stereo image message is called. The `scan_sync` node is started. This node sets the RPlidar A1 scan data time to the laptop system time, negating the often non synchronised system clock on the RP4.

Basic arguments and frame names are set. Essential TF between system coordinate frames are communicated using `static_transform_publisher` nodes. The argument `"hector_odom" default="false"` is set as true if LiDAR odometry is to be used instead of camera odometry for the session.

```
<group ns="/stereo" >
    <node pkg="stereo_image_proc" type="stereo_image_proc"
       name="stereo_image_proc"/>

    <!-- Odometry -->
    <node if="$(arg␣hector_odom)" pkg="hector_mapping" type="
       hector_mapping" name="hector_mapping" output="screen">
```

The `stereo_image_proc` node, subscribing to the rectified left and right image data and converting these stereo images to depth, is started. If LiDAR odometry is used for the session, the `hector_mapping` node is started.

```
    <node unless="$(arg␣hector_odom)" pkg="rtabmap_odom" type=
       "stereo_odometry" name="stereo_odometry" output="screen
       ">
```

If instead stereo camera odometry is used, the `stereo_odometry` node from the *rtabmap_odom* package is started.

```
<node pkg="rtabmap_sync" type="stereo_sync" name="stereo_sync"
    output="screen">
```

The `stereo_sync` node from the *rtabmap_sync* package is started for synchronising the left and right image to an RGB-D message.

```
<group ns="rtabmap">
  <node name="rtabmap" pkg="rtabmap_slam" type="rtabmap"
     output="screen" args="--delete_db_on_start">
```

The SLAM computations are performed using the `rtabmap` node from the *rtabmap_slam* package. Arguments within the node are set for subscribing to the correct topics and parameters for the specific SLAM setup are set.

```
  <node if="$(arg␣rviz)" pkg="rviz" type="rviz" name="rviz"
     args="-d␣$(find␣stereo_formatting)/config/setuprviz.rviz"
      output="screen" launch-prefix="bash␣-c␣'sleep␣20;␣$0␣$@'
     "/>
```

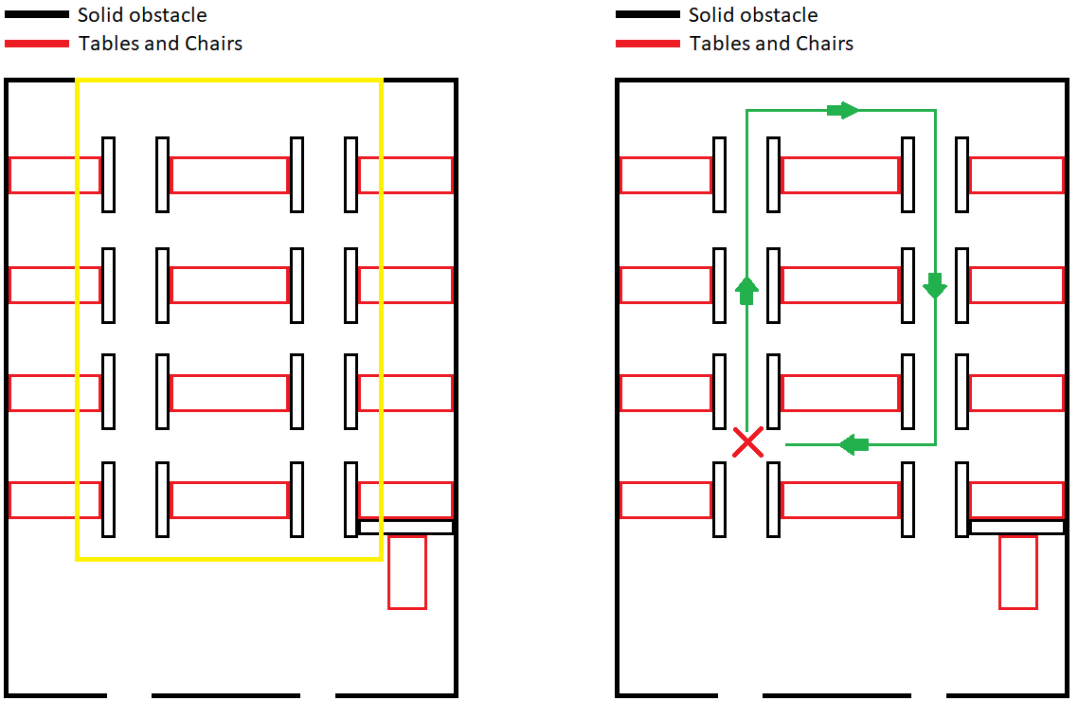
```
</group>
</launch>
```

Finally, the general purpose ROS user interface "Rviz" is started using the custom setup `setup.rviz` file and starting the `rviz` node. The launch file is then ended by `</launch>`.

3.7 Running the Robot

The robot SLAM setup was ran by performing the following steps. In the Appendix, the ROS workspaces for the project can be found with all the configuration and launch files 5.2. After connecting to the RP3 and RP4 from the laptop using ssh and confirming network connection proper, the following commands were run.

1. These commands were run on the RP4:
   (i) roscore
   (ii) roslaunch stereo_formatting stereo.launch
2. This command was run on the RP3:
   (i) roslaunch raspicam_node custom.launch
3. These commands were run on the laptop:
   (i) roslaunch stereo_formatting test.launch
   (ii) rosrun turtlesim turtle_teleop_key
4. Robot was placed in corridor of office space.
5. Robot was placed on the left hand side of the office and driven manually in a circular motion around the office space. The approximate office space layout and path is displayed in Figure 3.8.



(a) Sketch of office layout. The yellow box shows the main area that was mapped.

(b) Sketch of office layout. The red X shows where mobile robot was initially placed. Green line the trajectory.

Figure 3.8: Sketch of office layout, black lines marks walls and shelves, red areas marks tables and chairs. Left image displays main mapped area. Right image displays initial robot position and path.

3.8 Mapping and comparison

The SLAM problem, as described in Section 2.1, is theoretically solved [5]. However, in practise, that is not the case since conditions and data are not ideal. Most realised SLAM setups require two main sensor data components.

1. Sensor data of surroundings
2. Odometry

This corresponds to the remote mapping setup in Figure 3.3.

In this thesis, the sensor data was derived from the stereo camera setup and RPlidar A1 both separately and together. The choice of using either sensor or both at once was made using the parameters in the `test.launch` launch file as described in Section 3.6.

Odometry, as described in Section 2.5, was derived using data from one of the two sensor types, stereo camera and 2D LiDAR. The choice of the sensor from which to derive odometry, and switching between the two options, was performed in the `test.launch` launch file.

4 Results and discussions

4.1 Depth vision using cameras

The image displayed in Figure 4.1 is a captured screenshot of a live stereo image disparity feed. It was obtained using the *stereo_ image_ proc* ROS package with the `stereo_view` node. After stereo images were calibrated and the rectified stereo pair was published on the laptop, the following command was run in a laptop Linux terminal.

```
$ rosrun image_view stereo_view stereo:=/stereo image:=
    image_rect_color
```

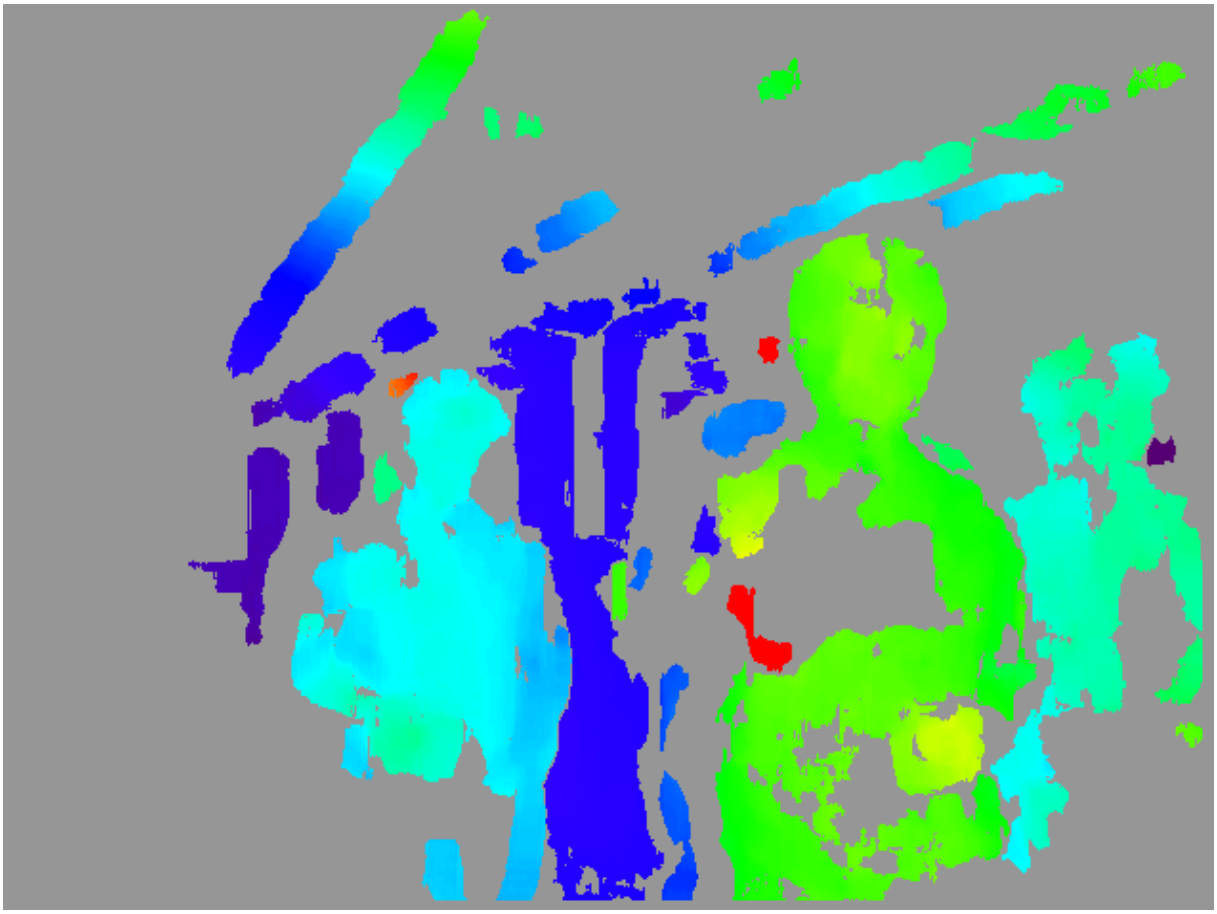


Figure 4.1: Initial disparity Image, with three people in image.

The calibrated Stereo Camera setup is able to see depth in an image, as shown in Figure 4.1. This shows a frame of the continuous disparity map. In the color gradient, red represents the objects closes to cameras and purple represents furthest away. The colors in between from closest to furthest are yellow, green, light blue, dark blue. Grey areas show parts where depth fails to be computed. This can occur due to distance being either too far away, too close to cameras, or due to the object being difficult to detect. The setup used is able to compute a depth map based on the position of pixels relative to the cameras, although accuracy is not measured. This proves the setup can in some way be used in SLAM.

4.2 Mapping of Office Space

4.2.1 First office map

This is the first SLAM session, using only the stereo camera setup with odometry from stereo cameras. This session used the custom ROS launch file `brtabmap.launch`.

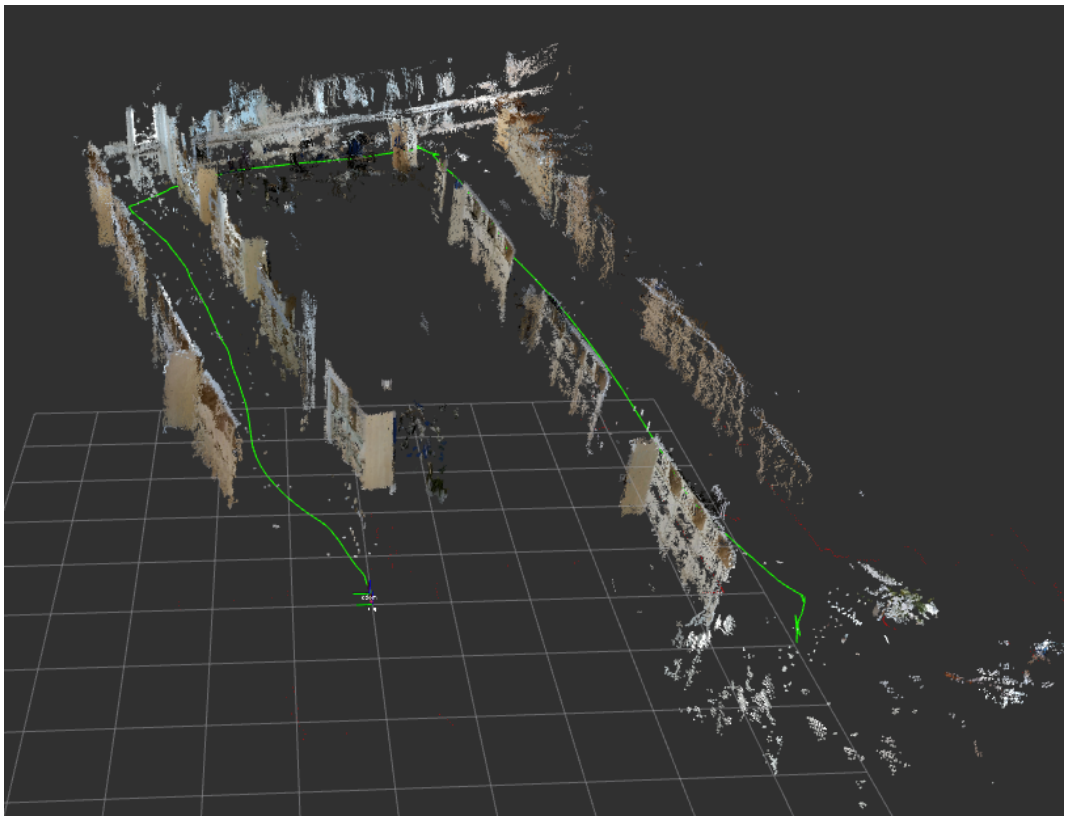


Figure 4.2: First office map using only the stereo camera setup. Image is a screenshot from Rviz.

The car started on left side at the green trajectory line and lost location at end of the green trajectory line on right side of image. At this point SLAM was not locked to three degrees of freedom and position drifts downwards throughout the mapping. This is sub optimal when mapping in a single plane.

Although the desks along the hallways are visible, they are grainy which negatively affects the system ability to preform loop closure detection.

The camera odometry is able to track the car during turns and loses position first when the car approached the open area displayed at bottom of office seen in sketch Figure 3.8. With a 640×480p resolution the setup looses accuracy quickly with distance and seems to struggle with vision when objects are further away than approximately 4 meters.

It can be seen that along robot path small pixels appear seemingly in the air. This noise could be due to multiple factors such as poor stereo camera calibration, poor synchronisation, robot instability, or poor camera quality. They could possibly be removed using another RTAB-map configuration.

4.2.2 Stereo camera only with stereo camera odometry

This SLAM session was performed using only the stereo camera setup with odometry from stereo cameras. Mapping was locked to one plane of mapping and not driven to larger open area. The image was captured from top view. The car was driven from the bottom left position in a circle around the office, then driven through office isles. This session used the custom ROS launch file `test.launch`.

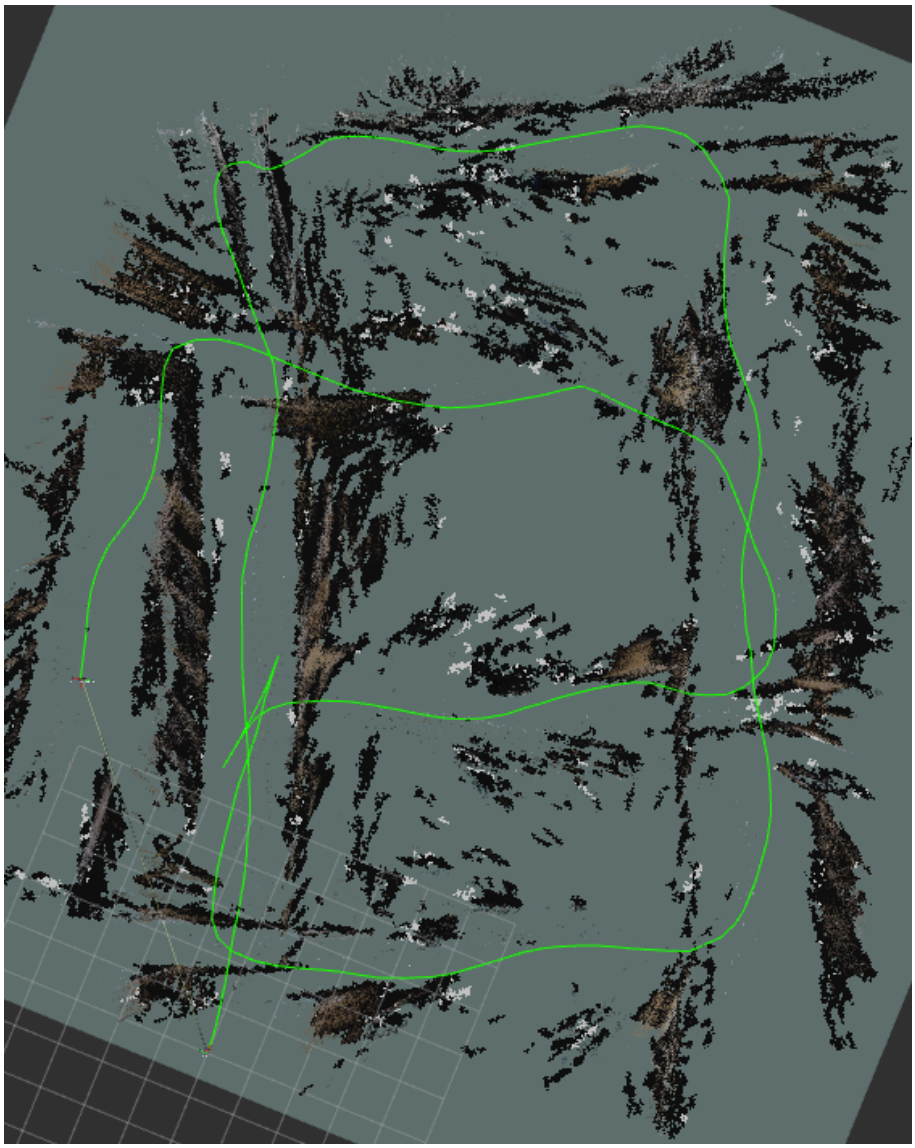


Figure 4.3: Improved Office map using the stereo camera setup only. Image is a screenshot from Rviz.

The odometry is seemingly sturdy but performing the loop closure when coming back to the bottom left corner fails. After this the car was driven between the halls among the tables but as loop closure failed the cumulative odometry drift caused the map to lose coherency. Still, the office layout for the hallways somewhat correspond to that of Figure 3.8.

4.2.3 Stereo camera and LiDAR with stereo camera odometry

This SLAM session was performed using both the LiDAR and the stereo camera setup with odometry from stereo cameras. Mapping was locked to one plane of mapping and not driven to larger open area. The car was driven from the trajectory line at top left of image and run was ended at bottom left side of image. This session used the custom ROS launch file `test.launch` with argument `hector_odom = "true"`.

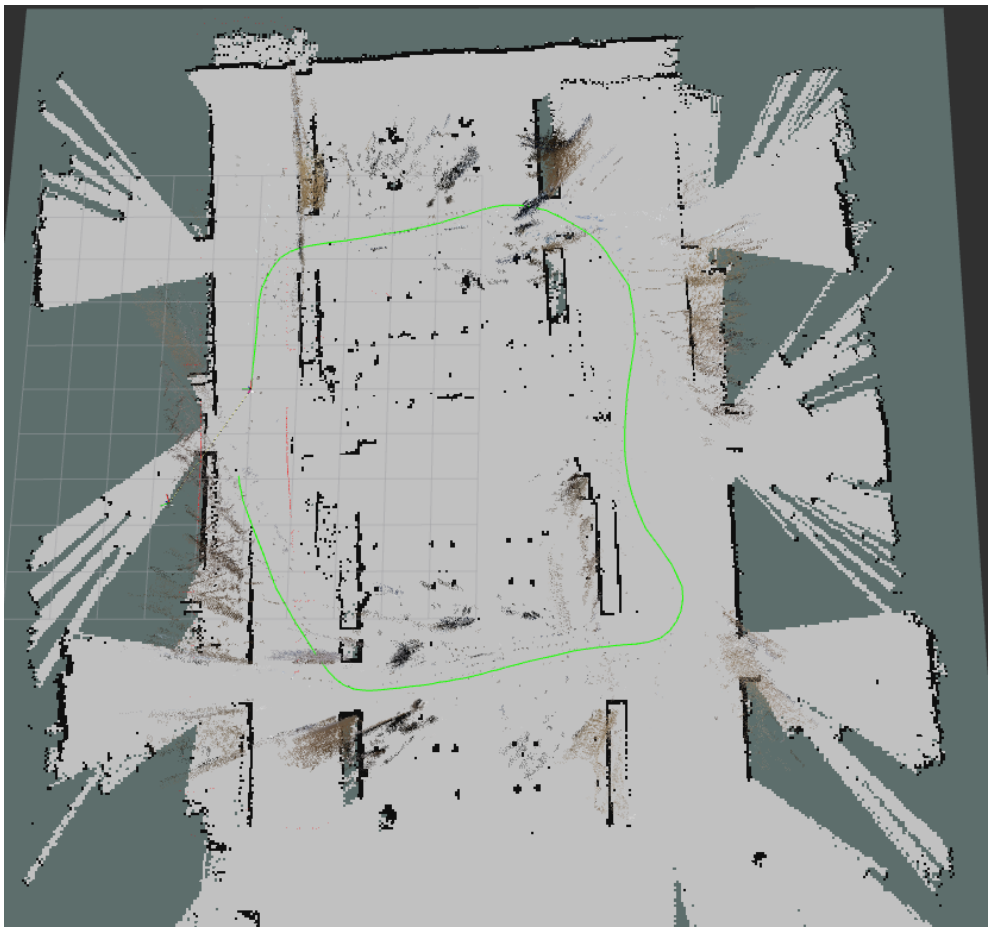


Figure 4.4: Office map with LiDAR and the stereo camera setup using stereo camera odometry. Image is a screenshot from Rviz.

This setup used the obstacle grid marked black in the image in Figure 4.4 from the LiDAR. The role of the stereo cameras was to compute odometry and attempt to perform loop closures. The LiDAR maps efficiently and accurately and a large room map is realised from a far shorter run than that in Figure 4.3. Some issues appear due to limitations of 2D light based sensor, chairs and tables seen in the centre row in Figure 4.4 are only recognized as dots.

In this run the map displays great likeness to the office layout shown in Figure 3.8. However, the odometry drifts as the robot mapped the room and the run was ended as it failed to close the loop to adjust for the odometry drift. The inability to perform loop closure shows the camera setup used to be too poor to perform reliable mapping. The color pixels in the image display area mapped from cameras but the performance is not good enough to detect individual objects and perform loop closures.

4.2.4 Stereo camera and LiDAR with LiDAR odometry

This SLAM session was performed using both the LiDAR and the stereo camera setup with odometry from LiDAR using RTAB-map implementation. Odometry was lost when turning and no full room map was achieved.

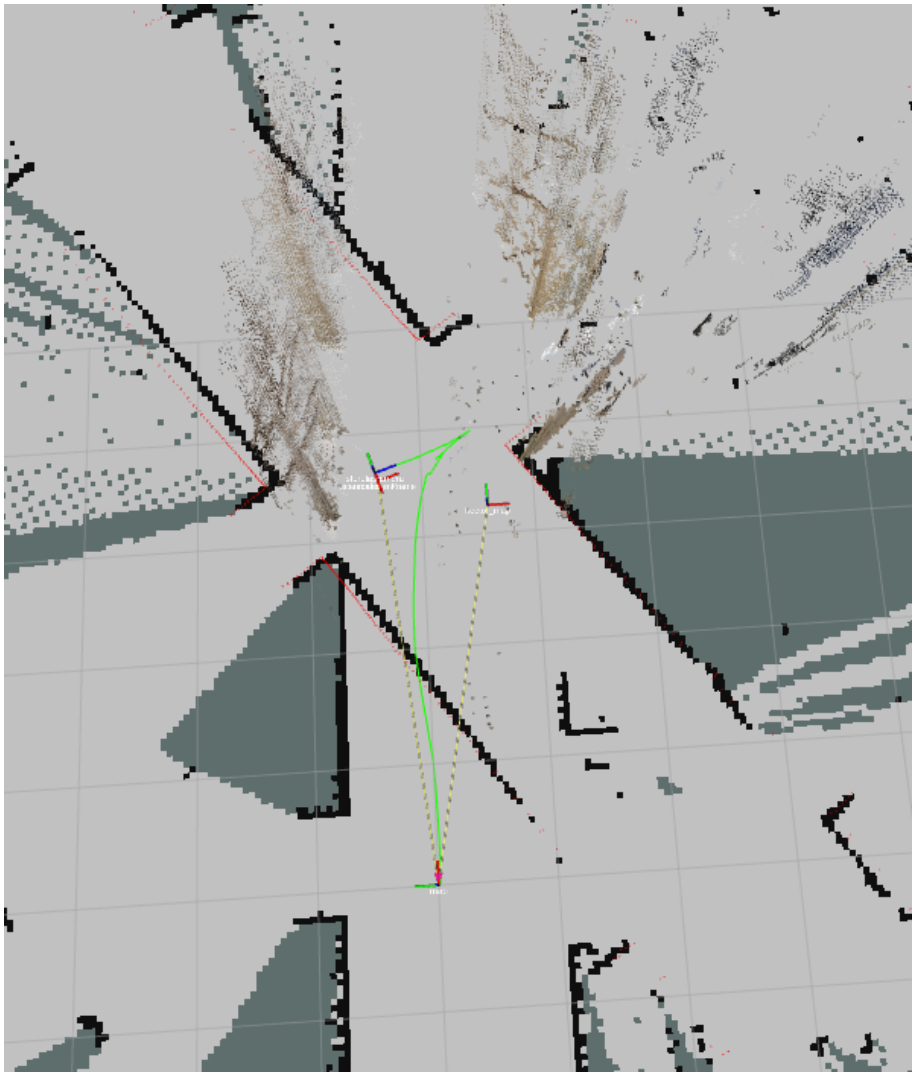


Figure 4.5: Failed office map with LiDAR and the stereo camera setup using LiDAR odometry. Image is a screenshot from Rviz.

The result using LiDAR odometry only was similar to that of Figure 4.5. No full map was achieved as odometry was lost.

The poor odometry, especially when turning, could be due to the car itself combined with this specific LiDAR sensor. This car with sensors mounted was too heavy for the simple motors, and caused them to struggle. This caused the car to struggle turning slowly and often turned to quick as was the case in Figure 4.5. A quick angular rotation, combined with the 5.5Hz rotation speed in the RPlidar A1, could cause odometry issues if the surrounding distance points differ too much one frame to another.

Another theory as to why this was the case would be a sub par odometry implementation

for this project. The ROS odometry message includes position covariance, denoting uncertainty which should increase when data changes quickly, for example when turning quickly. The odometry implementation for this project used static covariance, possibly causing poor odometry when turning.

4.3 Performance

The results displayed in Figures 4.3, 4.4 and 4.5 show that an office type environment could to some capacity be mapped using this setup. The types of sensors also display different strengths and weaknesses.

The movement and positioning information (odometry) derived from the cameras seemed more sturdy than that of the LiDAR. The camera odometry based mapping sessions managed to map the entire room and only lost position when placed in a large empty area at the bottom of the office not displayed in figures. However, the odometry drifted over the mapping session and this camera setup proved unable to perform large loop closures that are otherwise essential to the RTAB-map SLAM method [9]. The inability to perform loop closures could also be due to the difficulty of finding individual elements in an office environment with many repeating features and objects.

Comparing Figure 4.3 to Figure 4.4 suggests that the LiDAR is able to provide a more precise and further reaching data than that of the cameras. It is also able to map in 360 degree field and thus has no dependency of robot orientation, contrary to the cameras which are only able to see in a limited field forwards.

These results suggest a good mapping setup to be a combination of these two sensor types, perhaps with some added improvements. The LiDAR provides a precise occupancy grid but is prone to odometry drift as in Figure 4.5 and cannot relocate if drift is severe. It is non dependant on robot orientation and the LiDAR data is low size and fast to compute. RDB-D and Stereo camera setups has in many cases proved able to provide loop closures achieving great results of SLAM [8, 9]. The data of images is large in size, even on compressed form, and are more costly to compute than the 2D LiDAR data. They are also dependant of orientation and 360 degree mapping will be hard to achieve if robot does not turn around in place often or if a 360 degree panoramic camera is not used as achieved in 2005 [34].

4.4 Challenges

The greatest challenge of this project was configuring the stereo camera setup made of two low cost camera modules and the Raspberry Pi:s. It was shown that it is possible to achieve stereo capabilities from even a setup as inexpensive as the one used but the quality is ultimately sub par and purchasing a readily available stereo camera or RGB-D camera is advised.

With the two cameras forced to be mounted on the RP3 and RP4 respectively the problem of synchronising the image streams became apparent. At first both streamed separately to the laptop and showed asynchronous behaviour due to the difference in transfer speed over WiFi. The RP3 image stream was slower due to its 2.4 GHz WiFi in

comparison to the 5 GHz WiFi from the RP4. To solve this the image from the RP3 was transferred to the RP4 via Ethernet by using static Ethernet IP:s and allowing both Ethernet and WiFi to be used on the master unit RP4.

Simply relaying the image streams by the RP4 did not fix the full synchronisation problem as the image streams over WiFi was still separate, and prone to asynchronous behaviour if WiFi was unstable. Some alternatives using ROS synchronisation libraries was considered, but the working solution was implemented using a custom message as shown in the method.

Another challenge was the transfer and computation of data. A mapping setup will benefit greatly from performing the computation on the mobile robot. This was attempted but the RP4 risked overheating and computed too slowly when performing computation of the images, and was instead used to relay data to the laptop. This caused difficulty as the amount of data able to be transferred was limited by the transfer speed from the RP4 WiFi and limited by the WiFi used to connect units. The initial WiFi used fluctuated greatly and traffic was relayed to a WiFi router separate from other traffic. This setup allowed a stable transfer of two compressed images of 640×480 pixels resolution at 12Hz and was limited to 10Hz for stability. This proved sufficient for camera odometry but higher resolution could have improved results greatly.

The camera modules used, Raspberry Pi Camera Module v2, while good in specifications, proved hard to configure for SLAM. The base settings in the launch file had anti shake enabled and had no limit on shutter speed. This caused data to be blurred when moving or mainly rotating the car. This was changed as shown in section 3.5, to no anti shake and shutter speed at 7000μs. This shutter speed was chosen as it allowed enough time for a good image, but not enough to risk much motion blur. A shutter speed comparison is shown in Figures 4.6 and 4.7.

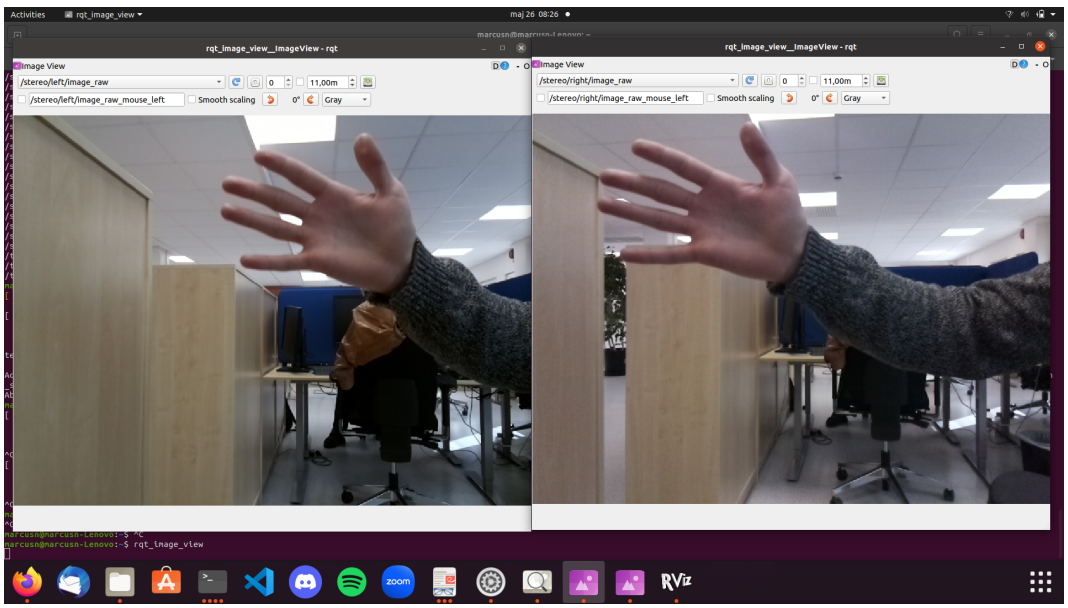


Figure 4.6: Images of moving hand. Left image displaying motion blur with `shutter_speed = 7000`, right image displaying motion blur with `shutter_speed = 3000`.

No motion blur is apparent in any of the two images but image quality and colors is better in case of `shutter_speed = 7000`.

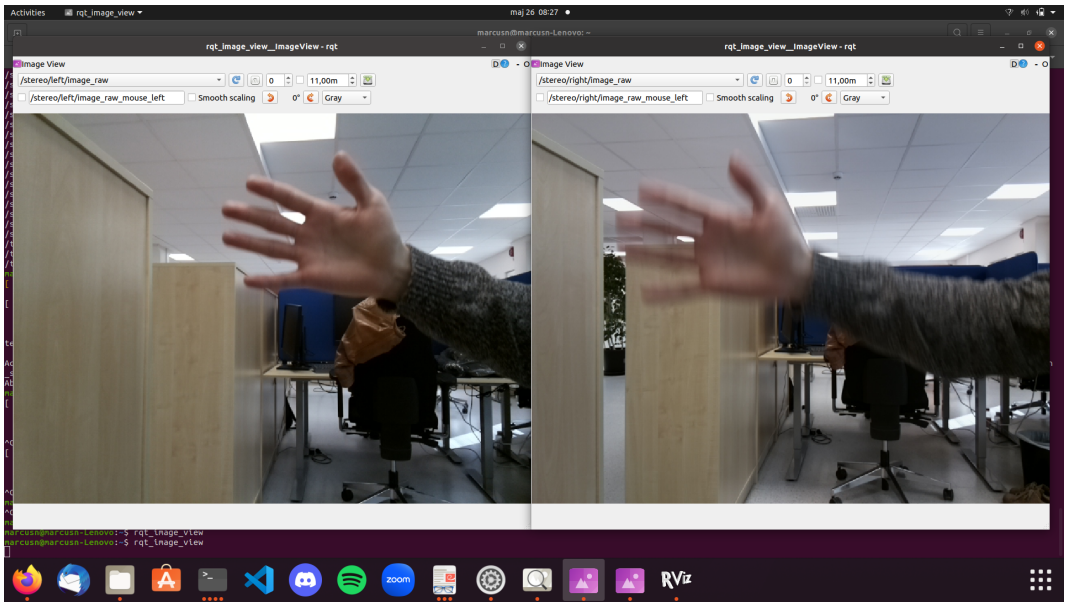


Figure 4.7: Images of moving hand. Left image displaying motion blur with `shutter_speed = 7000`, right image displaying motion blur with `shutter_speed=0` (no limit).

No difference in image quality and colors is apparent in any of the two images but motion blur is large in image with no limit on shutter speed `shutter_speed = 0`. This change greatly improved the stereo camera odometry as images were more clear when turning the car.

4.5 Possible Improvement

To this project there are lots of possible improvements as the results could have been much better. These include improvements on the RC car, improvements on the stereo camera setup and improvements on the ROS and RTAB-map configuration.

4.5.1 Stereo camera setup

Despite actions taken to synchronise the cameras, the cameras are not fully synchronised by hardware. At 10 Hz frame rate the difference in time between images that are counted as synchronised in time can differ up to $\frac{1}{10*2} = 0.05s$ in time. With slow moving cameras and with still background this would not equate to much error in distance and was deemed sufficient for this project but could contribute to error in distance or difficulty with odometry especially when turning the car. With a faster moving camera setup on for example a drone a possible error of this size could prove detrimental to performance. One way to improve upon this which was attempted would be to wire a GPIO pin and ground between the RP4 and RP3 and triggering the RP3 image capture from the RP4 by pulsing a digital signal to that GPIO pin. This would still not be equal to the synchronisation of commercial stereo cameras where capture is performed simultaneously across multiple cameras, but could improve results of this project.

4.5.2 RC-car

The base frame, motors and wheels for the RC car used in this project was the Kjell & Company "Playknowlogy" [26]. This is a very low cost car which leads to it being low performance and is not meant for automatic control and purposes with higher mechanical demand. Adding weight by mounting the RP3, RP4, LiDAR and powerbank made the frame and wheel axis sag and caused the car to be difficult to control. For example it would often turn not at all or far to fast applying the same voltage to the motors. It would also wobble greatly when driving which affected image data greatly and provided a poor basis for image processing in the rest of the project and measurements. The LiDAR performance is less affected by this as the distance in the planar field does not differ much when wobbling but the camera performance is greatly affected since pixel position in 3D differs more with this instability. This issue could have been improved and provided a better basis for the project by mounting the sensors on an RC car capable of the added weight and more easily controllable.

4.5.3 RTAB-map configuration

ROS and RTAB-map contains a multitude of configuration options and parameters that could be changed and used. In this project the conditions for the stereo camera setup SLAM was too poor for parameter tuning to be efficient. The parameters used was set in the launch file and largely based on default parameters from the RTAB-map ROS page [33]. The following two parameters made the greatest improvement on SLAM for this project.

1. `reg3dof = true`
2. `Kp/RoiRatios = 0.03 0.03 0.03 0.3`

The first parameter locked the robot movement to one plane and not allowing movement in height. This is suitable if robot only moved on one floor, as advised by Labbé [35]. The second parameter allowed RTAB-map to perform computation on only the upper 70% of the images. This prevented RTAB-map from making loop closures and drawing conclusion based on the repeating floor pattern of the office. This was used to great efficiency by the winners of the IROS 2014 Kinect Challenge [9].

Effects of other parameters that were not explored in this project include changing the visual SLAM approach and changing demands on mapping resolution.

4.5.4 ROS RTAB-map setup

Performing SLAM computation for images, or image computation in general, is costly. At the same time it is advised to to perform the calculations on the mobile robot and only use other units for data visualisation. Such a setup allows for better versatility and essentially no demands on WiFi transfer rate and WiFi stability. It would require greater computation power on the robot than the RP4 can provide. A ROS RTAB-map setup for computation on the robot using remote visualisation is displayed in Figure 4.8 and is more similar to setups used in other SLAM projects [8].

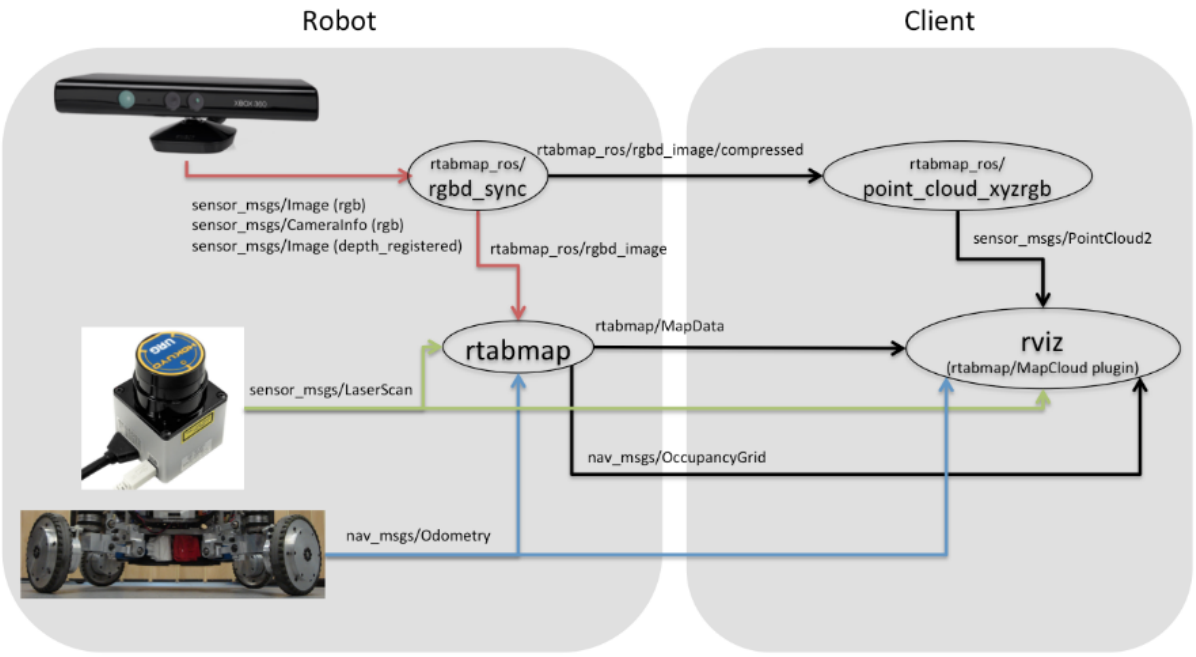


Figure 4.8: Remote visualisation with RTAB-map [33].

In this project the images were transferred from the RP4 to the laptop in a message containing compressed left image, compressed right image, left camera info and right camera info. A better setup could be to transform the stereo images to a compressed Red Green Blue Depth (RGB-D) image before sending to the laptop for computation. This could be done using the `stereo_sync` node from the *rtabmap_sync* package [36]. An RGB-D message is smaller than two RGB messages and would allow for higher frame rate or resolution allowing for better visual SLAM.

5 Conclusions and further work

5.1 Conclusion

This thesis has shown it is possible to perform SLAM given a 2D LiDAR and using two monocular cameras as a stereo camera both individually and in tandem.

The 2D LiDAR provided good mapping data regardless of robot orientation but was prone to odometry issues. The camera setup used provided sturdy odometry but did not give sufficient mapping data for loop closures in the difficult office environment. A combination of the two sensor types LiDAR and cameras can complement each other and has potential to provide good maps using SLAM.

5.2 Further work

For this thesis lots of further work is possible. Firstly the RC car used gave poor conditions for the camera to be sure of pixel positioning, and at points turned to quickly for the LiDAR odometry calculations. A better car frame, perhaps based on a more sturdy RC car could benefit the results greatly.

Secondly the camera modules used are sub optimal for SLAM implementation and after much work and configuration still proved insufficient to perform loop closure detection in SLAM. Using instead an RGB-D camera or commercial stereo camera could get better data and provide improved results.

Thirdly the RTAB-map is a complicated but seemingly good SLAM approach providing a great amount of different options for SLAM. The options explored in this project are but scratching the surface and improved parameter values and options could make a big impact.

Finally being forced to perform computation remotely on the laptop is sub optimal. Using a more powerful system for computation on the robot and only visualising remotely could allow the robot to operate regardless of WiFi speed, improving stability.

Appendix

Code used for this work can be found in GitHub repository https://github.com/Nises/SLAM_Exjobb/tree/master/catkin_ws.
Note that the two GitHub repositories "hector_slam" and "rtabmap_ros" are included but not accessible. These repositories has to be accessed separately.

Script `varied_control.cpp`

The code for controlling movement of the car

```
#include <ros.h>

#include <geometry_msgs/Twist.h>
#include <Arduino.h>

ros::NodeHandle nh;

int pinRB=6; // define pin6 as left back connect with IN1
int pinRF=9; // define pin9 as left forward connect with IN2
int pinLB=10; // define pin10 as right back connect with IN3
int pinLF=11; // define pin11 as right back connect with IN4
int x, z;
int speedR=0;
int speedL=0;
int increment=10;

void accelerate(int m);
void turn(int n);
void write();


void drive(const geometry_msgs::Twist& dir){
    x = dir.linear.x;
    z = dir.angular.z;
    nh.logdebug("Debugging");
    if(x){
        //accelerating
        Serial.print("␣Accelerating␣");
        accelerate(x);
    }
    if(z){
        //turning
        Serial.print("␣Turning␣");
        turn(z);
    }
}

ros::Subscriber<geometry_msgs::Twist> sub("turtle1/cmd_vel", &drive);

void setup()
{
Serial.begin(9600);
pinMode(pinLB,OUTPUT);
pinMode(pinLF,OUTPUT);
pinMode(pinRB,OUTPUT);
pinMode(pinRF,OUTPUT);
```

```
analogWrite(pinRB,0);
analogWrite(pinRF,0);
analogWrite(pinLB,0);
analogWrite(pinLF,0);

nh.initNode();
nh.subscribe(sub);
}
void accelerate(int x) // accelerate
{
speedR += x*increment;
speedL += x*increment;
if(speedR>250 || speedR<-250){
  speedR -= x*increment;
}
if(speedL>250 || speedL<-250){
  speedL -= x*increment;
}

write();
}

void turn(int z) //turn
{
speedR += z*increment;
speedL += -z*increment;
if(speedR>250 || speedR<-250){
  speedR -= z*increment;
}
if(speedL>250 || speedL<-250){
  speedL -= -z*increment;
}
write();
}

void write()
{
analogWrite(pinRB, (speedR>0 ? 0 : -speedR));
analogWrite(pinRF, (speedR>0 ? speedR : 0));
analogWrite(pinLB, (speedL>0 ? 0 : -speedL));
analogWrite(pinLF, (speedL>0 ? speedL : 0));
delay(50);
}

void loop()
{
nh.spinOnce();
delay(1);
}
```

ROS launchfile `test.launch`

The main launch file used on laptop.

```
<?xml version="1.0"?>
<!-- -->
<launch>
```

```
<include file="$(find stereo_formatting)/launch/laptop.launch"/>

<node name="scan_sync" pkg="stereo_formatting" type="scan_sync" />

<arg name="pi/2" value="1.5707963267948966" />
<arg name="optical_rotate" value="0 0 0.19 -$(arg pi/2) 0 -$(arg pi/2)" />
<node pkg="tf" type="static_transform_publisher" name="camera_base_link"
      args="$(arg optical_rotate) base_link stereo_camera 100" />

<arg name="hector" default="true" />
<arg name="hector_odom" default="false" />

<arg name="rviz" default="true" />

<node if="$(arg hector_odom)" pkg="tf" type="static_transform_publisher"
    name="scanmatcher_to_base_frame"
     args="0.0 0.0 0.0 0.0 0.0 0.0 /scanmatcher_frame /base_link 100" />
<node if="$(arg hector)" pkg="tf" type="static_transform_publisher" name="
    base_to_laser_broadcaster"
     args="0.0 0.0 0.19 0.0 0.0 0.0  base_link laser 100" />

<arg name="base_frame" default="base_link" />
<arg name="odom_frame" default="odom"/>

<!-- Run the ROS package stereo_image_proc -->
<group ns="/stereo" >
    <node pkg="stereo_image_proc" type="stereo_image_proc" name="
        stereo_image_proc"/>

    <!-- Odometry -->
    <node if="$(arg hector_odom)" pkg="hector_mapping" type="
        hector_mapping" name="hector_mapping" output="screen">

        <!-- Frame names -->
        <param name="map_frame" value="map" />
        <param name="base_frame" value="$(arg base_frame)" />
        <param name="odom_frame" value="$(arg odom_frame)" />

        <!-- Tf use -->
        <param name="pub_map_odom_transform" value="false"/>
        <param name="pub_map_scanmatch_transform" value="true"/>
        <param name="pub_odometry" value="$(arg hector_odom)"/>

        <!-- Map size / start point -->
        <param name="map_resolution" value="0.050"/>
        <param name="map_size" value="2048"/>
        <param name="map_multi_res_levels" value="2" />

        <!-- Map size / start point -->
        <param name="map_resolution" value="0.050"/>
        <param name="map_start_x" value="0.5"/>
        <param name="map_start_y" value="0.5" />

        <!-- Map update parameters -->
        <param name="update_factor_free" value="0.8"/>
        <param name="update_factor_occupied" value="0.8" />
        <param name="map_update_distance_thresh" value="0.4"/>
```

```
        <param name="map_update_angle_thresh" value="0.06" />
        <param name="laser_z_min_value" value = "-1.0" />
        <param name="laser_z_max_value" value = "1.0" />

        <!-- Advertising config -->
        <param name="scan_topic" value="/stereo/scan"/>
    </node>

    <node unless="$(arg hector_odom)" pkg="rtabmap_odom" type="
        stereo_odometry" name="stereo_odometry" output="screen">
        <remap from="left/image_rect"        to="left/image_rect"/>
        <remap from="right/image_rect"       to="right/image_rect"/>
        <remap from="left/camera_info"       to="left/camera_info"/>
        <remap from="right/camera_info"      to="right/camera_info"/>

        <param name="frame_id" type="string" value="base_link"/>
        <param name="odom_frame_id" type="string" value="odom"/>
        <param name="queue_size"     type="int" value="5"/> -->

        <param name="approx_sync"    type="bool" value="false"/>
        <param name="queue_size"     type="int" value="5"/>
        <param name="Vis/MinInliers" type="string" value="10"/>
        <param name="Vis/RoiRatios" type="string" value="0.03 0.03 0.04
            0.04"/>
        <param name="Vis/CorNNDR" type="string" value="0.8"/>
        <param name="Vis/MaxFeatures" type="string" value="1000"/>
        <param name="GFTT/MinDistance" type="string" value="10"/>
        <param name="GFTT/QualityLevel" type="string" value="0.00001"/>
            -->

        <param name="Reg/Force3DoF" type="bool" value="true"/>
    </node>

    <node pkg="rtabmap_sync" type="stereo_sync" name="stereo_sync" output=
        "screen">
        <remap from="left/image_rect"   to="/stereo/left/image_rect_color"
            />
        <remap from="right/image_rect"  to="/stereo/right/image_rect_color
            "/>
        <remap from="left/camera_info"  to="/stereo/left/camera_info"/>
        <remap from="right/camera_info" to="/stereo/right/camera_info"/>
    </node>
</group>

<group ns="rtabmap">
  <node name="rtabmap" pkg="rtabmap_slam" type="rtabmap" output="screen"
      args="--delete_db_on_start">
      <param name="frame_id" type="string" value="base_link"/>
      <param name="odom_frame_id" type="string" value="odom"/>
      <param name="subscribe_scan"  type="bool" value="true"/>
      <param name="subscribe_stereo" type="bool" value="false"/>
      <param name="subscribe_depth"  type="bool" value="false"/>
      <param name="subscribe_rgbd"  type="bool" value="true"/>
      <param name="subscribe_rgb"  type="bool" value="false"/>
      <param name="approx_sync"       type="bool" value="true"/>
      <param if="$(arg hector_odom)" name="subscribe_odom_info" type="bool"
          value="false"/>
```

```
      <remap from="right/image_rect"  to="/stereo/right/image_rect"/>
      <remap from="left/camera_info"  to="/stereo/left/camera_info"/>
      <remap from="right/camera_info" to="/stereo/right/camera_info"/> -->
      <remap from="rgbd_image"        to="/stereo/rgbd_image"/>

      <remap from="odom" to="/stereo/odom"/>

      <param name="queue_size" type="int" value="30"/>
      <remap from="scan" to="/stereo/scan"/>

       <!-- As hector doesn't provide compatible covariance in the odometry
           topic, don't use the topic and fix the covariance -->
      <param if="$(arg hector_odom)" name="odom_frame_id"             type="
          string" value="hector_map"/>
      <param if="$(arg hector_odom)" name="odom_tf_linear_variance"  type="
          double" value="0.0005"/>
      <param if="$(arg hector_odom)" name="odom_tf_angular_variance" type="
          double" value="0.0005"/>

      <!-- RTAB-Map's parameters -->
     <param name="Vis/MinInliers" type="string" value="12"/>

     <param name="Reg/Force3DoF" type="bool" value="true"/>
     <param name="Kp/RoiRatios" type="string" value="0.05 0.05 0.05 0.3"/>

     <param name="Rtabmap/TimeThr" type="string" value="700"/>

     <param name="Rtabmap/DetectionRate" type="string" value="1"/>

     <param name="Kp/DetectorStrategy" type="string" value="0"/>
     <param name="Kp/NNStrategy" type="string" value="1"/>

     <param name="SURF/HessianThreshold" type="string" value="1000"/>

     <param name="Vis/MinInliers" type="string" value="10"/>

     <param name="RGBD/LoopClosureReextractFeatures" type="string" value="
         true"/>
     <param name="RGBD/NeighborLinkRefining" type="string" value="true"/>
     <param name="RGBD/ProximityBySpace" type="string" value="true"/>
     <param name="RGBD/AngularUpdate" type="string" value="0.01"/>
     <param name="RGBD/LinearUpdate" type="string" value="0.01"/>
     <param name="RGBD/OptimizeFromGraphEnd" type="string" value="false"/>
     <param name="Grid/FromDepth" type="string" value="false"/>


     <param name="Vis/MaxWords" type="string" value="500"/>
     <param name="Vis/MaxDepth" type="string" value="5"/>

     <param if="$(arg hector)" name="Grid/Sensor" type="int" value="0"/>
         <!-- occupancy grid from lidar -->
     <param name="Reg/Strategy" type="string" value="1"/>  <!-- ICP -->

     <!-- ICP parameters -->
     <param name="Icp/VoxelSize" type="string" value="0.05"/>
     <param name="Icp/MaxCorrespondenceDistance" type="string" value
         ="0.1"/>
```

```
      <param name="cloud_noise_filtering_radius" value="0.05"/>
      <param name="cloud_noise_filtering_min_neighbors" value="3"/>

      <param name="proj_max_ground_angle" value="50"/>
    </node>

    <!-- Visualisation RTAB-Map, master on remote machine so rtabmap takes
        ~50s to start. -->
    <node unless="$(arg rviz)" pkg="rtabmap_ros" type="rtabmapviz" name="
        rtabmapviz" args="-d $(find stereo_formatting)/launch/config/rgbd_gui.
        ini" output="screen" launch-prefix="bash -c 'sleep 30; $0 $@' ">
      <param name="subscribe_stereo"    type="bool"   value="true"/>
      <param unless="$(arg hector)" name="subscribe_odom_info" type="bool"
        value="true"/>
      <param if="$(arg hector)" name="subscribe_laserScan" type="bool"
        value="true"/>
      <param name="queue_size"          type="int"    value="10"/>
      <param name="frame_id"            type="string" value="base_link"/>
      <remap from="left/image_rect"     to="/stereo/left/image_rect_color"/>
      <remap from="right/image_rect"    to="/stereo/right/image_rect"/>
      <remap from="left/camera_info"    to="/stereo/left/camera_info"/>
      <remap from="right/camera_info"   to="/stereo/right/camera_info"/>
      <remap from="odom_info"           to="/stereo/odom_info"/>
      <remap from="odom"                to="/stereo/odom"/>
      <remap if="$(arg hector_odom)" from="scan" to="/stereo/scan"/>
      <remap unless="$(arg hector_odom)" from="scan" to="/stereo/scan"/>

      <param name="imageView_odometry\features_shown" type="bool" value="
        false"/>
      <param if="$(arg hector_odom)" name="odom_frame_id" type="string"
        value="hector_map"/>
    </node>

    <!-- Visualisation RVIZ. -->
    <node if="$(arg rviz)" pkg="rviz" type="rviz" name="rviz" args="-d $(
        find stereo_formatting)/config/setuprviz.rviz" output="screen" launch-
        prefix="bash -c 'sleep 20; $0 $@'"/>

  </group>
</launch>
```

References

[1] X. Wang, H. Pan, K. Guo, X. Yang, and S. Luo, "The evolution of lidar and its application in high precision measurement", *IOP Conference Series: Earth and Environmental Science*, vol. 502, no. 1, p. 012 008, May 2020. doi: 10.1088/1755-1315/502/1/012008.

[2] J. Velasco, "Lidar is the gold standard of robot vacuum navigation, but it's not perfect", July, 2021. [Online]. Available: https://www.digitaltrends.com/home/lidar-gold-standard-robot-vacuum-navigation/. [Accessed May. 23, 2023].

[3] A. Dubey, "Stereo vision-Facing the challenges and seeing the opportunities for ADAS applications", *Texas Instruments*. June, 2020 [Online]. Available: https://www.ti.com/lit/wp/spry300a/spry300a.pdf?ts=1673869574680&ref_url=https%253A%252F%252Fwww.google.com%252F. [Accessed Jan. 28, 2023].

[4] University of Auckland, "COMPSCI773S1T: Vision Guided Control", 2019. [Online]. Available: https://www.cs.auckland.ac.nz/courses/compsci773s1t/lectures/773-GG/topCS773.htm. [Accessed Feb. 13, 2023].

[5] H. Durrant-Whyte and T. Bailey, "Simultaneous localization and mapping: part I the essential algorithms", in *IEEE Robotics Automation Magazine*, vol. 13, no. 2, pp. 99-110, June 2006, doi: 10.1109/MRA.2006.1638022.

[6] J. Wenshuai, "Application of autonomous navigation in robotics", *Journal of Physics: Conference Series*, vol. 1906, p. 012 018, May, 2021. doi: 10.1088/1742-6596/1906/1/012018.

[7] K. Schmid *et al.*, "Lidar 101: An Introduction to Lidar Technology, Data, and Applications", *National Oceanic and Atmospheric Administration (NOAA) Coastal Services Center*, November, 2012. [Online]. Available: https://coast.noaa.gov/data/digitalcoast/pdf/lidar-101.pdf. [Accessed Feb. 02, 2023].

[8] M. Labbé and F. Michaud, "RTAB-Map as an open-source lidar and visual simultaneous localization and mapping library for large-scale and long-term online operation", *Journal of Field Robotics* vol. 36, pp. 416–446, 2 Mar. 2019. issn: 15564967. doi: 10.1002/rob.21831.

[9] M. Labbé, "IROS 2014 Kinect Challenge", June, 2020. [Online]. Available: https://github.com/introlab/rtabmap/wiki/IROS-2014-Kinect-Challenge. [Accessed Apr. 23, 2023].

[10] Open Robotics, "The ROS Ecosystem", 2021. [Online]. Available: https://www.ros.org/blog/ecosystem/. [Accessed May. 20, 2023].

[11] Open Robotics, "Understanding Nodes", October, 2022. [Online]. Available: http://wiki.ros.org/ROS/Tutorials/UnderstandingNodes [Accessed Jan. 26, 2023].

[12] Open Robotics, "Understanding ROS Topics", October, 2022. [Online]. Available: http://wiki.ros.org/ROS/Tutorials/UnderstandingTopics [Accessed Jan. 26, 2023].

[13] Open Robotics, "Master", January, 2018. [Online]. Available: http://wiki.ros.org/Master [Accessed Jan. 26, 2023].

[14] Open Robotics, "roslaunch", October, 2019. [Online]. Available: http://wiki.ros.org/roslaunch. [Accessed May. 27, 2023].

[15] Y. Li, J. Ibanez-Guzman, "Lidar for Autonomous Driving: The Principles, Challenges, and Trends for Automotive Lidar and Perception Systems", in *IEEE Signal Processing Magazine*, vol. 37, no. 4, pp. 50–61, 2020. doi: 10.1109/MSP.2020.2973615.

[16] H. Tibebu, V. De-Silva, C. Artaud, R. Pina, and X. Shi, "Towards Interpretable Camera and LiDAR Data Fusion for Autonomous Ground Vehicles Localisation", *Sensors*, vol. 22, no. 20, p. 8021, Oct. 2022, doi: 10.3390/s22208021.

[17] Open Robotics, "nav_msgs/Odometry Message", Mars, 2022. [Online]. Available: http://docs.ros.org/en/noetic/api/nav_msgs/html/msg/Odometry.html. [Accessed Feb. 17, 2023].

[18] M. Labbé and F. Michaud, "Appearance-Based Loop Closure Detection for Online Large-Scale and Long-Term Operation", *IEEE Transactions on Robotics*, vol. 29, no. 3, pp. 734–745, 2013. doi: 10.1109/TRO.2013.2242375.

[19] OpenCV, "About", [Online]. Available: https://opencv.org/about/. [Accessed Feb. 15, 2023].

[20] M. Labbé, "Remote Mapping", September, 2019. [Online]. Available: http://wiki.ros.org/rtabmap_ros/Tutorials/RemoteMapping. [Accessed Apr. 19, 2019].

[21] Raspberry Pi, "Raspberry Pi 4 Model B specifications", [Online]. Available: https://www.raspberrypi.com/products/raspberry-pi-4-model-b/specifications/. [Accessed Jan. 28, 2023].

[22] Raspberry Pi, "Raspberry Pi 3 Model B", [Online]. Available: https://www.raspberrypi.com/products/raspberry-pi-3-model-b/. [Accessed Jan. 28, 2023].

[23] Arduino, "Arduino Uno Rev3", [Online]. Available: https://store.arduino.cc/products/arduino-uno-rev3. [Accessed Jan. 26, 2023].

[24] Slamtec, "DATASHEET RPLIDAR A1", June, 2016. [Online]. Available: https://www.generationrobots.com/media/rplidar-a1m8-360-degree-laser-scanner-development-kit-datasheet-1.pdf. [Accessed Feb. 23, 2023].

[25] Raspberry Pi, "Raspberry Pi Camera Module 2", [Online]. Available: https://www.raspberrypi.com/products/camera-module-v2/. [Accessed Feb. 17, 2023].

[26] Kjell Company, "Playknowlogy Arduino Robotbil Startpaket", [Online]. Available: https://www.kjell.com/se/produkter/hem-fritid/lek-lar/programmerbara-robotar/playknowlogy-arduino-robotbil-startpaket-p87288. [Accessed Jan. 26, 2023].

[27] Ubuntu, "How to install Ubuntu Server on your Raspberry Pi", [Online]. Available: https://ubuntu.com/tutorials/how-to-install-ubuntu-on-your-raspberry-pi#1-overview. [Accessed Jan. 24, 2023].

[28] Open Robotics, "Ubuntu install of ROS Noetic", [Online]. Available: http://wiki.ros.org/noetic/Installation/Ubuntu. [Accessed Jan. 26, 2023].

[29] Open Robotics, "Arduino IDE Setup", November, 2022. [Online]. Available: http://wiki.ros.org/rosserial_arduino/Tutorials/Arduino%20IDE%20Setup. [Accessed Feb. 05, 2023].

[30] anfederman, "Raspicam_node on ARM64 Raspberry pi", December, 2021. [Online]. Available: https://discourse.ros.org/t/raspicam-node-on-arm64-raspberry-pi/23311. [Accessed Mar. 17, 2023].

[31] Ubiquity Robotics, "raspicam_node", June, 2020. [Online]. Available: https://github.com/UbiquityRobotics/raspicam_node. [Accessed Feb. 23, 2023].

[32] M. Labbé *et al.*, "rtabmap_ros", *Introlab*, April, 2023. [Online]. Available: https://github.com/introlab/rtabmap_ros. [Accessed Feb. 23, 2023].

[33] M. Labbé, "Setup RTAB-Map on Your Robot!", April, 2023. [Online]. Available: http://wiki.ros.org/rtabmap_ros/Tutorials/SetupOnYourRobot. [Accessed Apr. 19, 2023].

[34] H. Andreasson, A. Treptow and T. Duckett, “Localization for Mobile Robots using Panoramic Vision, Local Features and Particle Filter”, 2005. 3348 - 3353. 10.1109/ROBOT.2005.1570627.

[35] M. Labbé, “Advanced Parameter Tuning”, April, 2023. [Online]. Available: http://docs.ros.org/en/noetic/api/nav_msgs/html/msg/Odometry.html. [Accessed Apr. 23, 2023].

[36] M. Labbé, “rtabmap_sync”, April, 2023. [Online]. Available: http://wiki.ros.org/rtabmap_sync. [Accessed May. 07, 2023].