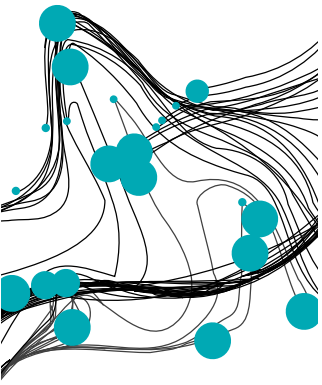
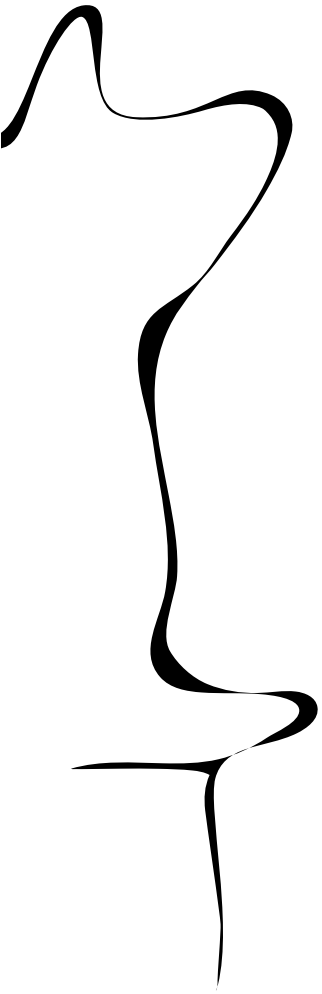


RAM ROBOTICS AND MECHATRONICS



REVERSE ENGINEER DISCONTINUED KUKA I-DO ROBOTS AND MAKE THEM APPLICABLE FOR EDUCATIVE USE

C. (Cliff) ten Berge

BSC ASSIGNMENT

Committee:
dr. ir. E. Dertien
dr. ir. J.F. Broenink
D.P. Davison, Ph.D

April, 2022

013RaM2022
Robotics and Mechatronics
EEMathCS
University of Twente
P.O. Box 217
7500 AE Enschede
The Netherlands

UNIVERSITY OF TWENTE. | TECHMED CENTRE UNIVERSITY OF TWENTE. | DIGITAL SOCIETY INSTITUTE

Abstract

Recently the University of Twente has been donated 5 I-do social robot systems from KUKA (a German company). Due to the absence of documentation, these robots were not directly usable by the university. The first aim of this research is to reverse engineer the robotic platform and to create new documentation on it. In this report, all the relevant functionalities of the components on the I-do platform are described. The second part of this research is to use the new insights on the robot to adapt it to use at the university. Upon discovery that it would be very difficult to reuse the original software, the decision was made to partially replace the custom hardware, allowing the circumvention of the original software. The design choices that came along with this new set of hardware are all explained in the report, along with the restructuring of the power management. The new hardware made it possible to develop a new ROS-based software architecture from the ground up. This research goes into detail on which ROS packages were chosen and how they work together. The end product of this research is a working robot, that can be easily modified by students for use in projects. Because of this, a step-by-step approach to operating the basic functionalities of the robot is supplied in the appendix.

Contents

1 Introduction

Recently the University of Twente has been donated 5 I-do robot systems by KUKA in Germany. These prototypes were part of a now-canceled social robot project at this company. With the canceling of this project, the prototypes served no clear use anymore at KUKA, which sparked the idea to donate them to the University of Twente where they could possibly be put to better use. The robots are interesting for the University of Twente, as the prototypes come with numerous interesting hardware components, and therefore have a lot of potential.

As the robots' purpose at KUKA is not necessarily the same as the purpose they would serve at the university, the robots have to be adapted to fit this new role. The main problem with this project is that documentation from KUKA appears to be completely missing, making it very challenging to reuse the robots. Therefore the first part of this research focuses on reverse engineering the entire I-do platform in the state that it was donated to the university, this report serves as the new documentation of the robot. It was important to also evaluate the state of every individual component (like the batteries), as the robot had been sitting for a while at the start of this research.

The second stage of this research is about the adaptation of this robot platform into the educational environment. As the entire research is oriented on not letting a possibly good robot go to waste, it is highly preferred to reuse as many components of the original I-do platform as possible. For clarity purposes, the expected role for the robot within this educational environment is not about having a role as a robotic tutor, but more as an easily modifiable robotic platform for student projects.

Although this research is specific to the I-do platform, it is also certainly possible to use the same hardware and software structure in other (differential drive) robots.

2 Analysis

2.1 Background

The I-do robot platform has been developed in Germany by KUKA. Their commercial products consist mainly of robotic arms for factories, meaning that with the development of the I-do platform the company has been researching a new segment outside of its normal product range. KUKA showcased a prototype at the Hannover Messe of 2018, which is one of the biggest industrial technology fairs in the world. The prototypes differed in the pieces of equipment mounted onto them, showcasing that the I-do platform is very modular. The equipment options consist of a sound system from Bang & Olufsen, an air conditioner, a photo camera, and even a coffee machine. The I-dos with the mounted photo camera were driving around taking pictures of people that posed in front of them. One of the attendees of the event was the then German chancellor Angela Merkel, who was photographed by one of the I-do's. After the Hannover Messe, KUKA decided to discontinue the entire I-do project. As the prototypes that were produced had no clear use anymore, they moved the I-do's into storage, after which the University of Twente (UT) convinced KUKA to donate the prototypes to them with the idea to use them for education and experiments.

2.1.1 KUKA

As mentioned above, the company that developed the I-dos is a German company called KUKA. KUKA is one of the world's leading robotic systems suppliers and was founded in 1898 in Augsburg. The company offers modular systems capable of (semi) automating entire production lines in a wide variety of sectors. Their most sold product line is their distinctive orange (company color) robotic arms segment. KUKA has a large selection of these robotic arms, each with its own capabilities like maximum lift capacity. These arms are heavily used by the German automotive industry. Besides this, the company also offers different robotic systems ranging from automatic foundry systems to (rotational) welding systems (3).

2.1.2 Savioke

When we take a brief look at the I-do platform, we can see that another company had a big part in the project and that KUKA did not build the I-do's from the ground up. KUKA used a mobile robot base from a company called Savioke, which is a robotics company based in San Jose, California in the United States of America. Savioke has one main product for sale, and that is the Relay robot. The Savioke Relay is an autonomous delivery robot designed to operate in hotels and hospitals, it tries to distinguish itself from possible competitors by being relatively easy to install and a cost-effective option to make deliveries autonomous. According to the Savioke website, one of the main reasons for this is that the Relay is able to mechanically push buttons of elevators, this allows the robot to operate in buildings with multiple floors without the need for expensive adjustments (2). The I-do platform shared the differential drive moving base with the relay robot and also uses the same software as was made for the relay.

The company also has a connection to the well-known Robotic Operating Software (ROS), that is because the CEO of Savioke, Steve Cousins, used to be the CEO of Willow Garage (7). Willow Garage was a US-based company and had a large role in developing ROS as we know it today.

2.2 Hardware

Like most autonomous robots, the I-dos have a lot of different components working together to give the robot the desired functionalities. The I-dos have numerous high-tech sensors which

made the platform very desirable to have for the UT. In the sections below, I will cover all the different components that made up the I-do platform. In these sections, I will often refer to the segment division that I made myself while taking the robot apart. This segment division is purely for clarity purposes. The three segments that I divided the robot into are the tower, the entertainment system, and the Savioke base. The I-dos had different entertainment systems, therefore this analysis will be based on the robot that I took apart, recognizable by the sticker with "Eugeen" written on it, this is the name the robot was given by the KUKA engineers.

Figure 2.1: Section division

2.2.1 Batteries

The robot Eugeen has four batteries in total, which are all Li-Ion-based batteries. The main question at the start of the project was whether the batteries had survived the long storage time at KUKA. It was completely unknown whether the batteries had been looked after, therefore it was decided to take a closer look at each battery individually, before connecting the entire system up to the chargers.

One battery is located halfway up the tower segment and is a 12 Volts 192 Wh battery made by Super-B (product name: 12V-15Ah). The battery has an integrated battery management system (BMS) in it, this protects the battery from very high (dis)charge currents and also provides cell balancing(16). The battery voltage after the long storage period was 13.31 Volts, which was a good indicator that the battery had survived the inactivity. To confirm that it was indeed in pristine condition, the battery was connected to a load, and no strange behavior was observed. This battery is charged using the round charging port located in the tower. This battery has a small breakout board connected to it, this board has a USB connection with the main computer, giving the computer the ability to disconnect the battery from the system.

The entertainment system differs per robot, Eugeen has a battery pack in the entertainment system consisting of two (quite similar to the above-mentioned 12 Volts Super B Li-Ion batteries) Andrena 12V25aH-SC connected in parallel. These batteries have a capacity of 320Wh each and also come with an integrated BMS (17). The exact same tests were performed for this battery pack, the starting voltage was measured at 13.29 Volts and also this battery pack appeared to function properly under a load. This battery pack is charged by using the loose connector that was hanging close to the battery pack.

Finally, there is also a battery all the way in the bottom part of the Savioke base. Unlike the other batteries, this was a 24 Volts battery made by U-charge. The battery belongs to the RT series of the manufacturer, the exact product name is U1-24RT. The measured voltage of the battery upon removing it from the robot was approximately 0.0 Volts, meaning that this battery was drained far below its minimum voltage level. The U1-24RT has an integrated battery management system (BMS)(18), these circuits are normally used to prevent wrongful use and thus should protect the battery and the load. Although it is impossible to know for sure, the power usage of the BMS (although very small) may have been the reason why the battery slowly self-discharged over time (more than the Super-B batteries), and thus was destroyed. In order to make sure that the voltage over the Li-ion cells themselves did not differ from the voltage over the output terminals of the BMS, the cover of the battery was removed to measure the voltage on the cells directly. Unfortunately, this did not result in a different reading. The battery used to be charged through the motor control board (which is covered below), which in its turn was connected to the 24 Volts power connector on the dashboard, located in the lower back of the robot.

2.2.2 Powerboard

Because of the large variety of components on the I-do platform, the robot has a custom powerboard that is used to supply the correct voltage to a large part of the components, like the ethernet switch, the LiDAR scanner, the main computer, and the Cradlepoint modem. The powerboard is not just responsible for powering electronics, as it also handles the data from the ultrasound modules at the top of the robot. The power distribution is explained more extensively in the power management section.

The powerboard is also directly connected to the dashboard, meaning that while charging the U1-24RT the current goes through the powerboard into the motor control board, which then directs it into the battery itself. During discharge, the powerboard is powered through the motor control board by the battery.

2.2.3 Motor control board

The main function of the motor control board is as the name suggests, controlling the motors. In order to accomplish that, the board has two full H-bridges integrated into the circuit, which control what current can flow into the DC motors. Additional functionalities of the motor control board are receiving the data from both encoders and also receiving the data from the bumper board (elaborated further in the bumpers section below). The last purpose of the board is to (de)charge the U1-24RT battery and also to monitor its state through the Ethernet cable which connects the board to the BMS.

2.2.4 Computers

ROS computer

The I-do system has 2 separate computers running various tasks. The main computer does not have a brand and is therefore referred to as the ROS computer. This computer runs all the robot movement-related programs. In order to do that it is connected to the motor control board, the

powerboard, the internet modem, and the LiDAR scanner through the Ethernet switch. The computer uses an Intel i7-6700TE as the processor and comes with 16 GB RAM memory. The computer does not have a dedicated graphics card, but uses the integrated MESA Intel HD Graphics 530.

Intel NUC

The second computer mounted on the robot is an Intel NUC, which is Intel's small size computer series, because of the compact design it is very useful for onboard robot applications. The Intel NUC on the I-do is a 7th generation of the type NUC7i5BNH. It used to control the tablet (which is missing), besides that is also controls the camera head on top. The NUC is connected to the Canon camera using USB, this allows the computer to take pictures and instantly save them on the hard drive. The NUC also has a USB connection to the camera control board, giving it control over the stepper motors in the camera head.

2.2.5 Internet capabilities

The local network on the robot is created using a TRENDnet TI-G80 Ethernet switch, this switch connects the various Ethernet-dependent components like the powerboard, the motor control board, the LiDAR scanner, and the ROS computer to each other. The switch is mounted in the Savioke base. As will be further explained in the software section, the I-do platform has a great dependency on a stable connection to the internet. To not depend on WiFi networks, the I-do platform comes with a Cradlepoint modem, which gives the robot the option to connect to the 4G network.

2.2.6 Sensors

The I-do platform uses a combination of sensors to determine what the surrounding area looks like and what the robot should do next. In figure 2.2 and figure 2.3 all the sensors on the I-do platform can be seen with their respective field of view roughly drawn into it. The field of view of the ultrasound sensors is drawn in blue, the field of view of the LiDAR is drawn in red, the bumpers are drawn as light green blocks and the angle of the depth cameras is represented by the grey arrows.

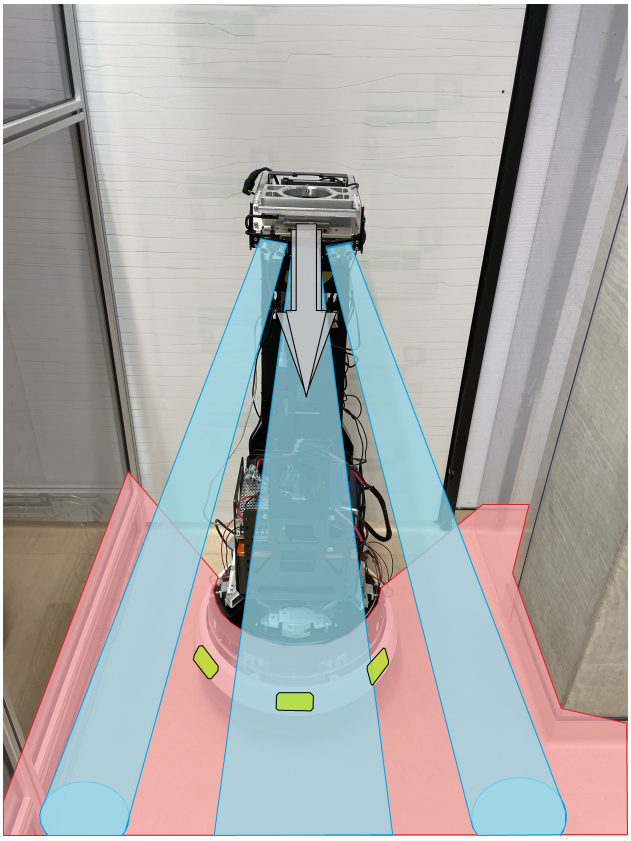


Figure 2.2: Front view of the I-do with the sensors and their respective field of view

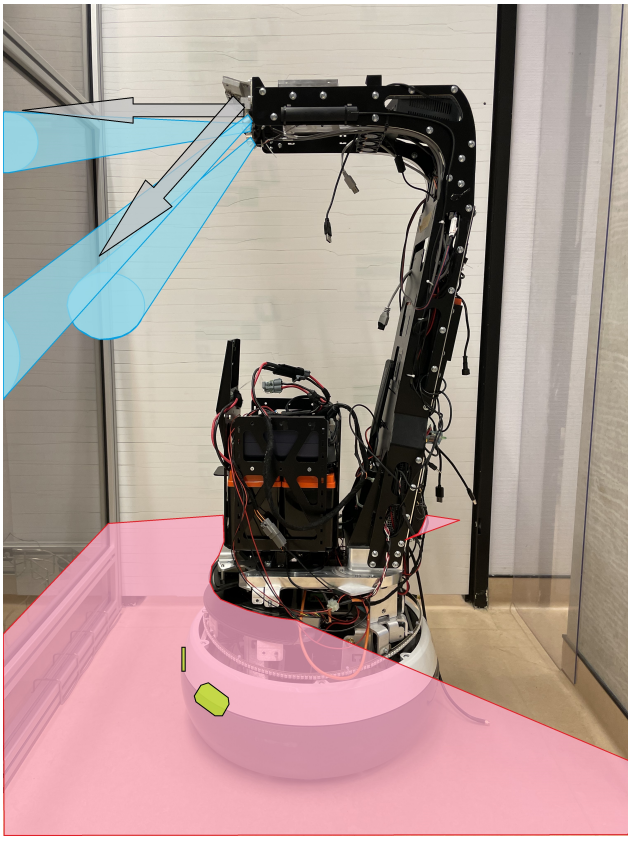


Figure 2.3: Side view of the I-do with the sensors and their respective field of view

LiDAR

One of the sensors on the I-do platform is a SICK TiM571 2D LiDAR sensor. LiDAR stands for light detection and ranging and is a method for determining how far an object is from the sensor. The way it works is that the sensor emits a very short laser beam pulse in the preferred measured direction, when the laser beam hits an object, part of the beam reflects back in the direction of the sensor. By measuring the time delay between the emitted beam and the measured reflected beam, the distance to the object can be calculated. The main advantage of using light for these measurements is that the response rate is very high, due to the extremely high velocity of light.

As the name of the sensor would imply, the LiDAR on the I-do platform is a 2D sensor, meaning that the sensor only performs measurements in one plane compared to the sensor. The SICK TiM571 model uses a light source with a wavelength of 850 nm, which falls under the infrared category. Performance-wise, the TiM571 has a field of view of 270 degrees and has a typical range of up to 8m, this can be seen in figure 2.4. The resolution of the scanner is 0.33 degrees, which results in 810 measurements per full 270 degrees scan (14). The sensor itself is located on the front part of the Savioke base and is thus mounted very close to the floor.

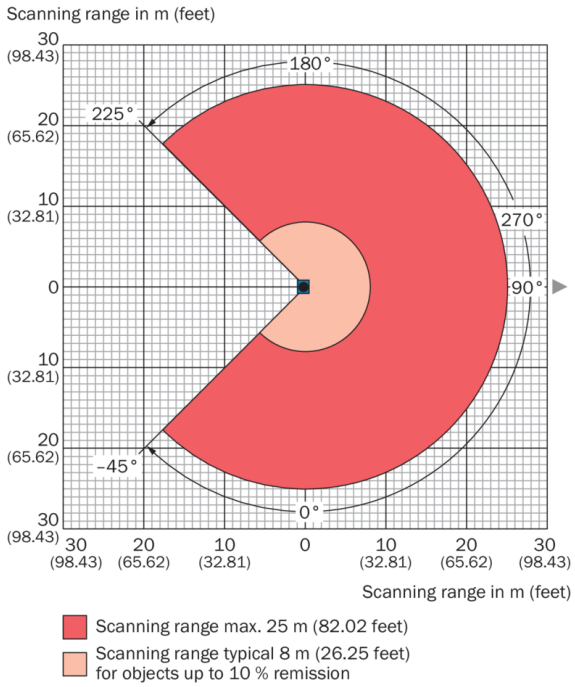


Figure 2.4: Range diagram of the SICK TiM571(13)

Ultrasound

The robot also has three Parallax PING))) ultrasound modules mounted on the top front part of the robot. These ultrasound sensors have the same principle as the LiDAR sensor explained above, but instead of using light reflection, it uses sound reflection to determine the distance. These sensors have a fixed measurement direction and are not capable of constructing a full surrounding scan (without moving the sensor) as the SICK LiDAR can. The PING module is able to measure objects up to 3 meters away and uses sound waves of 40kHz for the operation.

The added value of the ultrasound sensors can be explained by taking a look at the field of view of the LiDAR sensor. In order to ensure that the laser beams of the LiDAR do not skip over objects that could cause a problem for the wheels, the sensor has to be mounted very close to the floor. The downside of this is that the LiDAR data could indicate that an area with an overhang is clear (like the middle of a table for example), while the top of the robot would bump into it. This is where the ultrasound sensors come in, these give the robot a little bit more information about the area in front of the robot at the same height as the top. This could also be solved by installing an additional LiDAR sensor near the top, but since these are a lot more expensive than ultrasound sensors, and the robot doesn't need a very high-quality precise picture of the objects near the top of the robot, the ultrasound sensors suffice.

Bumpers

The I-do platform also comes with three sensors that detect whether the robot has hit anything. The robot has a separate structure, the bumper, surrounding the main hull of the robot. This separate structure is being held in place by springs but is able to move a little bit when a force is applied directly on the bumper, like when the robot runs into a wall. In order for the robot to measure whether it has hit anything, it must be able to measure whether the bumper structure has moved. To do that it uses three Hall effect sensors. These sensors use the Hall effect, which creates a voltage based on the magnetic field around the sensor and the current through it. When the iron structure moves over the sensor it alters the magnetic field, which is measured by the sensors and sent to the custom breakout board at the back of the robot. The three Hall effect

sensors only measure bumps from the front left, the front, and the front right. Meaning that the robot needs to take into account that it could hit objects while driving backward without noticing it.

Realsense D410

Another way that the robot can get an overview of its surroundings is by using its two Intel RealSense D410 depth cameras. These cameras have a resolution of 1280x720 pixels and are able to determine the depth of each pixel up to 10 meters. The cameras are mainly used for navigation on the I-do platform, one of the cameras is looking straight to the front and one of them is looking downwards at an angle. This means that the downward-facing camera can also be used to determine whether the robot is able to move forwards, as almost every path blocking obstacle would have to be in the field of view of this camera.

2.2.7 Canon camera

Some of the I-do platforms come with a head that has a Canon camera inside. The I-dos used this to take pictures of posing people. Because the primary focus of this research is to get the basics of the robots operational again, the camera head portion is considered out of the scope of this research.

2.2.8 DC motors

In order to move the robot, the I-do platform has a differential drive system, meaning that two fixed wheels are powered by motors and that the other rotation wheels are there to keep the robot stable. The robot is able to turn by having the wheels rotate at a different speed, therefore it does not require skid steering to steer the robot. To power the wheels, the robot has two DC motors rated at an operating voltage of 18 Volts, while using a current of up to 2 Ampere. The two DC motors combined therefore can have a typical power consumption of up to 72 Watt.

Optical encoders

To be able to measure the position of the shaft through the DC motor, an optical encoder is mounted on top of it. An optical encoder is a type of rotary encoder, which uses light to measure the rotational position of the shaft. There are multiple designs for these types of encoders, one of the designs has a disk mounted at the end of the DC motor's shaft. This disk has an evenly spaced set of holes in it. By placing a light source on one end of the disk and a light detector on the other end with a mask between them, a binary signal is created that goes high as long as the holes in the disk and the mask line up with the light source and the detector. This means that a pulsing signal is created whenever we spin the shaft, giving us the possibility to calculate the position of the shaft by counting the pulses. The only problem with this is that this system would not be able to measure the movement direction, which is why (optical) rotary encoders have two sets of these light sources and detectors. By measuring which one pulses first when a hole approaches, the movement direction can be determined. The inner workings of an optical encoder is visualized in figure 2.5.

The optical encoders mounted on the I-do platform use a differential output, meaning that the signals are not outputted over a single wire, but rather over a (differential) pair, where one is the reverted signal of the other. The original signal can be constructed on the receiving end by taking the difference between the wires and dividing the result by two. The advantage of this approach is that the wires are far less susceptible to interference because the noise would impact both wires, meaning that the difference between the two wires would not be influenced. This is especially useful for situations that require the wires to be relatively long.

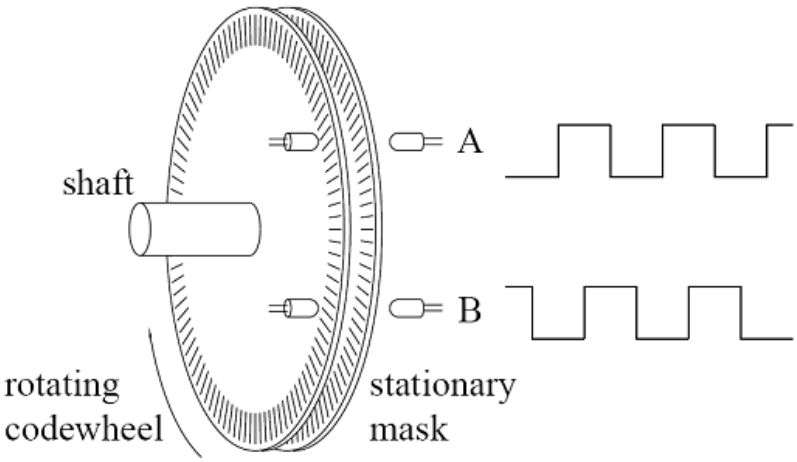


Figure 2.5: The inner workings of an optical encoder

2.2.9 Power management

The original power management on the I-do platform is sketched in figure 2.6. The same color coding is used as in figure 2.1, meaning that the blue components are mounted in the entertainment pack, the red components in the Savioke base, and the green components in the tower. The individual power requirements of the components can be found in Appendix A.1.

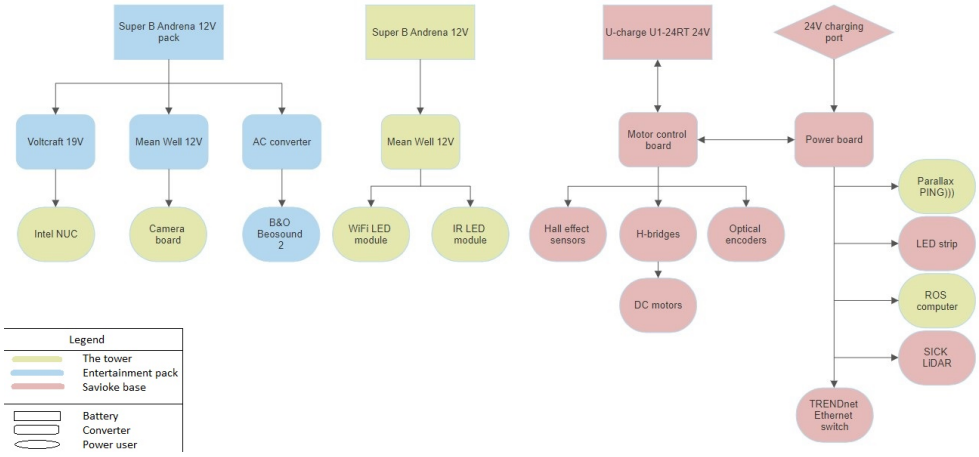


Figure 2.6: Overview of the power distribution

2.2.10 Bang & Olufsen sound system

Eugeen comes with a very high-quality speaker from Bang & Olufsen. The speaker is a Beosound 2 and uses Bluetooth as audio input. It is normally connected straight to the grid at 230 Volts, on the robot it is powered by the battery pack, where a transformer turns the 12 Volts DC into 230 Volts AC.

2.2.11 LEDs

The I-do platform comes with three different LED segments that can be independently controlled. The LED strip at the top of the robot (behind the "KUKA" letters) is controlled by a WiFi LED controller, this controller operates at 12V and can control the patterns and the colors either through a WiFi LED remote or through a phone application. The LED strip on the casing of the robot is controlled with an IR LED controller, which is controlled by an IR remote. This

controller also operates at 12V and is also able to set a lighting pattern and color of your choice. The final LED strip on the robot is mounted around the Savioke base and contains 193 LEDs, which operate at 5V. This LED strip does not come with a controller but is controlled through the powerboard. The cable of the LED strip has 3 inputs, one 5V supply, one ground, and a digital (green) input that is used to control the LEDs.

2.3 Software

In order to be able to use the robots for a different task than was originally intended by KUKA, the software needs to be understood. This turned out to be the biggest challenge with this robot platform, as the computers did not seem to be very user-friendly. Upon starting up the ROS computer, the computer seemed to cut off the video output as soon as it had finished the booting procedure. The ROS computer runs on Ubuntu 12.04 Xenial and without access to the file system, it would be impossible to reuse the original software. To circumvent the booting procedure from KUKA, the computer was booted in recovery mode, this made it possible to open a root terminal, which gave insight into the file system. After spending a lot of time going through every single folder on the computer, it appeared that the software has a large dependency on an active connection to the servers of Savioke. Unfortunately, the majority of the interesting named programs were all pre-compiled, making it very hard to reverse engineer the software. The main focus was on finding out how the robot communicated with the motor control board, as that would be crucial in order to let the robot execute tasks to our liking. Even at this low-level control, evidence was found that the software used external servers of Savioke to control the motors through WebSocket connections. It is unclear why Savioke decided to take this approach, but it does explain why there was a need to give the I-do platform access to the 4G network.

2.3.1 ROS

With the history of Savioke, it was to be expected that the company had used the Robotic Operating System (ROS) as the framework for the robot. After another extensive search through the file system looking for anything related to ROS, a few programs were found that sporadically contained pieces that looked like they were to function inside a ROS network. Unfortunately, no "normal" ROS network was found anywhere in the file system, further confirming the earlier finding that the I-do platform mostly relied on WebSocket connections. It is up to a different engineer with more knowledge of ROS, whether there was a modified ROS network operating on the system, but for the scope of this research, it was decided that it would be better to refrain from putting more hours into sifting the file system and instead focus on how to work around the need to reuse the existing software.

2.4 Design goals

In order to make changes to the robot, it is very important to have a good of idea what the I-do platform is going to be used for. As the platform is so versatile, the University of Twente has numerous options. First of all the University could use the robots during events, just like KUKA did. The purpose of the robots is then to have as many people interact with the robot, boosting the reputation of the University as a technologically advanced institute.

The second purpose is to use the robots for research purposes, possibly useful fields would be to study the relationship between humans and (social) robots. As discussed before, the I-do platform comes with a variety of equipment, all of which are focused on aiding humans, making the robot very suitable for this kind of research.

Another use of the I-do platform could lie in the educational field. The main functionalities of the robots are quite basic, making the platform a good learning project for students. A possi-

bility would be to remove a piece of software and let the students redesign that specific part as a project.

2.5 Requirements

To make the list of requirements as concrete as possible, it is important to keep the previously mentioned design goals in mind. First of all the robot should be able to construct a map of the environment and navigate itself around.

In order to make the robot applicable for a wide variety of uses, the robot should be able to accept simple coordinate targets for the robot to navigate to. The speed and the accuracy for this navigation should be reasonably modular, as there are so many different use cases possible, with each a different environment and thus different desirable traveling speeds and accuracies.

It is quite likely that the robot will be used indoors, therefore it should be able to handle confined spaces (like door openings) without any problems.

Safety is also incredibly important in this project, as the robot could be used by students, therefore the possibility of unsafe commands should be considered. In order to guarantee safety for all people involved, the robot should contain a low-level safety feature that stops the robot no matter what commands are sent from the top-level structure when the robot gets into a dangerous situation.

As this research is about reusing an original robot, the new set requirements should be met while recycling as many components on the original I-do as possible.

3 Design

3.1 Hardware

As the original pre-compiled software was not easy to reverse engineer, it was concluded that the custom motor board and power board could not be reused. Therefore both of these boards were removed from the robot. This presented the opportunity to design a system that is more focused on the specific design goals that were set in the analysis chapter. To save as much original hardware as possible, it was decided to try to keep the main robot programs on the existing ROS computer, freeing processing power from the custom PCBs' replacements.

With the removal of the custom PCBs, a new set of hardware had to be found that could execute the following tasks.

- Read the analog signals of the bumper sensors (3 analog pins).
- Read the digital signals of the ultrasound sensors and the optical encoders (5 digital pins + 2 interrupt pins).
- Supply the DC motors with the intended voltage for driving the robot.
- Supply a constant voltage to all of the components.
- Write digital signals to the LED strip (1 digital pin).
- Communicate with the intended ROS network on the ROS computer.

3.1.1 Arduino Mega

Because of the before-mentioned decision to leave the majority of the processing up to the ROS computer, the new hardware should be mostly focused on handling the inputs and the outputs of the signals. Therefore a microcontroller would be better suited for this job compared to a single-board computer like a Raspberry Pi. However, that still leaves a lot of possible options on the table. Keeping the design goal in mind to have the total system easily understandable for students, a microcontroller which the majority of students have previously worked with would be of great value. Therefore it was decided to go with one of the Arduino microcontrollers. The Arduino family consists of multiple boards that could be well suited for the tasks at hand.

The first candidate is the Arduino Uno, this is the microcontroller the majority of EE students at the University of Twente have started out with in earlier projects. The board has an operating voltage of 5 Volt and is based on the ATmega328p microcontroller. The board comes with 14 digital I/O ports, from which 6 are able to produce a PWM signal and only 2 are suited for interrupts. Furthermore, the board also comes with 6 analog input pins and has 32 KB of flash memory.

The second candidate is the Arduino Nano, this board has exactly the same specifications as the Arduino Uno, but it comes in a different size package and thus lacks some features like the DC power plug.

The last candidate is the Arduino Mega, the Mega is a board that unlike the Uno and the Nano uses the ATmega2560 microcontroller. This microcontroller also operates at 5 Volt and comes with 54 digital I/O pins, from which 15 are able to produce a PWM signal and 8 are able to handle interrupts. Finally, the board comes with 16 analog input pins and 256 KB of flash memory.

All 3 candidates have enough digital I/O and analog pins to accommodate all of the sensors, the relevant differences lie mostly in the flash memory and the available interrupt pins.

The flash memory determines mostly how many variables can be maintained at one point in time, a larger flash memory would therefore allow a bigger program with more (global) variables to run on the microcontroller. Considering that the I-do platform could be further developed by students, a larger flash memory could be beneficial in the future.

The amount of pins that are able to handle interrupts determines how many signals can be received that can temporarily stop the program. This will be used by both the encoders to ensure that position is incremented or decremented with each pulse of the encoder. This means that at least 2 interrupt pins will be used by the sensors. As this is already the maximum amount on both the Arduino Uno and the Nano, and it is preferred to have a few spare pins for further development, it is concluded that with the previously set design goals the Arduino Mega is the best-suited microcontroller for the I-do platform.

3.1.2 Pololu motor drivers

As the Arduino Mega is unable to supply the DC motors with 18 Volts, especially with the rated current, an extra component is needed to handle this task. On the custom-made PCBs, this was done in the Motor Control board, which has an integrated full H-bridge for each DC motor. But with the removal of this motor control board, external motor drivers are required. As briefly discussed in the analysis section, the DC motors have a maximum rated voltage of 18 V, and have a typical current draw of 2 A each. With these specifications, there is a wide variety of medium to high power motor drivers available. For the first prototype of the I-do platform, it was decided to use the High-power 36v20 CS motor driver from Pololu, the main reason for this is that there was a set of these motor drivers already available. The specifications (12) of these drivers are quite a bit better than necessary for this project, therefore it could be investigated to replace these with cheaper options when the other I-do robots are rebuilt.

3.1.3 Step down DC-DC converter

Another component that was powered by one of the custom PCBs was the LED strip mounted on the Savioke base. This LED strip uses 5V and can use up to 1A, which is far too high for the Arduino to deliver (the only 5V source so far), therefore an additional Step down DC-DC converter is required to supply the LED strip with sufficient power. The converter that was chosen is the LM2596S DC-DC adjustable step-down converter, the main reason for this is that it was one of the cheaper options while being able to supply more than enough current (up to 2A).

3.1.4 Power management

The original power management of the I-do platform is drawn out in figure 3.1. The shapes in the figure represent what kind of component it is, the squares are the batteries, the rounded-off squares are the converters, and the ovals are the power users. The colors represent in which section of the robot the component is located, blue stands for the entertainment system, red is the Savioke base, and the yellow/green components are located in the tower segment. The exact voltage and power ratings of the (power using) components can be found in Appendix A.1, the power ratings of the converters can be found in Appendix A.2.

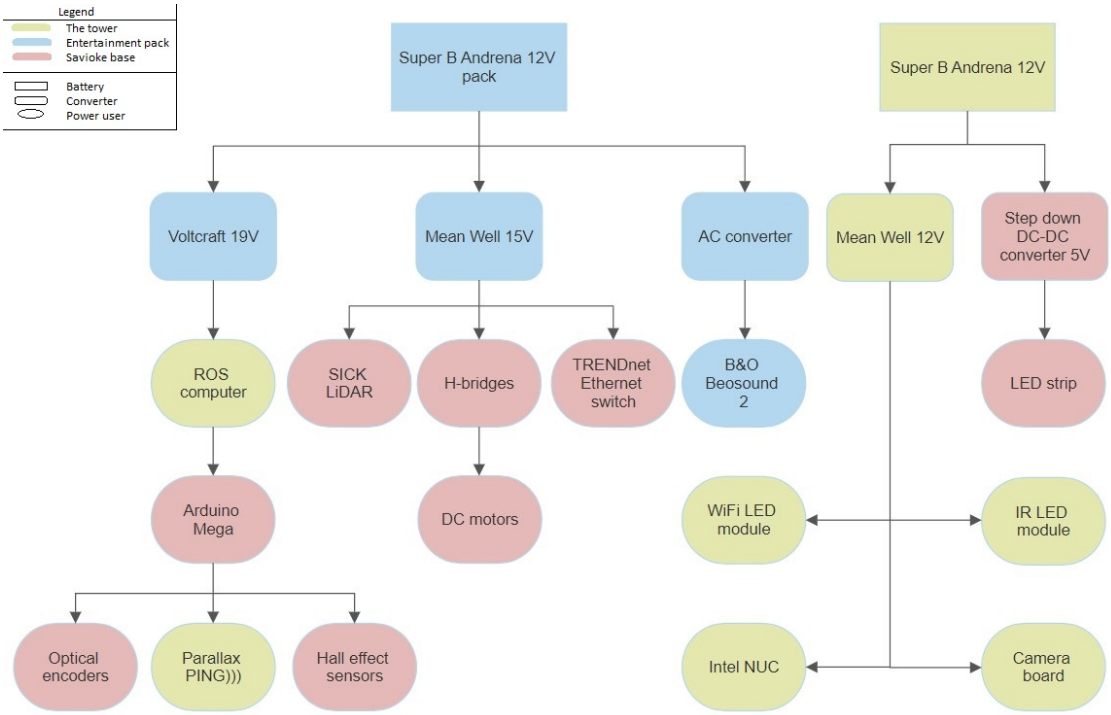


Figure 3.1: Overview of the power distribution

3.2 Software

Since the original software was not reusable, the entire software structure needs to be completely rebuilt. In order to do that it was decided that the software should be built on the previously mentioned Robotic Operating System, or ROS in short. The main reasons for this are that ROS makes a robotic system very modular, this is because ROS is completely open-source and the users can always add new packages to the existing software. This plays very well into the requirement, that was earlier set, that the platform should be easily modifiable by students. Next to this, the original software also (partly) uses ROS, this leaves the option open to reuse the original software if it was somehow still extracted from the I-dos. The advantages that ROS brings to any robotic system make it very popular within the developers' community, this makes ROS a very useful framework to learn as a student. All of these advantages combined make ROS a very good base to use for the software. In figure 3.2 a software overview can be seen from the entire platform. The squares represent software packages (on the ROS computer) or software sections (on the Arduino). The oval shapes are used for sensors and the hexagons are the (physical) H-bridges.

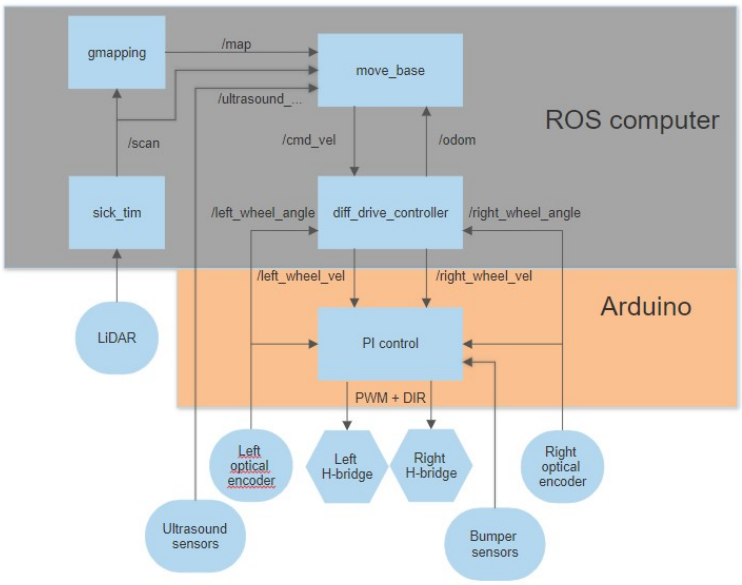


Figure 3.2: Overview of the total navigation network

3.2.1 ROS architecture

In appendix A.2 the rqt graph of the ROS network is shown, containing all the relevant nodes, topics, and services on the network. The packages with the corresponding nodes, topics, and services are further explained in the sections below.

move_base

The package that is responsible for the path planning and the creation of velocity commands is the move_base package, an overview is sketched in figure 3.3. The move_base package uses a global planner, which calculates a path using a global static costmap, and a local planner, which uses a local dynamic costmap. The package combines these two planners to navigate the robot through the environment towards its goal. When comparing figure 3.3 with figure 3.2, the implementation of the platform-specific nodes can be seen. The I-do has two types of sensor sources, the SICK LiDAR, and the ultrasound sensors. In order to use the ultrasound sensors, the package "range_sensor_layer" was added, which adds an additional layer to the costmap of both planners. The optional amcl package is normally used to determine the pose (location) of a robot on an existing map, the I-do uses SLAM (covered below) and therefore does not require amcl. The output of the move_base package is a velocity command that gets sent to the diff_drive_controller package. This velocity command contains both the preferred translational velocity and the preferred rotational velocity. Because the I-do has two fixed powered wheels, it can only move forward, backward, and turn around its axis, therefore only the velocities on and around these axes will be sent to the diff_drive_controller package.

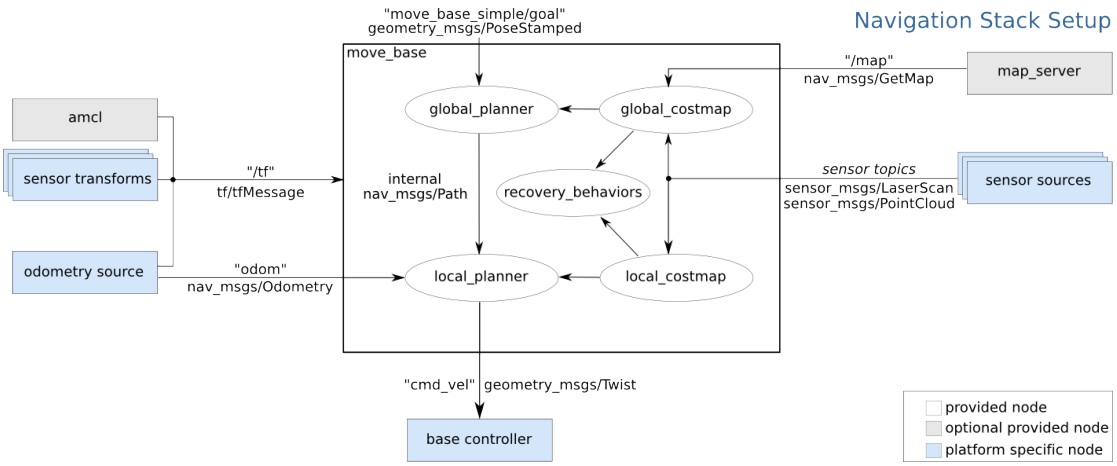


Figure 3.3: Overview of the move_base node(11)

gmapping

In order to construct a map from the laser data, a package called "gmapping" is used. The package is specifically designed to perform Simultaneous Localization and Mapping (SLAM) using the laser scan message type incorporated into the sensor messages in ROS.

In order to build an accurate map, the algorithm needs to have an accurate pose of the robot on said map. This creates a problem because odometry data from the wheel encoders alone has a long-term drift from the actual position. Therefore the gmapping package also needs to localize the robot on the map that it has been building using the laser scanner. The need to simultaneously perform localization and mapping creates a chicken-and-egg problem which is further explained in paragraph 5.2 of (6). The Rao-Blackwellised Particle Filter that is discussed there, is also the approach that is used by the gmapping package.

Sick_tim

In order to be able to use the SICK TiM571, a ROS package is needed named "sick_tim", this is a wrapper that wraps the sensor data into the LaserScan message from ROS. The SICK scanner from the original I-do platform was configured with a certain IP address, therefore every SICK scanner needs to be reset using the SOPAS engineering tool from SICK (15) before they can be used in the robots.

Diff_drive_controller

As previously described, the move_base package sends a single velocity command that contains both the translational and the rotational target velocities. The I-do platform however has two independent motors, which means that the single velocity command needs to be split up into two separate target speeds for the individual motors. This is done by the diff_drive_controller, which besides splitting the velocity command, also does the exact opposite, as it combines the data from both the wheel encoders into a single odometry message(10). The odometry message consists of the pose of the robot and the current translational and rotational velocity.

The diff_drive_controller package is part of ros_control, which is a set of controllers that all work according to a template. The data flow of this type of controller is sketched in figure 3.4. A controller manager loads the controller and functions as the node for the controller. The orange part in figure 3.4 is the hardware interface, which is the link between the controller and the hardware. To make ros_control more flexible, this hardware interface is supposed to be written for every type of robot independently, as a robot can have many different ways of

getting the commands across to the motors. The robot-specific code is mostly written in the write() and the read() functions. These functions are called by the controller framework and allow programmers to implement their own communication protocol with the hardware. In the case of the I-do, the communication is done through ros_serial (see below), therefore the write() and the read() functions publish and subscribe to the topics of ros_serial. The full code of the hardware interface that was written for the I-dos can be found in appendix A.6.

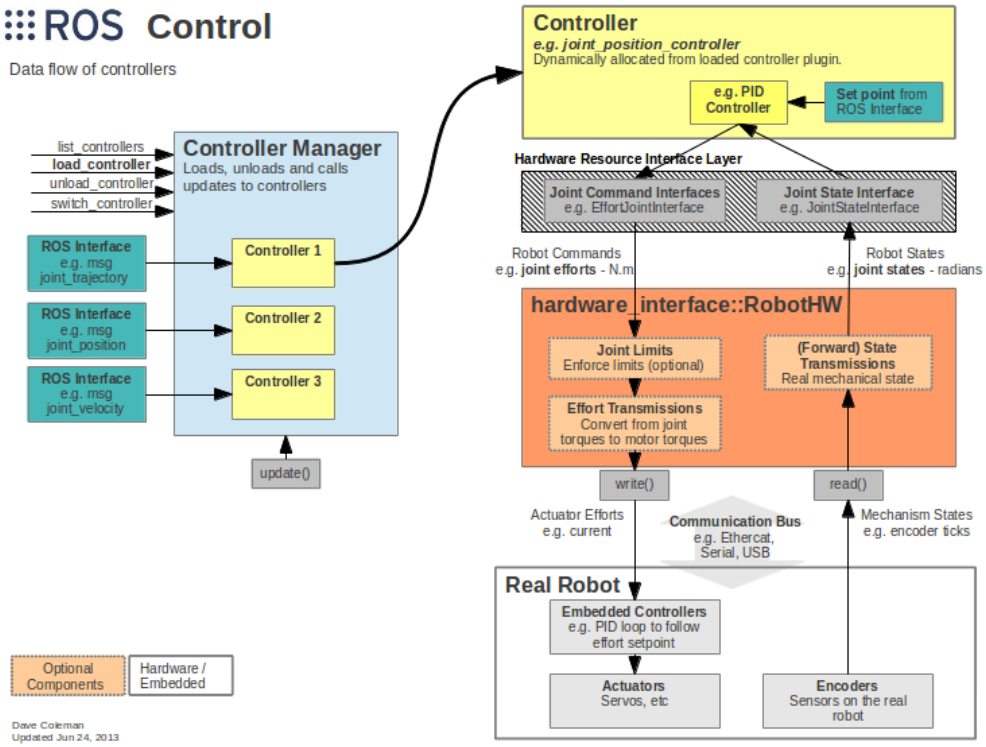


Figure 3.4: General functioning of ROS_control (5)

Rosserial

In figure 3.2 the connection between the ROS computer and the Arduino Mega can be seen. To accomplish this connection a package called "rosserial" is used. This package makes it possible to establish a serial connection to other devices. This means that every connection between the grey and the orange part in figure 3.2 goes through the rosserial package. When looking at figure A.1 in appendix A.2 it can be seen that the rosserial package creates a node that functions as the Arduino. This node can be thought of as a sort of black box containing everything that is behind the serial port. The serial connection with the Arduino Mega does have downsides, due to the limited resources (like SRAM) on the Arduino there is a maximum amount of subscribers and publishers. The ATMega2560 that is used by the Arduino Mega is able to have 25 publishers and 25 subscribers simultaneously on the serial port, which is more than enough for the I-dos, which at the current configuration use 5 publishers and 2 subscribers.

TF

The package "TF" is crucial for the robot because it links the different frames together. All the sensors publish their data with the name of their frame attached to it. This frame is linked to the base frame (base_link) of the robot by several static_transform_publishers, who send TF

messages containing the position and orientation of the frames. These static publishers are located in the general launch files for the I-dos. In general, frames are not static, an example of this is the base frame of the robot that moves over the (global) map frame. In appendix A.3 the TF tree can be seen, this shows how all the frames are connected to each other.

3.2.2 Arduino architecture

The Arduino Mega has the following tasks on the I-do platform.

- Publish and subscribe to the required topics through rosserial_arduino
- Read encoder data to calculate the position of the wheels
- Read the ultrasound sensors and filter the data
- Determine the wheel velocities and filter the data
- Calculate the required duty cycle to set the motor speeds to the target speeds
- Read the bumper sensors and implement a low-level kill switch
- Control the LED strip

Wheel position calculation

Both encoders have one output wired up to an interrupt pin on the Arduino. This allows the Arduino to interrupt the loop whenever a pulse is sent from the encoders, indicating that the motor shaft has rotated. In order to check the direction of the movement, the callback function reads the state of the other output of that encoder (and thus the direction) that is wired up to a normal I/O pin. Whenever the shaft of one of the motors rotates, the Arduino increments or decrements, depending on the direction, the position counter variable of that motor. This variable represents the position of the wheel in ticks, which can then be converted into radians. The number of ticks per wheel revolution was unfortunately not known, because the motor has a gearbox mounted on it with an unknown gearing ratio. Therefore the amount of encoder ticks per wheel revolution is calculated by letting the wheel spin a large number of times and then taking the average amount of ticks per revolution.

PI control

In order to use the motors effectively, the Arduino Mega should be able to adjust the duty cycle of the PWM signal to both motor drivers in such a way that the individual target speeds for both motors are (quickly) reached. Because of the number of factors that impact the steady-state speed of the motor at a specific duty cycle, a closed-loop control system is highly preferred. In order to keep the implementation relatively simple, the PID controller approach was chosen to control the system. The target of the closed-loop system is in radians per second, therefore to keep the system intuitive, the entire control loop will be in radians per second. In order to calculate the current wheel velocity, the position (from the encoders) needs to be differentiated. This could lead to noise-related problems, therefore the resulting velocity will have to be low-pass filtered. A full PID controller differentiates the error, resulting in extra additional noise. Therefore it was decided to leave the differentiating block out of the controller, resulting in a PI controller. The downside to this is that the PI controller could have a larger overshoot and a longer settling time compared to a full PID controller(9). The PI controller should in theory suffice for the project, as the majority of the control relies in higher layers of the software (like in move_base). This means that the performance demands (overshoot and settling time) are not very strict. These loose requirements make it possible to tune the controller with a trial-and-error approach.

3.3 How to use the robot

In this section I will sketch the bigger picture on how the robot can be used, an exact step-by-step approach can be found in appendix A.4.

3.3.1 Connectivity

The required ROS nodes have to be launched with the terminal window (shell prompt) on the ROS computer. In order to do that one can either use a monitor and keyboard directly on the computer or open the terminal window through a different laptop (or computer) using SSH. As it would be quite cumbersome to attach a monitor and keyboard for every command, the SSH method is recommended. The local IP address on the local network (made by the Ethernet switch) is 192.168.0.5 (static). As mentioned above, the step-by-step approach can be found in appendix A.4. It is also possible to use rviz on the remote laptop, but this requires some additional steps as the ROS network needs to be set up properly (see the same appendix).

3.3.2 Initialization

The entire system can be initialized at once with the use of launch files, these files can be found in the "launch_ido" package in the folder "launch". There are 3 launch files available to the user, the "full.launch" file starts every single node that is required for the normal operation of the robot. The "full_rviz.launch" file does the same thing, but also launches a preconfigured rviz environment, rviz is the graphical interface built within ROS. The last launch file that was written for the I-do platform is "teleop.launch", this launch file (only) launches the required nodes to create velocity commands with the PlayStation 3 controller.

3.3.3 Controlling the robot

The robot itself can be controlled in two ways. The first one is the most straightforward and is through the "cmd_vel" topic. All the commands that are published on this topic, will be executed by the moving base. Although the localization and the mapping still work, the navigation (move_base) system is simply on standby. One of the easier ways to do this is by launching the above-mentioned PlayStation 3 controller launch file, this allows the user to drive the robot with the controller. This manual driving could be very useful, as the robot learns and maps the environment while it's being driven around. This information can then be used to navigate autonomously.

The other option to work with the robot is by publishing a goal pose on the "move_base_simple_goal" topic. This sends the command to the navigation stack to start planning and following a path to the desired location. The easiest way to do this is through the rviz environment, which allows you to send the goal location by clicking on the map.

4 Results

4.1 PI control

As described in the design chapter, the controller parameters (K_p and K_i) are found using a trial-and-error approach(8). As a starting point, K_i was set at 0, this means that the controller is only proportional. Then the proportional gain (K_p) was increased until the system had a noticeable overshoot but still recovered to the steady-state within a few oscillations. The proportional gain that matched that criteria turned out to be 57.

With the proportional gain set, the integral gain could be determined. The integral component is there to remove the remaining error to the reference velocity. A correct integral gain results in a system that has (roughly) the same quick settling time that was achieved before, but without the offset in the steady-state. The integral gain that was found to be working well is 62.

The step response of the system with these parameters can be seen in figure 4.1, this graph contains the measured responses of both motors to a reference velocity of 3 rad/s.

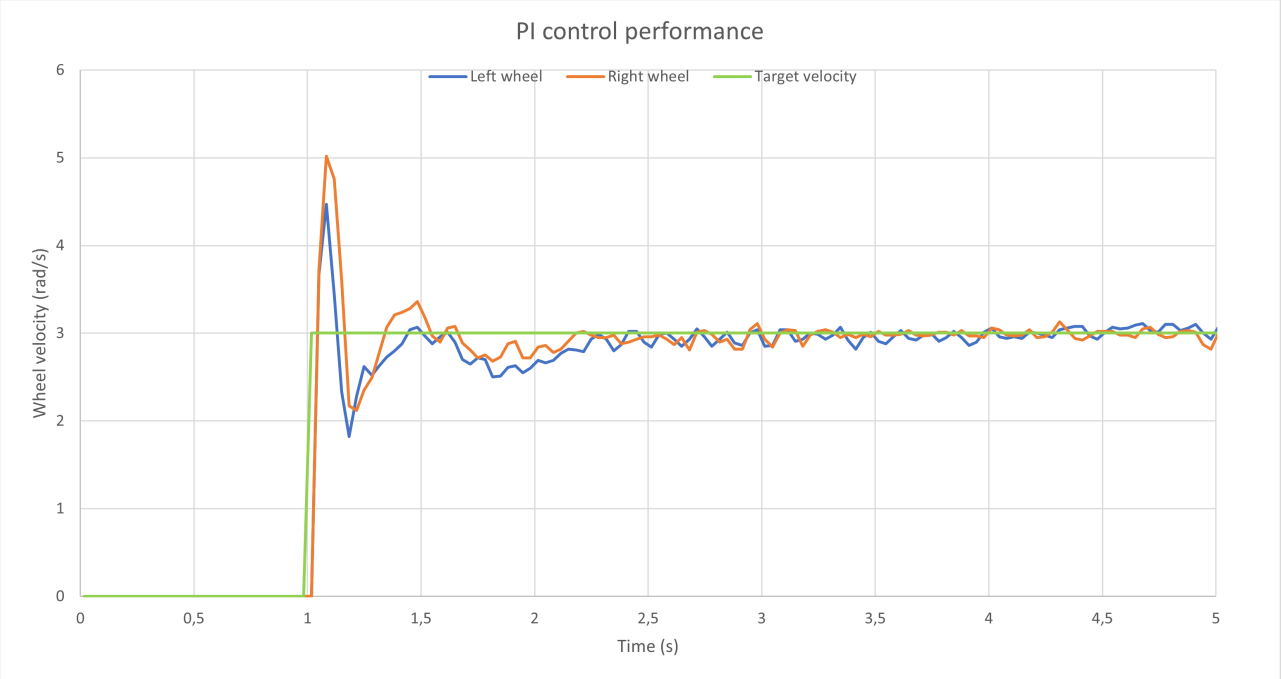


Figure 4.1: Performance of the PI controller

4.2 Navigation

Figure 4.2 contains a map that was created by an I-do (Eugeen) of RaM lab 1. The I-do was driven around manually using a PlayStation 3 controller. There are three doorways visible on the map, two in the lower part and one on the left side. During this session the robot was not driven outside of the lab, therefore the mapping of the areas behind the doorways was purely created by the view that the robot was able to get from within the lab.

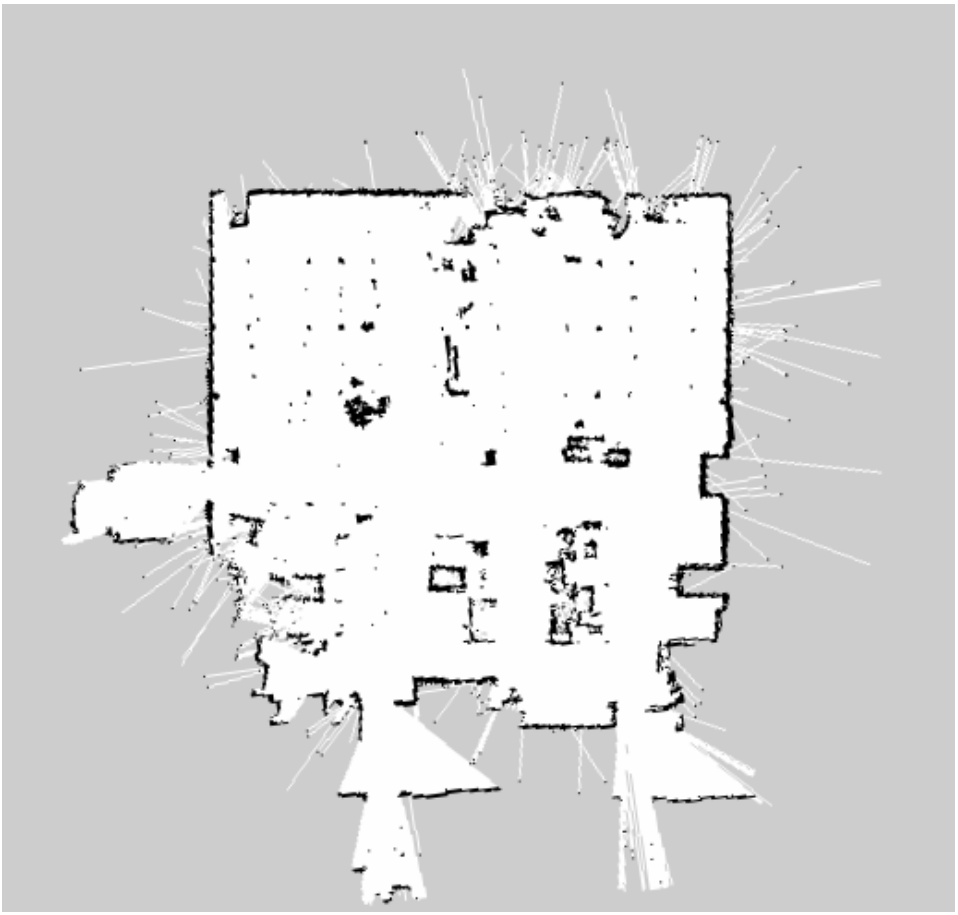


Figure 4.2: A map that was created of RaM lab 1

4.3 Applicability for educative use

In the design goals section of the analysis chapter, it was discussed how the I-do platform could be of use in the educational field. This made it very desirable to have a system that is easily understandable by students, who could use the I-do platform during projects. The knowledge that is required by these students to be able to work with the robot depends a lot on the type of project. These projects can be divided into two types, the first type requires students to use the current functions of the robot, while the other type also requires students to add new features.

Looking at the steps that need to be followed to get the robot operational, it is safe to say that students do not necessarily need to have any knowledge of ROS to work on projects of the first type. This is mainly because the step-by-step approach (found in appendix A.4) allows students to treat the I-do platform as a black box, only making use of the intuitive graphical interface rviz.

The second type of project requires students to have more knowledge on ROS, as they have to write new nodes that use the existing topics on the network. To get an idea of how long it would take for students to get comfortable enough with ROS to make meaningful changes to the system, we can take a look at my own experience with ROS. When I started with this research I had no prior knowledge of ROS whatsoever. After finishing the first 11 tutorials on the ROS website (1) I was able to understand the key concepts and write my own nodes, this process took me about a full day. Adding some additional time for students to understand the I-do specific software structure, it is to be expected that even students that do not have any prior knowledge of ROS are able to get up to speed within a few days.

5 Discussion

The performance of the PI controllers can be assessed by looking at the step response of the individual motors to a new reference velocity in figure 4.1. Here it can be seen that with the exception of the first peak (which is exaggerated due to wheel slip), the system settles at the reference velocity without multiple oscillations, this means that the robot is quite responsive while maintaining the required accuracy for indoor usage. The PI controllers' parameters could be further tuned to different priorities, an example of this would be to reduce the wheel slip and overshoot to ensure that the robot does not make unexpected movements, this could be beneficial in extremely confined spaces. For the requirements that were set in the analysis chapter the current performance is good enough, this is mainly due to the higher level of control that is implemented in the ROS network. This extra layer of feedback control will intervene and change the reference velocities whenever the PI controllers' imperfections threaten the operation of the I-do.

The mapping performance of the robot is assessed with the map that was created by the I-do (see the results chapter). Looking at this result it can be concluded that the robot is able to map the environment reliably and that there are no problematic localization issues. This can be derived from the fact that the walls in figure 4.2 of the results are straight and that there are no clear faulty angles between them.

In the results chapter, my own learning experience with ROS was used as an indication on how difficult it would be for other students start working on the I-do platform. Further research on this could be done by a trial with an actual set of students that have never worked with ROS before. This information could then also be used to improve the step-by-step guide in the appendix.

6 Conclusion

Looking back on the first stage of the research, the reverse engineering, we see that new documentation on the robot platform has been made. Because of this, it should now be possible for a student with no prior knowledge of the I-do platform to start adding new features without having to disassemble and research the entire robot.

The design stage focused mainly on restoring the basic functionalities, like driving and mapping, while keeping the entire system easy to use for students. Comparing the driving performance of the robot with the requirements that were set in the analysis chapter, we can conclude that the robot is indeed able to move around precisely enough to operate safely in confined spaces.

The safety is ensured by the speed limits in the ROS network, which is configurable in the configuration files, and by the low-level kill switch that is implemented on the Arduino. This kill switch is automatically activated by the bumper sensors and hereby tries to prevent additional damage after the first bump.

The robot combines the previously mentioned mapping and driving features to navigate autonomously. As set in the requirements, for this to work the robot only needs the goal coordinates on the map frame. This makes it very easy to control the robot and to change its goal during operation. With the use of ultrasound sensors, the robot is able to navigate itself around objects that are not directly perceivable by the LiDAR sensor, making the robot much more effective for autonomous indoor usage.

The newly designed robot only comes with 3 batteries instead of the original 4, this makes it easier for the university to maintain them, especially as they are all of the same manufacturer and operating voltage. Besides it being easier to maintain, all the hardware and software components are now all open-source, removing all the "black box" components that were previously on the platform. These aspects combined make the robot less complex to work on, making it more suitable for use in an educative environment.

Although the robot passed every requirement that was mentioned in the analysis chapter, it is not perfect. Further fine-tuning of the PID parameters could prevent wheel slip, which could reduce the long-term drift in the odometry data, making it more reliable. This problem is now compensated by the gmapping package, but this uses more resources on the ROS computer and is therefore not desirable.

Further development on the robot could bring additional interesting features to the robot, an example of this is the incorporation of the D410 Realsense depth cameras. These could be used to implement the people tracking software that was developed for the EU FP7 project SPENCER (4). Another use for the depth cameras could be to use them as additional sensors for mapping the environment. Besides these possible features, additional research could also be done on the camera head of the original I-do platform.

A Appendix

A.1 Power ratings

Component(s)	Voltage (range) rating	Maximum power or maximum current usage
B&O Beosound 2	230V AC	100 W
ROS computer	19V DC	unknown
DC motors	18V DC	4 A
TRENDnet Ethernet switch	12V - 56V DC	5 W
SICK LiDAR	9V - 28V DC	4 W
WiFi LED module	12V DC	unknown
IR LED module	12V DC	unknown
Intel NUC	12V - 19V DC	65 W
Camera board	12V DC	unknown
LED strip	5V DC	5 W
Optical encoders	5V DC	72 mA (total)
Parallax PING)))	5V DC	105 mA (total)
Hall effect sensors	5V DC	15 mA (total)

Table A.1: Power rating components

Converter	Voltage range	maximum power or maximum current
Belkin F5C412Eb300W	230V AC	300 W
Voltcraft SMP-125 USB	15V - 24V DC	120 W
Mean Well SD-100A-12	10V - 16V DC	102 W
LM2596S DC-DC step-down converter	1.5V - 30V DC	2 A
Arduino Mega	5V DC	0.5 A (USB powered)

Table A.2: Power ratings of the converters

A.2 Rqt graph of the (reduced) ROS network

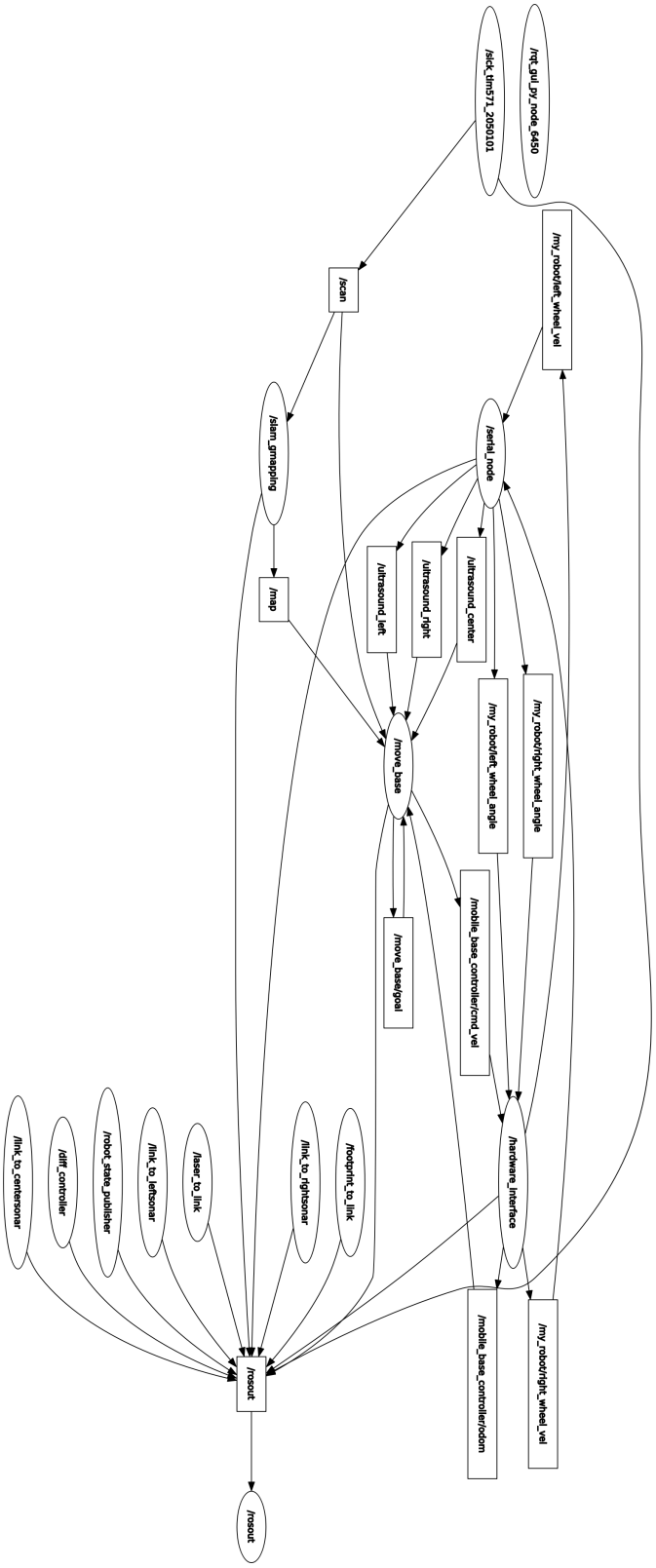


Figure A.1: Rqt graph of the ROS network

A.3 TF tree of the ROS network

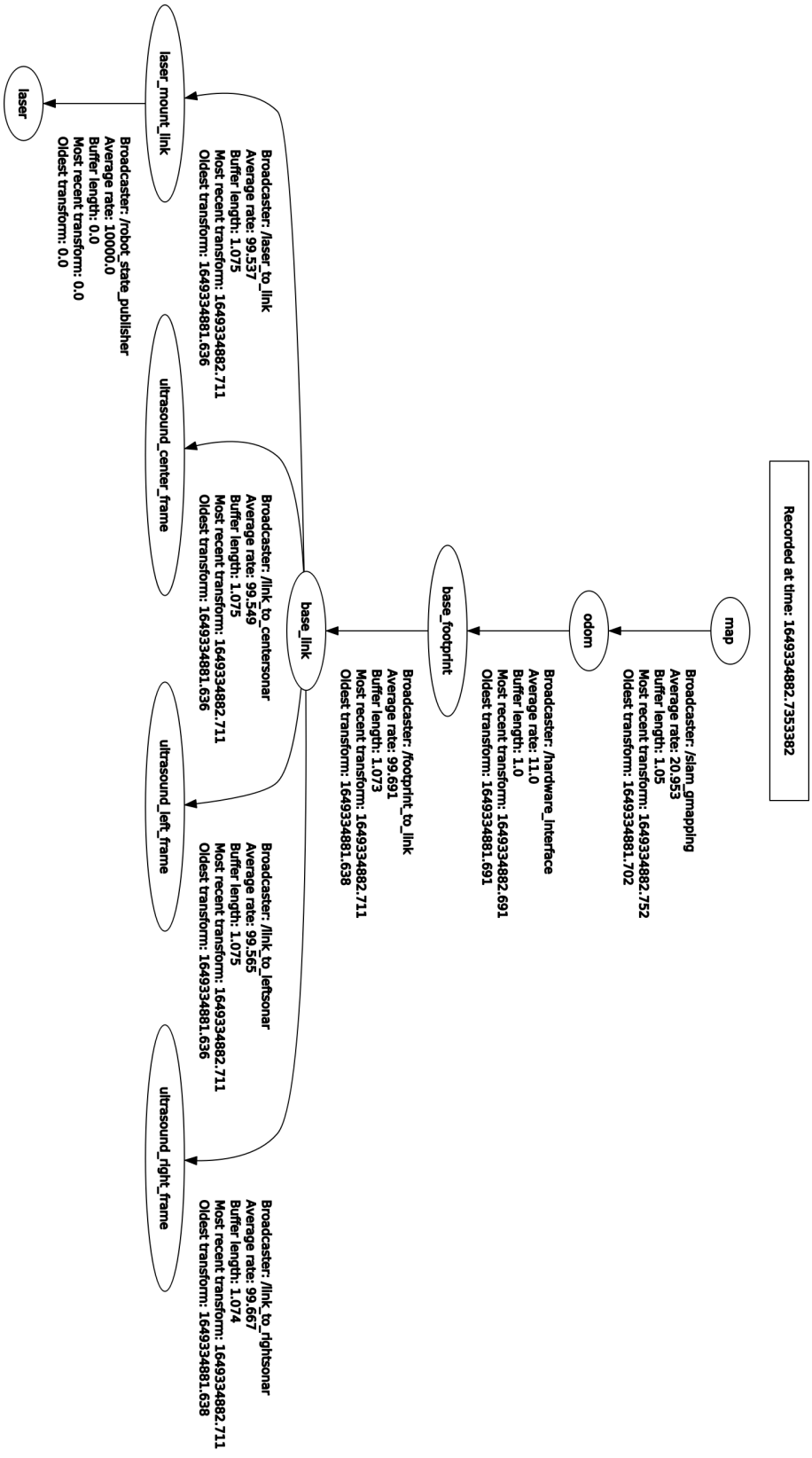


Figure A.2: The TF tree of the ROS network, containing all the frames

A.4 Step-by-step approach to using the basics of the I-do platform

Note: Skip the setup if you want use the command prompt on the local ROS computer on the robot (not recommended).

Note: It is possible to SSH into the local ROS computer using a computer that runs on Windows (using Putty), however this makes it a lot more difficult to use the graphical interface rviz (not recommended).

A.4.1 Setup

1. Download and install Ubuntu as your (second) operating system, the dual booting procedure is described here: https://itsfoss.com/install-ubuntu-1404-dual-boot-mode-windows-8-81-uefi/.

2. Download and install ROS (noetic) on your laptop using the steps described here (this is to run rviz on your laptop): http://wiki.ros.org/noetic/Installation/Ubuntu.

3. Connect the loose power cables on the I-do, this starts up the system.

4. Connect your laptop or computer to the robot with an Ethernet cable, the Ethernet port on the robot is located on the backside, close to the floor.

5. Set up a static IP address for your laptop by following this guide (scroll down to the desktop variant guide): https://pimylifeup.com/ubuntu-20-04-static-ip-address/. The required static IP address is "192.168.0.2", the required gateway is "192.168.0.1" and the netmask is "255.255.255.0".

6. Start an SSH connection with the robot, you do this with the following command: "ssh eugeen@192.168.0.5". The password is "eugeen". This shell prompt is now accessing the ROS computer on the robot.

7. In order to run rviz on your own laptop, the ROS network needs to be setup properly. To do that you will have to start a second (new) shell prompt and run the following two commands (you need to do this every time you start a new shell prompt): "export ROS_MASTER_URI=http://192.168.0.5:11311" and "export ROS_HOSTNAME=192.168.0.2". You could also add these two commands to the /.bashrc file (command: "gedit /.bashrc"), this executes it automatically every time you open a new shell prompt.

A.4.2 Launching ROS nodes

You can launch the required ROS nodes by using the command "roslaunch", with this command we can run launch scripts, which essentially launches all the required nodes at once. All the commands that you want to execute on the robot (like the command right below), you'll have to execute in the shell prompt which runs SSH.

1. Execute the follow command to launch the required ROS nodes for the robot to function: "roslaunch launch_ido full.launch".

Optional If you have a monitor connected to the ROS computer, you can also run: "roslaunch launch_ido full_rviz.launch". This command launches the same nodes as the first command, but also brings up the preconfigured rviz environment on the ROS computer.

A.4.3 Using rviz to control the robot on your own laptop

1. After running the above mentioned commands, you are able to launch rviz on your own computer. You do this by executing the following command: "rviz" in the shell prompt that you did not use for SSH!

Optional Download the config file at the link supplied by the project supervisor.

Optional In the rviz environment, go to file -> Open Config and load the supplied file. This loads a preconfigured rviz environment.

2. You can send a goal to the robot by first clicking on the "2D Nav Goal" button followed by a click on the position of the map you want the robot to move to.

A.4.4 Controlling the robot with a PlayStation 3 controller

In order to drive the robot around manually, you can send velocity commands directly to the robot base with a PlayStation 3 controller.

1. First connect the PlayStation 3 controller to the ROS computer with a USB cable.
2. Press the PlayStation logo, now the player 1 light should light up.
3. In the SSH shell, run: "roslaunch launch_ido teleop.launch". The robot should now respond to the controller. You can drive forwards and backwards with the left stick, and turn the robot with the right stick.

A.5 Arduino code

```
#include <ros.h>
#include <std_msgs/Float32.h>
#include <sensor_msgs/Range.h>
#include <NewPing.h>
#include <SimpleKalmanFilter.h>
#include <util/atomic.h> // For the ATOMIC_BLOCK macro
#include <FastLED.h>

// Pins right motor
#define PWM_R 6 // orange   - PWM out
#define DIR_R 22 // Yellow  - DIGI out
#define ENA_R 40 // Green    - DIGI in
#define ENB_R 2 // White    - Interrupt pin

// Pins left motor
#define PWM_L 7   // Orange  - PWM out
#define DIR_L 30 // Blue     - DIGI out
#define ENA_L 34 // Green    - DIGI in
#define ENB_L 3   // White         - Interrupt pin

// Pins Sonar
#define leftSonarPin 45 // Orange
#define centerSonarPin 47 // Yellow
#define rightSonarPin 43 // White

// Pin LED strip
#define DATA_LED 50 // Green
```

```
// Pins Bumper
#define leftBumper A0
#define frontBumper A1
#define rightBumper A2

// Sonar constants
#define SONAR_NUM 3                 //The number of sensors.
#define MAX_DISTANCE 100            //Mad distance to detect
    obstacles.
#define PING_INTERVAL 33            //Looping the pings after 33
    microseconds.
unsigned int cm[SONAR_NUM];         // Variable where the ping
    distances are stored.
unsigned int kal[SONAR_NUM];        // Variable where the filtered
    ping distances are stored
unsigned long _timerStart = 0;      // Global used for setting the
    starting time of the timer
int LOOPING = 40;                   // Constant that determines how
     often the ultrasound sensors measure the distance

// Led strip globals and constants
#define NUM_LEDS 193
CRGB leds[NUM_LEDS];
int LED_nr = 0;
uint8_t hue = 0;       // The color that the led strip is displaying


#define bumperThreshold 10  // Threshold value used for the
    colission detection (lower -> more sensetive detection)



ros::NodeHandle nh;         // ROS node handler definition

// Messages used for the angle feedback
std_msgs::Float32 msgs_pubL;
std_msgs::Float32 msgs_pubR;

// Message definitions used for the ultrasound data
sensor_msgs::Range range_left;
sensor_msgs::Range range_center;
sensor_msgs::Range range_right;


// Globals

bool allowedToDrive = true;         // The boolean that sets
    whether the robot is allowed to drive
```

```
 // Defining the ultrasound sensors as an array
NewPing sonar[SONAR_NUM] = {
  NewPing(leftSonarPin, leftSonarPin, MAX_DISTANCE), // Trigger pin
      , echo pin, and max distance to ping.
  NewPing(centerSonarPin, centerSonarPin, MAX_DISTANCE),
  NewPing(rightSonarPin, rightSonarPin, MAX_DISTANCE)
};

// Defining the corresponding filters (for the ultrasound) as an
    array
SimpleKalmanFilter kalfilt[SONAR_NUM] = {
SimpleKalmanFilter (2, 2, 0.01),
SimpleKalmanFilter (2, 2, 0.01),
SimpleKalmanFilter (2, 2, 0.01)
};

// Motor control:
long prevT = 0;        // Variable used for determining the time
    difference for the differentiating

// Left:
volatile double pos_i_L = 0;       // Variable used to store the
    wheel position in ticks
long posPrev_L = 0;                // Variable used for storing the
    previous position for the velocity calculation
float eintegral_L = 0;             // Variable used for integrating
    the error for the PI controller

float radsFilt_L = 0;              // Variable for storing the
    filtered velocity in rad/s
float radsPrev_L = 0;              // Variable for storing the
    previous velocity, used in the filtering

float target_L = 0;                // The reference velocity for the
     PI controller


// Right:
volatile double pos_i_R = 0;       // Variable used to store the
    wheel position in ticks
long posPrev_R = 0;                // Variable used for storing the
    previous position for the velocity calculation
float eintegral_R = 0;             // Variable used for integrating
    the error for the PI controller

float radsFilt_R = 0;              // Variable for storing the
    filtered velocity in rad/s
float radsPrev_R = 0;              // Variable for storing the
    previous velocity, used in the filtering
```

```
float target_R = 0;                 // The reference velocity for the
    PI controller

// Callback function to set the new reference velocity
void setVelL(const std_msgs::Float32& left_wheel_vel) {
  target_L = left_wheel_vel.data;
}

// Callback function to set the new reference velocity
void setVelR(const std_msgs::Float32& right_wheel_vel) {
  target_R = right_wheel_vel.data;
}

// Create the subscribers and publishers objects
ros::Subscriber <std_msgs::Float32> subL("/my_robot/left_wheel_vel"
    , &setVelL);
ros::Subscriber <std_msgs::Float32> subR("/my_robot/right_wheel_vel
    ", &setVelR);

ros::Publisher pubL("my_robot/left_wheel_angle", &msgs_pubL);
ros::Publisher pubR("my_robot/right_wheel_angle", &msgs_pubR);

ros::Publisher pub_range_left("/ultrasound_left", &range_left);
ros::Publisher pub_range_center("/ultrasound_center", &range_center
    );
ros::Publisher pub_range_right("/ultrasound_right", &range_right);



void setup() {
  Serial.begin(9600);

  TCCR4B = TCCR4B & B11111000 | B00000001;    // Set the PWM
      frequency to 31372.55 Hz, which is outside the audible range

  // Intitialize the ROS node and subscribe/advertise the required
      topics
  nh.initNode();
  nh.subscribe(subL);
  nh.subscribe(subR);
  nh.advertise(pubL);
  nh.advertise(pubR);
  nh.advertise(pub_range_left);
  nh.advertise(pub_range_center);
  nh.advertise(pub_range_right);

  // Setup the pins for the left motor
  pinMode(DIR_L, OUTPUT);
  pinMode(ENA_L, INPUT);
  pinMode(ENB_L, INPUT);
```

```
    pinMode(PWM_L, OUTPUT);
    attachInterrupt(digitalPinToInterrupt(ENB_L), readEncoder_L,
        RISING);      // Attach the callback function to the interrupt
         pin

    // Setup the pins for the right motor
    pinMode(DIR_R, OUTPUT);
    pinMode(ENA_R, INPUT);
    pinMode(ENB_R, INPUT);
    pinMode(PWM_R, OUTPUT);
    attachInterrupt(digitalPinToInterrupt(ENB_R), readEncoder_R,
        RISING);      // Attach the callback function to the interrupt
         pin

    // Initialize the messages for the ultrasound sensors
    sensor_msg_init(range_left, "ultrasound_left_frame");
    sensor_msg_init(range_center, "ultrasound_center_frame");
    sensor_msg_init(range_right, "ultrasound_right_frame");

    // Initialize the LED strip
    FastLED.addLeds<WS2812,DATA_LED,RGB>(leds,NUM_LEDS);
    FastLED.setBrightness(84);
}

void loop() {

    double pos_L = 0;
    double pos_R = 0;
    ATOMIC_BLOCK(ATOMIC_RESTORESTATE) {   // To avoid any misreads,
        the positions are read in an atomic block
      pos_L = pos_i_L;
      pos_R = pos_i_R;
    }

    // Compute the time delay for the loop, this is used for the
        differentation
    long currT = micros();
    float deltaT = ((float) (currT - prevT)) / 1.0e6;

    // Compute the left wheel velocity
    float velocity_L = (pos_L - posPrev_L) / deltaT;
    posPrev_L = pos_L;

    // Compute the right wheel velocity
    float velocity_R = (pos_R - posPrev_R) / deltaT;
    posPrev_R = pos_R;

    prevT = currT;


    // Convert count/s to rad/s
```

```
  float rads_L = velocity_L * 0.000427;
  float rads_R = velocity_R * 0.000427;

  // Low-pass filter left wheel velocity (25 Hz cutoff)
  radsFilt_L = 0.854 * radsFilt_L + 0.0728 * rads_L + 0.0728 *
      radsPrev_L;
  radsPrev_L = rads_L;

  // Low-pass filter right wheel velocity (25 Hz cutoff)
  radsFilt_R = 0.854 * radsFilt_R + 0.0728 * rads_R + 0.0728 *
      radsPrev_R;
  radsPrev_R = rads_R;


// PI controller parameters
  float kp = 57.3;     // Proportional gain
  float ki = 62.1;     // Integration gain


  float e_L = target_L - radsFilt_L;          // Calculate the
      error
  eintegral_L = eintegral_L + e_L * deltaT;   // Integrate the
      error
  float u_L = kp * e_L + ki * eintegral_L;    // Calculate the
      control signal

  float e_R = target_R - radsFilt_R;          // Calculate the
      error
  eintegral_R = eintegral_R + e_R * deltaT;   // Integrate the
      error
  float u_R = kp * e_R + ki * eintegral_R;    // Calculate the
      control signal


// Limit the control signal to 255 (max dutycycle)
  int pwr_L = (int) fabs(u_L);
  if (pwr_L > 255) { //255
    pwr_L = 255;
  }

// Limit the control signal to 255 (max dutycycle)
  int pwr_R = (int) fabs(u_R);
  if (pwr_R > 255) { //255
    pwr_R = 255;
  }

// If the robot needs to stop, cut the motors off (this prevents
    small oscillations)
     if (target_L == 0 && target_R == 0) {
  pwr_L = 0;
  pwr_R = 0;
```

```
    }

  // Send the motor command if the robot is allowed to drive
  if (allowedToDrive) {
    setMotor(readDir(u_L), DIR_L, pwr_L, PWM_L);
    setMotor(readDir(u_R), DIR_R, pwr_R, PWM_R);
  }
  else {  // Cut of the power if it is not allowed to drive
    setMotor(readDir(u_L), DIR_L, 0, PWM_L);
    setMotor(readDir(u_R), DIR_R, 0, PWM_R);
  }

  //  Calculate the wheel position in radians (from ticks)
    msgs_pubL.data = pos_L / 2342;
    msgs_pubR.data = pos_R / 2342;

  //  Publish the wheel position data
    pubL.publish(&msgs_pubL);
    pubR.publish(&msgs_pubR);


    sonarLoop();      // Run the sonar loop
    checkBump();      // Check whether the bumper sensors have exceeded
        the threshold
    nh.spinOnce();   // Update the ROS network

  //  If the LED counter is at the last LED, go back to the first
    if (LED_nr > NUM_LEDS-1) {
      LED_nr = 0;
    }
    leds[LED_nr++] = CHSV(hue, 255, 255); // Set the LED color and
        brightness

    FastLED.show();                       // Show the leds
    fadeall();                            // Fade every LED, this
        creates the fading tail
  }

  // Determine the required direction of movement
  int readDir (int u) {
    if (u < 0) {
      return 0;
    }
    else {
      return 1;
    }
  }

  // Set the correct PWM dutycycle and direction pin
  void setMotor(int dir, int dirPort, int pwmVal, int pwmPort) {
```

```
  digitalWrite(dirPort, dir);      // Set the direction on the motor
      driver
  analogWrite(pwmPort, pwmVal);    // Adjust the dutycycle
}

// Callback function for the left wheel
void readEncoder_L() {
  if (digitalRead(ENA_L) == LOW) {   // Determine the movement
      direction of the motor shaft
    pos_i_L++;
  }
  else {
    pos_i_L--;
  }
}

// Callback function for the right wheel
void readEncoder_R() {
  if (digitalRead(ENA_R) == LOW) {   // Determine the movement
      direction of the motor shaft
    pos_i_R++;
  }
  else {
    pos_i_R--;
  }
}


//looping through the ultrasound sensors
void sensorCycle() {
  for (uint8_t i = 0; i < SONAR_NUM; i++) {
    cm[i] = sonar[i].ping_cm();                  // Read the
        distance in cm
    kal[i] = kalfilt[i].updateEstimate(cm[i]);   // Filter the data
    if (cm[i] < 10) {                            // If the reading
        is too close, set the output as maximum, this prevents the
        costmap from putting false objects on the robots position
      kal[i] = MAX_DISTANCE;
    }
  }
}

// Store the current time
void startTimer() {
  _timerStart = millis();
}

// Check whether a certain interval has passed
bool isTimeForLoop(int _mSec) {
  return (millis() - _timerStart) > _mSec;
}
```

```
// The loop that handles the ultrasound sensors
void sonarLoop() {
    if (isTimeForLoop(LOOPING)) {
    sensorCycle();                          // Loop through the
        sensors
    startTimer();                           // Start the timer

    // save the data into the message
    range_left.range   = (float) kal[0] / 100;
    range_center.range = (float) kal[1] / 100;
    range_right.range  = (float) kal[2] / 100;

    // Implement the time
    range_left.header.stamp = nh.now();
    range_center.header.stamp = nh.now();
    range_right.header.stamp = nh.now();

    // Publish the data
    pub_range_left.publish(&range_left);
    pub_range_center.publish(&range_center);
    pub_range_right.publish(&range_right);
  }
}

// Fill the sensor message
void sensor_msg_init(sensor_msgs::Range &range_name, char *
    frame_id_name)
{
  range_name.radiation_type = sensor_msgs::Range::ULTRASOUND;
  range_name.header.frame_id = frame_id_name;
  range_name.field_of_view = 0.1;
  range_name.min_range = 0.1;
  range_name.max_range = 1.0;
}

// Fade every LED
void fadeall() {
  for(int i = 0; i < NUM_LEDS; i++) {
    leds[i].nscale8(252);
   }
}

// Check whether one of the sensors has exceeded the threshold
    value
void checkBump() {
  if (max(analogRead(leftBumper), max(analogRead(frontBumper),
      analogRead(rightBumper))) > bumperThreshold && allowedToDrive
      ) {
    emergencyStop();
  }
}
```

```
// Perform an emergency stop
void emergencyStop () {
  allowedToDrive = false;          // Prevent any motor control
      commands from being send
  hue = 96;                        // Set the LED color to red

// Set all the LEDs immediately to the red color
  for(int i = 0; i < NUM_LEDS; i++) {
    leds[i] = CHSV(hue, 255, 100);
    // Show the leds
    FastLED.show();                // Update the LEDs
}
}
```

A.6 Hardware interface code

```
#include <ido_interface/MyRobot_hardware_interface.h>

MyRobot::MyRobot(ros::NodeHandle& nh) : nh_(nh) {


// Declare all JointHandles, JointInterfaces and
   JointLimitInterfaces of the robot.
    init();

// Create the controller manager
    controller_manager_.reset(new controller_manager::
        ControllerManager(this, nh_));

//Set the frequency of the control loop.
    loop_hz_=10;
    ros::Duration update_freq = ros::Duration(1.0/loop_hz_);

//Run the control loop
    my_control_loop_ = nh_.createTimer(update_freq, &MyRobot::
        update, this);


}


MyRobot::~MyRobot() {
}

void MyRobot::init() {

// Create joint_state_interface for JointA (left wheel)
    hardware_interface::JointStateHandle jointStateHandleA("
        wheel_left_joint", &joint_position_[0], &joint_velocity_
        [0], &joint_effort_[0]);
```

```
    joint_state_interface_.registerHandle(jointStateHandleA);
// Create effort joint interface as JointA accepts effort command.
    hardware_interface::JointHandle jointVelocityHandleA(
        jointStateHandleA, &joint_velocity_command_[0]);
    velocity_joint_interface_.registerHandle(jointVelocityHandleA);


// Create joint_state_interface for JointB (right wheel)
    hardware_interface::JointStateHandle jointStateHandleB("
        wheel_right_joint", &joint_position_[1], &joint_velocity_
        [1], &joint_effort_[1]);
    joint_state_interface_.registerHandle(jointStateHandleB);
// Create effort joint interface as JointB accepts effort command.
    hardware_interface::JointHandle jointVelocityHandleB(
        jointStateHandleB, &joint_velocity_command_[1]);
    velocity_joint_interface_.registerHandle(jointVelocityHandleB);



// Register all joints interfaces
    registerInterface(&joint_state_interface_);
    registerInterface(&velocity_joint_interface_);

    // Advertise the publishers
    left_wheel_vel_pub_ = nh_.advertise<std_msgs::Float32>("
        my_robot/left_wheel_vel", 1);
    right_wheel_vel_pub_ = nh_.advertise<std_msgs::Float32>("
        my_robot/right_wheel_vel", 1);

    // Subscribe to the required topics
    left_wheel_angle_sub_ = nh_.subscribe("my_robot/
        left_wheel_angle", 1, &MyRobot::leftWheelAngleCallback,
        this);
    right_wheel_angle_sub_ = nh_.subscribe("my_robot/
        right_wheel_angle", 1, &MyRobot::rightWheelAngleCallback,
        this);
}


//This is the control loop
void MyRobot::update(const ros::TimerEvent& e) {
    elapsed_time_ = ros::Duration(e.current_real - e.last_real);
    read();
    controller_manager_->update(ros::Time::now(), elapsed_time_);
    write(elapsed_time_);
}

// This function gets called to input data into the controller
void MyRobot::read() {

  joint_position_[0] = _wheel_vel[0];
```

```
    joint_position_[1] = _wheel_vel[1];

}

// This function gets called to output data from the controller
void MyRobot::write(ros::Duration elapsed_time)
{
        //Create a new message and fill it with the data, then
            publish it
        std_msgs::Float32 left_wheel_vel_msg;
        std_msgs::Float32 right_wheel_vel_msg;
        left_wheel_vel_msg.data = joint_velocity_command_[0];
        right_wheel_vel_msg.data = joint_velocity_command_[1];
        left_wheel_vel_pub_.publish(left_wheel_vel_msg);
        right_wheel_vel_pub_.publish(right_wheel_vel_msg);

}


int main(int argc, char** argv)
{

    //Initialze the ROS node.
    ros::init(argc, argv, "MyRobot_hardware_interface_node");
    ros::NodeHandle nh;

    // Create the object of the robot hardware_interface class and
        spin the thread.
    MyRobot ROBOT(nh);

    //Separate Spinner thread for the Non-Real time callbacks such
        as service callbacks to load controllers
    ros::MultiThreadedSpinner spinner(0);
    spinner.spin();

    return 0;
}
```

A.7 Launch file code (full.launch)

```
<launch>

    <!-- Start the rosserial node, with the correct settings. If
        there is an error, check whether the board is connected at
        /dev/ttyACM0 -->
    <node name="serial_node"        pkg="rosserial_python"
        type="serial_node.py">
    <param name="port"              type="string"
        value="/dev/ttyACM0"/>
    <param name="baud"              type="int"
        value="57600"/>
```

```
    </node>

    <!-- Start the hardwrae interface -->
    <node name="hardware_interface"         pkg="ido_interface"
          type="MyHardware_interface">
    </node>

    <!-- Run all the required launch files -->
    <include file="$(find␣ido_interface)/launch/MyInterface.launch"
        />

    <include file="$(find␣sick_tim)/launch/sick_tim571_2050101.
       launch" />

    <include file="$(find␣gmapping)/launch/ido_gmapping.launch" />


    <include file="$(find␣my_2d_nav)/launch/move_base.launch" />

    <!-- Start all the static TF publishers -->
    <node pkg="tf" type="static_transform_publisher" name="
       footprint_to_link" args="0␣0␣0.08␣0␣0␣0␣base_footprint␣
       base_link␣10" />
    <node pkg="tf" type="static_transform_publisher" name="
       laser_to_link" args="0␣0␣0.23␣0␣0␣3.14159␣base_link␣
       laser_mount_link␣10" />

    <node pkg="tf" type="static_transform_publisher" name="
       link_to_leftsonar" args="0.11␣0.05␣1.04␣0.21␣0.785␣0␣
       base_link␣ultrasound_left_frame␣10" />
    <node pkg="tf" type="static_transform_publisher" name="
       link_to_centersonar" args="0.11␣0␣1.04␣0␣0␣0␣base_link␣
       ultrasound_center_frame␣10" />
    <node pkg="tf" type="static_transform_publisher" name="
       link_to_rightsonar" args="0.11␣-0.05␣1.04␣-0.21␣0.785␣0␣
       base_link␣ultrasound_right_frame␣10" />

</launch>
```

Bibliography

[1] Ros wiki tutorials. http://wiki.ros.org/ROS/Tutorials.

[2] Savioke relay. https://www.savioke.com/relay-plus.

[3] Website kuka. https://www.kuka.com/en-de/company/about-kuka.

[4] Motion planning under socially normative constraints. http://www.spencer.eu/deliverables/d5_3.pdf, 2014.

[5] Sachin Chitta, Eitan Marder-Eppstein, Wim Meeussen, Vijay Pradeep, Adolfo Rodríguez Tsouroukdissian, Jonathan Bohren, David Coleman, Bence Magyar, Gennaro Raiola, Mathias Lüdtke, and Enrique Fernández Perdomo. ros_control: A generic and simple control framework for ros. *The Journal of Open Source Software*, 2017.

[6] N. Murphy K Doucet, A. de Freitas and S Russel. *Rao-Blackwellised Particle Filtering for Dynamic Bayesian Networks.* 2000.

[7] Erico Guizzo Evan Ackerman. Wizards of ros: Willow garage and the making of the robot operating system how a small band of silicon valley engineers started a global robotics revolution. https://spectrum.ieee.org/wizards-of-ros-willow-garage-and-the-making-of-the-robot-operating-system, 2017.

[8] Manuel Gräber. Practical pid tuning guide. https://tlk-energy.de/blog-en/practical-pid-tuning-guide, 2021.

[9] Kiam Heong Ang, Gregory Chong, and Yun Li. Pid control system analysis, design, and technology. *IEEE Transactions on Control Systems Technology*, 2007.

[10] Bence Magyar. Ros package: diff_drive_controller. http://wiki.ros.org/diff_drive_controller.

[11] Eitan Marder-Eppstein. Ros package: move_base. http://wiki.ros.org/move_base.

[12] Pololu. Pololu High-Power Motor Driver 36v20 CS. https://www.pololu.com/product/1457/specs.

[13] SICK. Sick tim571 working range diagram. https://www.sick.com/be/en/detection-and-ranging-solutions/2d-lidar-sensors/tim5xx/tim571-2050101/p/p412444.

[14] SICK. SICK TiM571 datasheet. https://cdn.sick.com/media/pdf/4/44/444/dataSheet_TiM571-2050101_1075091_en.pdf, 2020.

[15] SICK products. SOPAS. https://www.sick.com/nl/nl/sopas-engineering-tool/p/p367244",.

[16] Super B. Andrena 12V15aH. https://s3.eu-central-1.amazonaws.com/superb-com/sulu/uploads/media/08/datasheet-andrena_12v15ah_v1-1.pdf.

[17] Super B. Andrena 12V25aH-SC. https://s3.eu-central-1.amazonaws.com/superb-com/sulu/uploads/media/07/datasheet-mason-12v25ah_v1-1.pdf.

[18] Valence. U-Charge U1-24RT. https://www.rdbatteries.com/upload/729/U1-24RT-Datasheet-Aug-2015.pdf, 2015.