Hochschule für Angewandte Wissenschaften Hamburg
Hamburg University of Applied Sciences

Hamburg University of Applied Sciences

5th semester project

MS NHAWIGATORA

Burau, Henri
henri.burau@haw-hamburg.de

Mang, Maximilian
maximilian.mang@haw-hamburg.de

Naumann, Felix
felix.naumann@haw-hamburg.de

Schnirpel, Thorben
thorben.schnirpel@haw-hamburg.de

Stark, Franek
franek.stark@haw-hamburg.de

supervised by
Prof. Dr. Tim Tiedemann
Prof. Dr. Stephan Pareigis

April 1, 2019

Abstract

The main target of this project, was to build a autonomous boat which can navigate through a simultaneously created map or a given one. The development of this boat was made possible through a cooperation with the Miniatur Wunderland Hamburg. A Framework used entirely throughout our project is the Robot Operating System (ROS). It offers Support for all sensors and actors used in this project and enables a easy method for remote logging and distributed processing. We created a modular, scalable and functional base for a completely autonomous ship.

Contents

1 Introduction **3**

2 Sensors & Actors **4**
2.1 Sensors . . . 5
2.1.1 Lidar . . . 5
2.1.2 Short Distance Sensors . . . 5
2.1.3 Camera . . . 5
2.1.4 Internal Measurement Unit . . . 6
2.2 Actors . . . 7
2.2.1 The PWM Controller . . . 7

3 Ship Platform **8**
3.1 Ship Architecture . . . 8
3.1.1 Hull . . . 8
3.1.2 Intermediate Decks . . . 8
3.1.3 3D Prints . . . 9
3.2 Electrical Components . . . 9
3.2.1 The Motor Driver . . . 9
3.2.2 The Power wiring . . . 10

4 Robot Operating System **12**
4.1 About ROS . . . 12
4.2 Installation . . . 12
4.3 ROS Tools . . . 13
4.3.1 Recording . . . 13
4.3.2 Distributed Processing . . . 13
4.4 Node-Graph . . . 14
4.4.1 Rangefinder-Node . . . 15
4.4.2 Lidar-Node . . . 15
4.4.3 IMU-Node . . . 16
4.4.4 PylonCamera-Node . . . 16
4.4.5 Controller-Node . . . 16

4.4.6 CollisionWarning-Node . . . 17
4.4.7 Marker-Node . . . 19
4.4.8 MotorController-Node . . . 19
4.5 Launch Configurations . . . 20

5 Controlling **21**
5.1 Xbox-Controller . . . 21
5.1.1 Driver . . . 21
5.1.2 Bluetooth Pairing and Connecting . . . 21
5.1.3 Ros-Usage . . . 22
5.1.4 Use the **joy_node** . . . 22

6 Visualisation **23**
6.1 TF-Frames . . . 23

7 Navigation **25**
7.1 Hector Slam . . . 25
7.1.1 Tests at the University . . . 25
7.1.2 At the Miniaturwunderland Hamburg . . . 26
7.2 AMCL . . . 26

8 Future Outlook **28**

9 Images **29**

Chapter 1

Introduction

The project is part of our study in technical computer science at the University of Applied Sciences in Hamburg. As part of our fifth semester project, our plan was to create an autonomous vehicle for the Miniatur Wunderland in Hamburg. Since our team thought that building an autonomous ship would be very interesting and challenging we agreed on facing this challenge.

First of all we started looking for a framework, which would support everything we planned on doing. Soon we stumbled upon the Robot Operating System, which is a very powerful framework that supported distributed progressing, visualization of our data, SLAM and many other features that were required for autonomy.

Chapter 2

Sensors & Actors

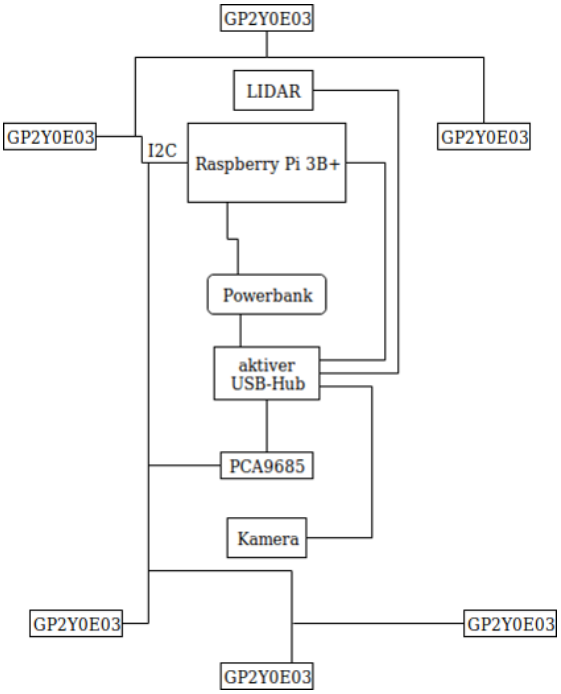

Figure 2.1: Hardware setup

The diagram in figure 2.1 shows the control system. The control system is powered by an active USB Hub. The power bank has two USB ports. The first is connected to the Raspberry Pi and the other is the power source of the active USB hub. The upstream port of the USB hub is connected to the Raspberry Pi. The downstream ports are connected to the LIDAR and the camera. One downstream port is used to power the PCA9685.

2.1 Sensors

2.1.1 Lidar

As the main component to sense our environment we chose the 2D lidar (light detection and ranging) device URG-04LX-UG01 from Hokuyo. It scans its surroundings with 10 Hz and a resolution of 0.36 degrees with a total scan angle of 240 degrees. Its maximum distance is up to 4 meters (also depending on object sizes).

Mapping Currently we are able to generate maps with the lidar as a sensor alone. Normally this would not be possible without proper odometry. But we faked it by using a ros node which takes lidar scan data to generate crude odometry.

Power Consumption Since the lidar is powered by 5V USB supply voltage it could be powered directly plugged in to the raspberry pi. But because of its power consumtion of up to 800mA this will lead to power consumption dips resulting in reduced availability of the raspberry pi. We solved this problem by using a powered USB-Hub.

Figure 2.2: URG-04LX-UG01 Lidar

2.1.2 Short Distance Sensors

In order to get the distance to objects which are not covered by the lidar, we chose GP2Y0E03 infrared distance sensors. Those sensors can measure ranges between 4 to 50 centimeters with an maximum angle of 5 Degrees. Currently we have six sensors installed, two on each sides and one to the front and the back. Although we installed the sensors only to assist the lidar the idea came up that we might be able to replace the lidar by put more sensors on board. This way they we could have enough distances to locate the boat in a future map.

2.1.3 Camera

The camera we chose and which was kindly sponsored by Basler, is a daA1920-30uc. It offers a full-HD resolution while delivering 30 frames per second. We decided to use the color

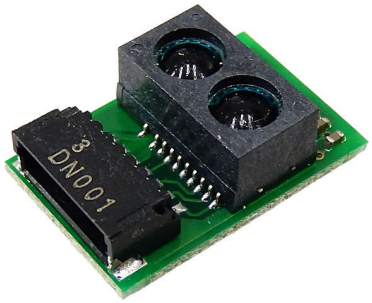

Figure 2.3: GP2Y0E03 Infrared distance sensor

version of this camera in order to detect potentially colored landmarks in the surrounding terrain. It is connected to our system via USB. To get as much terrain as possible onto the sensor we chose a wide angled lens. It as an opening angle of 98 degrees. By using the official Basler driver provided at the Basler website[1]. On the ROS side, you can simply use the pylon_camera package found on the ROS website[2]

Figure 2.4: Camera and objective

2.1.4 Internal Measurement Unit

For measuring the boats orientation we chose the MPU9255 which is a 10 degrees of freedom Internal Measurement Unit (IMU). It combines a gyroscope, an accelerometer and a magnetometer.

Problems Due to its cheap build quality its capable of measuring rotations at x- and y-Axis but only very inaccurate at the z-Axis. Another factor is that the magnetometer (which is mainly responsible for the z-axis) is very close to magnetic motors which power the boats thrusters and also the lidar. This is why the z-Axis rotation from the IMU is

[1] **www.baslerweb.com**
[2] **http://wiki.ros.org/pylon_camera**

currently unusable. A solution could be to move the IMU further away from magnetic parts or filter and calibrate it better on the software side.

2.2 Actors

2.2.1 The PWM Controller

The PCA9685 is a 16-channel PWM controller. It is used to control the Motor Driver and the Servo. The Board is connected to the Raspberry via an I2C-bus. Have a look at the Reference[3] on how to change the I2C-Address.

Motor Drivers

The data sheet provides the following information:

$$PWM - Period = 20ms(\text{Could vary})$$

Highlevels:

$$\begin{aligned} FullForward &= 1,1ms \\ (Stop)Neutral &= 1,5ms \\ FullReverse &= 1,8ms \end{aligned}$$

Highlevels between above values will result in nuances between Full and Stop.

Servo

For the Servo we can used the standard PWM-Values as shown in figure 2.5.

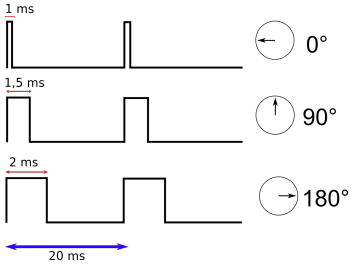

Figure 2.5: Tiempos servo

[3] **https://cdn-learn.adafruit.com/downloads/pdf/16-channel-pwm-servo-driver.pdf**

Chapter 3

Ship Platform

3.1 Ship Architecture

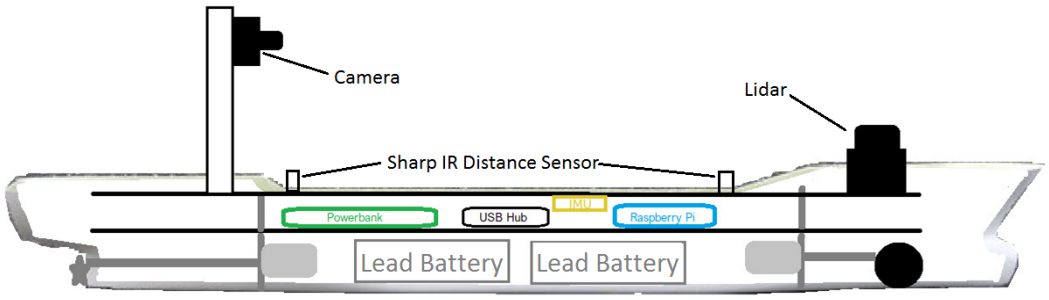

Figure 3.1: Ship architecture

3.1.1 Hull

The hull of the ship was created by Steinhagen Modelltechnik in Kiel. It is made of fiberglass, which makes it very stable. The dimensions are 98 centimeters in length, 13.7 cm in width and 12 cm in height. To match our requirements in the Miniatur Wunderland the ship is scaled 1:100 which is close to the ideal 1:87.

3.1.2 Intermediate Decks

The ship has two intermediate decks in order to create a modular design, which can be disassembled quickly. These where created from a plastic material with a metal core. It is specially designed to leave about 0.5 cm room to the sides of the hull, to potentially allow cables to be mounted there. They intermediate decks are mounted to 90 degree brackets, which are glued to the hull.

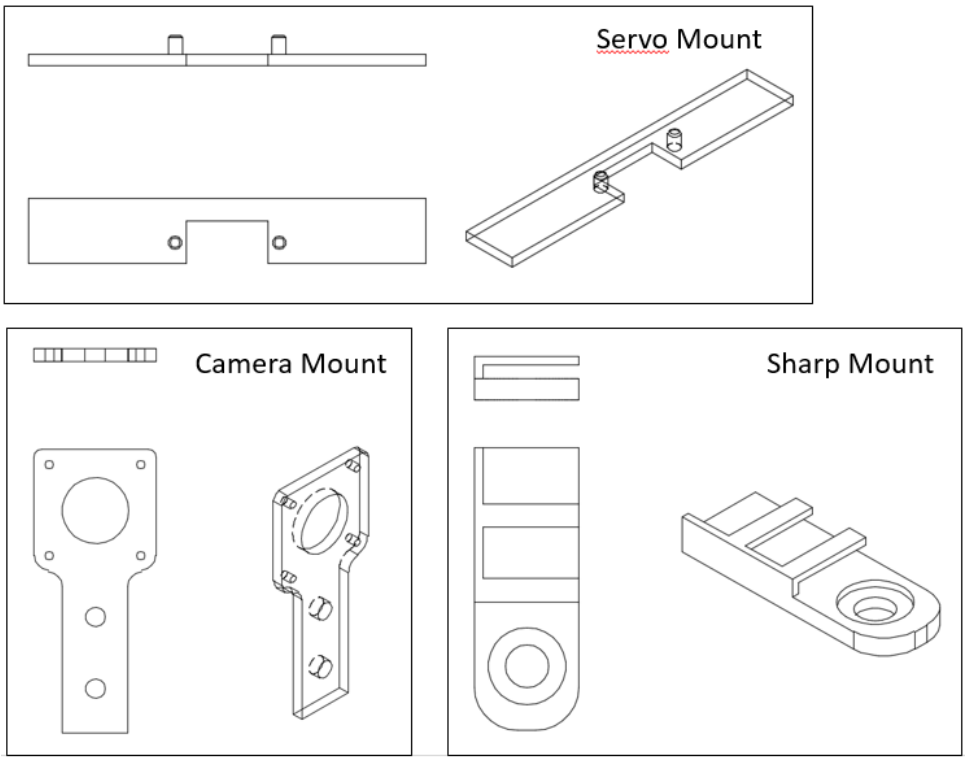

Figure 3.2: 3D printed sensor mounts

3.1.3 3D Prints

To mount actors and sensors to the ship we occasionally had to 3D-printed mounts. This was the case for the short distance sharp sensors, the Basler camera and the servo mount. All models were designed using Fusion360.

3.2 Electrical Components

The main computing unit of this project is a Raspberry Pi, which is a low cost mini computer. To control the motors for the thruster and the propeller we have used a PCA9685 PWM Controller. It offers 16 PWM channels which can all be interfaced via I2C directly from the Raspberry Pi.

3.2.1 The Motor Driver

We have used a "Graupner Navy V15R". It is a special motor driver for wet environment. The load capacity is 15 A and the weight is 48g. The controllers are powered with 12V. They get their control signals via PWM.

3.2.2 The Power wiring

Red wires are used for VCC connection and black wires are used for ground. We have two separate power systems. The engine power system is 6V based and is powered by two parallel 6V battery's.

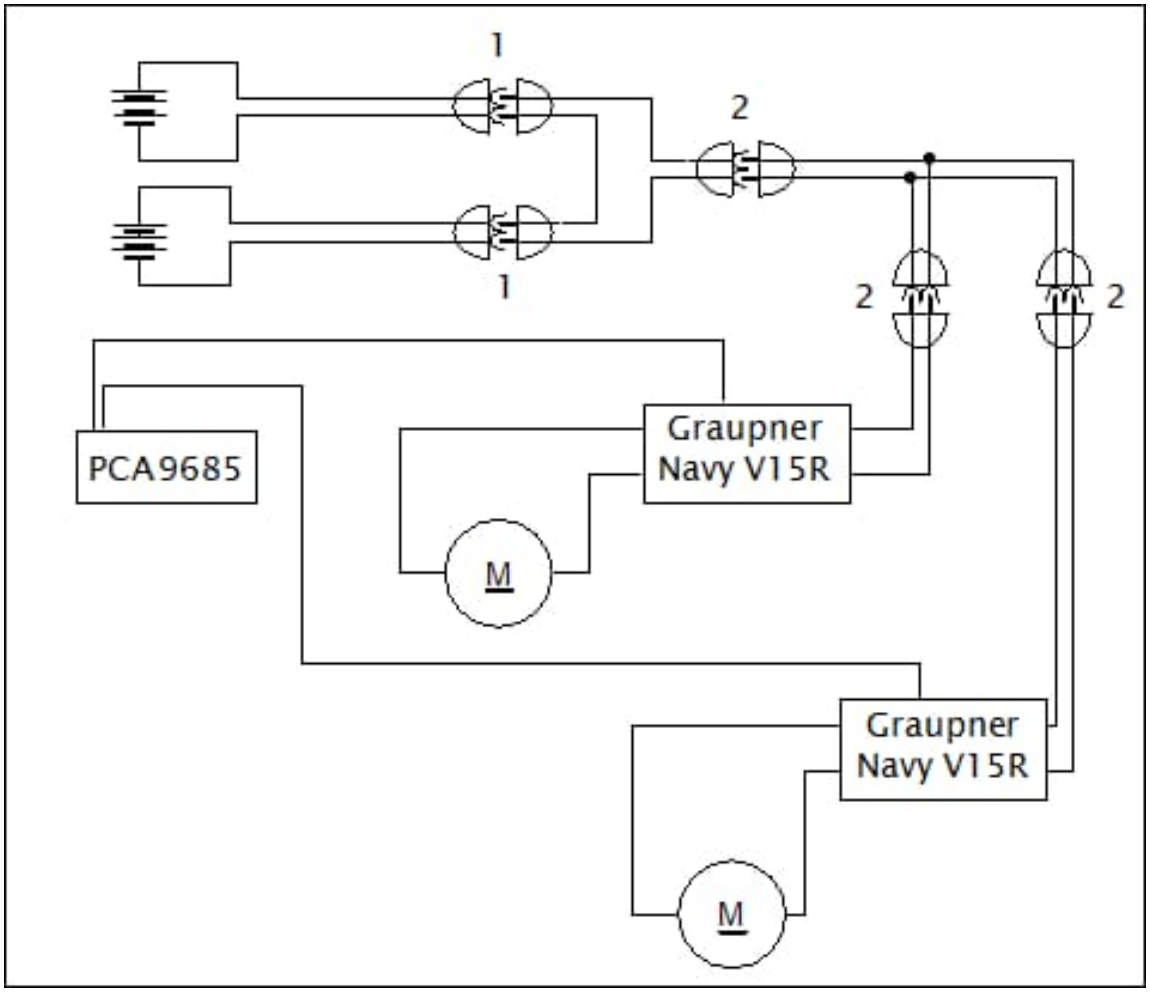

Figure 3.3: Power wiring

Figure 3.3 shows the power electronics. The boat has two brushed motors installed which we use to drive the propeller and the thruster.

Battery pack

We have used two 6V Lead–acid battery. The Type is "Sonnenschein A506/3,5 S". The battery's are parallel connected. The capacitance is 3,5 Ah. Both weigh 1400 grams. They are located in the lowest deck to put the center of mass as as deep as possible.

Connector typ 1

The T-connector is used for the 6V system to connect the battery's to the System.

Figure 3.4: 6V Connector (Typ 1)

Connector typ 2

For our 12V System we have used Tamiya connectors.

Figure 3.5: 12V Connector (Typ 2)

Chapter 4

Robot Operating System

4.1 About ROS

As a middleware we chose the *Robot Operating System (ROS)* . ROS contains many very powerfull tools, a messaging concept and a `Python` or `C++` API. To learn more about ROS heave a look at the ROS-Website[1]. In the following we assume basic knowledge about ROS and explain only project-specific or difficult things.

4.2 Installation

To install ROS in a simple and maintainable way there are a few things that you have to take into consideration.

First of all we tried to install ROS onto the default linux distribution for the Raspberry Pi Raspbian. The problem here was that we could only compile ROS from scratch and had to solve the dependencies our selfs (Have a look at the ros installation page[2]). This caused a first installation expense of at least 8 hours. Another problem is, that whenever you add changed nodes to your catkin_workspace, you have to recompile the entire project which takes about one minute each time. Since we got tired of this process pretty fast, we searched for alternatives to this installation process.

We discovered that we could use another linux distribution called Ubuntu Mate. This distro made it possible to install a precompiled ROS-Version and resolve the dependencies almost completely automatically. See the ROS Documentation[3] for instructions. With this installation workflow we could finally save a lot of time in our development process.

[1] **http://www.ros.org**
[2] **http://wiki.ros.org/melodic/Installation/Source**
[3] **http://wiki.ros.org/melodic/Installation/Ubuntu**

4.3 ROS Tools

ROS provides a large set of tools which save developers a lot of time developing, testing and simulating nodes. In this section we want to present a selection of those tools which helped us most.

4.3.1 Recording

To record and play messages ROS provides a tool called *rosbag*. This tool allows the user to record selected messages which are being published to the ROS-Master in so called *Bags* and play them back later for debugging, testing and simulating purposes. Although it can also be used for logging it is not its main purpose. It is also possible to *rebag* Bags which means to form a new Bag out of selected messagetopics from another Bag.

4.3.2 Distributed Processing

With ROS more than one system can cooperate together and act as a *distributed system*. As known, ROS consists of the **ROS-Master** and several **ROS-Nodes**. In a distributed ROS-System one Computer is the **Master** and the other one is called **Slave**.

Preconditions To set up the distributed system the following preconditions must be fulfilled:

- The computers involved must be in the same network.
- Each computer has a *hostname*. (Mostly you get this in Linux by the keyword `hostname`.)
- The computers can ping each other by hostname. (It might be necessary to add the *hostnames* and *IP-addresses* in the local `/etc/hosts`.)

ROS-Setup Setting up ROS for distributed usage

- On **all** computers exists an *environment variable* which holds the *URI* to **master** and is called `ROS_MASTER_URI`: Use following console command:

  ```
  export ROS_MASTER_URI=http://<HOSTNAME>:<PORT>
  ```

  Replace `<HOSTNAME>` with the *hostname* of the **master-computer**. Replace `<PORT>` with the ROS-Port. Standard is `11311`.

- Moreover the ros-nodes on each computer have to know their **own** `hostname`. Use following command to set the *environment variable*:

```
export ROS_HOSTNAME=<HOSTNAME>
```

Replace `<HOSTNAME>` with the `hostname` of the respective **slave-computer**.

Remember that for each console-window the environment variables have to be set individually.

4.4 Node-Graph

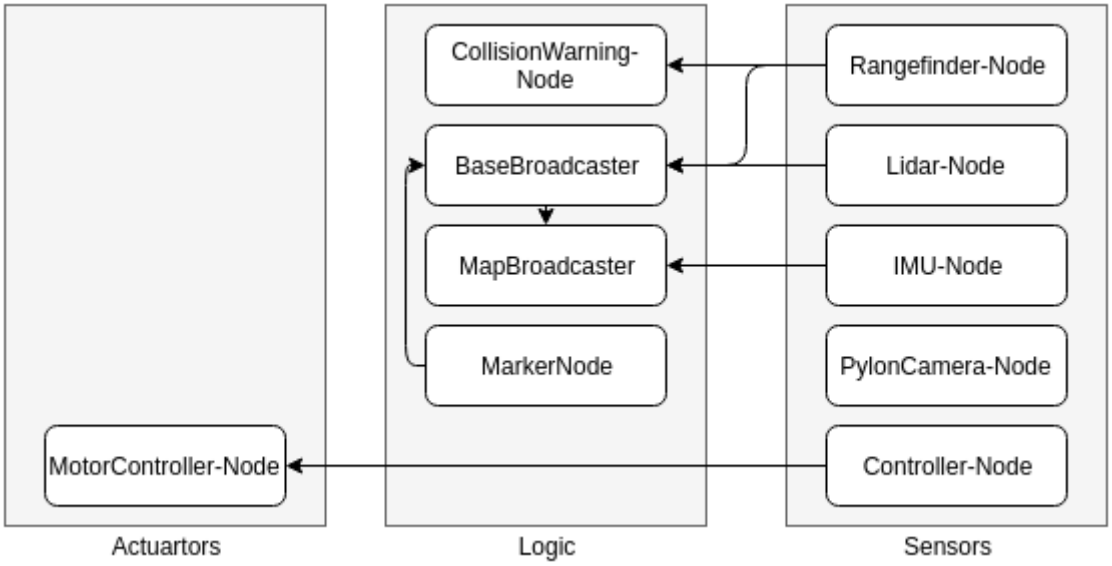

Figure 4.1: Node graph of the latest setup

Node	Topic	Message type
rangefinder	P: rangfinder/	sensor_msg/Range
imu	P: imu/	sensor_msg/Imu
pylon camera	P: pylon_camera_node/image_raw/	sensor_msg/Image
marker	P: marker/	visualization_msgs/Marker
motor_controller	S: motor_controller/<set_propeller/ set_rudder/set_thruster>	titanic/MotorThrottle titanic/ServoRad
lidar	P: laser/	sensor_msg/LaserScan

Table 4.1: All nodes with they're topics and messages types. P means publishes topic and S means subscribes to topic.

The Node-Graph can be structured in actuators, logic and sensors as shown in figure 4.1. Actuator-nodes are responsible for the movement of the ship. Only the MotorController-Node is in this section so far, it controls the bow thruster and the propeller motor. Nodes on the sensor layer sense the environment and nodes on the logic layer compute on data coming from the sensor layer. All nodes (with they're topics and message types) are listed in table 4.4.

4.4.1 Rangefinder-Node

The Rangefinder node is a wrapper for the Sharp GP2Y0E03 distance sensor as described in subsection 2.1.2. It reads data from a sensor connected via I2C and converts them into a distance. The distance is then published over a configurable Topic (default: **/rangefinder**) as a **Range**[4] message with a configurable frequency (default: 10Hz). The address is also configured using parameters. An example launchfile entry would be:

```
<node pkg="titanic" type="rangefinder_node">
    <param name="address" value="$(eval 0x40)"/>
    <param name="topic" value="rangefinder/heck_steuerbord"/>
</node>
```

4.4.2 Lidar-Node

For the integration of the hokuyo-lidar as described in subsection 2.1.1 we used the urg_node[5] provided from ros itself. It is designed to work with any SCIP 2.2 or earlier compliant laser range-finders.

As visible in the launch-file, we are starting the node with the following parameters:

```
<node name="lidar_node" pkg="urg_node" type="urg_node">
    ...
    <param name="publish_intensity" value="false"/>
    <param name="publish_multiecho" value="false"/>
    <param name="angle_min" value="-2.2689"/>
    <param name="angle_max" value="2.2689"/>
</node>
```

The *insensity-mode* is deactivated here. Therefore we only receive distance values and no brightness values. We also set a higher scan axis. As recommended in the ROS-Documentation the node publishes under the topic **scan** and the frame **laser**.

[4] **http://docs.ros.org/melodic/api/sensor_msgs/html/msg/Range.html**
[5] **http://wiki.ros.org/urg_node**

4.4.3 IMU-Node

This node was developed to publish the values from the MPU9255 as described in subsection 2.1.4. The IMU data is published over the **/imu**-Topic using the **IMU**[6] message with a frequency of around 100Hz.

Usability The Node as it is cannot be used right now. Due to the bad build quality of the used sensor only rotations around the x- and y-Axis can be measured reliable. To be functional the IMU data needs filtering which is not yet implemented.

4.4.4 PylonCamera-Node

This Node is responsible for the camera as described in subsection 2.1.3. We used an existing node which utilizes the Basler Pylon software suilt. For configuration details refer to the ros wiki[7].

4.4.5 Controller-Node

The Controller-Node communicates with the joy-Node which utilizes bluetooth to connect to a X-Box One controller. The controller events, which are present in the Topic **joy** and are published by the **joy_node**, have to be send to the motor controller via the **Motor Controller Node**. This requires another node in between in order to translate the events and is called **controller_node**.

Subscriptions As described earlier, the node has to subscribe to the Topic **joy** and has the message type **Joy**[8]. The values of the buttons and axis are sent in two arrays and for our use case can be selected and read with:

```
rightTrigger = data.axes[5]
leftTrigger = data.axes[2]
leftSteer = data.axes[0]
rightSteer = data.axes[3]
```

Topics This data can simply be calculated into our motor control data and published via the motor control Topics:

- **/motor_controller_node/set_propeller** To control the propller of the ship, we used the right and left trigger of the controller. As known from many video games, the right trigger is used for acceleration and the left trigger for breaking/reverse thrust. The control value for the motor controller is therefor calculated from the two trigger values:

[6] **http://docs.ros.org/melodic/api/sensor_msgs/html/msg/Imu.html**
[7] **http://wiki.ros.org/pylon_camera**
[8] **http://docs.ros.org/melodic/api/sensor_msgs/html/msg/Joy.html**

```
propeller = ((-rightTrigger + 1) / 2) - ((-leftTrigger + 1)/2)
```

This data is published using the custom message type **titanic::MotorThrottle** via the named topic.

```
propellerMsg = MotorThrottle()
//...
propellerMsg.percent = propeller
//...
pubPropeler.publish(propellerMsg)
```

This Messagetype holds control commands in a percentage **percent**.

- **motor_controller_node/set_thruster** Since the joy-Axes-Array holds values as percentages no value conversion has to be made:

```
thruster = leftSteer
//...
thrusterMsg.percent = thruster
```

We are also uing the custom message type: **titanic::MotorThrottle** here.

- **/motor_controller_node/set_servo** For the rudder we are using a control value range of $(+-100\%)$ which must be mapped into the control command range of $(+-0.785398RAD)$.

```
servo = rightSteer * 0.785398 # To Rad
```

This topic publishes the custom message type **titianic::ServoRad**, which sends the deflection of the rudder as a radiant in a float. *At this point the value range must be adapted if the rudder relents too much! During first tests we stumbled upon a problem, in which the steering rod dropped off the rudder connector.*

4.4.6 CollisionWarning-Node

As a first assistance system, a distance sensor based warning was introduced. Whenever the ship got too close to an obstacle (which could be detected by the sharp sensors) the Controller begins to rumble. This was made possible through the force feedback of the controller.

Subscribed Topics To realize the behaviour mentioned above, the Node has to subscribe to all sharp distance sensors and saves their distance values in a map.

```
distancesPrct = {"rangefinder/heck" : 0,
                 "rangefinder/heck_steuerbord" : 0,
                 "rangefinder/bug_steuerbord" : 0,
                 "rangefinder/heck_backbord" : 0,
```

```
                    "rangefinder/bug_backbord" : 0,
                    "rangefinder/bug" : 0}
```

The map is also used for extracting the names of the topics the node has to subscribe to:

```
for sensorNodeName in distancesPrct.keys():
    distancesPrct[sensorNodeName] = 0
    rospy.Subscriber(sensorNodeName, Range, handle_dsens, sensorNodeName)
```

The callback function used here is called *handle_dsens*. Its last parameter *sensorNodeName* tells the event handler the name of the triggering topic. This functionality is not explained properly in the ROS-Documentation. We adopted the usage from the example code included with ROS.
Before the measured distances are put into the map they have to be converted into percentages:

```
THRESHOLD = 0.10 #Threshold [m] between min_range and actual range
dif = data.range - data.min_range
procentNear =  (dif / THRESHOLD) #Percentage of Range between MINRange and
```

This percentage represents the range from the shortest distance measurable to a user defined threshold.

Topic To activate the corresponding vibration intensity on the controller, all values (of the map) are compared periodically and the largest percentage is published to the topic **/joy/set_feedback**
The topic has the message type **sensor_msgs::JoyFeedbackArray**, which constists of an array of **sensor_msgs::JoyFeedback**.
This message type is defined as follows:

```
# Declare of the type of feedback
uint8 TYPE_LED    = 0
uint8 TYPE_RUMBLE = 1
uint8 TYPE_BUZZER = 2

uint8 type

# This will hold an id number for each type of each feedback.
# Example, the first led would be id=0, the second would be id=1
uint8 id

# Intensity of the feedback, from 0.0 to 1.0, inclusive.  If device is
# actually binary, driver should treat 0<=x<0.5 as off, 0.5<=x<=1 as on.
float32 intensity
```

The X-Box controller we used has multiple motors for vibration. It has one in each trigger and two more in the center of the controller. Currently it is only possibe to control the motors in the center of the controller. The node **joy_node** is used here. Therefore:

```
type = TYPE_RUMBLE
```

must be used for the vibration.

```
id = 0
```

For the larger motor

```
id = 1
```

For the smaller motor which results in a softer vibration.

With help of the **JoyFeedbackArray** it is possible to control both motors at the same time.
For this node we are currently only using the larger motor:

```
msgFArray = JoyFeedbackArray()
msgF = JoyFeedback()
msgF.type = TYPE_RUMBLE
msgF.id = RUMBLE_STRONG_MOTOR
msgF.intensity = procentNear
msgFArray.array = [msgF]
publisher.publish(msgFArray)
```

Where *procentNear* is the top threshold mentioned above in the distance sensor mapping.

4.4.7 Marker-Node

The Marker-Node publishes a representation of the ship on the **/marker**-Topic via the **Marker**[9] message. The frequency is parametrized (default: 10Hz) and could be adjusted as follows:

```
<node pkg="titanic" type="marker_node">
    <param name="frequency" value="10"/>
</node>
```

4.4.8 MotorController-Node

This Node controls the thruster, propeller and rudder motors. The **motor_controller_node** subscribes following Messages:

[9] **http://docs.ros.org/melodic/api/visualization_msgs/html/msg/Marker.html**

/motor_controller_node/set_propeller It has the userdefinied messagtype **titanic::MotorThrottle**, which contains a thrustvalue as a percentage. Negativ values will result in a reverse thrust.
The topic sets the Speed of the propeller.

/motor_controller_node/set_thruster The same as above for the bow thruster.

/motor_controller_node/set_servo Message type is user defined **titanic::ServoRad** which contains the steering angel in radians. $(-45\,\mathrm{deg}\,to 45\,\mathrm{deg})$.

It simply maps the Messages to the right pwm-values using calculations.

4.5 Launch Configurations

ROS provides an easy way to start multiple nodes at once called **roslaunch**. This program takes as argument a so called launchfile where the nodes are specified.
In order to be flexible we created multiple launchfiles with different responsibilities, for example one launchfile for the sensor nodes. Then we combined those specific launchfiles in different setups, for example one for the nodes we want to start on the ship and one for the nodes we want to start on a remote laptop.
The launchfiles are located in the launch directory in the project root. The distribution of the launchfiles is shown in figure 4.2.

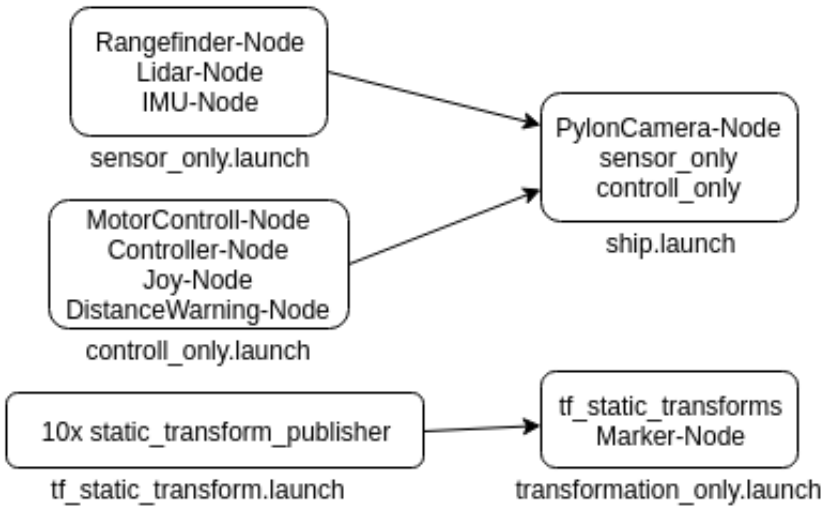

Figure 4.2: Distribution of the launchfiles

Chapter 5

Controlling

5.1 Xbox-Controller

For manual steering purpose, we decided us to use the **X-Box Wireless Controller**. It is connected via bluetooth and also has different kinds of force-feedback motors. Which offers some nice opportunities for assistance systems.

5.1.1 Driver

We used a third-Party Controller driver, which is especially designed for this Type of Controller [1].

5.1.2 Bluetooth Pairing and Connecting

This is a really complicated thing. This is my recommended workflow:

1. This first step you have to do on each reboot! (Sometimes it works without these lines of code)

```
sudo su
echo 1 > /sys/module/bluetooth/parameters/disable_ertm
service bluetooth restart
exit
```

2. The First Connection:
   Use the tool bluetoothctl, there are very nice manuals in Internet. Pair the Controller with the tool 'bluetoothctl'. Don't worry, if the Controller seems to connect and disconnect periodically. Then connect the Controller with the tool bluetoothctl. Restart the Controller by pushing the X-BOX-Button more than 5 Seconds. Then turn on again.

[1] **https://github.com/atar-axis/xpadneo**

Once the Controller is paired you only have to do Step 1 on each reboot. After that turn on the Controller and it will connect automatically.

5.1.3 Ros-Usage

In ROS you can use the joy package[2]. It contains the **joy_node**. Moreover it needs the following dependencies:

```
sudo apt-get install libspnav-dev
sudo apt-get install libbluetooth-dev
sudo apt-get install libcwiid-dev
```

5.1.4 Use the joy_node

For the correct feedback it is important to configure the device correct:

```
rosparam set joy_node/dev "/dev/input/<joystick>"
rosparam set joy_node/dev_ff "/dev/input/<joystickFF>"
rosparam set joy_node/autorepeat_rate <rate>
```

The First line sets the joystick input device. It is often *js0*.
The second line sets the joystick output device for force-feedback. It is often *event0* or *event1*. If one parameter is wrongly configured ROS will warn you with an Error.
The last line is the refresh rate. Its not well documented what it really does, but it seems to be important for force-feedback attack-rate. We get the best results with *50* Hz.

[2] **http://wiki.ros.org/joy/**

Chapter 6

Visualisation

6.1 TF-Frames

In order to process and visualize the data correctly, we are using the ROS concept **TF-Frames**. Our TF-Graph looks as shown in figure 6.1.

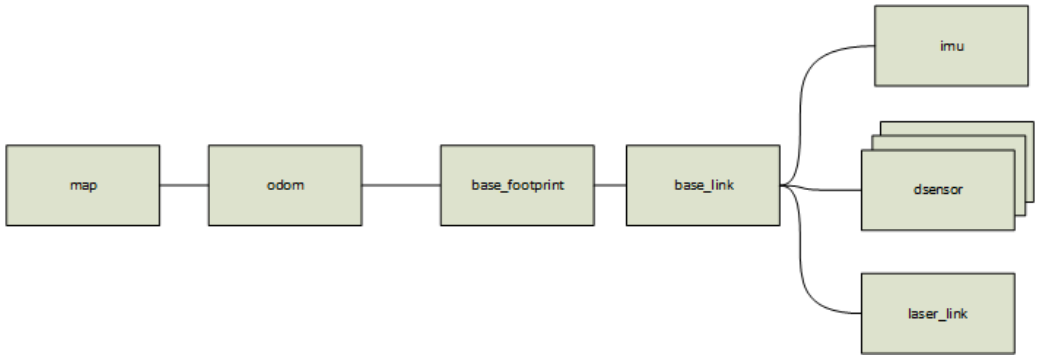

Figure 6.1: TF-Graph

This frame constellation is also recommended by ROS:

map This frame holds the map in which our ship is located.

odom This frame contains the odometry data. Usually this frame is used for the transformation with the **base_footprint** and odometry. But because our ship has no fully functional odometry we did not properly use this frame. Since some parts of the **ROS-Navigation-Stack** require the presence of the named frame, we either had to leave it empty or do the transformation statically. Later we will explain further how the odometry data can be faked. Nodes from the Navigation Stack then transform a **map** to **odom** while taking the transformation from **odom** to **base_footprint** into consideration.

base_footprint This frame is referring to the ship, without taking any horizontal or vertical tilting into consideration. It basically refers to a 2D shadow of the ship.

base_link This frame is referring to the ship, while taking any horizontal or vertical tilting into consideration. The tranformation it and the **base_footprint** should happen via the IMU-Data. In the frame **base_link** also the individual sensor will be transformed.

The transformations used here are generated by the *Tf2-Lib* :

laser_link_static_boradcaster this static broadcaster is transforming between **base_link** and **laser_link**.

dsensor_link_static_boradcaster_⟨Position⟩ This static Tf2-Broadcaster is transforming between the corresponding **dsensor/⟨Position⟩** and **base_link**.

imu_link_static_boradcaster This one is transforming **imu** and **base_link**.

maker_node This Node is broadcasting the marker (shipmodel) onto the **base_link**.

Chapter 7

Navigation

7.1 Hector Slam

At first we tried to generate a map of the environment just by using the lidar-data. ROS offers predefined nodes for this job in its **Navigation Stack**. Unfortunately these require a working odometry which we did not have, as mentioned above.
But then we stumbled upon the package **hector_slam** created by the **TU Darmstadt**. This node has the following advantages:

1. Fast
2. Low Cost
3. Robust Scanmatching
4. 6DOF suitable
5. Odometry not needed
6. Hard Realtime

The first two advantages are very important due to our very low processing speed on the ship. Since the system is 6DOF suitable it allows our ship to tilt along with waves without influencing the map creation. The fifth point is strongly required, because we have no working odometry.

7.1.1 Tests at the University

At first we tested the map creation at our Univesity. Since we already reported problems with our IMU at that time, the lack of tilt (no waves in our test laboratory) was an advantage here.

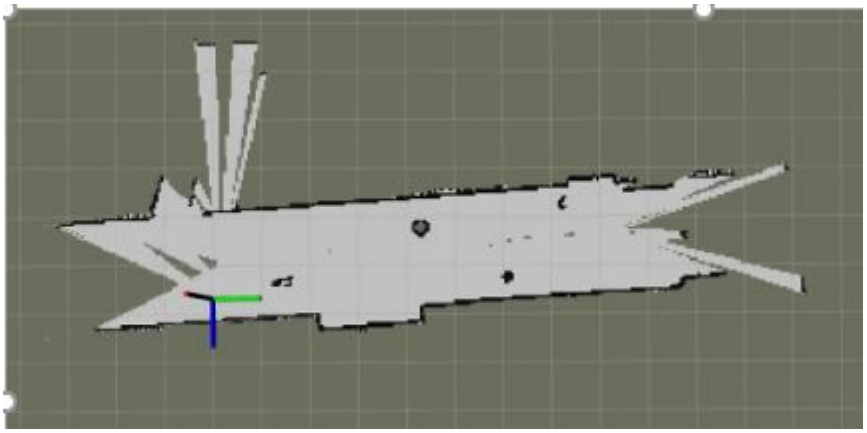

Figure 7.1: The first map we created. We placed a few obstacles into our hallway and dragged the ship along it.

As figure 7.1 shows, we achieved pretty good results here. All the obstacles were detected by our sensors and correctly placed into a pretty precise map of the ships environment. Only some problems were detected when parts of the hallway looked identical for the ship. But since the terrain at the Miniaturwunderland Hamburg is extremely detailed, we would probably not stumble upon this problem there.

7.1.2 At the Miniaturwunderland Hamburg

At the Miniaturwunderland the problem with the tilt caused by the waves became part of the tests. The navigation system was supposed to correct this problem using the IMU-values, but since we had problems with it (mentioned above), we completely deactivated it. The pilot in command tried to keep the ship as steady as possible, but still a slight tilting is visible in the collected data. This can be recognized by duplicate lines in the map plot. Another problem we detected, was that the border of the pool was not high enough. This caused the lidar to look through a class pane above the actual pool border. The navigation system then tried to include people passing by into the map creation. Also moving obstacles on the water (other ships) were included in the map creation and could only be removed by passing the are multiple times after the obstacles were gone. A generated map is shown in figure 7.2.

7.2 AMCL

> "amcl is a probabilistic localization system for a robot moving in 2D. It implements the adaptive (or KLD-sampling) Monte Carlo localization approach (as described by Dieter Fox), which uses a particle filter to track the pose of a robot

against a known map." (Package summary - amcl node[1])

The maps generated from our hector_slam node together with our laserscan data got fed into the amcl node and we where able to navigate inside the given map. We did those tests not in the live usage but generated the maps offline with recorded rosbags and tried the amcl node offline on this data. We got a problem when using amcl because it needs odometry data which we don't have because of our bad quality IMU. As a solution we generated an estimation of the 2D odometry using the rf2o node[2]. With that kind of hacked workaround we were able to estimate our correct position and orientation multiple times but the system is far from reliable. That might be because all information for the calculation come from the laser scan. In the future we will use a calibrated IMU and expect far better results.

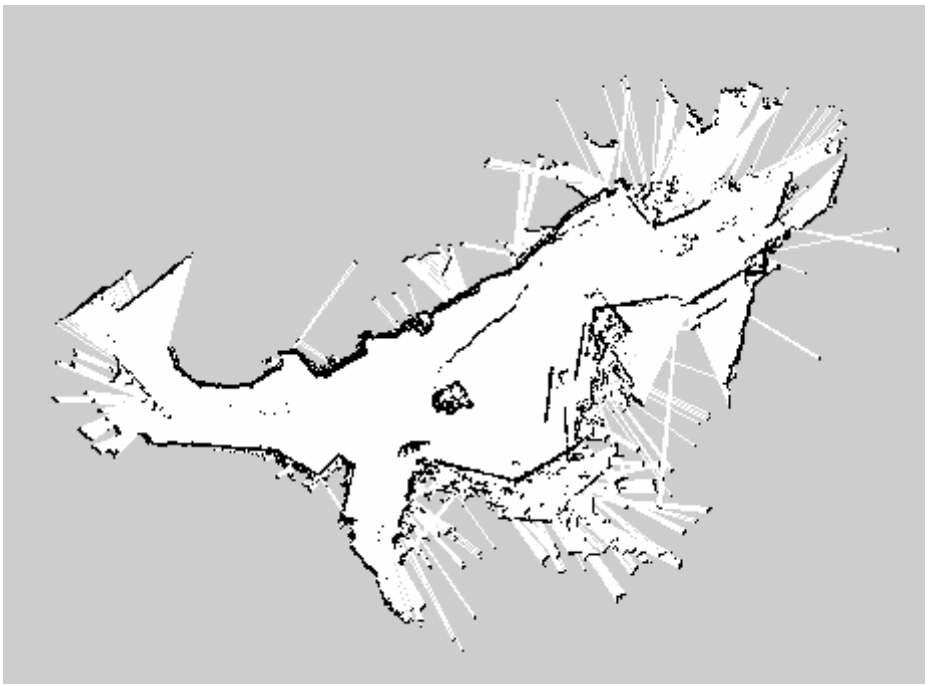

Figure 7.2: Generated at the Miniatur-Wunderland

[1] **http://wiki.ros.org/amcl**
[2] **http://wiki.ros.org/rf2o**

Chapter 8

Future Outlook

This project offers many opportunities to continue. Of course the biggest goal will be to make the ship fully autonomous. Our setup showed, that our sensors are capable of tracking the ship correctly. Therefor only software changes have to be made and an algorithm for complete autonomy has to be implemented.

Also something we learned from our tests was, that the Raspberry Pi 3B+ used as the "brain" of our ship is not as powerful as our use case requires. In particular when we had to log a lot of data onto the SD-card, we noticed significant performance drops, which caused the remote control to be extremely in-reactive. Although we could fix this issue through optimizing our log processes, it would probably be advisable to use a more powerful processing board in the future. Maybe something like the ODroid H2 would be better, because it offers a stronger cpu and has a faster harddrive interface. Also USB 3.0 support makes throughput via usb much faster.

Currently all our nodes (except one) have been written in Python, which is an interpreted programming language. Because it is interpreted and does not directly generate cpu instructions, we lose a little bit of performance, which in our case has to be taken into consideration. Therefor we are planning to rewrite the project into C++ Code, which will give us a noticeable performance boost.

Chapter 9

Images

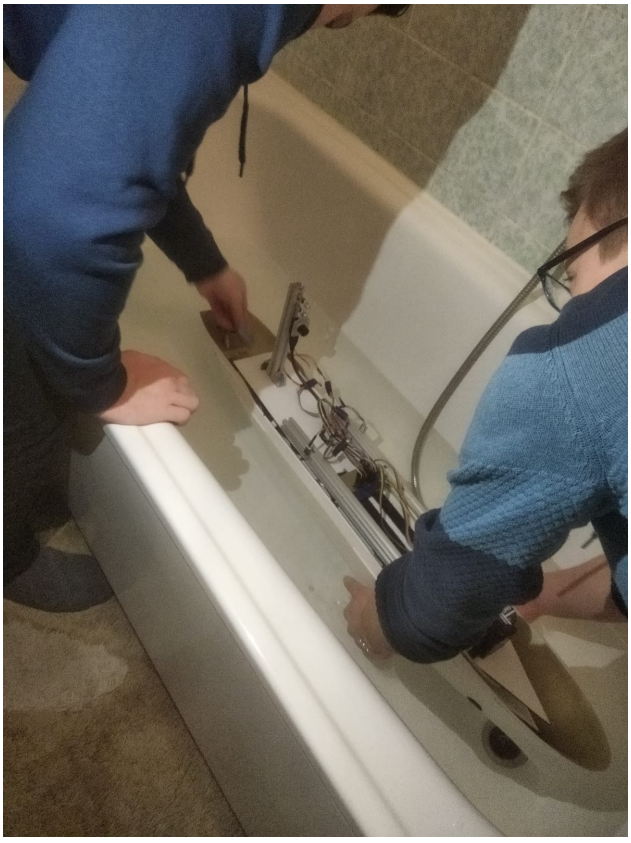

Figure 9.1: The first watering of the ship. Fortunately only a small amount of trimming was needed to keep it stable.

Figure 9.2: The first visit at the Miniaturwunderland Hamburg with our ship.

Figure 9.3: Although the ship was still controlled via our X-Box controller, the data we collected was very useful.

Figure 9.4: Collecting data of the extremely detailed terrain.

Figure 9.5: While collecting data, multiple laptops were used to monitor them in realtime and control the launchfiles.

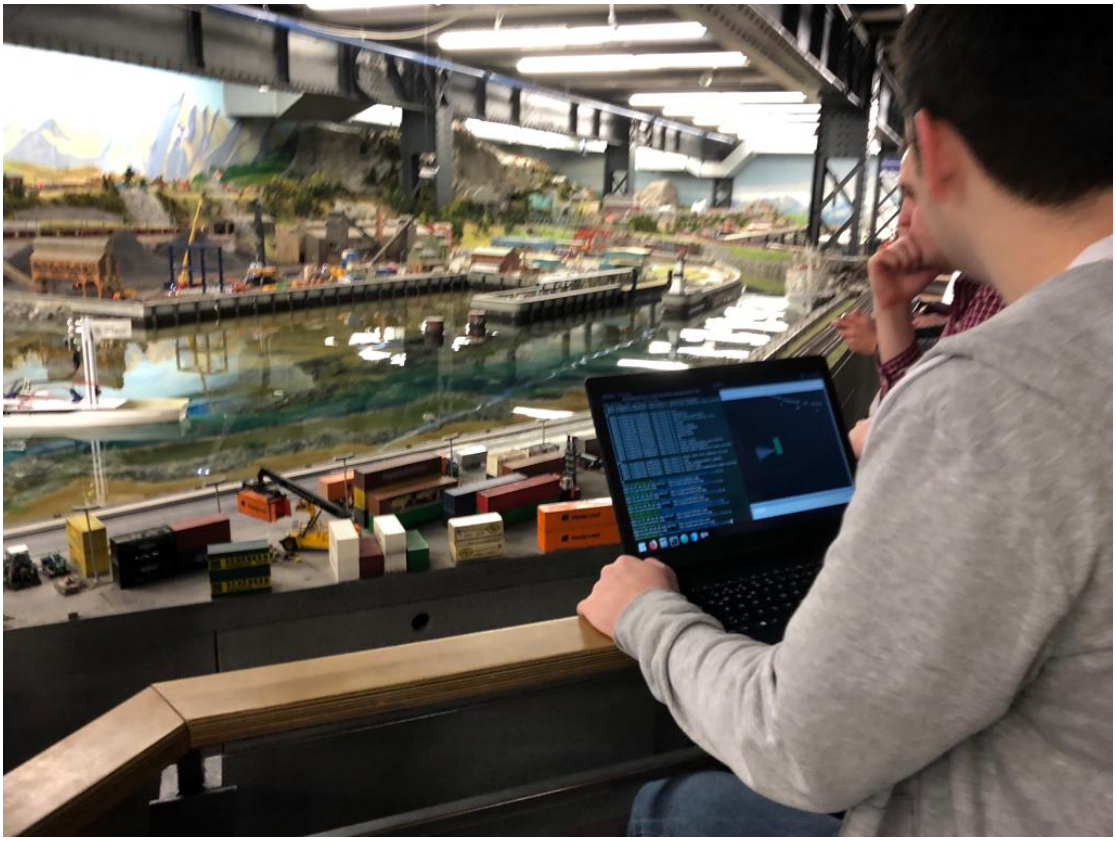

Figure 9.6: While driving around the area, we got a live evaluation of the collected data and received dierct feedback on how good our system could collect the information about its location and surrounding.

Figure 9.7: All in all the project was a great success and the team is looking forward to future cooperations with the Miniaturwunderland Hamburg and making the ship completely autonomous in the future.